

Relazione Progetto: Sistema di Rilevamento delle Intrusioni (IDS) Basato su Graph Neural Network (GNN) e Attacking Agent basato su Reinforcement Learning

Giorgio Colella (0522501752) e Maria Caterina D'Aloia(0522501852)

18 febbraio 2025

Indice

1 Introduzione	3
2 Metodologia	4
2.0.1 IDS basati su GNN	4
2.0.2 GAT	5
2.0.3 GraphSAGE	6
2.0.4 GCN	8
2.0.5 GCN multi-minaccia	9
2.1 IDS basati su classificatori binari	10
2.1.1 Random Forest	10
2.1.2 XGBoost	11
2.1.3 SVM	12
2.2 Agenti con Reinforcement Learning	13
2.2.1 DDQN	14
2.2.2 DQN	15
2.2.3 Q-Learning	17
2.2.4 SARSA	18
2.3 Motivazione delle nostre scelte	19
2.4 Ambiente	22
3 Risultati	23
3.1 Dataset	23
3.2 Metriche	23
3.2.1 Accuracy (Accuratezza)	23
3.2.2 Matrice di Confusione	24
3.2.3 Balanced Accuracy	24
3.2.4 Precision (Precisione)	24
3.2.5 Recall (Sensibilità o True Positive Rate)	24
3.2.6 F1-score	25
3.2.7 Matthews Correlation Coefficient (MCC)	25

3.2.8	ROC e AUC (Receiver Operating Characteristic e Area Under the Curve)	25
3.2.9	Chiave di lettura grafo agente	25
3.3	IDS basati su Graph Neural Network	26
3.3.1	GAT binario	26
3.3.2	GCN binario	28
3.3.3	GCN multi-classe	29
3.4	IDS binari con Agente DDQN	30
3.4.1	SVM	30
3.4.2	XGBoost	34
3.4.3	Random Forest	36
3.5	Agenti	41
3.5.1	DDQN con normalizzazione e Reward Imbalance Factor	41
3.5.2	DDQN con normalizzazione delle reward ma senza Reward Imbalance Factor	44
3.5.3	DDQN senza normalizzazione e senza Reward Imbalance Factor	47
3.5.4	DDQN senza normalizzazione ma con Reward Imbalance Factor	50
3.5.5	DDQN con normalizzazione delle reward ma senza Reward Imbalance Factor test su 2000 episodi	53
3.5.6	DQN con normalizzazione ed imbalance factor	57
3.5.7	DQN con normalizzazione ma senza imbalance factor	60
3.5.8	Q-Learning con rete neurale	63
3.5.9	Q-Learning standard con lookup table senza normalizzazione ma con imbalance factor	66
3.5.10	Q-Learning standard con lookup table con normalizzazione ma senza imbalance factor	69
3.5.11	SARSA con normalizzazione ed imbalance factor	72
3.6	Risultati agenti ed IDS	75
3.6.1	GAT/DQN con normalizzazione delle reward ed imbalance factor	75
3.6.2	GAT/DQN con learning rate alto, normalizzazione delle reward ed imbalance factor	75
3.6.3	GAT/DQN con normalizzazione delle reward ma senza imbalance factor	75
3.6.4	GAT/DDQN con normalizzazione ma senza imbalance factor e con 2000 episodi	75
3.6.5	GraphSage/DDQN con normalizzazione ma senza imbalance factor	75
3.6.6	GraphSage/DDQN con normalizzazione e con imbalance factor	75
3.6.7	GCN-multiclasse/DDQN	75
3.6.8	GCN/SARSA con normalizzazione ma senza imbalance factor	75
4	Conclusioni	75

Sommario

La sicurezza informatica è una delle principali sfide nei sistemi moderni, in particolare per la protezione delle reti contro attacchi informatici sempre più sofisticati. I Sistemi di Rilevamento delle Intrusioni (IDS) tradizionali si basano spesso su regole statiche o metodi di apprendimento supervisionato, risultando vulnerabili a minacce nuove ed evolutive. In questo lavoro, proponiamo un IDS basato su **Graph Neural Networks (GNN)**, testando due architetture principali: **Graph Convolutional Networks (GCN)** e **Graph Attention Networks (GAT)**.

Il nostro IDS viene sottoposto a un ambiente dinamico in cui un agente di attacco, basato su **Reinforcement Learning (RL)**, genera traffico benevolo e malevolo, adattandosi progressivamente per evadere il rilevamento. L'IDS e l'agente d'attacco imparano l'uno dall'altro in un processo di co-evoluzione, con il primo che raffina la propria capacità di classificazione mentre il secondo sviluppa strategie d'attacco più sofisticate.

Nel corso della sperimentazione, abbiamo analizzato l'efficacia delle GNN in più scenari, testando un **GCN multi-minaccia** per valutare la capacità di generalizzazione del modello rispetto a diversi tipi di attacco. Inoltre, abbiamo confrontato le GNN con **classificatori binari tradizionali**, per comprendere i vantaggi dell'approccio basato su grafi. Infine, abbiamo sperimentato diverse strategie di apprendimento per l'agente d'attacco, confrontando i metodi **DDQN**, **DQN**, **Q-Learning** e **SARSA**. I risultati ottenuti forniscono spunti significativi sul ruolo delle GNN nella sicurezza informatica e sull'efficacia dell'apprendimento dinamico per il rilevamento di attacchi avanzati.

1 Introduzione

La crescente complessità delle infrastrutture informatiche e l'aumento del traffico di rete hanno reso i **Sistemi di Rilevamento delle Intrusioni (IDS)** strumenti essenziali per la sicurezza delle reti. Tuttavia, i metodi tradizionali di rilevamento basati su **pattern matching** o **machine learning supervisionato** faticano a contrastare attacchi sofisticati, specialmente quelli che si evolvono nel tempo per eludere le difese statiche. L'applicazione di **modelli di apprendimento adattivo**, in grado di apprendere da attacchi sempre nuovi, rappresenta una direzione promettente per migliorare la sicurezza delle reti.

In questo contesto, proponiamo un IDS basato su **Graph Neural Networks (GNN)**, con l'obiettivo di modellare il traffico di rete come un grafo e sfruttare le proprietà topologiche per il rilevamento delle intrusioni. Le GNN, ed in particolare le **Graph Convolutional Networks (GCN)** e le **Graph Attention Networks (GAT)**, permettono di apprendere rappresentazioni strutturali che possono rivelarsi più efficaci rispetto alle tecniche di classificazione tradizionali.

Un elemento innovativo del nostro approccio è l'integrazione di un **agente d'attacco basato su Reinforcement Learning (RL)**. Questo agente genera sia traffico lecito che malevolo, adattando le proprie strategie nel tempo per eludere il sistema di rilevamento. In risposta, l'IDS rafforza progressivamente la propria capacità di identificare le intrusioni attraverso l'apprendimento continuo. Questa co-evoluzione tra attaccante e difensore simula scenari realistici in cui gli attaccanti modificano le loro tecniche per sfuggire al rilevamento, mentre i sistemi di sicurezza aggiornano le loro strategie di difesa.

Nel corso della sperimentazione, abbiamo analizzato diversi aspetti chiave:

- **Confronto tra GCN e GAT:** valutazione delle prestazioni delle due architetture GNN nel contesto della sicurezza di rete.
- **GCN multi-minaccia:** test sulla capacità di un modello GCN di adattarsi a molteplici tipologie di attacco.
- **Confronto con classificatori binari:** analisi comparativa con modelli di machine learning tradizionali per valutare i vantaggi e gli svantaggi dell'approccio basato su grafi.
- **Strategie di apprendimento per l'agente d'attacco:** confronto tra diversi algoritmi di reinforcement learning (**DDQN**, **DQN**, **Q-Learning** e **SARSA**) per valutare quale sia più efficace nel generare attacchi che eludano l'IDS.

L'obiettivo di questo lavoro è duplice:

1. **Valutare l'efficacia delle GNN (GCN e GAT) nel rilevamento delle intrusioni** rispetto ad altre tecniche di apprendimento automatico.
2. **Analizzare la capacità di adattamento dell'IDS** in presenza di un attaccante dinamico che modifica le proprie strategie per superare le difese.

I risultati ottenuti forniranno indicazioni sulla validità dell'uso delle GNN per il rilevamento di minacce e sulla fattibilità di un IDS adattivo basato su apprendimento interattivo.

Abbiamo effettuato un'estensiva ricerca in letteratura, gli articoli che abbiamo visionato sono presenti in bibliografia.

La repository GitHub con il codice da noi sviluppato è accessibile [qui](#).

2 Metodologia

2.0.1 IDS basati su GNN

Le Graph Neural Networks (GNN) sono una classe di reti neurali progettate per elaborare dati strutturati come grafi. I grafi sono strutture fondamentali per rappresentare dati relazionali, dove un insieme di nodi (vertici) è connesso tramite archi (edge). Le GNN sfruttano la struttura del grafo per apprendere rappresentazioni significative dei nodi, degli archi e dell'intero grafo.

Un grafo può essere formalmente definito come:

$$G = (V, E, X)$$

dove:

- $V = \{v_1, v_2, \dots, v_n\}$ è l'insieme dei nodi.
- $E \subseteq V \times V$ è l'insieme degli archi (connessioni tra nodi).
- $X \in \mathbb{R}^{|V| \times d}$ è una matrice di feature, dove ogni nodo è associato a un vettore di caratteristiche di dimensione d .

Le GNN propagano informazioni attraverso la struttura del grafo, permettendo ai nodi di aggregare le informazioni dai loro vicini e aggiornare le loro rappresentazioni.

L'apprendimento in una GNN avviene attraverso un meccanismo di message passing, in cui ogni nodo aggiorna il proprio stato sulla base delle informazioni ricevute dai nodi vicini. Ad ogni layer della rete, un nodo v_i aggiorna la propria rappresentazione $\mathbf{h}_i^{(k)}$ come segue:

$$\mathbf{h}_i^{(k+1)} = \sigma \left(W^{(k)} \cdot \text{AGGREGATE} \left(\{\mathbf{h}_j^{(k)} | j \in \mathcal{N}(i)\} \right) \right)$$

dove:

- $\mathcal{N}(i)$ rappresenta il vicinato del nodo i ,
- AGGREGATE(\cdot) è una funzione di aggregazione dei messaggi dai nodi vicini (es. somma, media, massimo),
- $W^{(k)}$ è una matrice di pesi appresa dal modello,
- σ è una funzione di attivazione non lineare (come ReLU o ELU).

Dopo più passaggi di propagazione, le rappresentazioni dei nodi incorporano informazioni sia locali (vicini diretti) che globali (interi componenti connesse del grafo).

2.0.2 GAT

Una variante dei GNN è il Graph Attention Network (GAT), introdotto per migliorare l'aggregazione delle informazioni tra nodi assegnando pesi adattivi agli edge.

Il meccanismo principale di GAT si basa su:

- **Self-attention:** ogni nodo calcola un peso di attenzione per i suoi vicini, enfatizzando le connessioni più informative.
- **Aggregazione pesata:** le feature dei vicini vengono combinate utilizzando i pesi di attenzione appresi dinamicamente.
- **Multi-head attention:** il modello utilizza più meccanismi di attenzione paralleli per aumentare la stabilità dell'apprendimento.
- **Funzioni di attivazione non lineari:** l'attivazione Exponential Linear Unit (ELU) viene comunemente utilizzata dopo ogni operazione di convoluzione.

Formalmente, l'attenzione viene calcolata come:

$$\alpha_{ij} = \frac{\exp \left(\text{LeakyReLU} \left(\mathbf{a}^T [\mathbf{h}_i || \mathbf{h}_j] \right) \right)}{\sum_{k \in \mathcal{N}_i} \exp \left(\text{LeakyReLU} \left(\mathbf{a}^T [\mathbf{h}_i || \mathbf{h}_k] \right) \right)}$$

dove:

- \mathbf{h}_i e \mathbf{h}_j sono le rappresentazioni dei nodi,
- \mathbf{a} è un vettore di parametri appreso,
- \mathcal{N}_i è il vicinato del nodo i .

L'output finale viene ottenuto aggregando le feature dei vicini con i pesi di attenzione:

$$\mathbf{h}'_i = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} \mathbf{h}_j \right)$$

dove σ è una funzione di attivazione e \mathbf{W} è una matrice di trasformazione appresa. Per modellare GAT come un IDS, abbiamo proceduto così:

1. Definizione del modello GAT:

- Due strati di convoluzione GATConv: il primo utilizza attenzione multi-head, il secondo aggredisce i risultati.
- Un livello lineare finale per produrre l'output binario (benigno/malevolo).
- Opzionalmente, viene applicato un dropout per ridurre l'overfitting.

2. Pre-elaborazione dei dati:

- Il dataset viene letto da un file CSV e convertito in un grafo utilizzando un K-Nearest Neighbors (KNN) Graph.
- Gli edge sono filtrati in base a una soglia di distanza (90° percentile) per rimuovere connessioni deboli.
- I dati vengono convertiti in tensori PyTorch e organizzati nel formato richiesto da PyTorch Geometric.

3. Pretraining e Bilanciamento delle Classi:

- Una funzione di pretraining estrae i dati e li converte in un grafo pronto per l'addestramento.
- Se i dati di rete sono sbilanciati (es. molte più connessioni legittime rispetto a quelle malevoli), vengono applicate tecniche di oversampling e undersampling per garantire un equilibrio tra classi.

4. Retraining con dati bilanciati:

- Il modello viene riaddestrato su dati bilanciati, con una normalizzazione delle feature per migliorare la stabilità dell'apprendimento.
- La funzione di perdita utilizzata è Binary Cross Entropy with Logits Loss (BCEWithLogitsLoss), adatta per classificazione binaria.

2.0.3 GraphSAGE

Una variante dei GNN è il Graph Sample and Aggregate (GraphSAGE), introdotto per migliorare l'aggregazione delle informazioni tra nodi utilizzando campionamento stocastico e aggiornamento delle feature in modo efficiente.

Il meccanismo principale di GraphSAGE si basa su:

- **Campionamento dei vicini:** invece di aggregare informazioni da tutti i vicini, GraphSAGE campiona un sottoinsieme per ridurre la complessità computazionale.

- **Aggregazione differenziabile:** le feature dei nodi campionati vengono combinate con il nodo centrale utilizzando funzioni di aggregazione specifiche come media, LSTM o pooling.
- **Aggiornamento delle feature:** le nuove feature dei nodi vengono calcolate combinando le informazioni aggregate con le feature originali del nodo stesso.
- **Trasferibilità a grafi non visti:** il modello appreso può generalizzare a nodi mai visti in grafi simili, rendendo GraphSAGE adatto a scenari in cui il grafo cambia nel tempo.

Formalmente, l'aggregazione viene definita come:

$$\mathbf{h}_i^{(k)} = \sigma \left(\mathbf{W}^{(k)} \cdot \text{AGG}^{(k)} \left(\{\mathbf{h}_j^{(k-1)}, \forall j \in \mathcal{N}_i\} \right) \right)$$

dove:

- $\mathbf{h}_i^{(k)}$ è la rappresentazione del nodo i al livello k ,
- $\mathbf{W}^{(k)}$ è la matrice di pesi appresa,
- $\text{AGG}^{(k)}$ è la funzione di aggregazione,
- \mathcal{N}_i è il vicinato del nodo i ,
- σ è una funzione di attivazione non lineare.

L'output finale viene ottenuto combinando l'aggregazione con le feature originali:

$$\mathbf{h}'_i = \text{normalize} \left(\text{concat} \left(\mathbf{h}_i^{(K)}, \mathbf{h}_i^{(0)} \right) \right)$$

Per modellare GraphSAGE come un IDS, abbiamo proceduto così:

1. Definizione del modello GraphSAGE:

- Due strati di convoluzione GraphSAGE: il primo utilizza aggregazione media, il secondo combina i risultati.
- Un livello lineare finale per produrre l'output binario (benigno/malevolo).
- Dropout applicato tra gli strati per ridurre l'overfitting.

2. Pre-elaborazione dei dati:

- Il dataset viene letto da un file CSV e convertito in un grafo mediante un K-Nearest Neighbors (KNN) Graph.
- Gli edge vengono filtrati in base alla distanza per rimuovere connessioni deboli.
- I dati vengono normalizzati e convertiti in tensori PyTorch per l'addestramento.

3. Pretraining e Bilanciamento delle Classi:

- I dati vengono campionati per garantire un equilibrio tra le classi.
- L'addestramento avviene in modo progressivo per stabilizzare l'aggiornamento delle feature.

4. Retraining con dati bilanciati:

- Il modello viene riaddestrato con tecniche di normalizzazione e bilanciamento delle classi.
- La funzione di perdita utilizzata è Binary Cross Entropy with Logits Loss (BCEWithLogitsLoss).

2.0.4 GCN

Le **Graph Convolutional Networks (GCN)** rappresentano una classe di reti neurali progettate per elaborare dati strutturati come grafi. A differenza delle reti neurali convenzionali, che operano su dati tabulari o immagini, le GCN utilizzano la struttura del grafo per apprendere rappresentazioni significative, propagando informazioni attraverso le connessioni tra nodi.

L'operazione principale delle GCN è la convoluzione sui grafi, definita come:

$$H^{(k+1)} = \sigma \left(\tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2} H^{(k)} W^{(k)} \right)$$

dove:

- $\tilde{A} = A + I$ è la matrice di adiacenza con l'aggiunta di connessioni residue (self-loops),
- \tilde{D} è la matrice diagonale delle somme dei gradi dei nodi,
- $H^{(k)}$ rappresenta le feature dei nodi allo strato k ,
- $W^{(k)}$ è una matrice di pesi appresa dal modello,
- σ è una funzione di attivazione (es. ReLU).

L'operazione di convoluzione nei grafi permette ai nodi di aggiornare le proprie feature in base ai vicini, propagando informazioni attraverso la topologia del grafo.

Il codice si compone di diverse componenti principali:

1. Definizione del modello GCN:

- Due strati convoluzionali GCNConv, con funzioni di attivazione ReLU.
- Un livello lineare finale per produrre un output binario (benigno/malevolo).
- Opzionalmente, viene applicato un dropout per ridurre l'overfitting.

2. Pre-elaborazione dei dati:

- Il dataset viene letto da un file CSV e convertito in un grafo utilizzando K-Nearest Neighbors (KNN).
- Gli edge vengono filtrati in base a una soglia di distanza per rimuovere connessioni meno significative.
- I dati vengono trasformati in tensori PyTorch e organizzati nel formato richiesto da PyTorch Geometric.

3. Pretraining del modello:

- Il modello può essere inizialmente addestrato su un dataset CSV per la generazione del grafo e l'estrazione delle feature.

4. Bilanciamento delle classi:

- Tecniche di oversampling e undersampling vengono applicate per riequilibrare la distribuzione tra traffico benigno e malevolo.
- Calcolo dei pesi di classe dinamici per gestire sbilanciamenti durante l'addestramento.

2.0.5 GCN multi-minaccia

Ad un certo punto della sperimentazione, ci è sembrato di non utilizzare tutte le potenzialità del GCN, abbiamo deciso d'implementare un modello multi-classe che ci permettesse di rilevare anche i tipi differenti di manacce:

1. Architettura del modello GCN:

- Il modello ora utilizza **tre strati convoluzionali** anziché due, con l'aggiunta di un terzo livello `GCNConv`.
- È stato introdotto il **Batch Normalization (BatchNorm1d)** dopo ogni strato convoluzionale per migliorare la stabilità del training.
- Il modello supporta un numero variabile di classi grazie alla modifica della **dimensione dell'output layer** (`num_classes` invece di un'uscita binaria).

2. Modifica della funzione di bilanciamento dei dati:

- Il bilanciamento dei dati ora combina undersampling e oversampling dinamici:
 - Le classi con meno del 70% della classe dominante vengono oversampdate.
 - Le classi con più del 30% della classe minoritaria vengono undersampdate.
- I dati riequilibrati vengono normalizzati prima dell'addestramento.

3. Uso di CrossEntropyLoss con pesi adattivi per classi sbilanciate:

- La loss function ora utilizza **CrossEntropyLoss** invece di `BCEWithLogitsLoss` per supportare più classi.
- I pesi delle classi vengono calcolati dinamicamente in base alla frequenza di ciascuna classe e normalizzati.
- Un `weight_tensor` assegna maggiore peso alle classi meno frequenti, migliorando la robustezza del modello rispetto a dataset sbilanciati.

4. Modifica alla pre-elaborazione dei dati:

- Il dataset viene letto separando esplicitamente le feature dalle etichette (`df.iloc[:, :-1]` e `df.iloc[:, -1]`).
- Le etichette vengono trasformate in valori numerici utilizzando `pd.factorize()`.

5. Miglioramento della gestione del grafo:

- La costruzione del grafo basata su K-Nearest Neighbors (KNN) è stata mantenuta, ma il filtraggio degli archi ora elimina connessioni sopra la soglia del **90° percentile** delle distanze.
- Il tensore delle etichette è stato modificato per supportare la classificazione multi-classe (`dtype=torch.long` invece di `torch.float32`).

2.1 IDS basati su classificatori binari

Abbiamo anche implementato dei classificatori binari in modo tale da poterli confrontare con i modelli basati su Graph Neural Network, le nostre scelte d'implementazione sono state le seguenti.

2.1.1 Random Forest

Random Forest (RF) è un algoritmo di apprendimento supervisionato basato su alberi decisionali. Si tratta di un modello di **ensemble learning**, in cui vengono addestrati più alberi di decisione su sottoinsiemi casuali del dataset, e la predizione finale viene ottenuta aggregando le uscite dei singoli alberi. L'idea principale è ridurre l'overfitting e migliorare la generalizzazione rispetto a un singolo albero decisionale.

Un Random Forest è composto da N alberi decisionali addestrati su dati campionati casualmente con rimpiazzo (*bootstrap sampling*). Ogni albero prende decisioni indipendentemente e la previsione finale è ottenuta tramite:

- Classificazione: maggioranza delle previsioni (voto a maggioranza).
- Regressione: media delle predizioni degli alberi.

Dati N alberi di decisione, la predizione finale per un input x è:

$$\hat{y} = \frac{1}{N} \sum_{i=1}^N h_i(x)$$

dove $h_i(x)$ è la predizione dell'i-esimo albero.

L'implementazione sfrutta una Random Forest con gestione della memoria persistente per il rilevamento delle intrusioni nel traffico di rete:

1. Definizione della classe RFIDS:

- Il modello utilizza una **RandomForestClassifier** con 100 alberi e class balancing attivo.
- Il sistema è dotato di memoria persistente per conservare e aggiornare i dati di addestramento nel tempo.

2. Forward pass per inferenza:

- Se il modello non è ancora addestrato, restituisce un tensore di zeri.
- Se il modello è addestrato, utilizza la probabilità stimata della classe 1 (malevolo) per calcolare i logits:

$$\log \left(\frac{p + \epsilon}{1 - p + \epsilon} \right)$$

con p probabilità previste e ϵ una costante per evitare errori numerici.

3. Gestione della memoria persistente:

- La memoria è divisa in tre buffer:
 - **memory_benign**: contiene campioni benigni.
 - **memory_malicious**: contiene campioni malevoli.
 - **core_benign**: una sottosezione della memoria benign che non viene eliminata.
- La memoria viene aggiornata in modo dinamico per evitare la perdita di informazioni storiche e per mantenere un bilanciamento tra classi.

4. Retraining del modello con memoria storica:

- Dopo ogni aggiornamento, il modello viene riaddestrato utilizzando una combinazione di dati nuovi e dati precedentemente memorizzati.
- Se disponibile un modello precedente, viene effettuato un **ensemble learning** con il nuovo modello tramite la classe `EnsembleRF`, che combina le previsioni di entrambi i classificatori con un peso predefinito.

5. Bilanciamento delle classi:

- I dati vengono bilanciati prima dell’addestramento mediante **undersampling stratificato**.
- Il dataset viene ridotto a un massimo di 10.000 campioni per evitare sovraccarichi computazionali.

2.1.2 XGBoost

XGBoost (Extreme Gradient Boosting) è un algoritmo di **boosting** basato su alberi di decisione, ottimizzato per prestazioni elevate e scalabilità. Si basa sull’idea di addestrare sequenzialmente una serie di alberi, in cui ogni nuovo albero cerca di correggere gli errori commessi dai precedenti. Questo approccio consente di costruire modelli altamente performanti, riducendo sia l’errore di bias che quello di varianza.

L’algoritmo di boosting utilizza una procedura iterativa in cui i modelli successivi apprendono dai residui degli errori dei modelli precedenti. L’aggiornamento del modello è guidato dalla funzione di perdita e dal gradiente della stessa. Formalmente, dato un insieme di dati di addestramento (X, y) , il modello XGBoost costruisce una funzione di predizione $F(x)$ attraverso l’ottimizzazione:

$$F_{t+1}(x) = F_t(x) + \eta h_t(x)$$

dove:

- $F_t(x)$ è la funzione di predizione all’iterazione t ,
- $h_t(x)$ è il nuovo albero costruito per correggere gli errori,
- η è il *learning rate*, che controlla l’aggiornamento del modello.

Per poterlo utilizzare nel nostro progetto, abbiamo proceduto come segue:

1. Definizione del modello XGBoost:

- La classe `XGBIDS` incapsula un'istanza di XGBoost e fornisce metodi per addestrare e valutare il modello.
- La predizione restituisce `logits` invece di probabilità, rendendola compatibile con funzioni di perdita basate su log-loss.

2. Forward pass per inferenza:

- Se il modello non è addestrato, restituisce un tensore di zeri.
- Se il modello è addestrato, trasforma i dati in una struttura `DMatrix` e calcola le predizioni utilizzando il margine di output di XGBoost.

3. Funzione di retraining con nuovi dati:

- Converte i dati di traffico e le etichette in formato numpy.
- Utilizza una `DMatrix` per il training di XGBoost.
- Imposta i parametri del modello:
 - `objective='binary:logistic'` per classificazione binaria.
 - `eval_metric='logloss'` per ottimizzare la funzione di perdita logaritmica.
 - `num_boost_round=100` per iterare 100 volte nel boosting.

4. Pretraining da file CSV:

- Carica i dati da un file CSV e separa le feature dalle etichette.
- Converte le etichette testuali in numeriche (`Benign` → 0, `Malicious` → 1).
- Addestra il modello su 100 round di boosting.

2.1.3 SVM

Le **Support Vector Machines (SVM)** sono una famiglia di algoritmi di apprendimento supervisionato utilizzati principalmente per problemi di classificazione. L'obiettivo di un SVM è trovare un iperpiano ottimale che separi le classi nel modo più efficace possibile, massimizzando il margine tra i punti dati più vicini delle classi opposte, noti come support vectors.

Dato un dataset di addestramento $\{(x_i, y_i)\}_{i=1}^N$, con $x_i \in \mathbb{R}^d$ e $y_i \in \{-1, 1\}$, il problema di ottimizzazione dell'SVM lineare è il seguente:

$$\begin{aligned} & \min_{w,b} \frac{1}{2} \|w\|^2 \\ & \text{soggetto a: } y_i(w \cdot x_i + b) \geq 1, \quad \forall i \end{aligned}$$

dove:

- w è il vettore dei pesi del classificatore,
- b è il bias,
- $\|w\|$ rappresenta la norma euclidea di w , che determina la dimensione del margine.

Il nostro utilizzo di SVM per IDS:

1. Definizione della classe SVMIDS:

- Il modello utilizza la classe `SVC` di `scikit-learn`, con il *kernel RBF* per modellare relazioni non lineari.
- La probabilità di appartenenza alle classi è attivata tramite l'opzione `probability=True`.

2. Forward pass per inferenza:

- Se il modello non è addestrato, restituisce un tensore di zeri.
- Se il modello è addestrato, calcola la probabilità di appartenenza alla classe positiva usando `predict_proba()`.
- Per ottenere logits anziché probabilità, il modello applica la funzione:

$$\log \left(\frac{p + \epsilon}{1 - p + \epsilon} \right)$$

con p probabilità previste e ϵ un valore piccolo per evitare errori numerici.

3. Funzione di retraining con nuovi dati:

- Converte i dati di traffico in array NumPy.
- Allena il modello SVM sui dati forniti tramite il metodo `fit()`.
- Dopo l'allenamento, imposta lo stato del modello come "trained".

4. Pretraining da file CSV:

- Carica i dati da un file CSV e separa le feature dalle etichette.
- Le etichette testuali vengono convertite in numeriche (`Benign` → 0, `Malicious` → 1).
- Il modello viene addestrato con `fit()` e lo stato viene aggiornato a "trained".

2.2 Agenti con Reinforcement Learning

Reinforcement Learning (RL) è un paradigma di **apprendimento automatico** in cui un agente interagisce con un ambiente per massimizzare una **funzione di ricompensa** nel tempo. A differenza dell'apprendimento supervisionato, in RL l'agente non riceve etichette corrette per ogni azione, ma deve apprendere tramite tentativi ed errori.

Un problema di RL è definito dai seguenti elementi:

- **Agente**: il modello che prende decisioni.
- **Ambiente**: il contesto con cui l'agente interagisce.
- **Stato (s)**: la rappresentazione della situazione attuale dell'ambiente.
- **Azione (a)**: la scelta effettuata dall'agente in base allo stato.
- **Ricompensa (r)**: un valore numerico che l'agente riceve dopo aver eseguito un'azione, indicando quanto sia stata positiva o negativa.

- **Policy** (π): la strategia che l'agente usa per scegliere le azioni.
- **Fattore di sconto** (γ): determina l'importanza delle ricompense future rispetto a quelle immediate.

L'agente cerca di massimizzare la somma delle ricompense nel lungo periodo, sviluppando una strategia ottimale attraverso esperienze passate.

2.2.1 DDQN

Le Deep Q-Networks (DQN) sono un metodo di **Reinforcement Learning (RL)** basato sull'apprendimento di una funzione di valore d'azione $Q(s, a)$, che rappresenta il valore atteso dell'esecuzione di un'azione a in uno stato s . Il modello utilizza una rete neurale profonda per approssimare questa funzione.

Tuttavia, le DQN tradizionali tendono a sovrastimare i valori Q , il che può portare a instabilità durante l'addestramento. Per risolvere questo problema, è stato introdotto il Double Deep Q-Network (DDQN), che separa il processo di selezione e valutazione dell'azione successiva.

A differenza delle DQN classiche, che usano la stessa rete sia per selezionare che per valutare le azioni, DDQN utilizza due reti neurali:

- Una rete principale (**online network**) per selezionare l'azione migliore nel prossimo stato.
- Una rete target (**target network**) per valutare il valore Q della migliore azione selezionata.

L'aggiornamento della funzione di valore d'azione segue la formula:

$$Q_\theta(s, a) = r + \gamma Q_{\theta^-}(s', \arg \max_{a'} Q_\theta(s', a'))$$

dove:

- $Q_\theta(s, a)$ è la stima attuale della rete principale.
- $Q_{\theta^-}(s', a')$ è la valutazione della rete target per l'azione selezionata.
- r è la ricompensa immediata.
- γ è il **discount factor**, che bilancia ricompense immediate e future.

L'implementazione utilizza un Double Deep Q-Network (DDQN) per addestrare un agente in un ambiente IDS. Il codice è strutturato nei seguenti componenti principali:

1. Definizione della classe **DDQNAgent**:

- L'agente utilizza due reti neurali: una per il training e una come **rete target**.
- La rete è composta da tre strati Linear con attivazioni ReLU.
- La memoria di replay (**Replay Buffer**) è implementata con una struttura FIFO di massimo 50 milioni di transizioni.

2. Funzione di selezione delle azioni (`act()`):

- Implementa una politica **epsilon-greedy**.
- Con probabilità ϵ , l'agente esplora scegliendo un'azione casuale.
- Altrimenti, seleziona l'azione con il valore Q massimo stimato dalla rete principale.
- Il valore di ϵ decresce esponenzialmente fino a un valore minimo per ridurre l'esplorazione nel tempo.

3. Memorizzazione delle esperienze (`remember()`):

- Ogni transizione $(s, a, r, s', done)$ viene memorizzata nel replay buffer.

4. Addestramento dell'agente (`replay()`):

- Viene campionato un batch casuale di esperienze dal replay buffer.
- Il valore Q viene aggiornato usando la formula di DDQN.
- Il **reward scaling adattivo** normalizza le ricompense in un intervallo $[-5, 5]$.
- Il loss viene calcolato tra il valore Q predetto e il target.
- I pesi della rete principale vengono aggiornati tramite backpropagation.

5. Aggiornamento della rete target (`update_target_network()`):

- La rete target viene aggiornata con i pesi della rete principale ogni certo numero di episodi per stabilizzare l'apprendimento.

2.2.2 DQN

Le Deep Q-Networks (DQN) sono un metodo di **Reinforcement Learning (RL)** che utilizza reti neurali profonde per approssimare la funzione di valore $Q(s, a)$, che rappresenta il valore atteso dell'esecuzione di un'azione a in uno stato s . L'obiettivo è apprendere una strategia ottimale per la selezione delle azioni in un ambiente sequenziale.

L'aggiornamento del valore Q segue la formula:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

dove:

- $Q(s, a)$ è il valore attuale della funzione di valore Q.
- r è la ricompensa ricevuta dopo aver eseguito l'azione a .
- γ è il **discount factor**, che bilancia il valore delle ricompense future.
- $Q(s', a')$ è il valore Q massimo nello stato successivo s' .

Nel nostro progetto è stata implementata così:

1. Definizione della classe `DQNAgent`:

- La classe definisce un agente con una rete neurale che approssima la funzione $Q(s, a)$.

- Sono presenti due reti neurali: una principale per il training e una **rete target** per la stabilizzazione dell'apprendimento.
- La memoria di replay è implementata con una struttura FIFO da 50 milioni di transizioni per migliorare la diversità dei dati di training.

2. Funzione di selezione delle azioni (`act()`):

- Implementa una politica **epsilon-greedy**.
- Con probabilità ϵ , l'agente esplora scegliendo un'azione casuale.
- Altrimenti, seleziona l'azione con il valore Q massimo stimato dalla rete principale.
- Il valore di ϵ decresce esponenzialmente fino a un valore minimo per ridurre l'esplorazione nel tempo.

3. Memorizzazione delle esperienze (`remember()`):

- Ogni transizione $(s, a, r, s', done)$ viene memorizzata nel replay buffer.

4. Varianti della funzione di replay (`replay()`):

- `replayOldMaxReward()`:
 - Utilizza il massimo reward storico per normalizzare le ricompense.
 - Aggiorna il valore Q target in base all'azione migliore trovata dalla rete principale.
- `replayNoScheduler()`:
 - Normalizza le ricompense con un massimo calcolato dinamicamente sugli ultimi 1000 esempi.
 - Aggiorna il valore Q target con la rete target.
- `replaySkewed()`:
 - Utilizza un **learning rate scheduler** per adattare dinamicamente il tasso di apprendimento.
 - Adotta una strategia di normalizzazione adattiva.
- `replay()`:
 - Implementa un meccanismo di Adaptive Reward Scaling basato su una finestra mobile delle ultime 1000 esperienze.
 - Clippa i valori delle ricompense tra $[-5, 5]$ per evitare gradienti estremi.
 - Utilizza la rete target per la selezione dell'azione ottimale nello stato successivo.

5. Aggiornamento della rete target (`update_target_network()`):

- La rete target viene aggiornata con i pesi della rete principale ogni certo numero di episodi per stabilizzare l'apprendimento.

2.2.3 Q-Learning

Q-Learning è un algoritmo di **Reinforcement Learning (RL)** basato su valori, che permette ad un agente di apprendere una strategia ottimale per la scelta delle azioni massimizzando una funzione di ricompensa nel tempo. L'obiettivo è apprendere la funzione di valore d'azione $Q(s, a)$, che rappresenta il valore atteso dell'esecuzione di un'azione a in uno stato s .

L'aggiornamento della funzione Q segue la formula:

$$Q(s, a) = Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

dove:

- $Q(s, a)$ è il valore attuale della funzione Q .
- α è il **learning rate**, che determina il peso dell'aggiornamento.
- r è la ricompensa immediata ricevuta dopo aver eseguito l'azione a .
- γ è il **discount factor**, che bilancia il valore delle ricompense future.
- $\max_{a'} Q(s', a')$ è il valore massimo dell'azione nello stato successivo s' , usato per approssimare il valore atteso delle ricompense future.

L'implementazione di Q-Learning standard utilizza una lookup Q -table per memorizzare e aggiornare i valori associati a ogni coppia stato-azione. Questo approccio è adatto per ambienti con spazi di stato di dimensioni contenute, dove la memorizzazione esplicita della tabella è gestibile.

Per modellare Q-Learning, abbiamo proceduto così:

1. Definizione della classe `QLearningAgent`:

- La classe definisce un agente con una tabella Q inizializzata a zeri per tutte le coppie possibili di stato-azione.
- La selezione delle azioni avviene con una strategia epsilon-greedy per bilanciare esplorazione e sfruttamento.
- Parametri chiave:
 - $\gamma = 0.99$ (fattore di sconto).
 - $\epsilon = 1.0$ con decrescita fino a 0.01 (strategia epsilon-greedy).
 - $\alpha = 0.001$ (learning rate).

2. Selezione delle azioni (`choose_action()`):

- Implementa una politica epsilon-greedy.
- Con probabilità ϵ , l'agente esplora scegliendo un'azione casuale.
- Altrimenti, seleziona l'azione con il valore Q massimo stimato dalla tabella.
- Il valore di ϵ decresce esponenzialmente fino a un valore minimo per ridurre l'esplorazione nel tempo.

3. Aggiornamento della funzione Q (`update()`):

- Recupera il valore corrente di $Q(s, a)$ e del massimo Q nello stato successivo s' .
- Calcola il valore target usando la formula dell'aggiornamento di Q-Learning.
- Aggiorna la tabella Q utilizzando il learning rate α .

4. Normalizzazione adattiva delle ricompense:

- La normalizzazione viene aggiornata dinamicamente in base a una finestra di 1000 esperienze recenti.
- Il valore massimo delle ricompense viene aggiornato con una media esponenziale.
- Il valore della ricompensa è scalato tra $[-5, 5]$ per stabilizzare l'apprendimento.

2.2.4 SARSA

SARSA (State-Action-Reward-State-Action) è un algoritmo di **Reinforcement Learning (RL)** basato su valori, che permette ad un agente di apprendere una strategia ottimale aggiornando la sua funzione di valore d'azione $Q(s, a)$ in base alle transizioni osservate. A differenza di Q-Learning, che utilizza il massimo valore futuro stimato, SARSA apprende seguendo la politica attuale dell'agente, risultando in un comportamento più stabile e meno esplorativo rispetto a Q-Learning. Questa particolare implementazione di SARSA utilizza una rete neurale e quindi prende il nome di Deep SARSA.

L'aggiornamento della funzione Q segue la formula:

$$Q(s, a) = Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$$

dove:

- $Q(s, a)$ è il valore attuale della funzione Q .
- α è il **learning rate**, che determina il peso dell'aggiornamento.
- r è la ricompensa ricevuta dopo aver eseguito l'azione a .
- γ è il **discount factor**, che bilancia il valore delle ricompense future.
- $Q(s', a')$ è il valore della funzione Q nello stato successivo s' , per l'azione successiva a' scelta dalla politica corrente.

Utilizziamo un SARSA basato su reti neurali per addestrare un agente in un ambiente IDS:

1. Definizione della classe SARSAAgent:

- La classe definisce un agente con una rete neurale che approssima la funzione $Q(s, a)$.
- La memoria di replay è implementata con una struttura FIFO da 5 milioni di transizioni per migliorare la diversità dei dati di training.
- Parametri chiave:
 - $\gamma = 0.99$ (fattore di sconto).

- $\epsilon = 1.0$ con decrescita fino a 0.01 (strategia epsilon-greedy).
- $\alpha = 10^{-2}$ (learning rate).

2. Architettura della rete neurale (`build_model()`):

- Tre strati completamente connessi:
 - Input Layer: `Linear(state_size, 64)`
 - Hidden Layer: `ReLU + Linear(64, 64)`
 - Output Layer: `Linear(64, action_size)`, che produce i valori $Q(s, a)$.

3. Memorizzazione delle esperienze (`remember()`):

- Ogni transizione $(s, a, r, s', a', done)$ viene memorizzata nel replay buffer.

4. Selezione delle azioni (`act()`):

- Implementa una politica epsilon-greedy.
- Con probabilità ϵ , l'agente esplora scegliendo un'azione casuale.
- Altrimenti, seleziona l'azione con il valore Q massimo stimato dalla rete neurale.
- Il valore di ϵ decresce esponenzialmente fino a un valore minimo per ridurre l'esplorazione nel tempo.

5. Aggiornamento della funzione Q (`replay()`):

- Estraie un batch casuale dal replay buffer.
- Calcola il target Q utilizzando l'equazione di aggiornamento del SARSA.
- Applica la reward normalization adattiva, scalando la ricompensa tra $[-5, 5]$.
- Ottimizza la rete neurale tramite backpropagation.

6. Normalizzazione adattiva delle ricompense:

- La normalizzazione viene aggiornata dinamicamente in base a una finestra di 1000 esperienze recenti.
- Il valore massimo delle ricompense viene aggiornato con una media esponenziale.

2.3 Motivazione delle nostre scelte

Partiamo con alcune scelte che riguardano l'agente:

- L'utilizzo di un `deque` con un `maxlen` molto elevato (1 milione di esperienze) permette di mantenere un archivio di esperienze molto ampio, riducendo il rischio di overfitting e migliorando la generalizzazione.
- Si inizia con $\epsilon = 1.0$ per favorire una forte esplorazione iniziale, mentre il decadimento controllato (0.995) permette di ridurla gradualmente fino ad un minimo fissato (0.01);

- Il valore scelto per il learning rate (0.001) è comunemente usato in molte applicazioni di rete neurale e rappresenta un buon compromesso tra velocità di apprendimento e stabilità dell'aggiornamento dei pesi;
- L'uso di una rete target è una scelta consolidata nel DDQN e DQN per migliorare la stabilità durante l'apprendimento. Copiando i pesi dalla rete principale alla rete target a intervalli regolari, si evita che il target venga modificato troppo rapidamente, rendendo il training più robusto;
- L'implementazione della policy epsilon-greedy garantisce un buon equilibrio tra esplorazione e sfruttamento.

Abbiamo poi anche l'evoluzione della funzione di replay() implementata. La prima implementazione presentava le seguenti caratteristiche:

- **Normalizzazione del Reward:** Per ogni esperienza, veniva aggiornato un fattore di normalizzazione (`max_reward`) prendendo il massimo tra il valore corrente e il valore assoluto del reward. Questa normalizzazione era calcolata a livello di singolo campione, portando a possibili fluttuazioni e incoerenze.
- **Calcolo del Target:** Il target veniva calcolato utilizzando direttamente il modello principale (`self.model`) per stimare il valore del prossimo stato, rendendo l'aggiornamento meno stabile.

I miglioramenti apportati nella versione successiva includevano:

- **Normalizzazione più Stabile:** Il fattore di normalizzazione viene ora calcolato utilizzando un *sliding window* delle ultime 1000 esperienze. Ciò fornisce una stima più robusta e coerente del valore massimo dei reward.
- **Clipping dei Reward:** I reward normalizzati vengono limitati nell'intervallo $[-5, 5]$, in modo da evitare che valori estremi possano destabilizzare l'apprendimento.
- **Utilizzo della Target Network:** Per il calcolo del target nei passi non terminali, si utilizza la rete target (`self.target_model`) invece della rete principale. Questa pratica è standard nei DQN per migliorare la stabilità durante l'addestramento.

La versione finale:

- **Adaptive Reward Scaling:** Invece di basarsi esclusivamente sul valore massimo dei reward delle ultime 1000 (parametro modificabile) esperienze, viene ora mantenuto un fattore di normalizzazione adattivo (`reward_norm_factor`) calcolato tramite una media esponenziale mobile. Questo approccio consente di:
 - **Adattarsi Gradualmente:** La media esponenziale attenua gli effetti di picchi o cali improvvisi, garantendo una normalizzazione più fluida.
 - **Migliorare la Robustezza:** La normalizzazione non risente eccessivamente di valori anomali, rendendo il processo di apprendimento più stabile.

Consideriamo ora la funzione `step()`, nella sua prima versione:

- **Elaborazione con il GNN:** Le feature dei nodi vengono accumulate e, attraverso la funzione `get_edge_index()` che utilizza un concetto fully-connected basato su somiglianza per la creazione degli edge. La somiglianza si basa su protocolli, comportamento, volume, prossimità temporale e similarità nel flow.
- **Calcolo del Reward:** Qui l'agente scopre il risultato della detection della GNN tramite un valore che va da 0 ad 1 fornito da una funzione sigmoide ed interpretato come probabilità. In base a questo viene assegnato un reward fisso (positivo per traffico malevolo non rilevato, negativo se rilevato; e viceversa per il traffico benigno).

Nella sua ultima versione:

- **Bilanciamento Basato su Sliding Window:** Le reward base vengono modificate moltiplicandole per un imbalance factor che penalizza l'agente se produce traffico poco diversificato (troppo traffico benevolo o troppo traffico malevolo) ed il calcolo dell'*imbalance factor* si basa su una finestra scorrevole che considera gli ultimi 1000 (parametro modificabile) campioni di traffico anziché il totale, fornendo un adattamento più reattivo alle variazioni della distribuzione del traffico.

La funzione `generate_traffic()` atta alla generazione del traffico è diventata sempre più ricca: siamo passati da una versione molto semplice che creava solo traffico benevolo o "incorrecto" ad una funzione in grado di creare traffico riconducibile a specifici attacchi.

La generazione degli edge nel grafo utilizzava all'inizio un approccio Fully Connected ma la complessità computazionale elevata e la possibilità di generare un numero eccessivo di edge introducendo rumore, ci hanno portati a passare a K-NN che è più veloce e parametrizzabile.

Per la retrain della GNN abbiamo scelto di utlizzare `BCEWithLogitsLoss` per diverse ragioni:

- **Stabilità Numerica:** integra la funzione `sigmoid` internamente, riducendo il rischio di errori numerici che possono verificarsi applicando manualmente la sigmoide e successivamente calcolando il loss.
- **Semplificazione del Codice:** Non è necessario gestire separatamente l'attivazione e il clamping dei valori, il che rende il flusso di addestramento più semplice e meno incline a errori.
- **Flessibilità con il Bilanciamento:** Questa funzione permette di impostare il parametro `pos_weight` per gestire facilmente lo squilibrio delle classi, migliorando la capacità del modello di apprendere dai dati sbilanciati.

L'ultima versione della `retrain_balanced()` funziona nel seguente modo:

- I dati vengono combinati e suddivisi in due gruppi (benign e malicious). In seguito, si effettua un **oversampling** o **undersampling** per garantire che entrambe le classi abbiano un numero equilibrato di campioni.
- Dopo il bilanciamento, i dati vengono preprocessati e normalizzati ed il training viene effettuato utilizzando `BCEWithLogitsLoss`.
- L'approccio garantisce che il modello non sia influenzato negativamente da uno squilibrio nelle classi, migliorando così la capacità predittiva soprattutto per la classe meno rappresentata.

2.4 Ambiente

L'ambiente è definito come una rete di traffico in cui un Intrusion Detection System (IDS) basato su una Graph Neural Network (GNN) deve identificare attività malevole. L'ambiente simula flussi di rete ed è formalmente modellato come un processo decisionale di Markov (MDP):

- **Stati (S):** ogni stato rappresenta un insieme di feature di traffico di rete, descritto da un vettore numerico di dimensione fissa $s \in \mathbb{R}^d$.
- **Azioni (A):** il modello può generare traffico benigno ($a = 0$) o malevolo ($a = 1$).
- **Transizioni (T):** La dinamica dell'ambiente è determinata dalla generazione di nuovi flussi di traffico e dalla loro classificazione da parte dell'IDS.
- **Ricompense (R):** il modello di reward è progettato per bilanciare l'identificazione corretta degli attacchi e la riduzione dei falsi positivi.

Il traffico viene generato sulla base dell'azione selezionata dall'agente. Ogni istanza di traffico è descritta da un vettore di caratteristiche contenente informazioni come:

- **Durata del flusso:** tempo totale del flusso di rete.
- **Numero di pacchetti:** totale dei pacchetti inviati e ricevuti.
- **Dimensione media dei pacchetti:** dimensione media dei pacchetti nel flusso.
- **Protocolli:** TCP, UDP, ICMP e altri protocolli.
- **Flag dei pacchetti:** contatori dei flag SYN, ACK, FIN.
- **Tasso di pacchetti al secondo:** misura dell'attività della connessione.

Se il traffico è malevolo, vengono introdotte anomalie nelle feature per simulare attacchi come flooding, brute-force, SQL injection, ARP spoofing, ecc.

L'ambiente è modellato come un grafo dinamico in cui i nodi rappresentano flussi di rete e gli archi rappresentano relazioni tra essi. La topologia è costruita utilizzando:

- k-Nearest Neighbors (k-NN): i nodi vengono connessi ai k nodi più simili in termini di feature.
- Soglia di distanza: due nodi sono connessi se la loro distanza è inferiore a una soglia definita.
- Completamento delle connessioni: Se un nodo non ha connessioni, viene forzata una connessione con il nodo più vicino.

L'IDS basato su GNN utilizza questa struttura per apprendere pattern di comportamento anomali.

Il sistema di reward è progettato per incentivare l'agente a generare traffico malevolo che sfugga alla rilevazione e, allo stesso tempo, penalizzarlo se il traffico benigno viene classificato erroneamente. La funzione di reward è data da:

$$R(s, a) = \begin{cases} +25, & \text{se il traffico malevolo non viene rilevato (falsa negatività)} \\ -20, & \text{se il traffico malevolo viene rilevato (vera positività)} \\ +15, & \text{se il traffico benigno viene correttamente identificato (vera negatività)} \\ -10, & \text{se il traffico benigno viene etichettato come malevolo (falsa positività)} \end{cases}$$

Ogni episodio inizia con lo stato iniziale e prosegue con un numero fisso di step, in cui l'agente genera traffico e riceve feedback dal GNN-IDS. L'episodio termina se:

- Viene raggiunto il numero massimo di step.
- La memoria dell'ambiente supera una soglia di campioni (ad esempio, 5000 campioni).

L'addestramento è diviso in due fasi:

1. **Pre-training:** il modello viene inizializzato con dataset statici per apprendere una base di feature di traffico normale e malevolo.
2. **Apprendimento interattivo:** l'agente RL genera nuovi dati e aggiorna il modello IDS in un ciclo continuo, migliorando la capacità del sistema di rilevare minacce reali.

Ogni N episodi, il modello GNN viene riaddestrato con un dataset aggiornato utilizzando un mix di undersampling e oversampling per bilanciare le classi.

3 Risultati

3.1 Dataset

I dataset utilizzati sulle varie prove, modificati anche in parte da noi per alcuni casi:

- [IDS 2018 Intrusion CSVs \(CSE-CIC-IDS2018\)](#)
- [CSE-CIC-IDS 2018 sampled by 25% and cleaned from inf and nan data](#)
- [CSE-CIC-IDS2018 \(divisi per tipi di attacco\)](#)

3.2 Metriche

3.2.1 Accuracy (Accuratezza)

L'accuratezza misura la proporzione di istanze correttamente classificate rispetto al totale delle istanze. È una metrica utile nei dataset bilanciati, ma può risultare fuorviante se le classi sono sbilanciate.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Dove:

- TP (True Positives): Veri positivi, campioni correttamente classificati come positivi.
- TN (True Negatives): Veri negativi, campioni correttamente classificati come negativi.
- FP (False Positives): Falsi positivi, campioni negativi classificati erroneamente come positivi.
- FN (False Negatives): Falsi negativi, campioni positivi classificati erroneamente come negativi.

3.2.2 Matrice di Confusione

La *matrice di confusione* è uno strumento fondamentale per valutare le prestazioni di un classificatore. Essa rappresenta il confronto tra le etichette vere e le previsioni del modello, organizzando i risultati in una tabella 2×2 (per problemi di classificazione binaria) o più grande (per classificazioni multi-classe).

	Predetto Positivo	Predetto Negativo
Vero Positivo (TP)	TP	FN (Falso Negativo)
Vero Negativo (TN)	FP (Falso Positivo)	TN

Tabella 1: Esempio di matrice di confusione per una classificazione binaria.

3.2.3 Balanced Accuracy

La Balanced Accuracy è una variante dell'accuracy, più adatta per dataset sbilanciati, in quanto calcola la media dell'accuracy per ogni classe.

$$\text{Balanced Accuracy} = \frac{1}{2} \left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP} \right) \quad (2)$$

Questa metrica assicura che entrambe le classi contribuiscano equamente al punteggio finale.

3.2.4 Precision (Precisione)

La precisione misura la qualità delle predizioni positive del modello, ovvero quanti dei campioni classificati come positivi sono effettivamente positivi.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3)$$

Un'alta precisione significa che il modello produce pochi falsi positivi. Tuttavia, non tiene conto dei falsi negativi.

3.2.5 Recall (Sensibilità o True Positive Rate)

Il recall indica la capacità del modello di identificare correttamente tutte le istanze positive.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (4)$$

Un alto recall significa che il modello cattura la maggior parte delle istanze positive, ma può generare molti falsi positivi.

3.2.6 F1-score

L'F1-score è la media armonica tra precision e recall e rappresenta un buon equilibrio tra le due metriche.

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (5)$$

L'F1-score è particolarmente utile quando si ha uno squilibrio tra le classi, poiché combina sia la precisione che il recall in un unico valore.

3.2.7 Matthews Correlation Coefficient (MCC)

L'MCC è una metrica che tiene conto di tutti gli elementi della matrice di confusione e offre un valore bilanciato anche in presenza di classi squilibrate.

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (6)$$

L'MCC varia tra -1 (predizioni completamente errate) e 1 (predizioni perfette), con 0 che indica una classificazione casuale.

3.2.8 ROC e AUC (Receiver Operating Characteristic e Area Under the Curve)

La curva ROC mostra la relazione tra il tasso di veri positivi (True Positive Rate, TPR) e il tasso di falsi positivi (False Positive Rate, FPR) per diverse soglie di classificazione.

$$TPR = \frac{TP}{TP + FN} \quad (7)$$

$$FPR = \frac{FP}{FP + TN} \quad (8)$$

L'area sotto la curva ROC (AUC-ROC) è una misura della capacità del modello di distinguere tra le classi:

$$AUC = \int_0^1 TPR(FPR) d(FPR) \quad (9)$$

Un modello perfetto ha un'area AUC pari a 1, mentre un modello che predice casualmente ha un AUC pari a 0.5.

3.2.9 Chiave di lettura grafo agente

Per il grafo che riguarda le reward dell'agente, si tenga conto che le reward sono normalizzate con un fattore di normalizzazione che è adattivo. Per questo, l'elemento chiave per valutare le performance dell'agente è la curva gialla nello stesso grafo che rappresenta il rapporto (per episodio) del numero di reward positive fratto il numero di reward totali.

3.3 IDS basati su Graph Neural Network

3.3.1 GAT binario

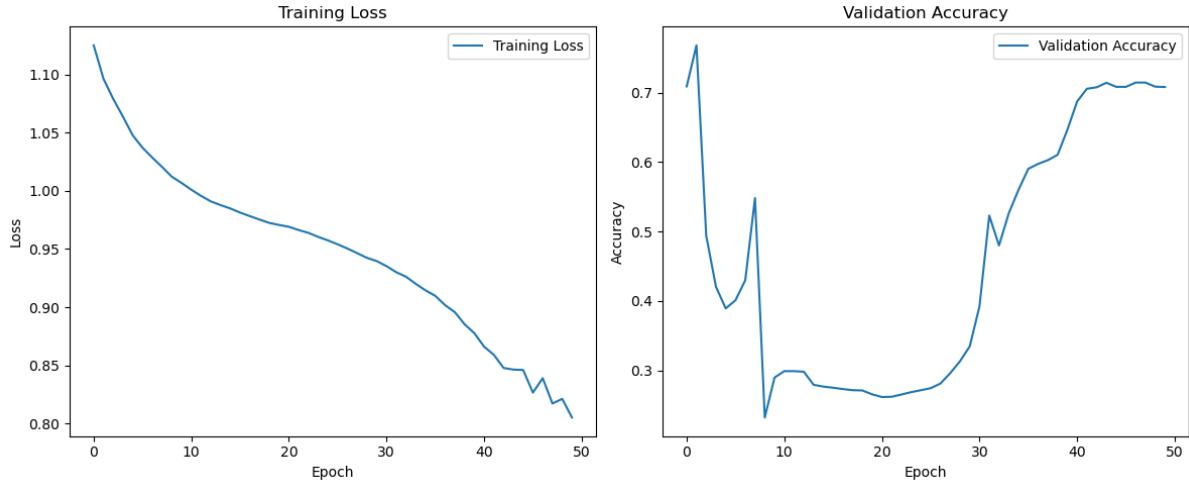


Figura 1: Andamento della perdita di addestramento (Training Loss) e dell'accuratezza di validazione (Validation Accuracy) di un Graph Attention Network (GAT) durante 50 epoche di training. La perdita di addestramento mostra una decrescita graduale, segnalando l'ottimizzazione del modello. Tuttavia, l'accuratezza di validazione presenta una forte instabilità iniziale e una brusca caduta attorno all'epoca 15, seguita da una lenta ripresa, suggerendo possibili problemi di overfitting o difficoltà nell'apprendimento della struttura del grafo

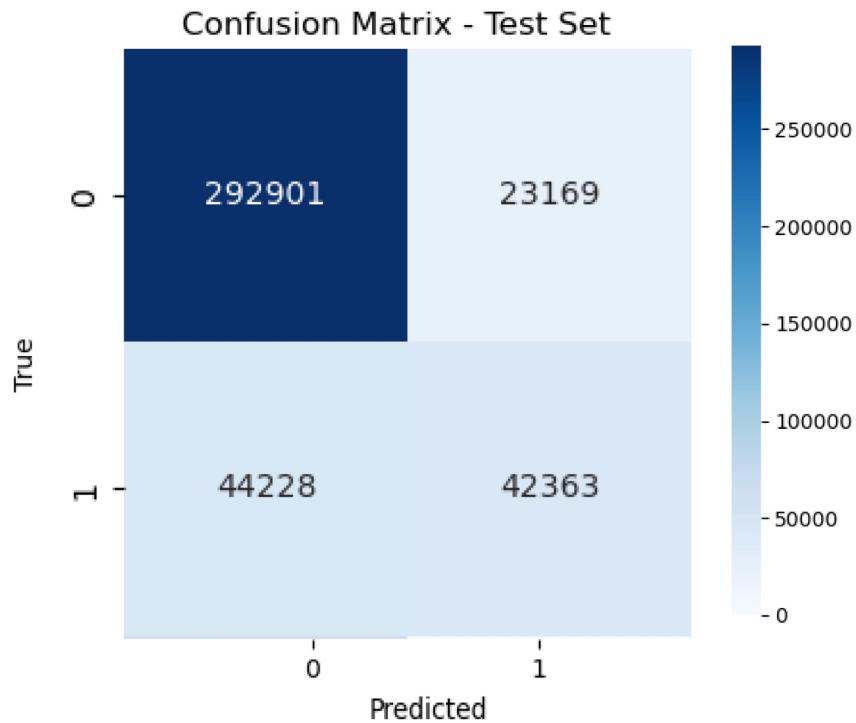


Figura 2: Matrice di confusione del Graph Attention Network (GAT) sul set di test.

Il modello mostra un numero significativo di falsi negativi, suggerendo che potrebbe avere difficoltà nel rilevare correttamente la classe positiva. Questo potrebbe indicare la necessità di migliorare il bilanciamento dei dati, la funzione di perdita o il tuning dei parametri del modello.

3.3.2 GCN binario

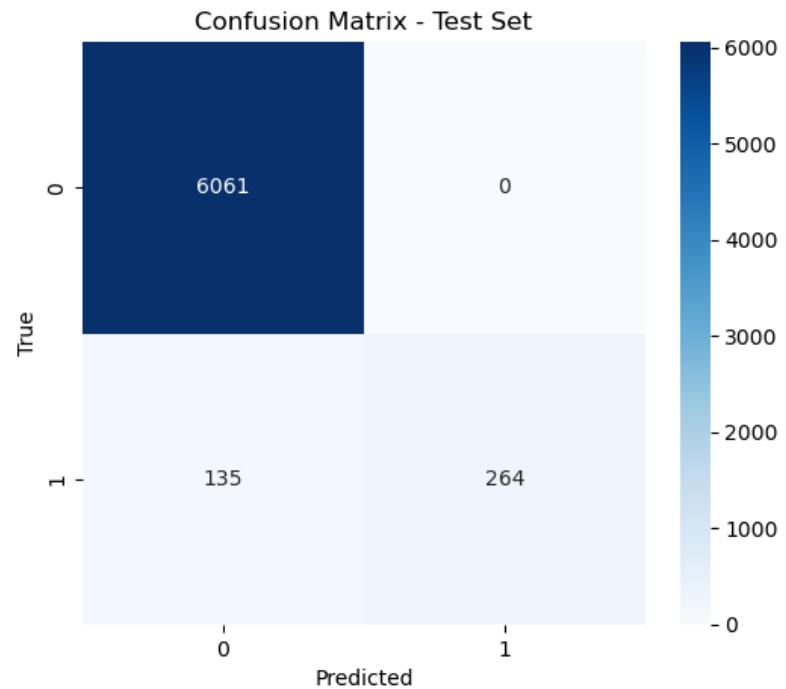


Figura 3: Matrice di confusione dell'IDS basato su GCN, binario.

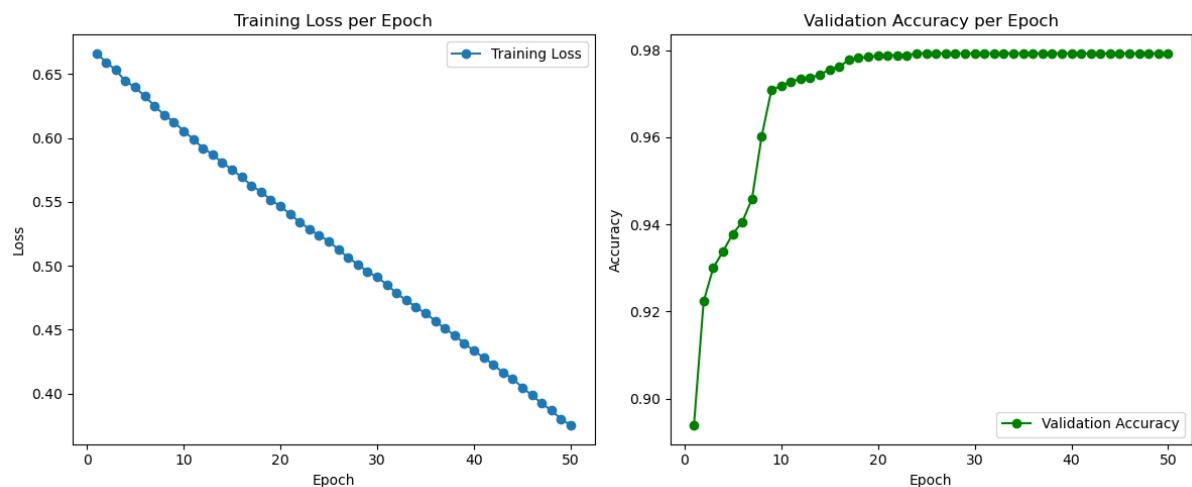


Figura 4: Training Loss e Validation Loss per IDS basato su GCN, binario. Il grafico mostra l'andamento su 50 epoche.

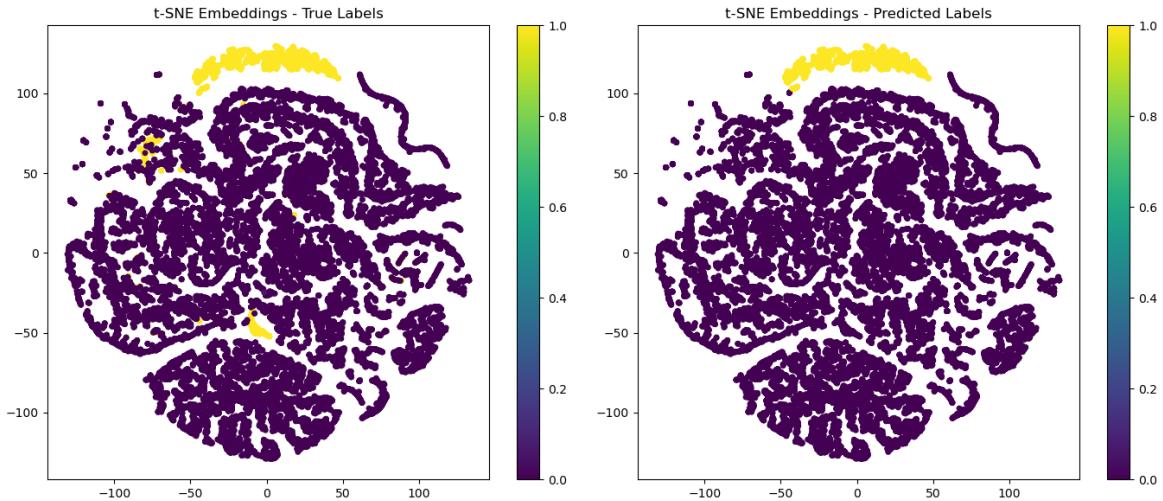


Figura 5: Visualizzazione delle embedding t-SNE dei dati tramite una GCN. A sinistra, i veri label, mentre a destra, le etichette predette dal modello. Il colore indica la classe, con il giallo che rappresenta la classe positiva (malicious) e il viola la classe negativa (benign). La separabilità delle classi mostra la capacità del modello di distinguere tra traffico benigno e malevolo.

3.3.3 GCN multi-classe

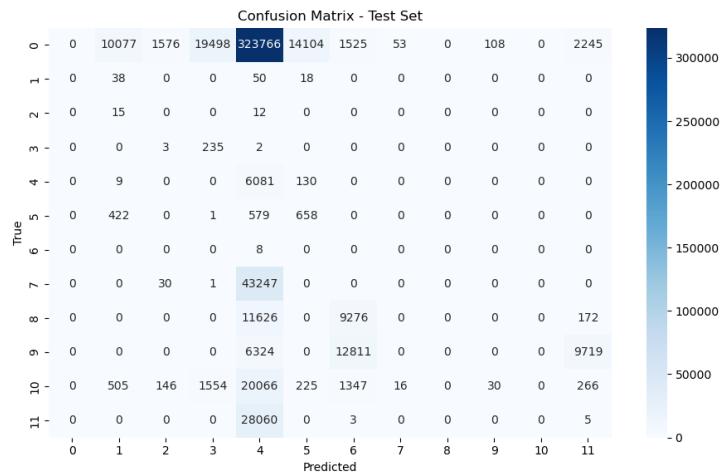


Figura 6: Matrice di confusione del Graph Convolutional Network (GCN) sul set di test per un problema di classificazione multi-classe. Le righe rappresentano le classi reali (True), mentre le colonne indicano le classi predette (Predicted).

L'osservazione dei valori diagonali evidenzia che il modello ha una forte accuratezza per alcune classi (es. classe 4 con 323,766 predizioni corrette), mentre altre classi sono mal classificate o quasi ignorate.

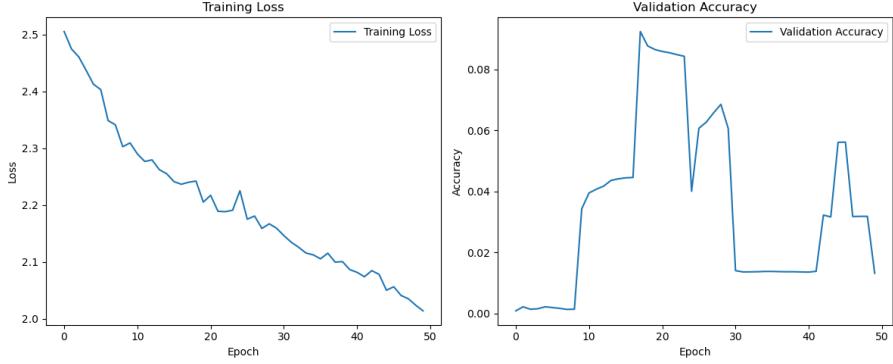


Figura 7: Andamento della perdita di addestramento (Training Loss) e dell’accuratezza di validazione (Validation Accuracy) per un Graph Convolutional Network (GCN) su un problema di classificazione multi-classe.

La perdita di addestramento mostra un miglioramento progressivo, suggerendo che il modello sta convergendo.

L’accuratezza di validazione presenta forti oscillazioni e un valore molto basso (<0.1), indicando che il modello potrebbe avere difficoltà ad apprendere rappresentazioni discriminative per le classi.

3.4 IDS binari con Agente DDQN

3.4.1 SVM

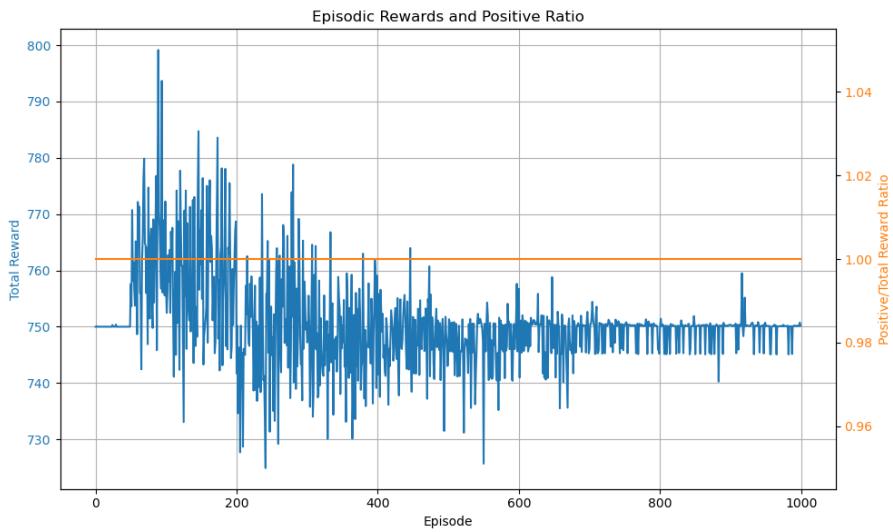


Figura 8: Andamento delle ricompense episodiche e del rapporto tra ricompense positive e totali in un ambiente di Reinforcement Learning. Il totale delle ricompense mostra una fase iniziale di crescita seguita da una stabilizzazione. Il rapporto tra ricompense positive e totali si mantiene intorno a 1, indicando che quasi tutte le ricompense ricevute dall’agente sono positive.

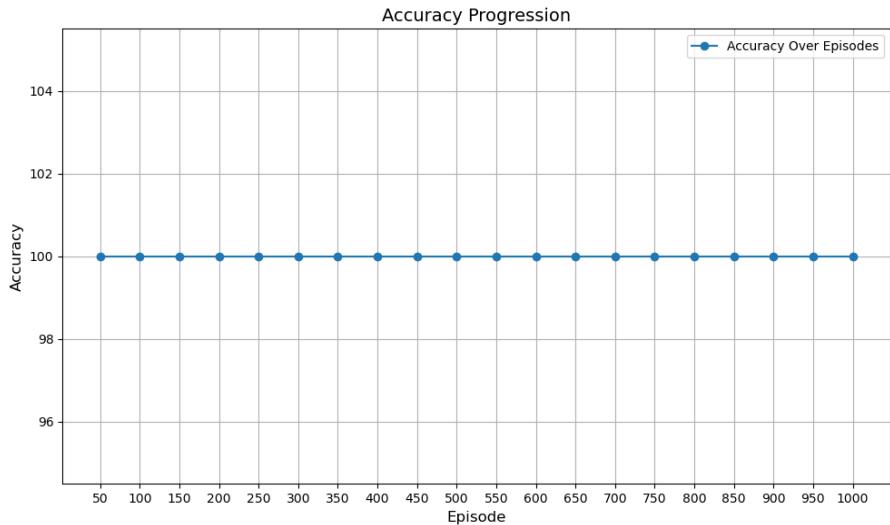


Figura 9: L'accuratezza si mantiene costante al 100%, indicando che il modello classifica perfettamente tutte le istanze. Un'accuratezza così elevata potrebbe derivare da un dataset non bilanciato o da un problema di overfitting.

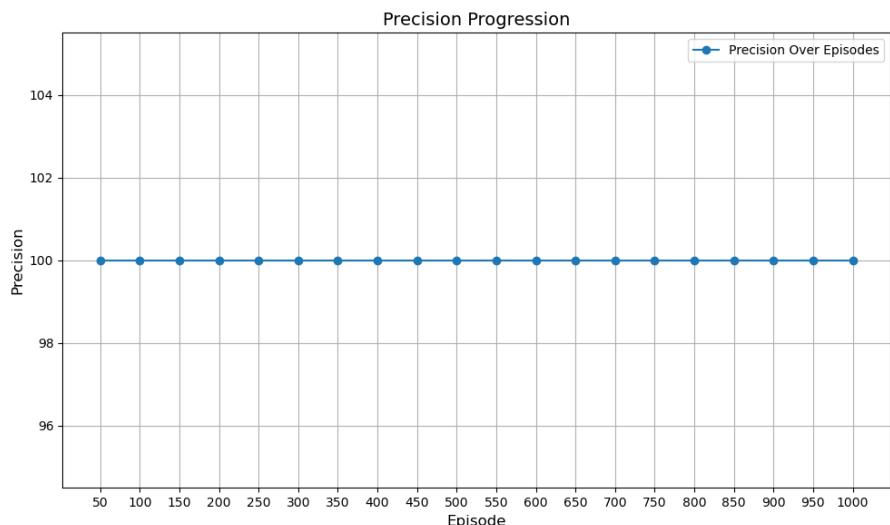


Figura 10: La precisione è costante e pari al 100%, suggerendo che tutte le predizioni positive fatte dal modello sono corrette. Se il dataset non è bilanciato, questo valore potrebbe non essere indicativo di una reale capacità generalizzativa del modello.

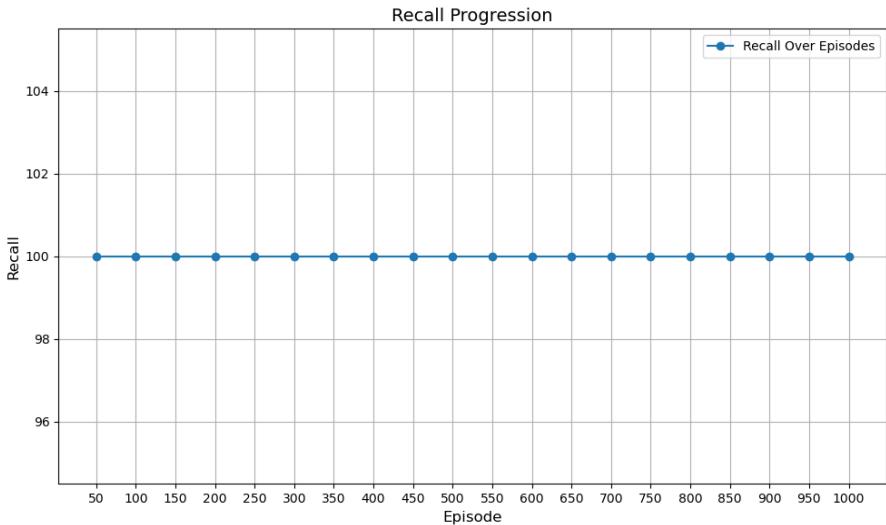


Figura 11: Il recall rimane costante al 100% durante tutte le iterazioni, suggerendo che il modello sta classificando correttamente tutti gli esempi positivi. Questo potrebbe indicare un dataset sbilanciato o un comportamento del modello che favorisce il richiamo a discapito di altre metriche.

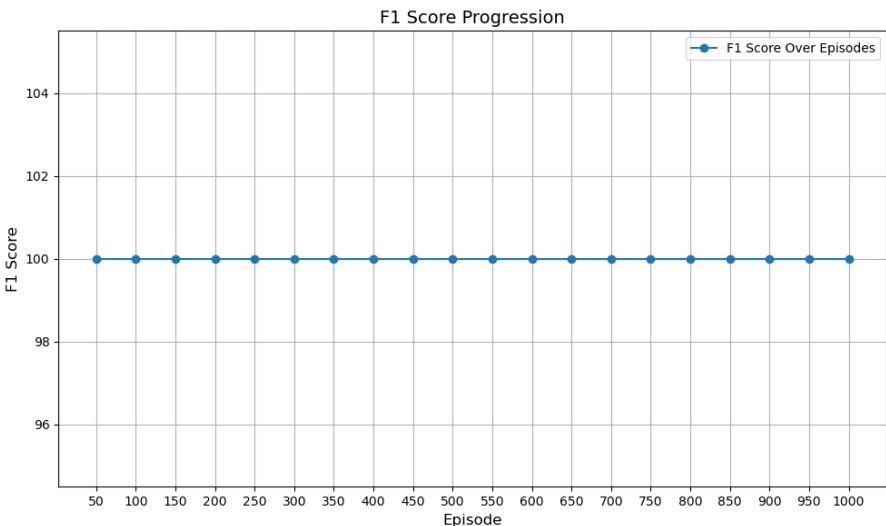


Figura 12: L'F1-score, che rappresenta il bilanciamento tra precisione e recall, è costante e pari a 100%. Questo è coerente con gli altri risultati, ma potrebbe indicare un comportamento artificiale del modello, ad esempio se sta predicendo sempre una sola classe.

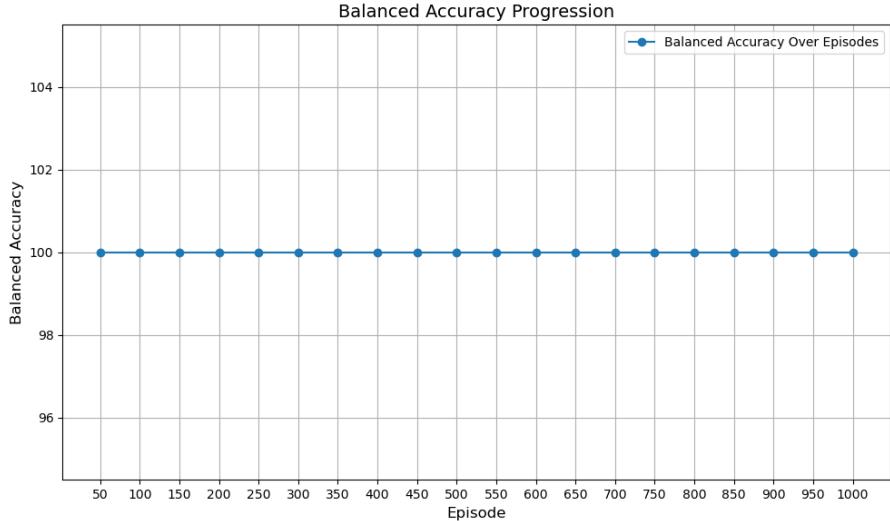


Figura 13: L'accuratezza bilanciata è perfetta (100%), il che potrebbe essere sintomo di un problema nel dataset o di un modello che sta sovra-adattandosi a un insieme di dati non realistico.

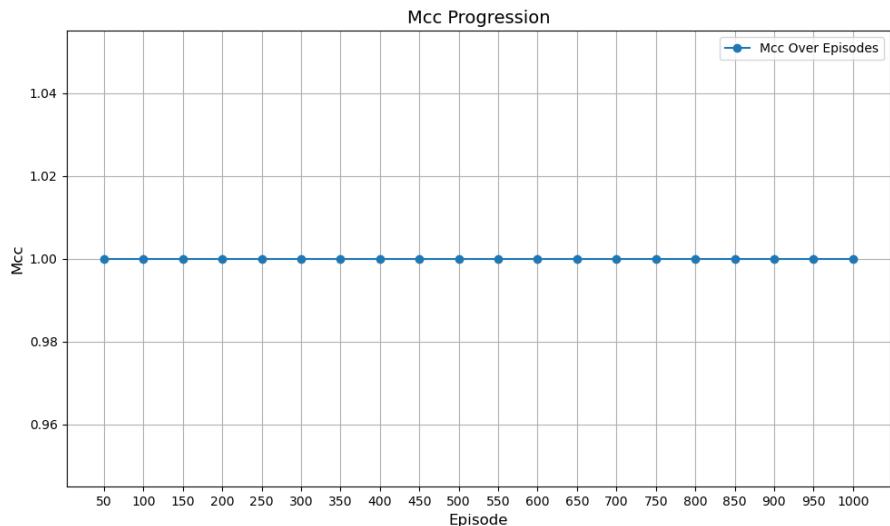


Figura 14: Il coefficiente di correlazione di Matthews (MCC) è pari a 1 per tutte le iterazioni, segnalando una correlazione perfetta tra predizioni e classi reali. Questo è un risultato raro e può essere indicativo di un dataset non rappresentativo o di una configurazione non realistica del modello.

Per quanto possa sembrare irrealistico questo tipo di dato, dopo tante prove siamo convinti alla veridicità. Nel corso delle nostre prove ha prodotto risultati più realistici, anche se poco al di sotto del 100%.

3.4.2 XGBoost

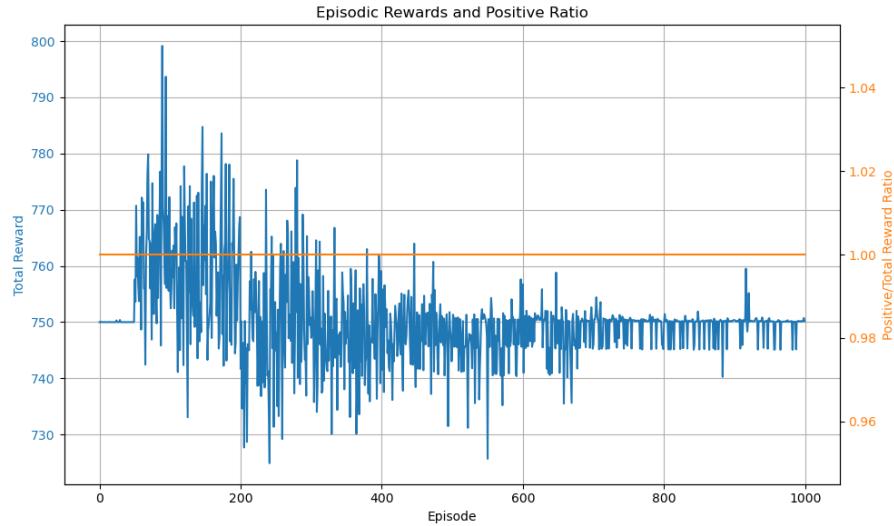


Figura 15: Andamento delle ricompense episodiche e del rapporto tra ricompense positive e totali durante il training di un agente di Reinforcement Learning.

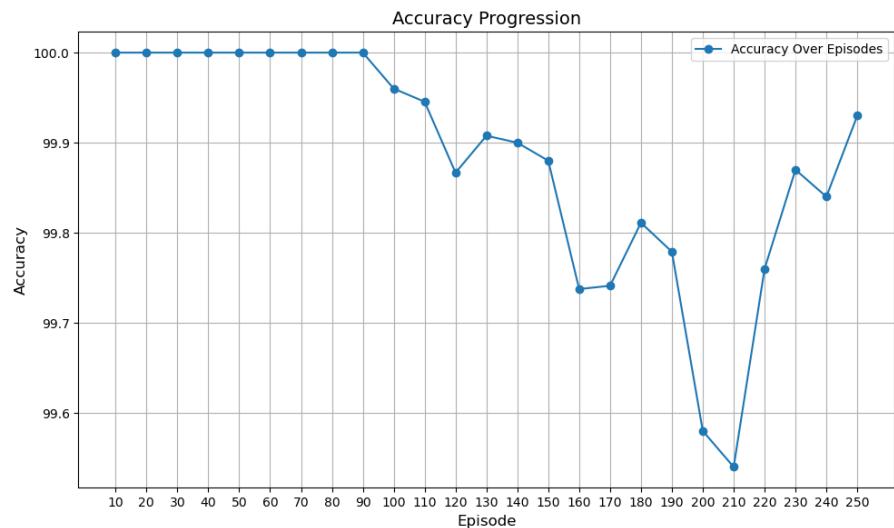


Figura 16: Accuratezza durante il training.

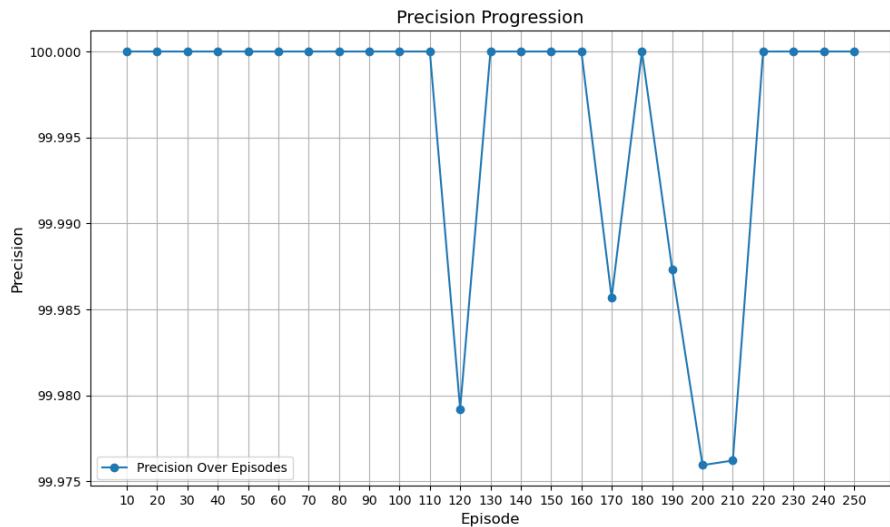


Figura 17: Precisione durante il training.

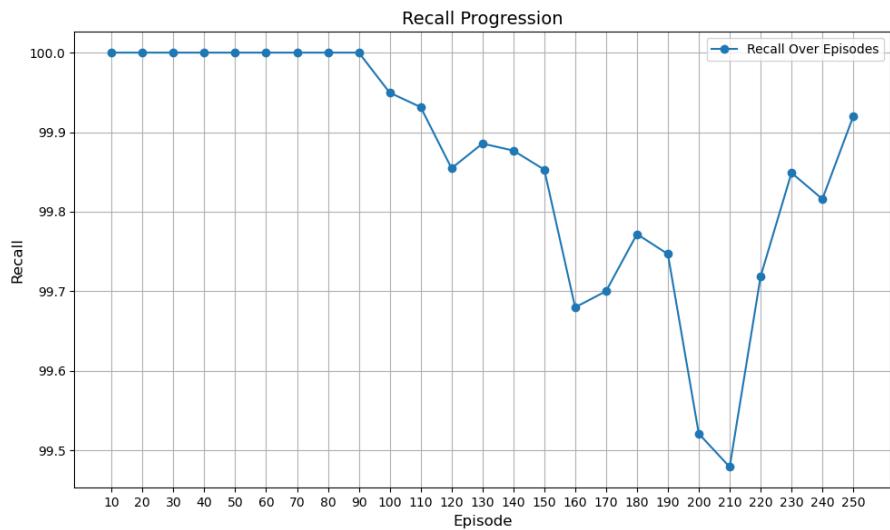


Figura 18: Recall durante il training.

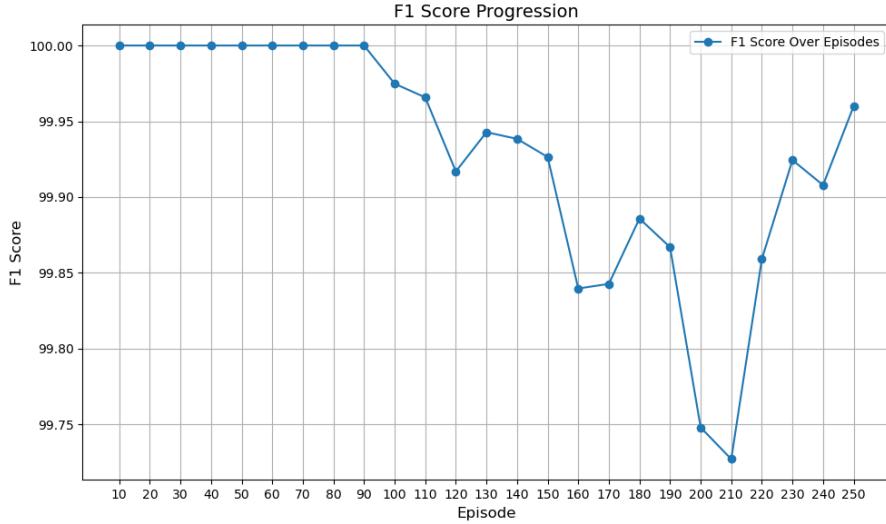


Figura 19: F1-score durante il training.

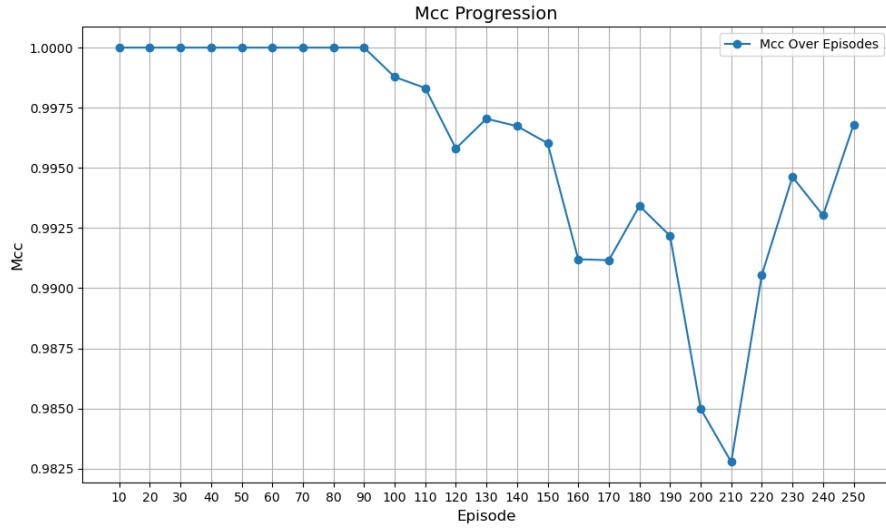


Figura 20: MCC durante il training.

I grafici mostrano un’evoluzione delle metriche di classificazione su più episodi di addestramento. Complessivamente, il modello sembra mantenere prestazioni molto elevate in termini di accuratezza, precisione, recall, F1-score e MCC, con valori che oscillano intorno al 99 – 100%.

Tuttavia, si osservano lievi fluttuazioni a partire dalla metà degli episodi, suggerendo possibili cambiamenti nella distribuzione dei dati o effetti legati alla convergenza del modello.

3.4.3 Random Forest

Per questo terzo classificatore abbiamo inizialmente riscontrato delle performance elevatissime, sintomo di un possibile overfitting che abbiamo provato a risolvere includendo un meccanismo che tiene conto del modello vecchio che è stato trainato e di quello nuovo

appena aggiornato e computa le probabilità di predizione su entrambi i modelli ritornando una media pesata (viene essenzialmente fatto un ensembling dei modelli). Includendo inoltre una memoria di campioni benigni da usare durante ogni retrain in modo da avere sempre un modo per distinguere le due classi anche nel caso di abbondanza di campioni malevoli da parte dell'agente. Questi cambiamenti hanno parzialmente risolto il problema dell'overfitting.

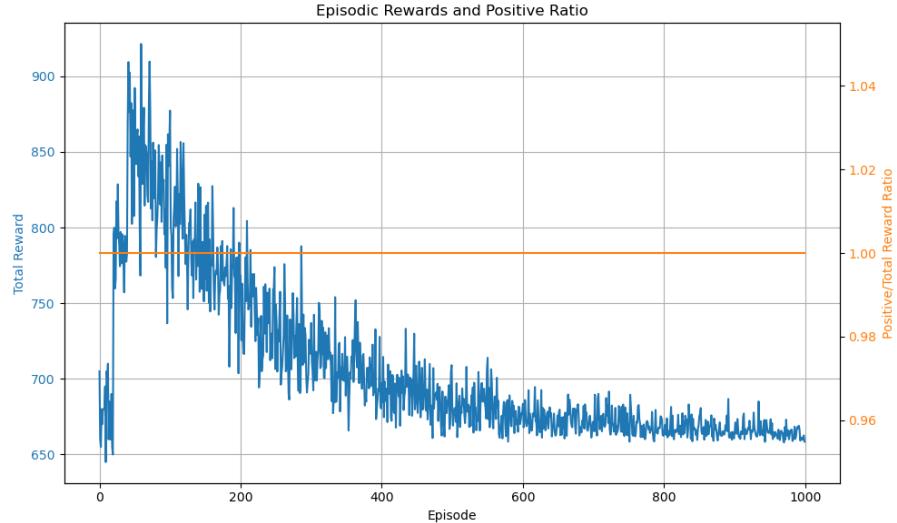


Figura 21: Grafico dell'andamento delle ricompense episodiche e del rapporto tra ricompense positive e totali in un ambiente di Reinforcement Learning.

L'asse sinistro mostra il totale delle ricompense per episodio, che inizialmente cresce, ma poi mostra un calo progressivo e una maggiore stabilità dopo circa 400 episodi.

L'asse destro rappresenta il rapporto tra ricompense positive e ricompense totali, che si mantiene intorno a 1, suggerendo che la maggior parte delle ricompense ottenute dal modello sono positive.

L'andamento decrescente delle ricompense totali potrebbe indicare un cambiamento nella strategia dell'agente, la presenza di exploration-exploitation trade-off, oppure un aggiustamento dei parametri che influisce sull'accumulo delle ricompense nel tempo.

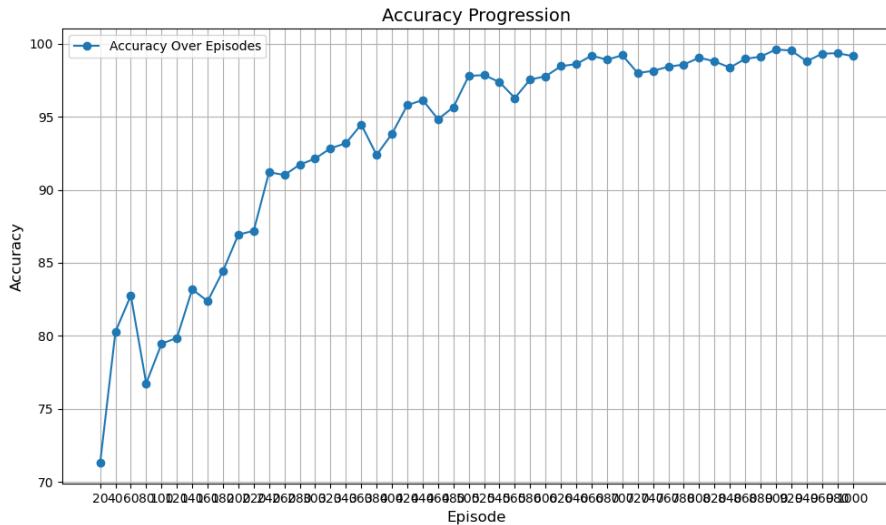


Figura 22: Evoluzione dell'accuratezza durante le iterazioni. Il valore mostra una crescita costante, stabilizzandosi sopra il 95% nelle ultime iterazioni. Questo suggerisce un apprendimento efficace, ma l'accuratezza da sola potrebbe non essere un indicatore affidabile in caso di squilibrio tra classi.

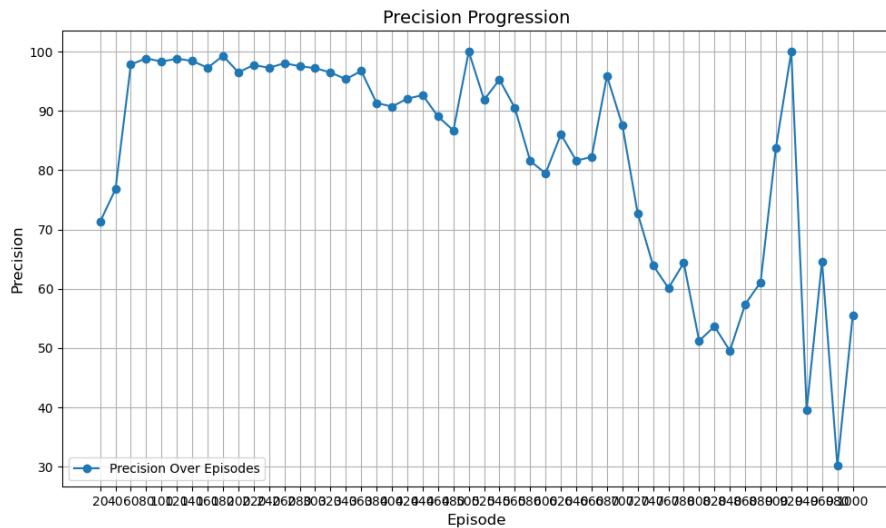


Figura 23: Progresso della precisione nel tempo. Il modello raggiunge rapidamente un'alta precisione ($> 90\%$), ma presenta una forte variabilità nelle fasi avanzate. Questo potrebbe indicare un comportamento poco stabile nella classificazione delle classi positive.

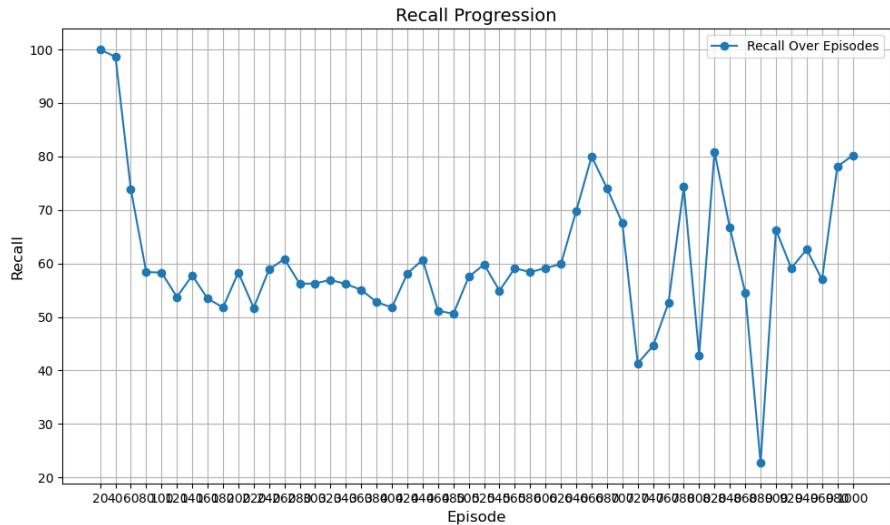


Figura 24: Andamento del recall nel tempo durante le iterazioni dell'algoritmo. Dopo un valore iniziale elevato, il recall cala rapidamente e rimane instabile, con oscillazioni significative nelle fasi successive. Questo potrebbe indicare difficoltà nel catturare esempi positivi o una forte variabilità nei dati.

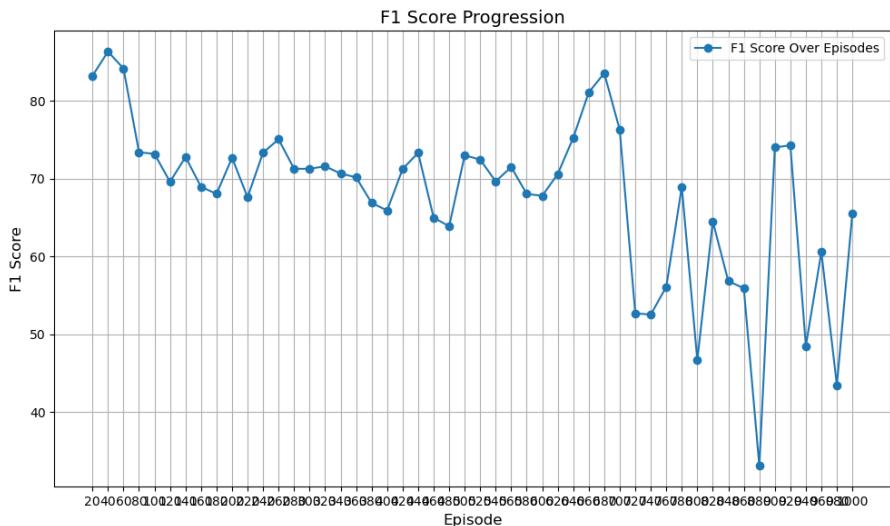


Figura 25: Andamento dell'F1-score, una metrica che bilancia precisione e recall. Dopo una fase iniziale instabile, il valore si mantiene relativamente stabile, ma con fluttuazioni. La diminuzione in alcune fasi suggerisce potenziali problemi di equilibrio tra precisione e recall.

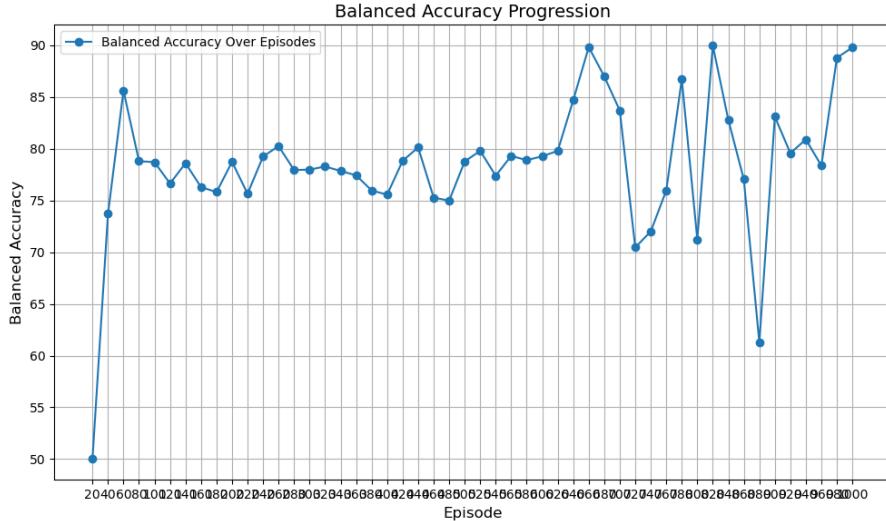


Figura 26: Evoluzione dell'accuratezza bilanciata, utile per gestire dataset con classi sbilanciate. Mostra un miglioramento progressivo, con alcune oscillazioni che potrebbero riflettere la difficoltà nel classificare le classi meno rappresentate.

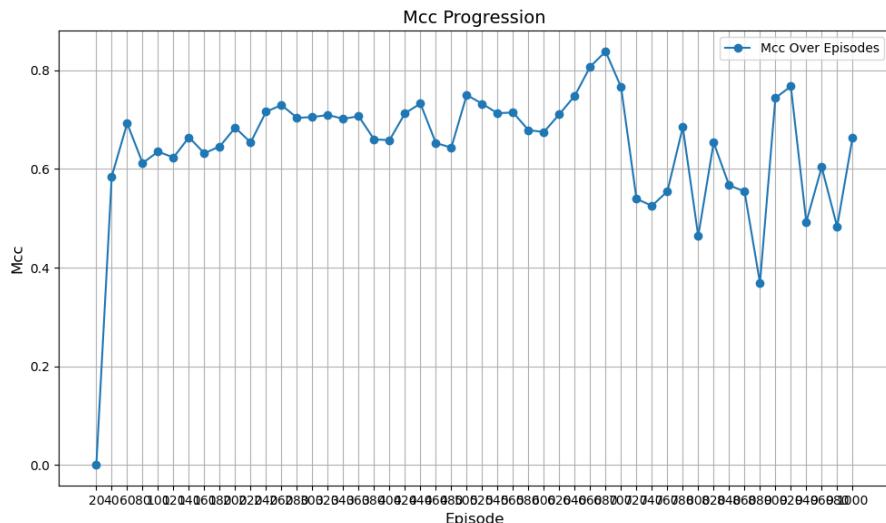


Figura 27: Progresso del MCC, una metrica robusta per la qualità delle classificazioni binarie. Il valore aumenta nelle prime iterazioni, stabilizzandosi sopra 0.6-0.8, indicando una correlazione positiva tra predizioni e valori reali.

L'accuracy e la precisione migliorano significativamente e si stabilizzano, suggerendo che il modello sta imparando in modo efficace.

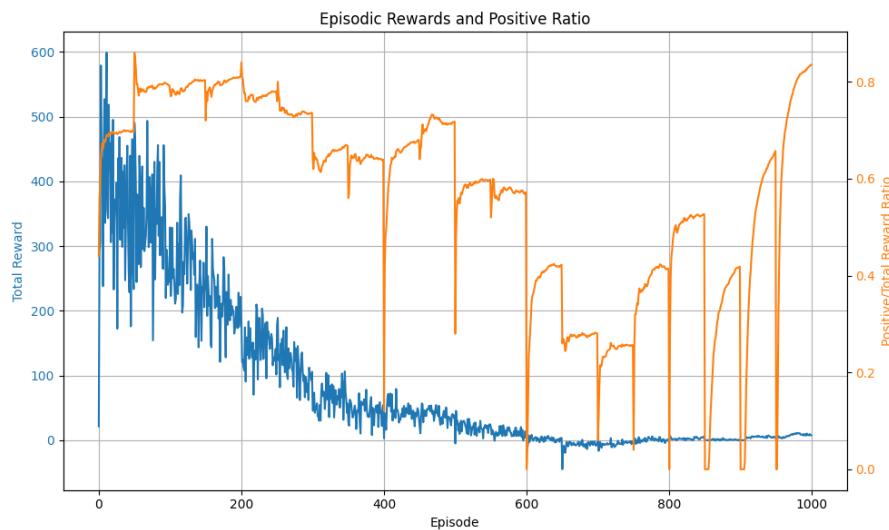
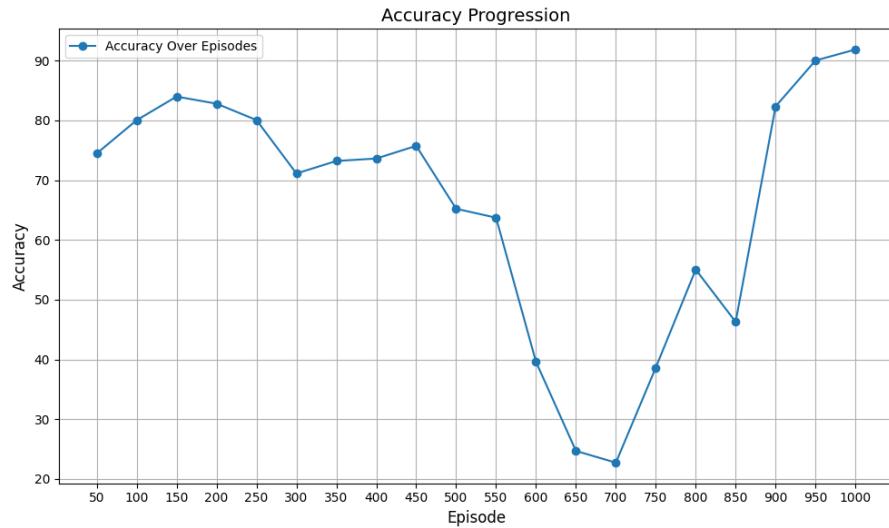
Il recall e l'F1-score presentano maggiore variabilità, indicando che il modello potrebbe avere difficoltà nel rilevare tutte le istanze positive.

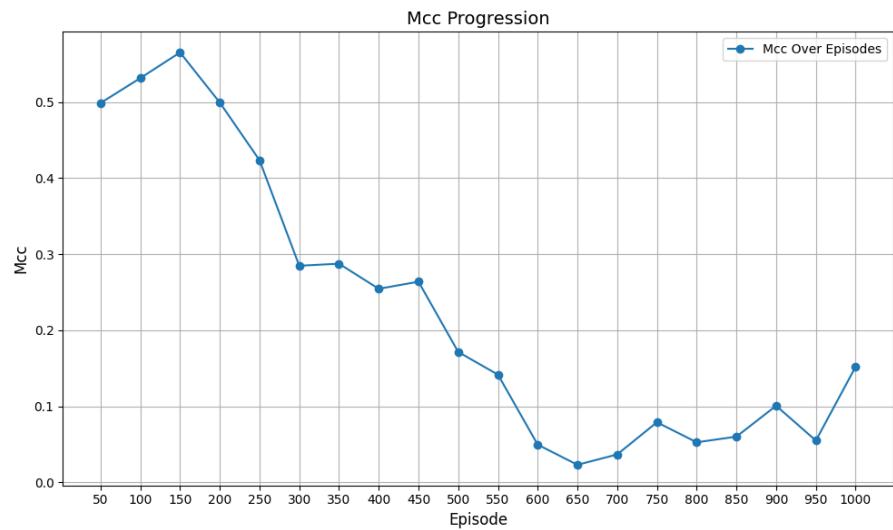
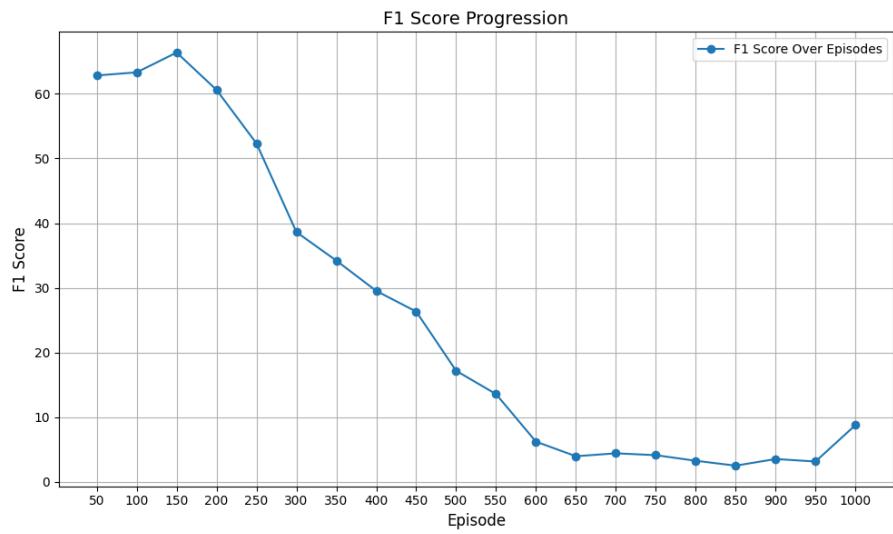
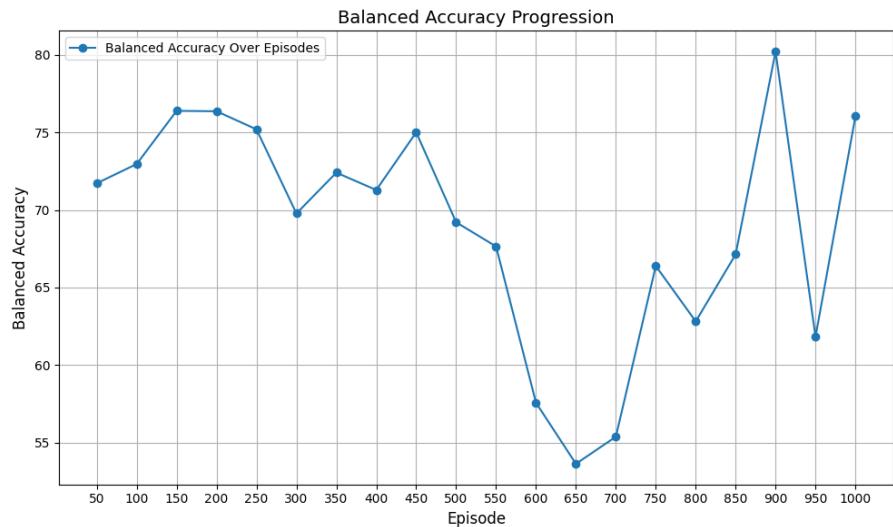
L'MCC e l'accuracy bilanciata suggeriscono che il modello mantiene un buon bilanciamento tra classi, nonostante alcune oscillazioni.

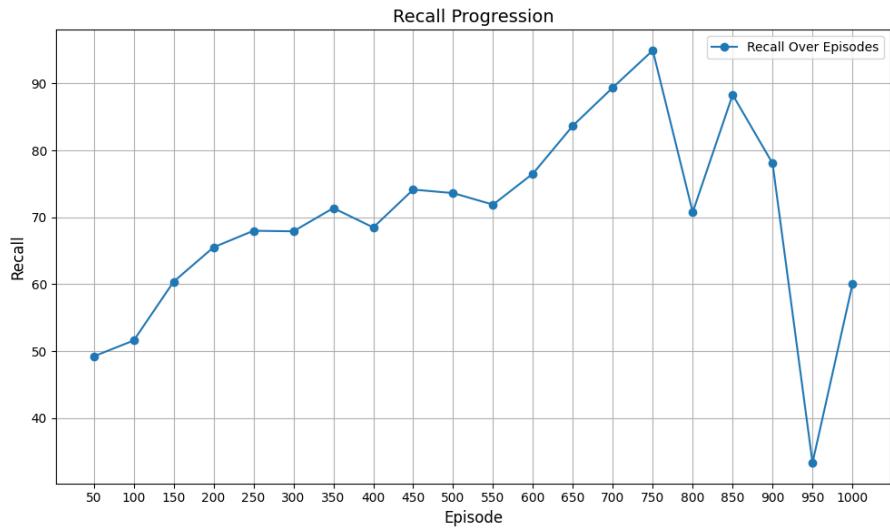
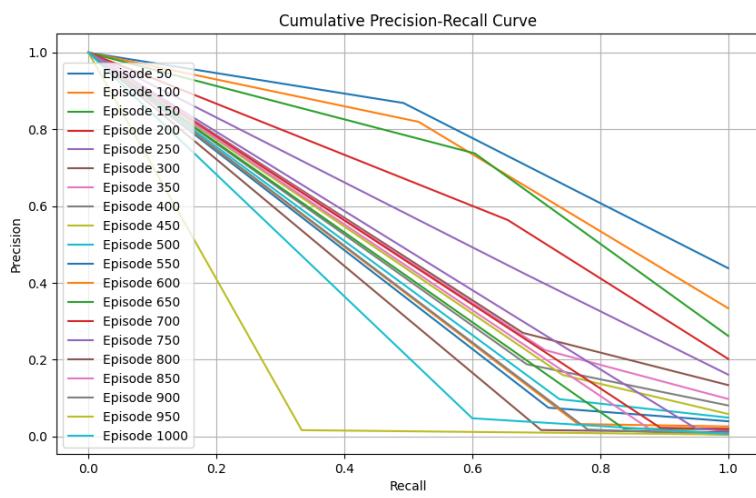
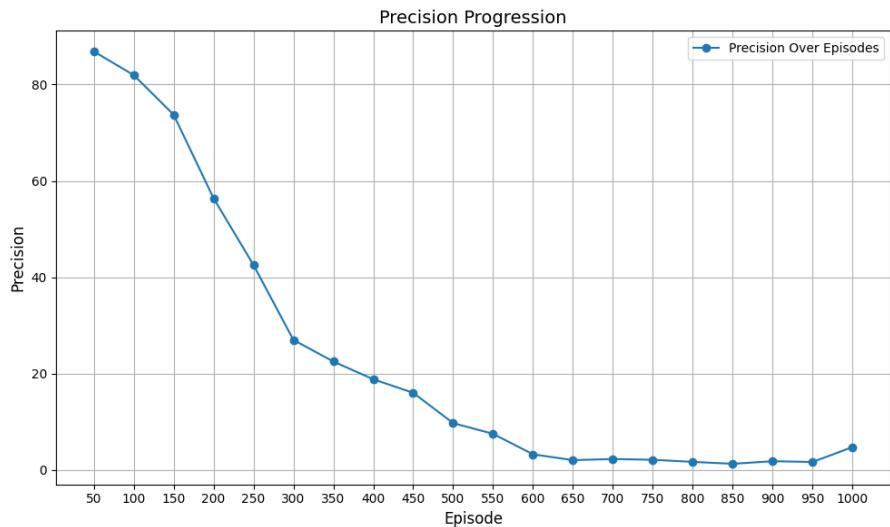
3.5 Agenti

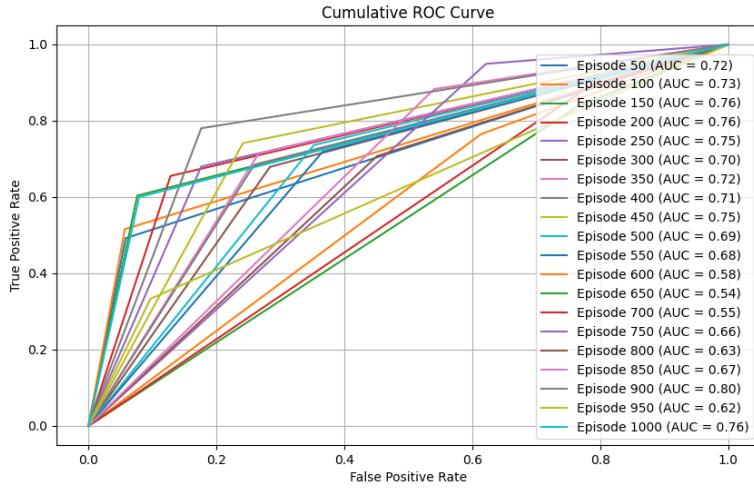
Tutti i test sono stati effettuati attaccando l'IDS implementato con GCN.

3.5.1 DDQN con normalizzazione e Reward Imbalance Factor



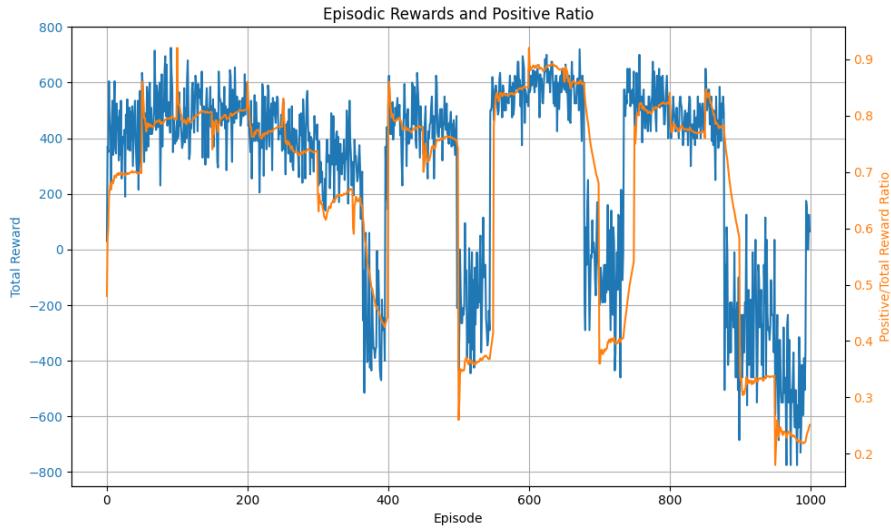


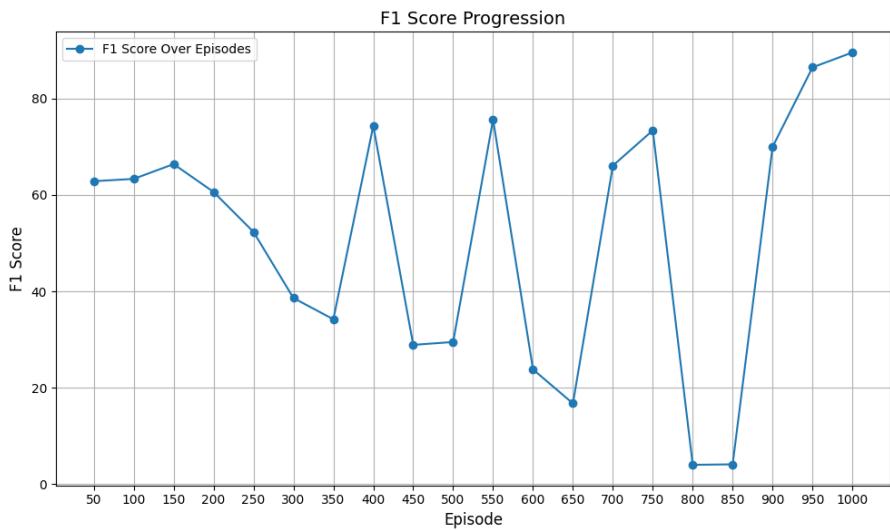
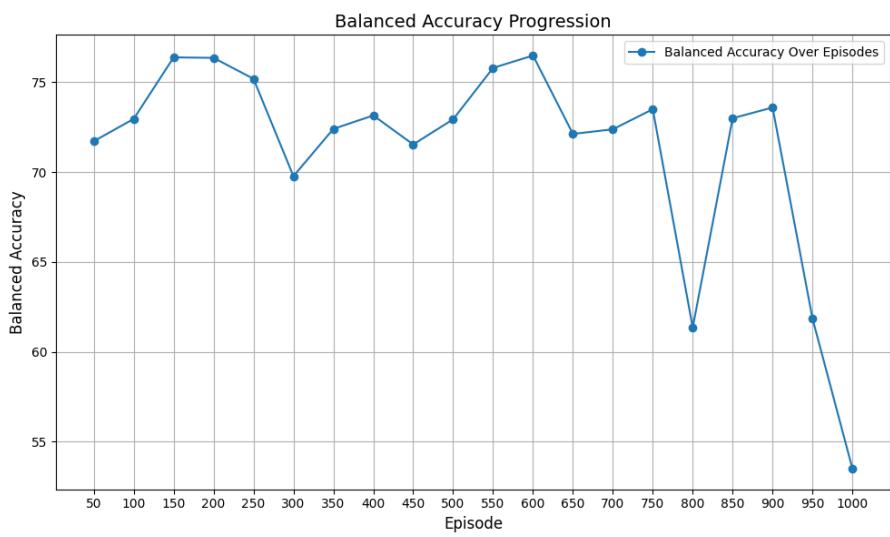
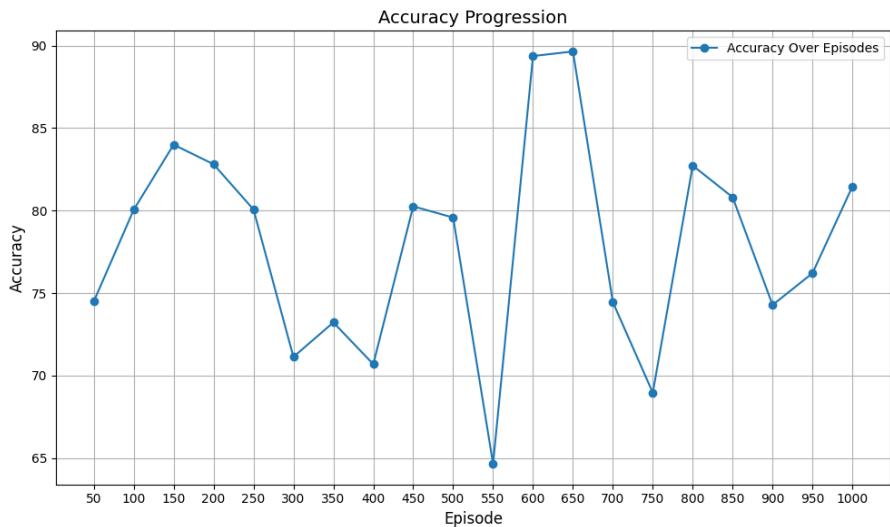


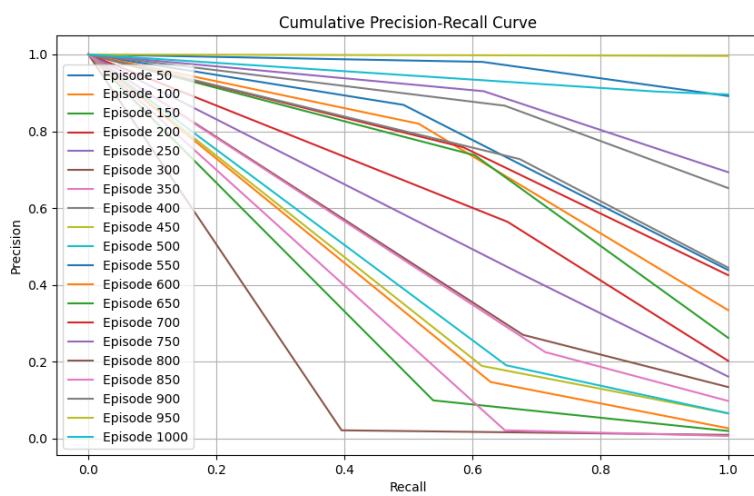
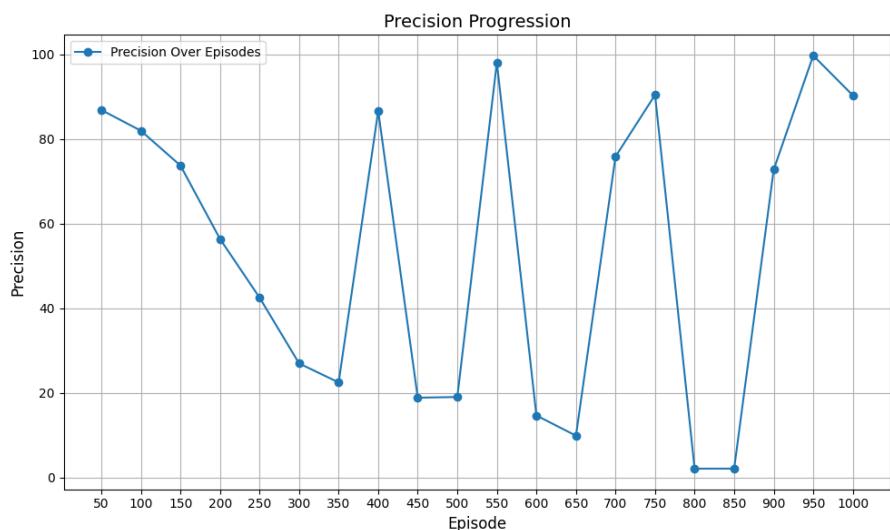
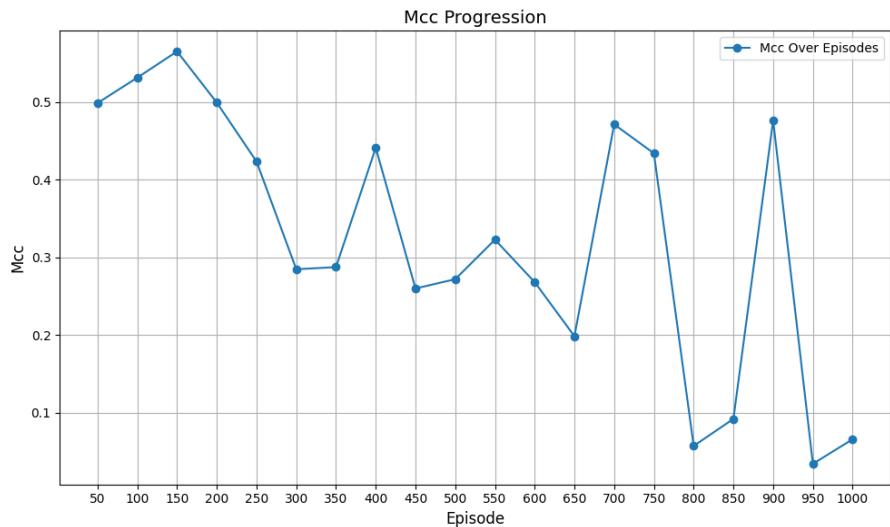


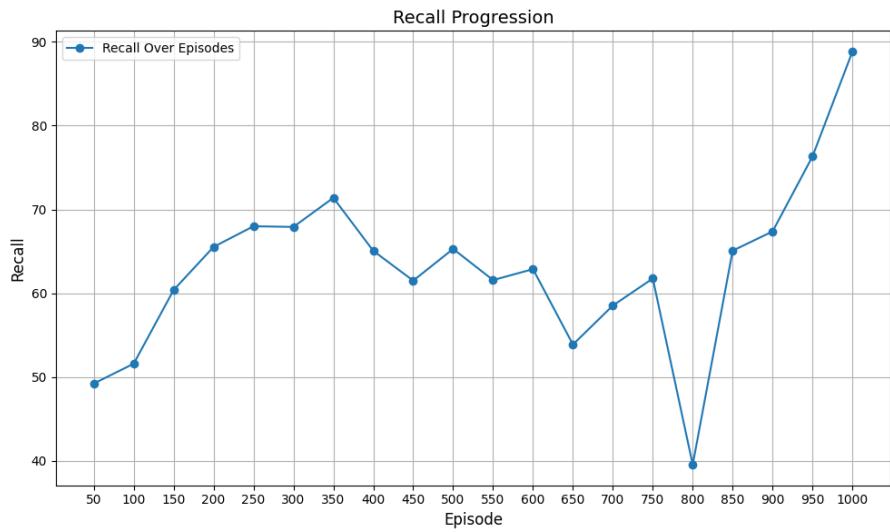
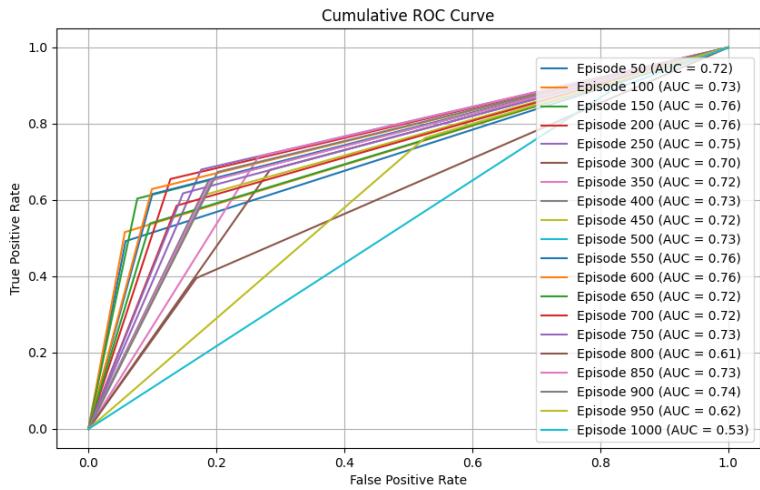
3.5.2 DDQN con normalizzazione delle reward ma senza Reward Imbalance Factor

Questo test è stato fatto per rappresentare l'andamento rimuovendo l'imbalance factor delle reward che serviva per cercare di evitare che l'agente generasse troppo traffico benevolo o troppo traffico malevolo moltiplicando le reward per un imbalance factor calcolato su una sliding window di traffico generato.

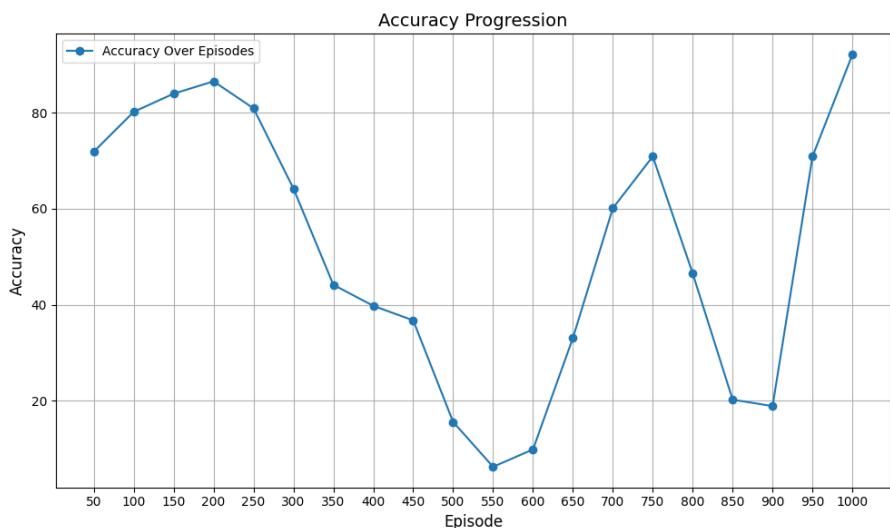


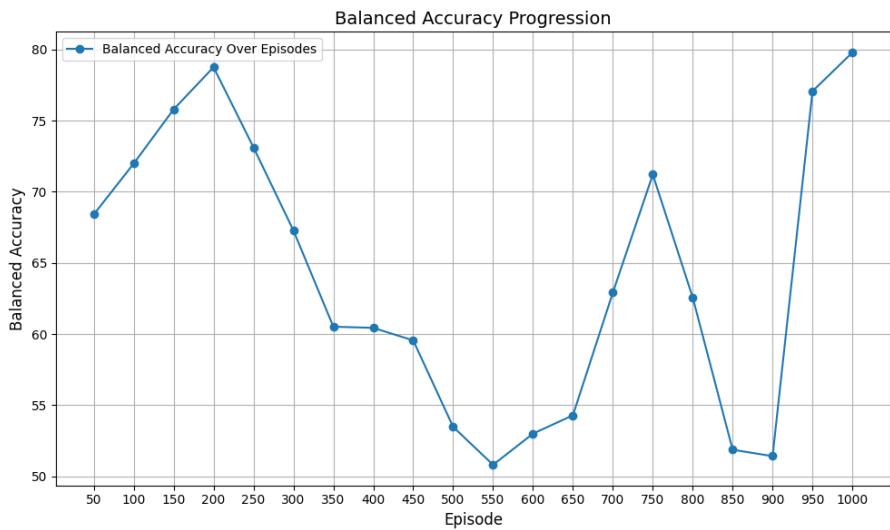
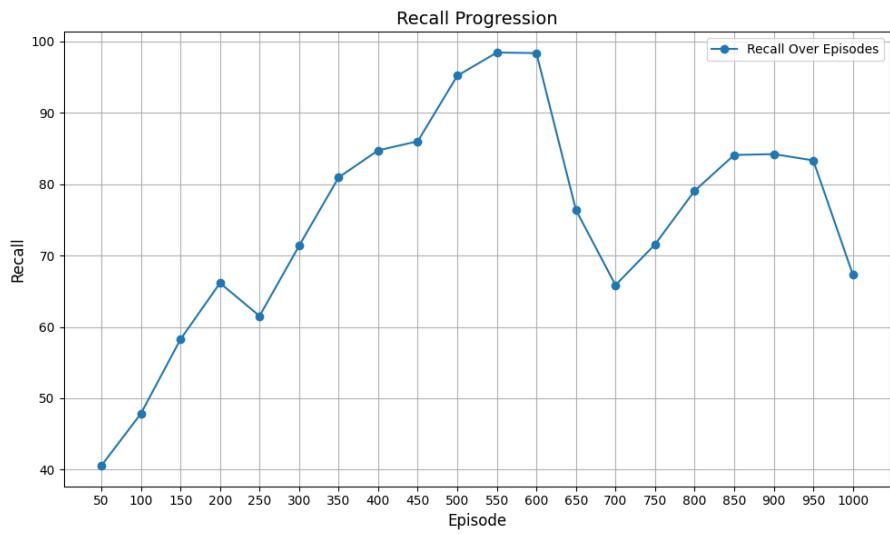
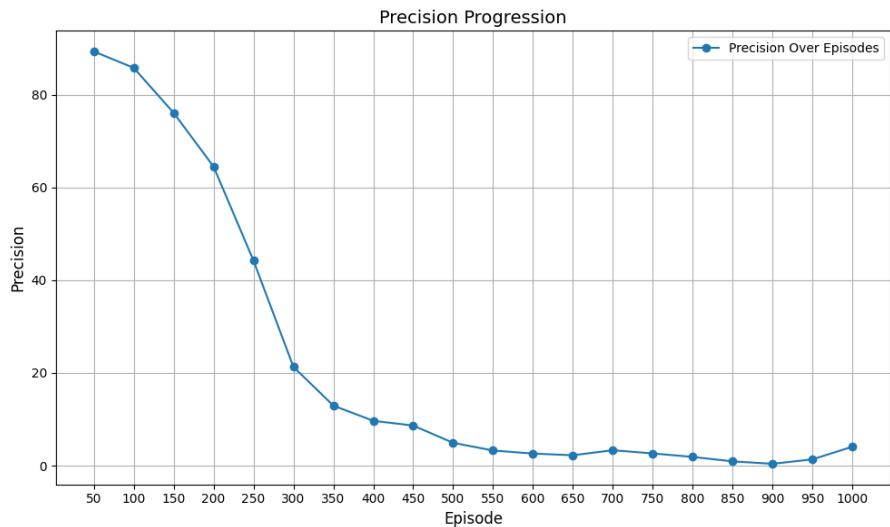


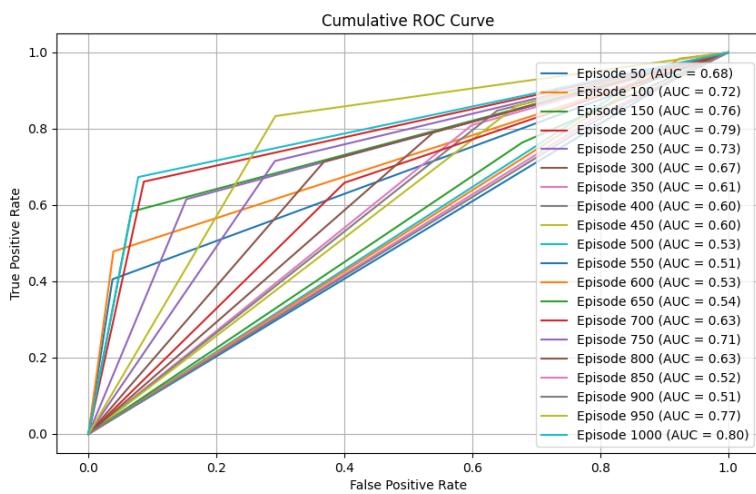
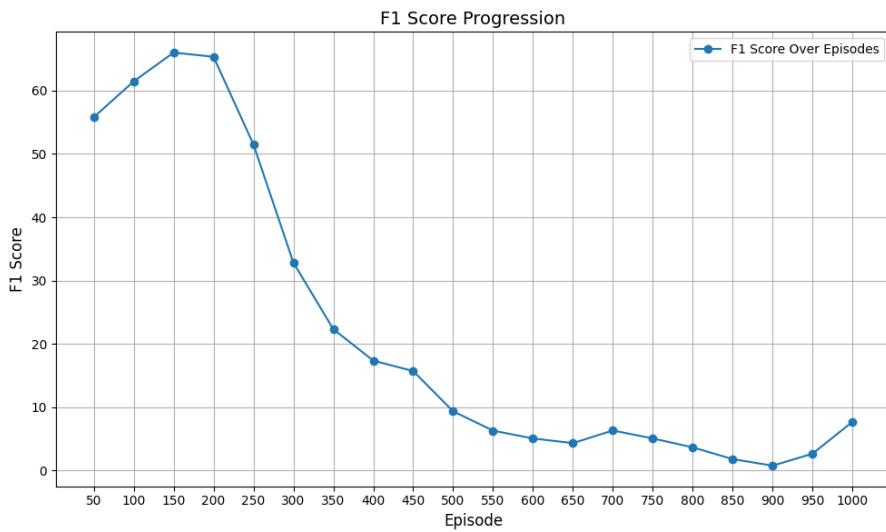
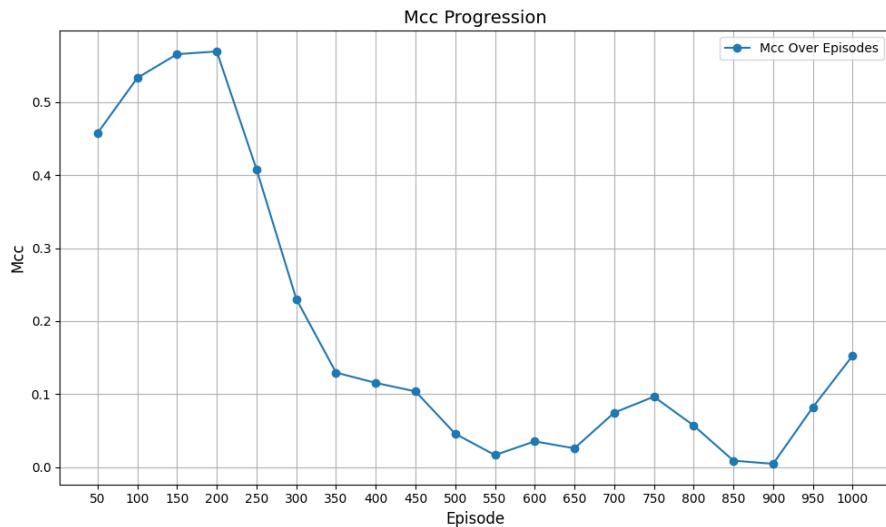


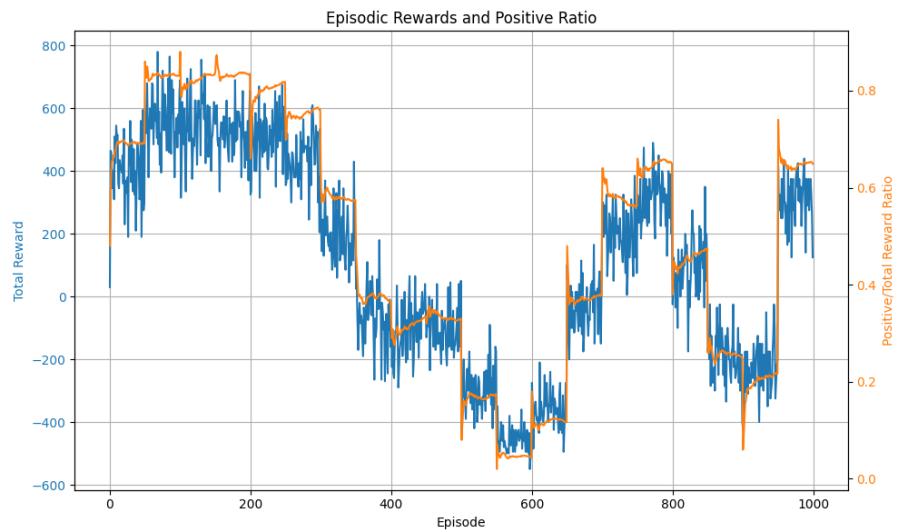
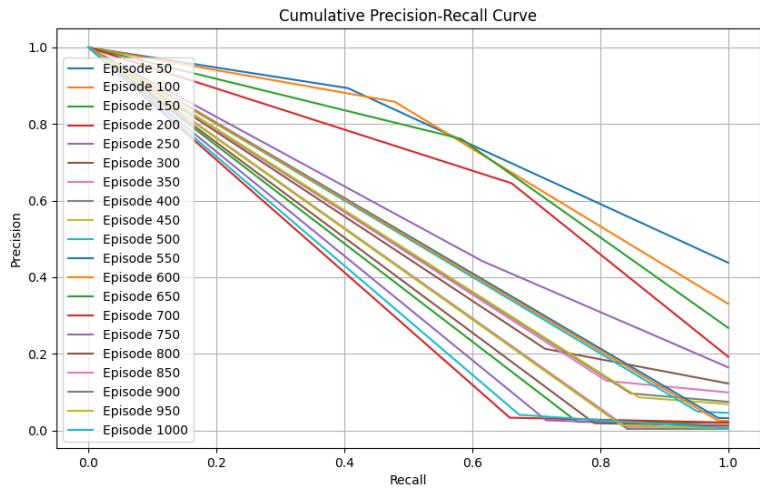


3.5.3 DDQN senza normalizzazione e senza Reward Imbalance Factor

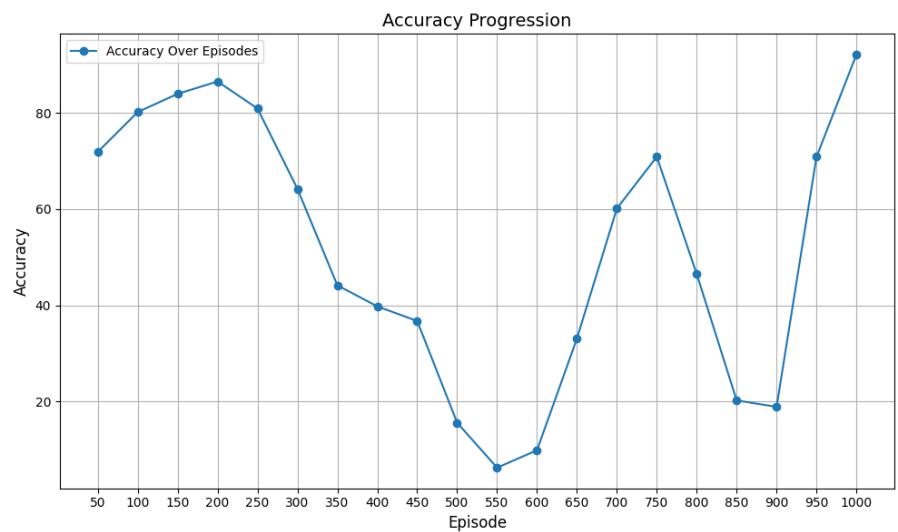


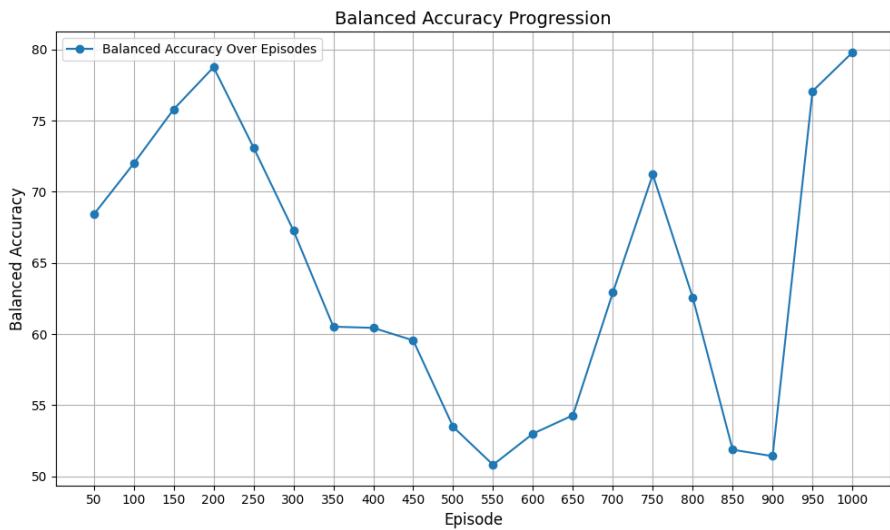
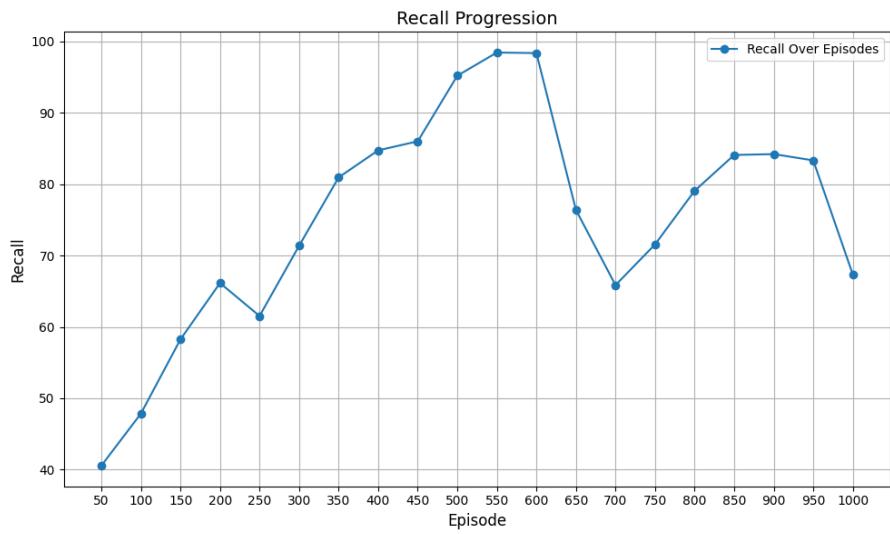
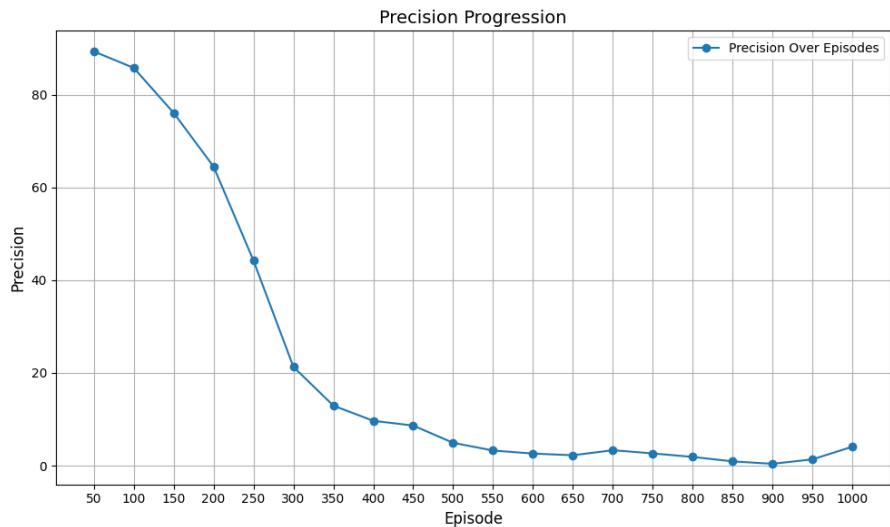


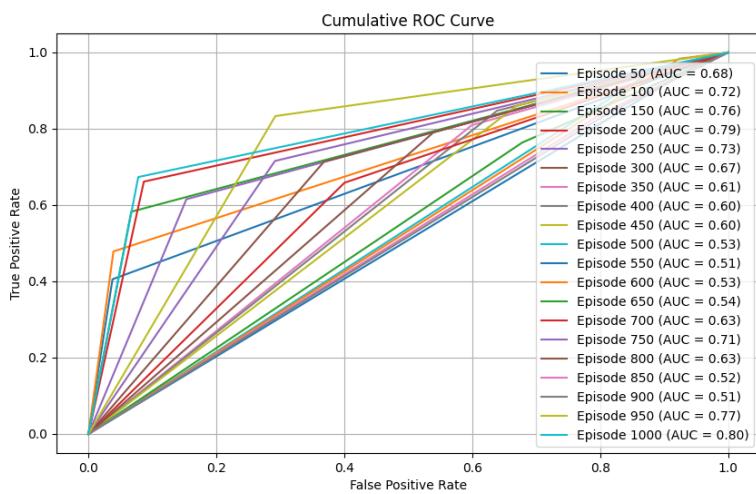
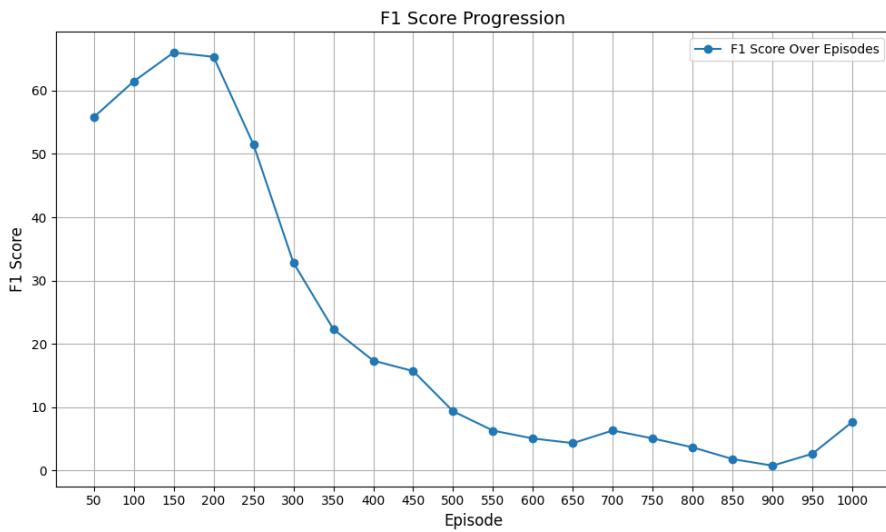
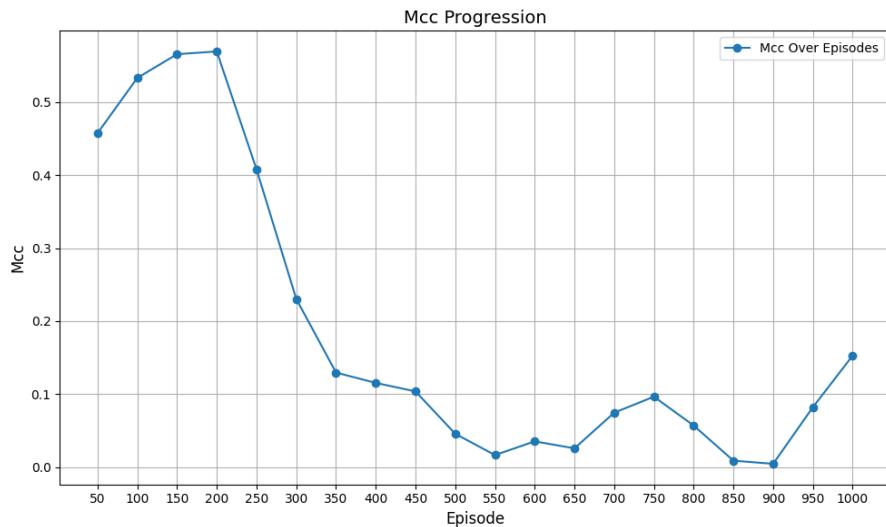


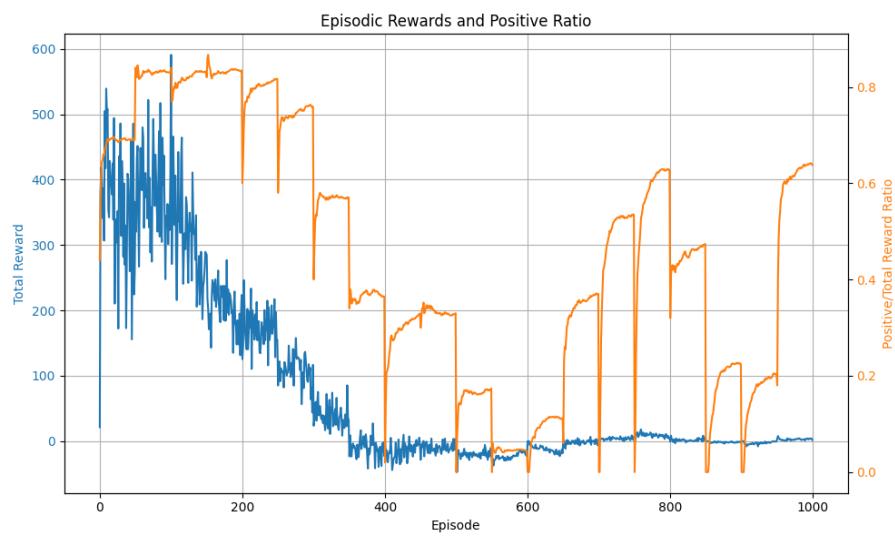
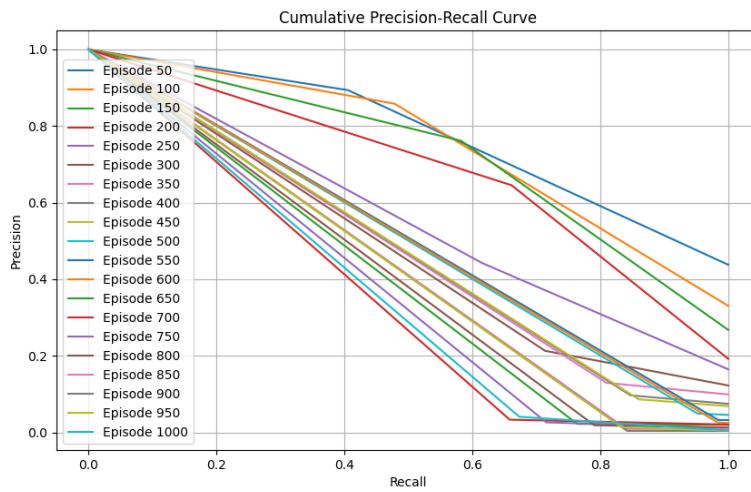


3.5.4 DDQN senza normalizzazione ma con Reward Imbalance Factor



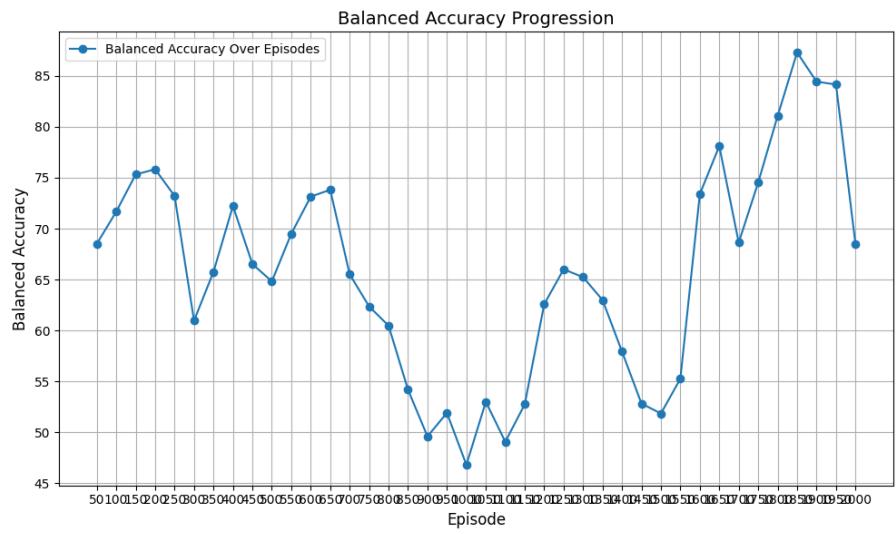
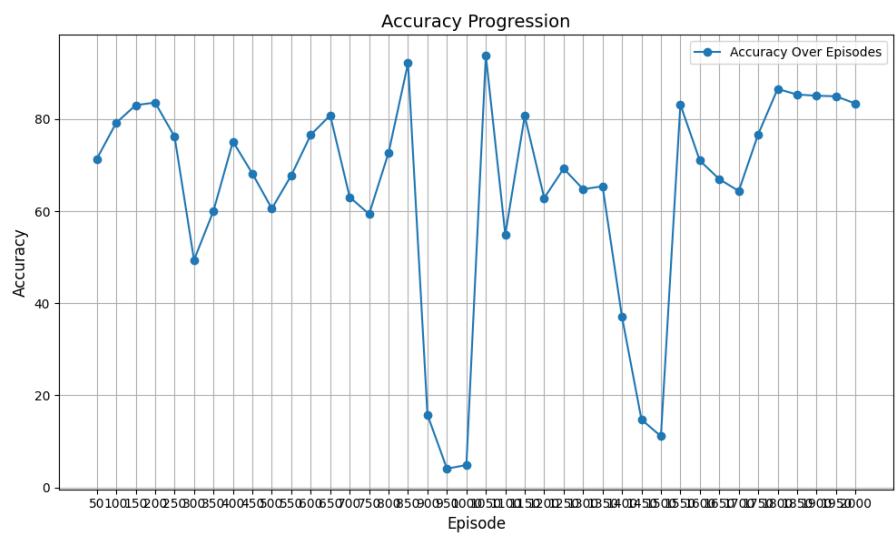
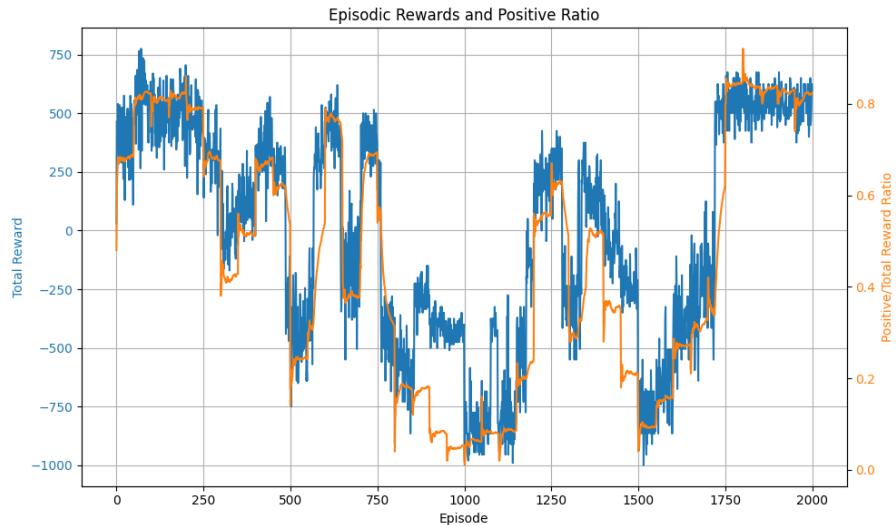


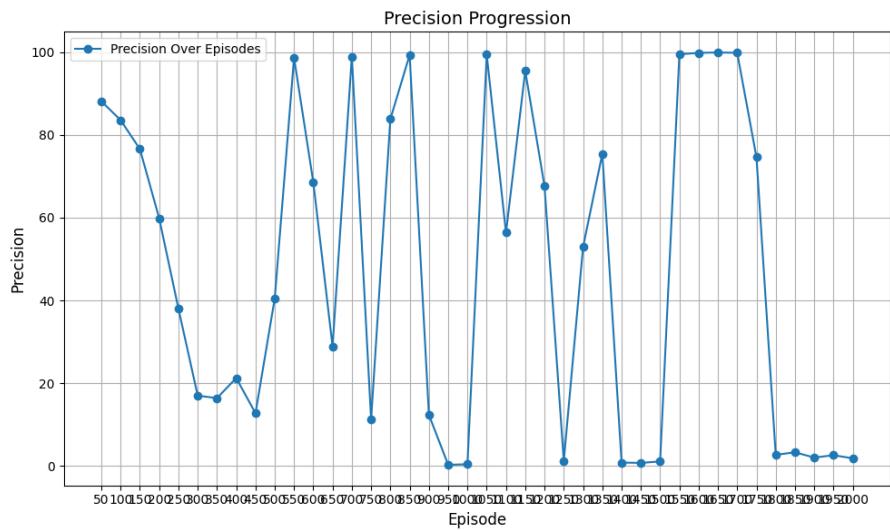
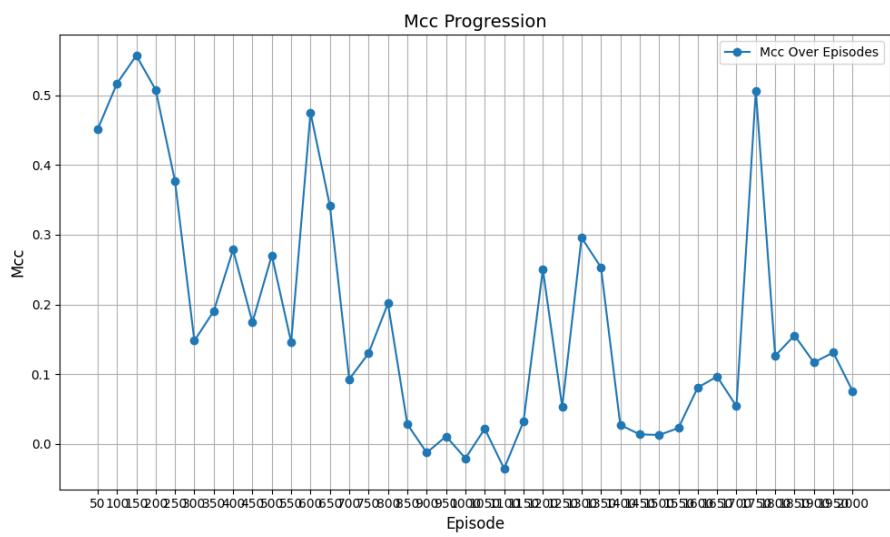
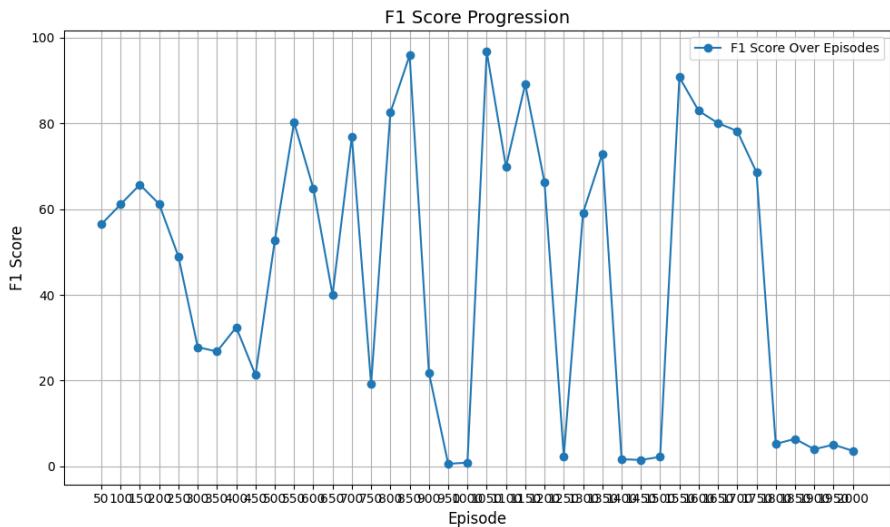


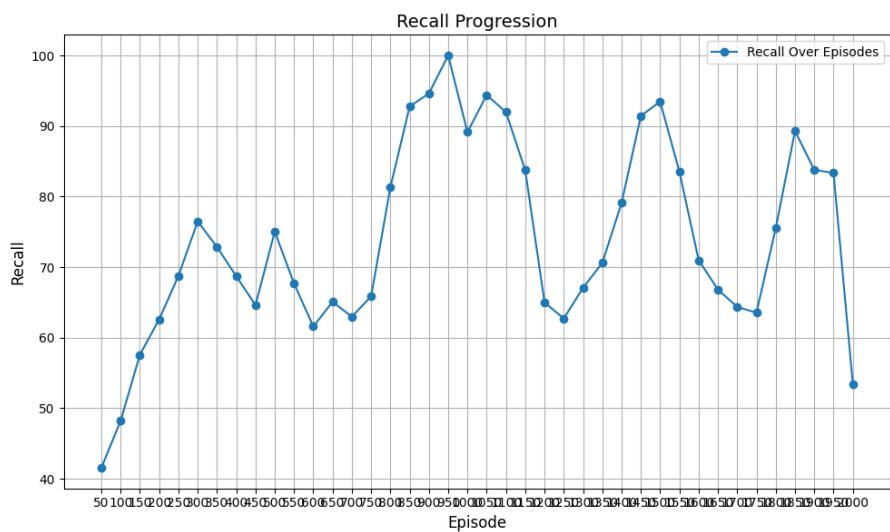
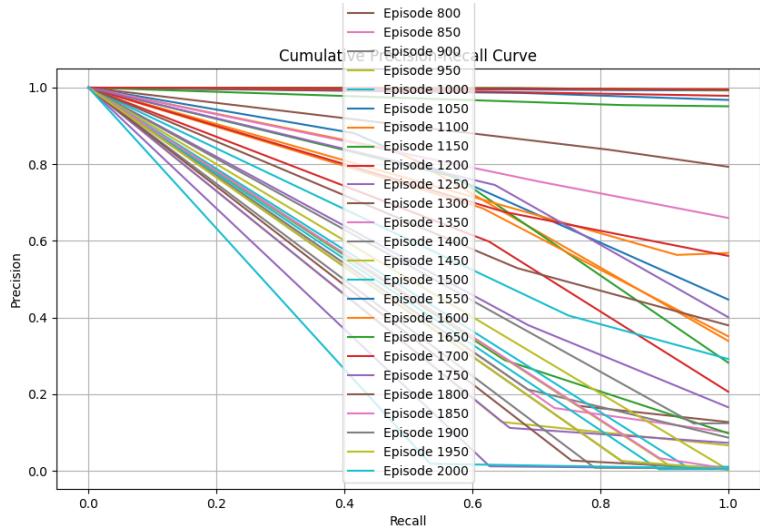
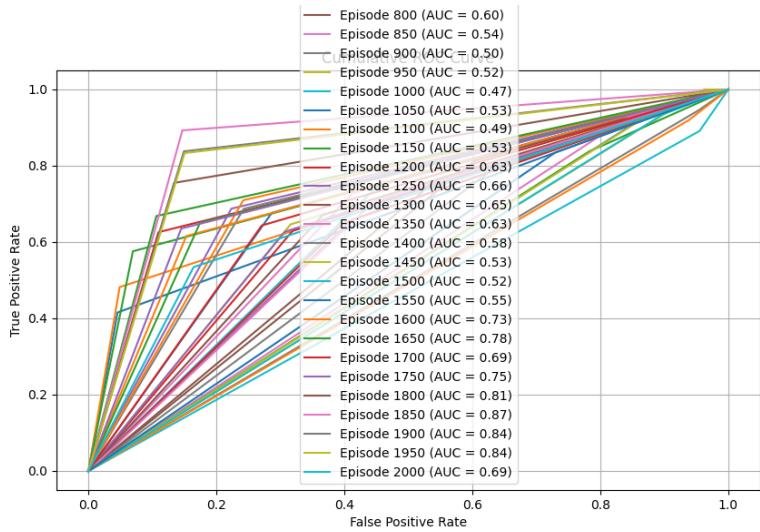


3.5.5 DDQN con normalizzazione delle reward ma senza Reward Imbalance Factor test su 2000 episodi

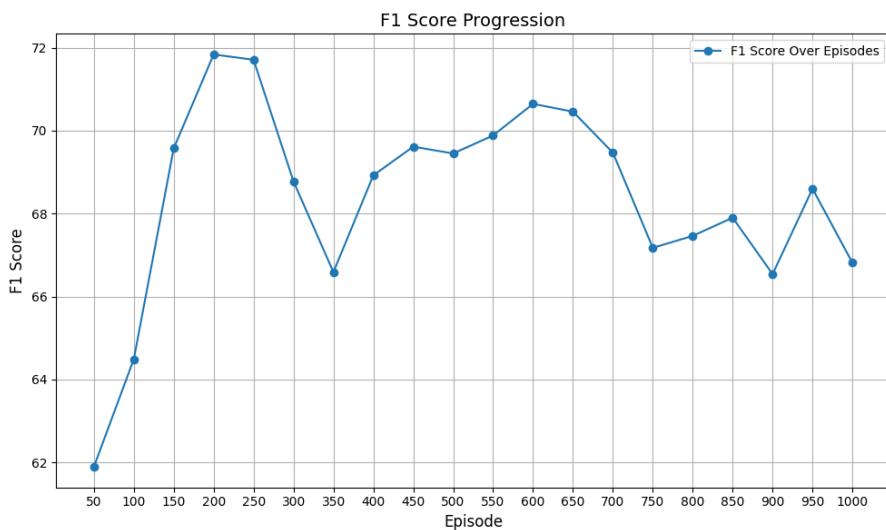
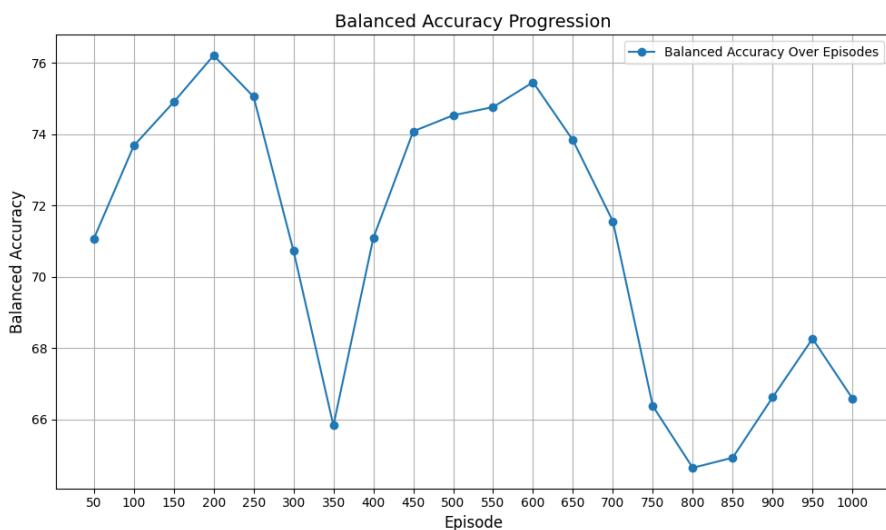
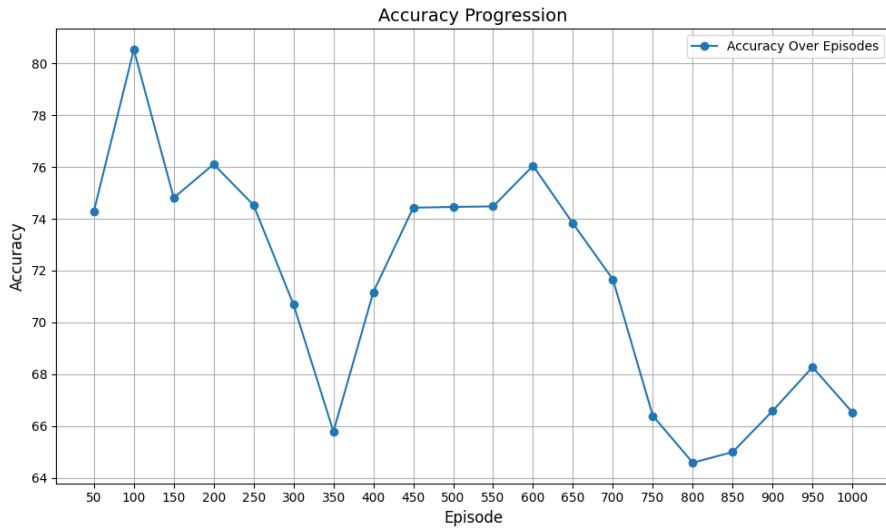
Questo è lo stesso test precedente ma fatto su 2000 episodi.

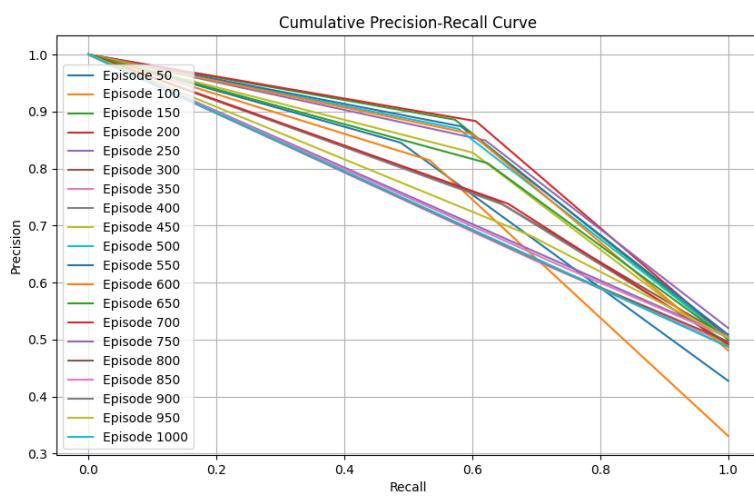
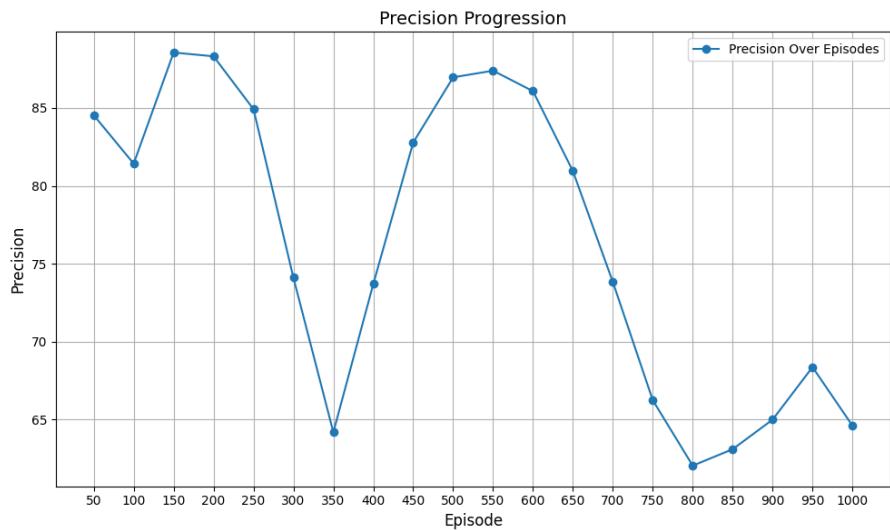
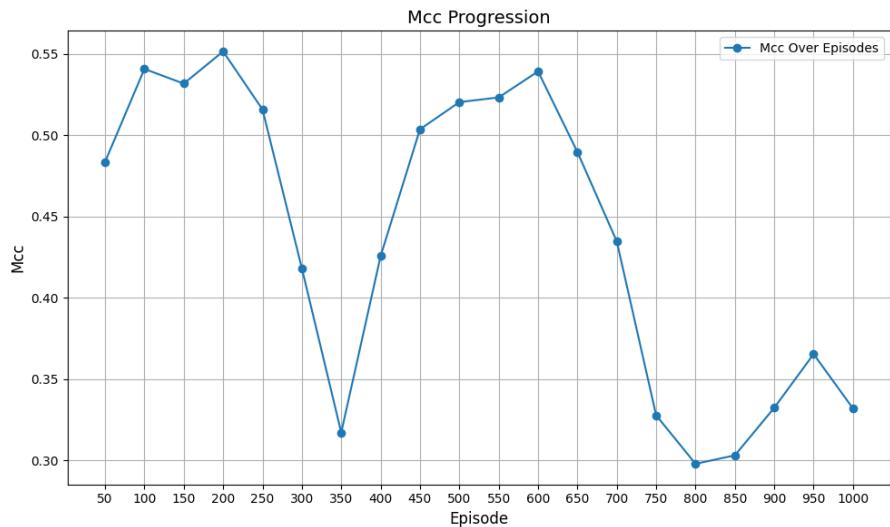


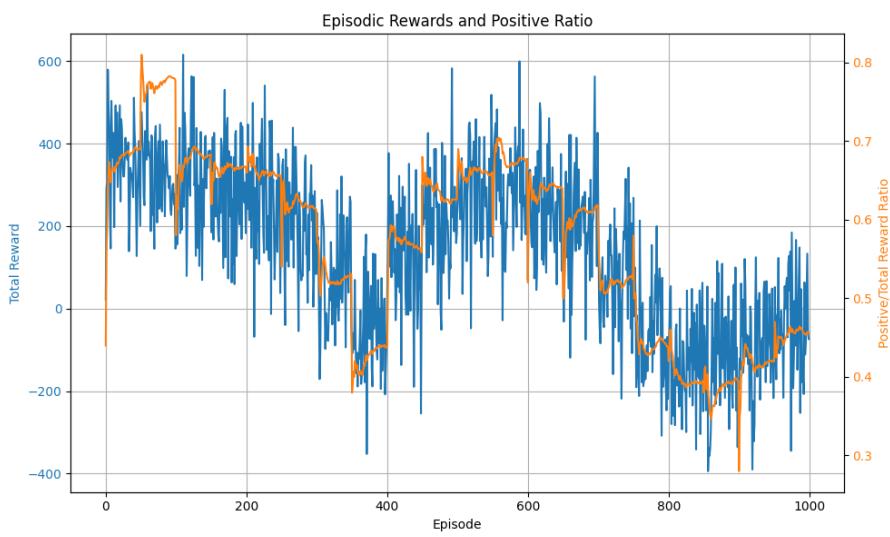
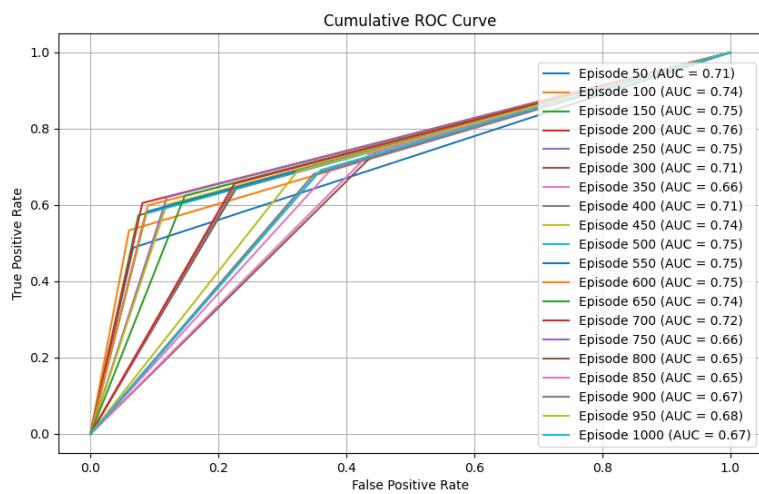
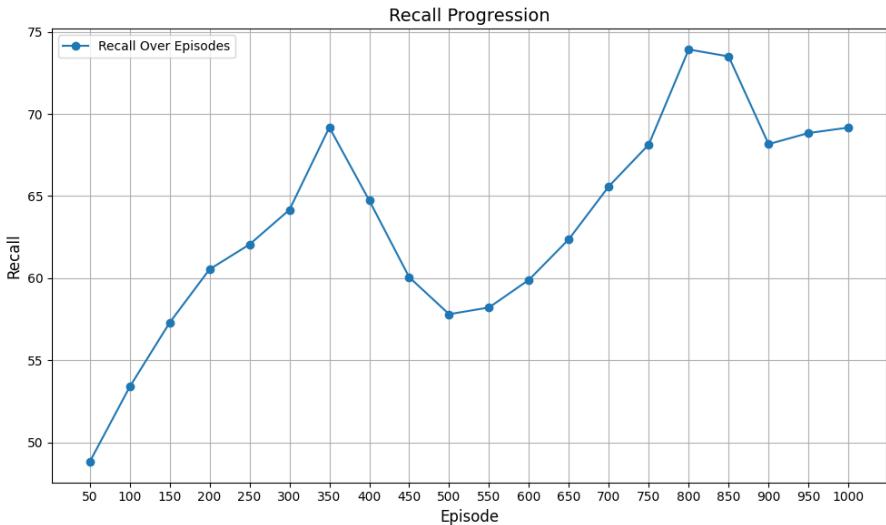




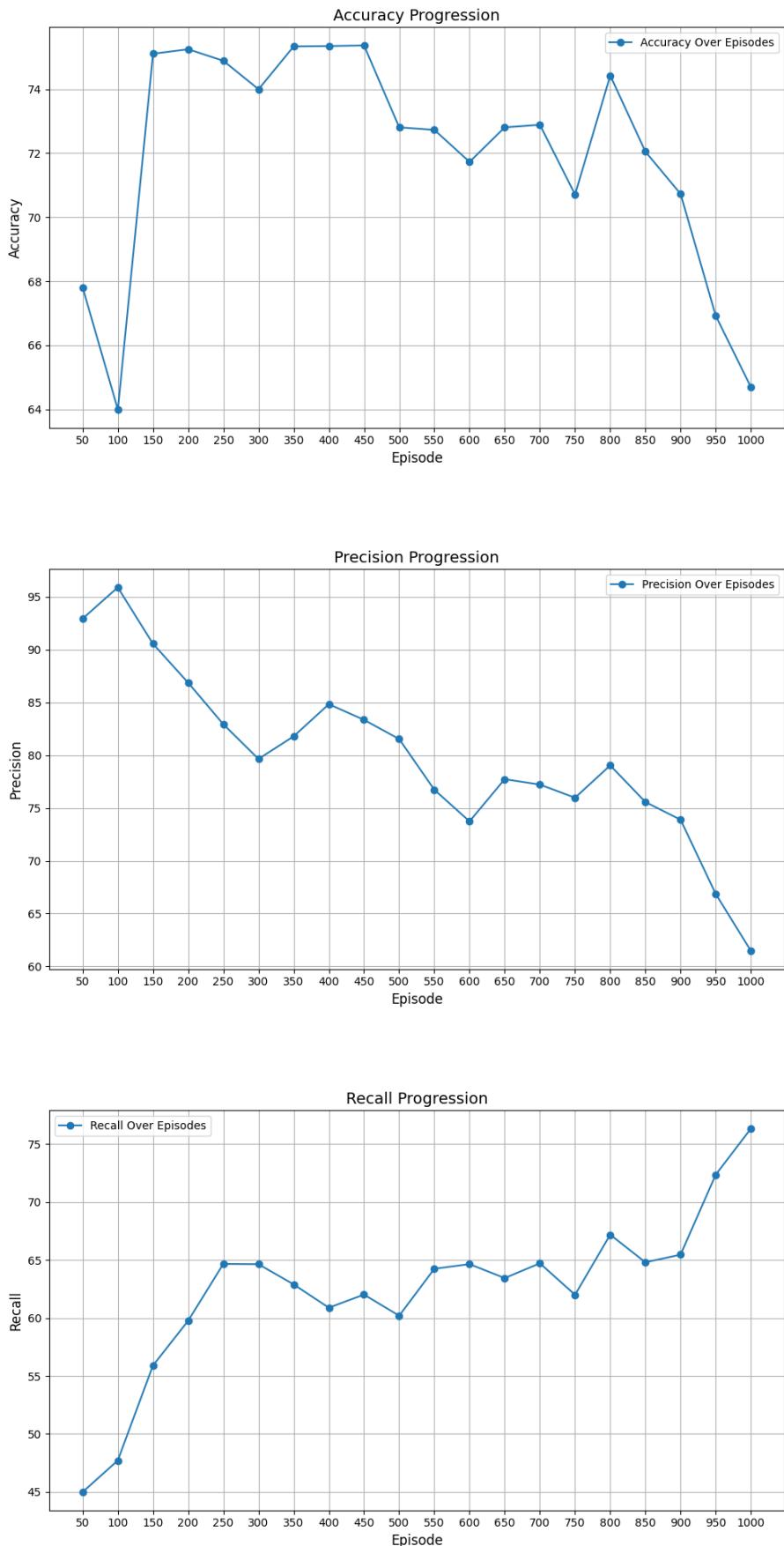
3.5.6 DQN con normalizzazione ed imbalance factor

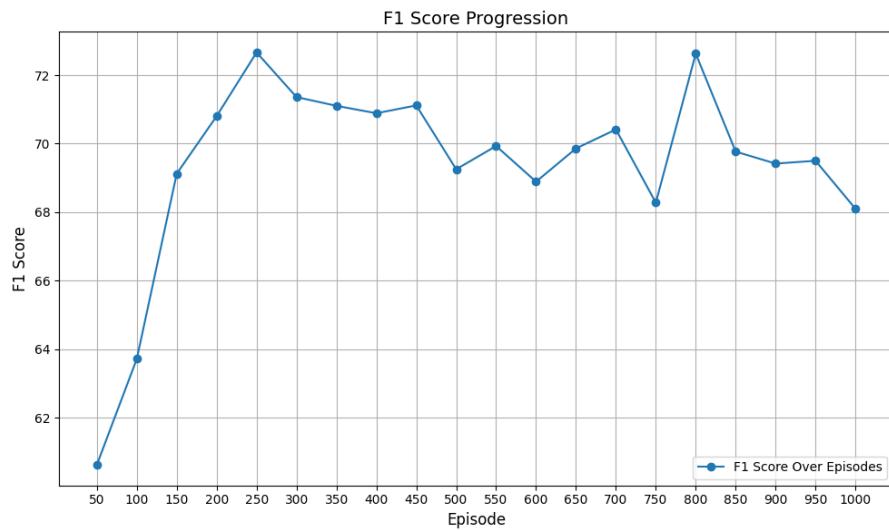
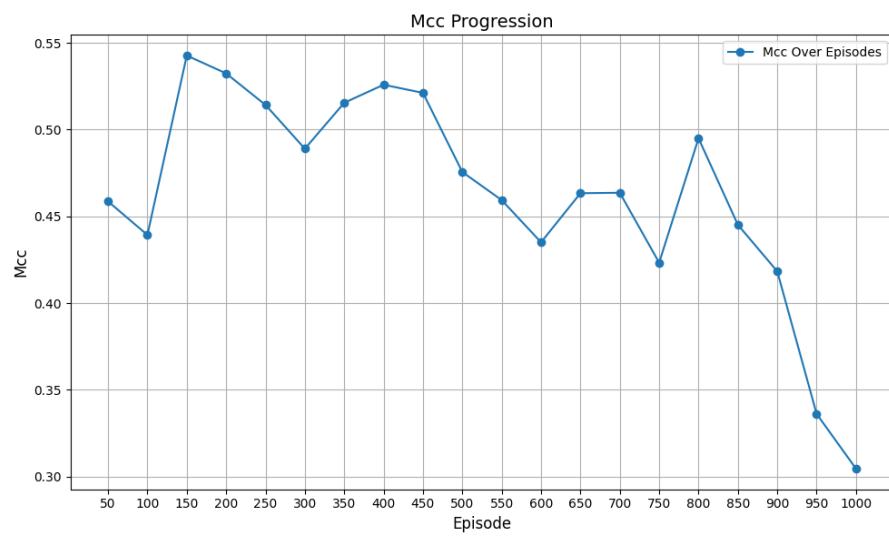
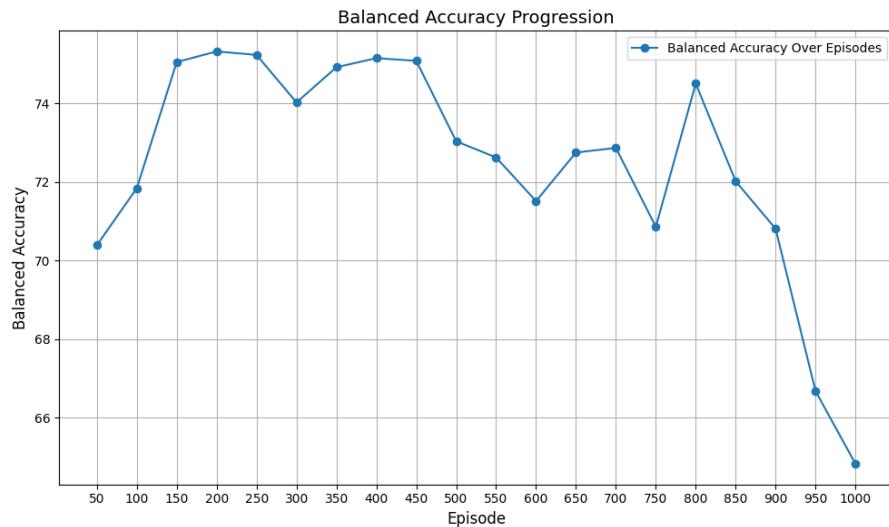


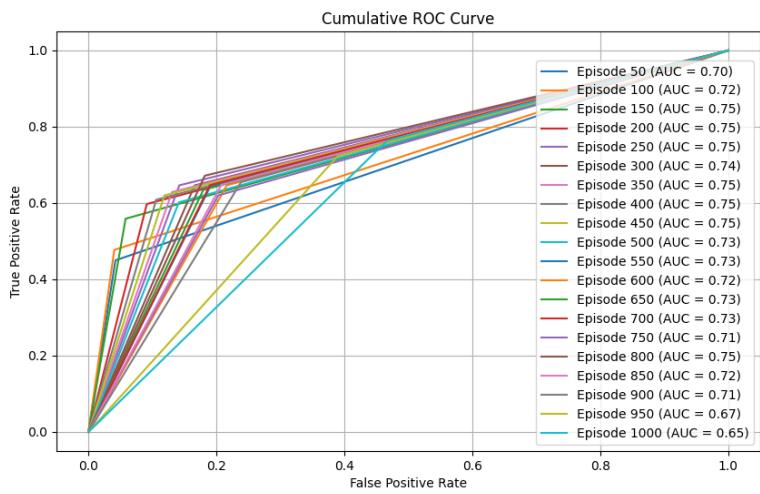
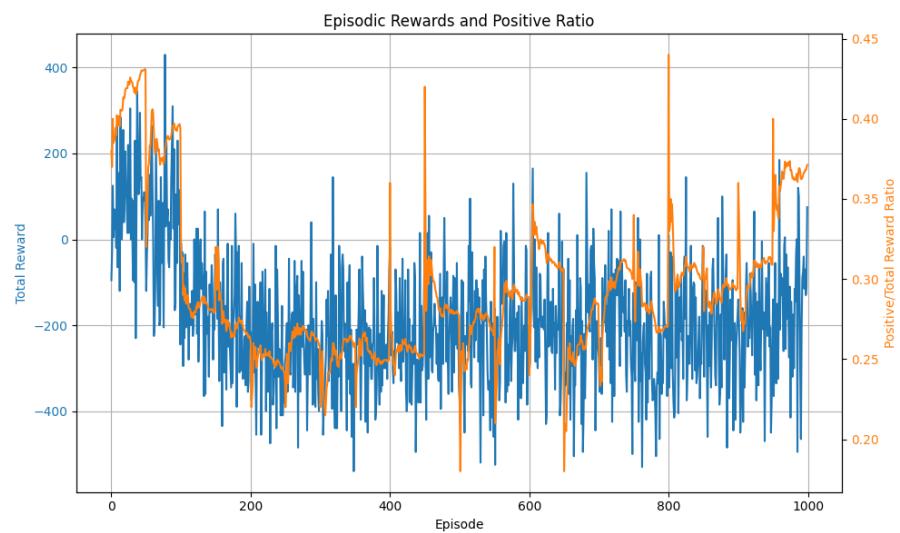
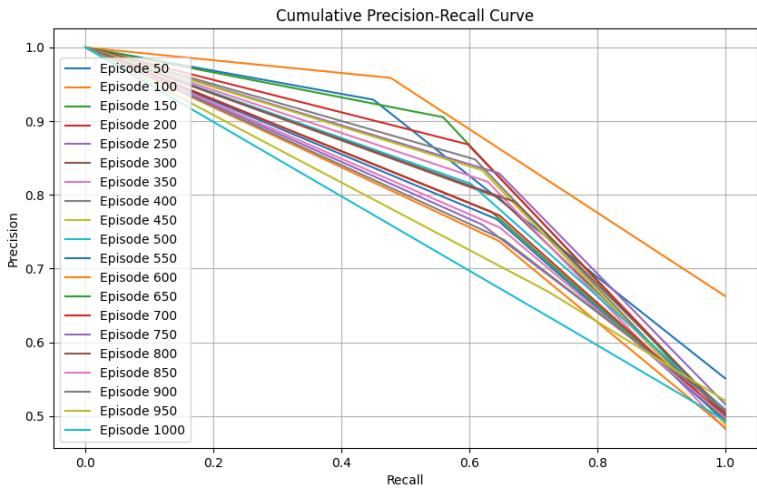




3.5.7 DQN con normalizzazione ma senza imbalance factor

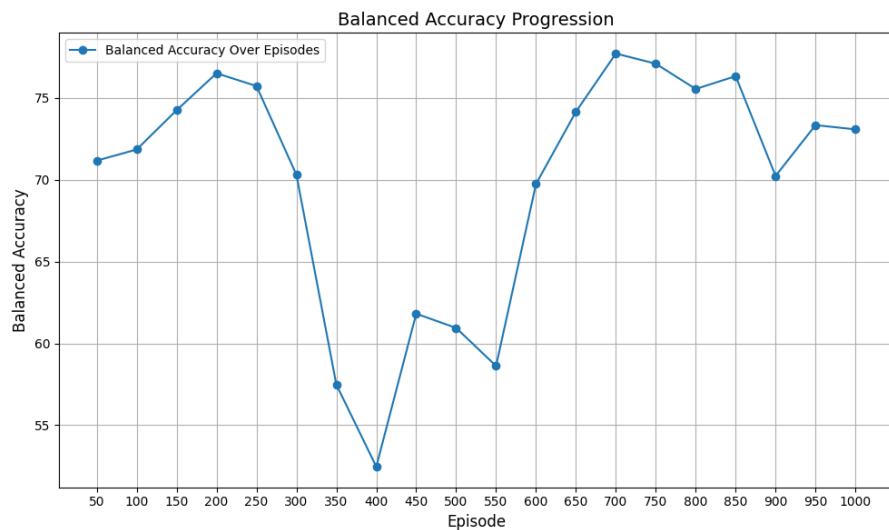
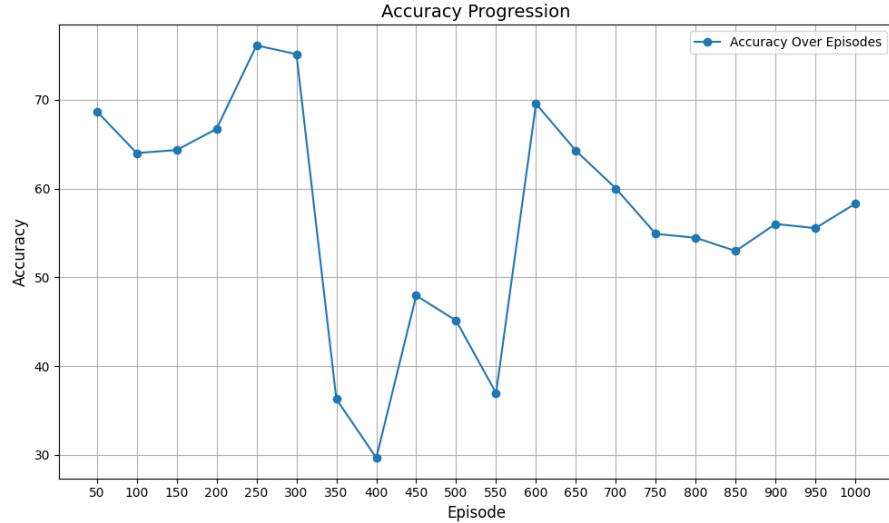


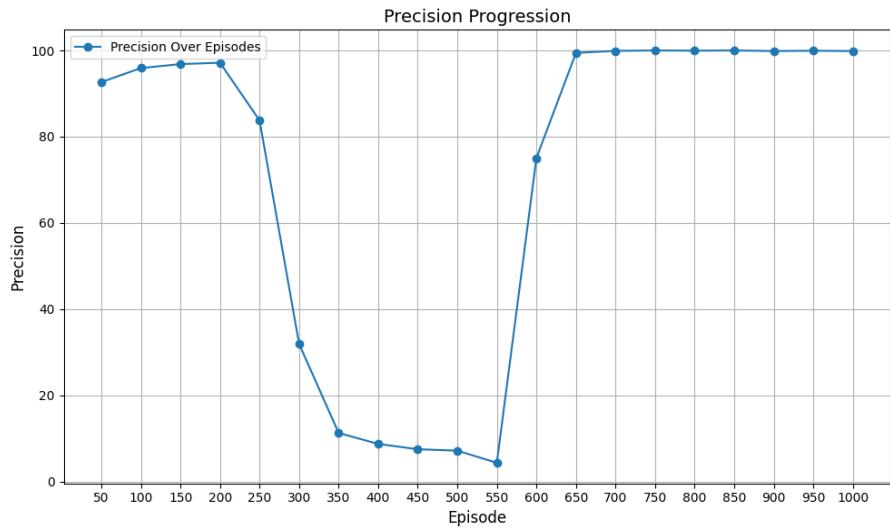
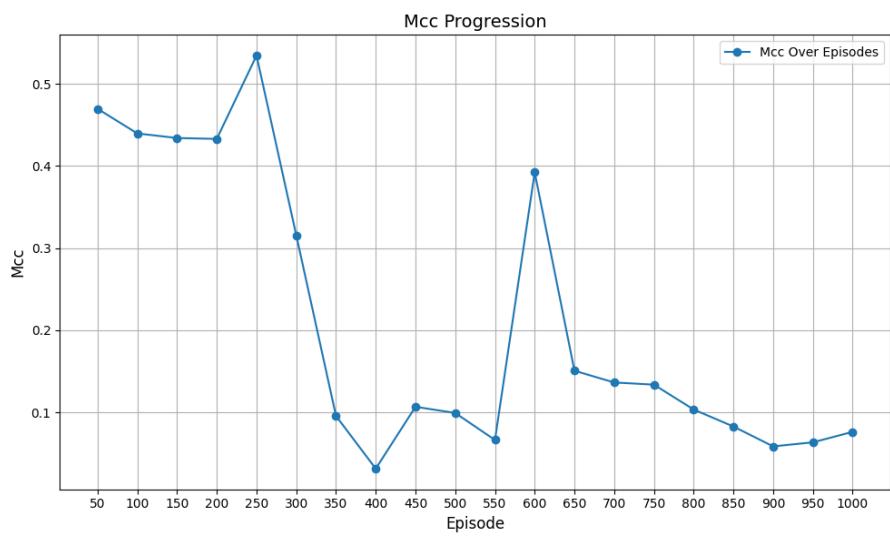
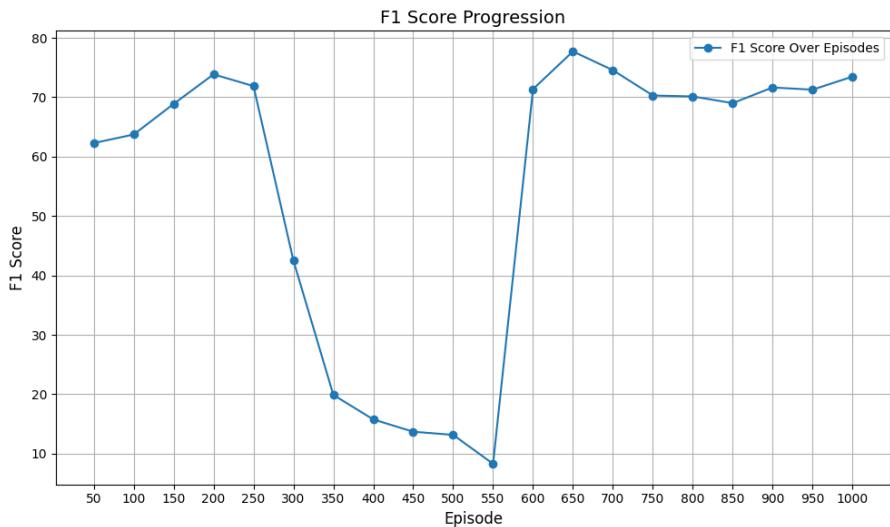


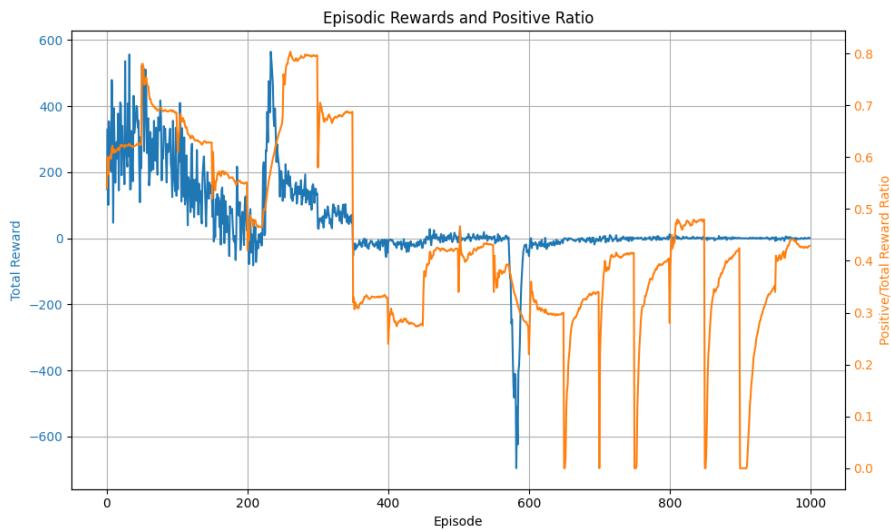
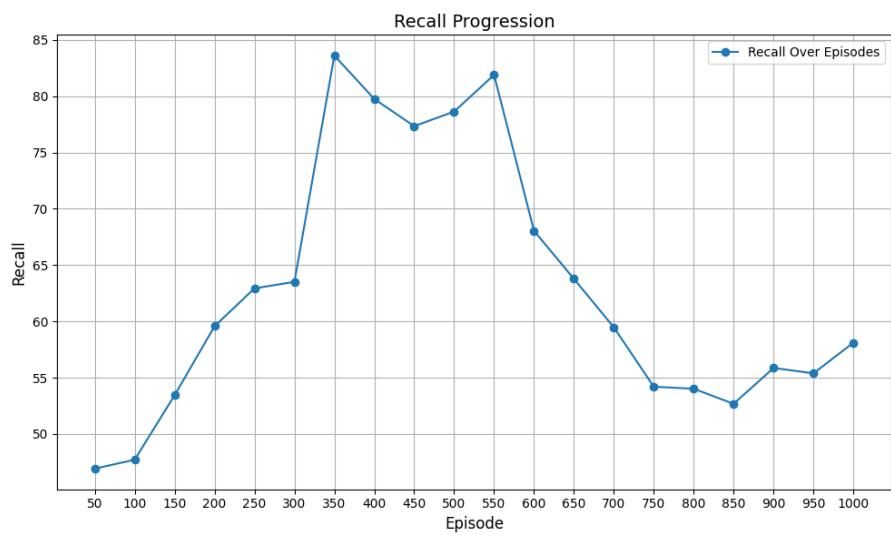
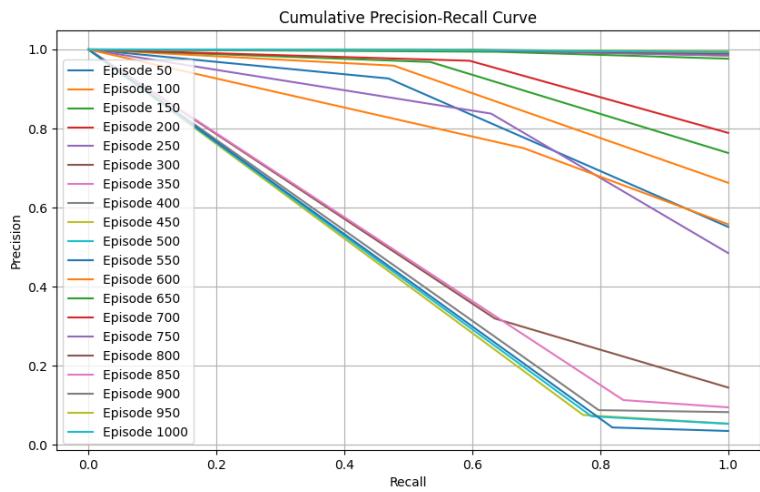


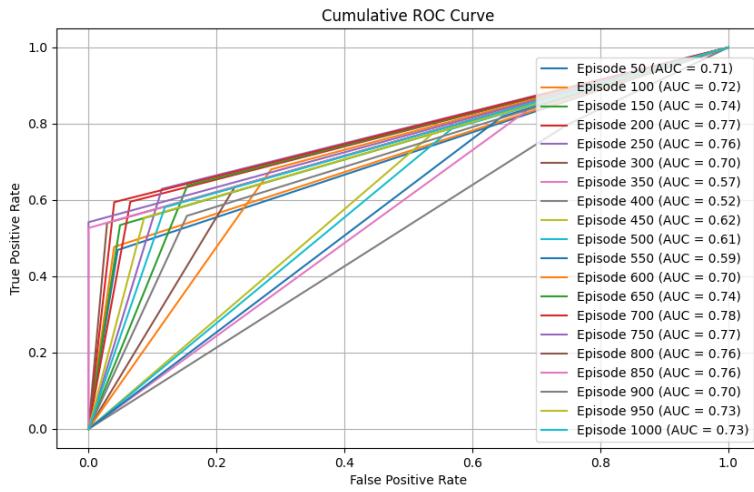
3.5.8 Q-Learning con rete neurale

Questo è un tentativo un pò inusuale che usa la formula di aggiornamento di QL ma con una rete neurale invece di una lookup table. Sotto poi è presente la versione standard con lookup table.

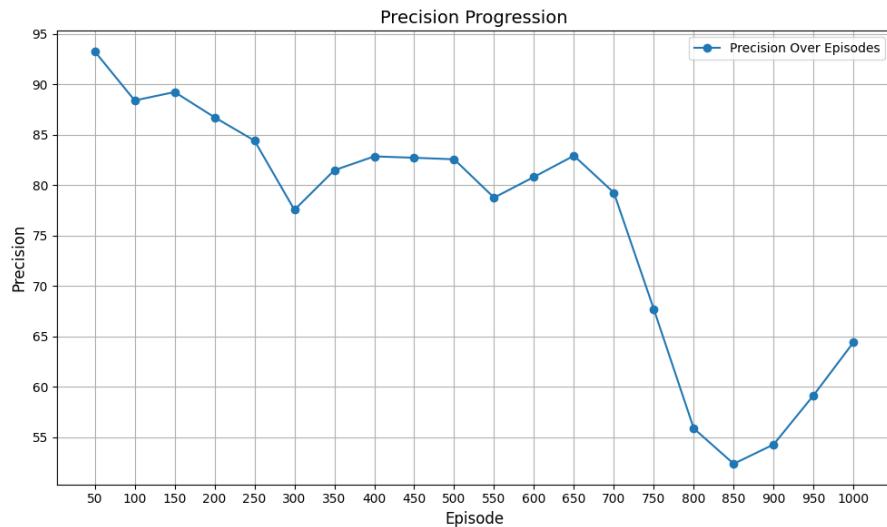
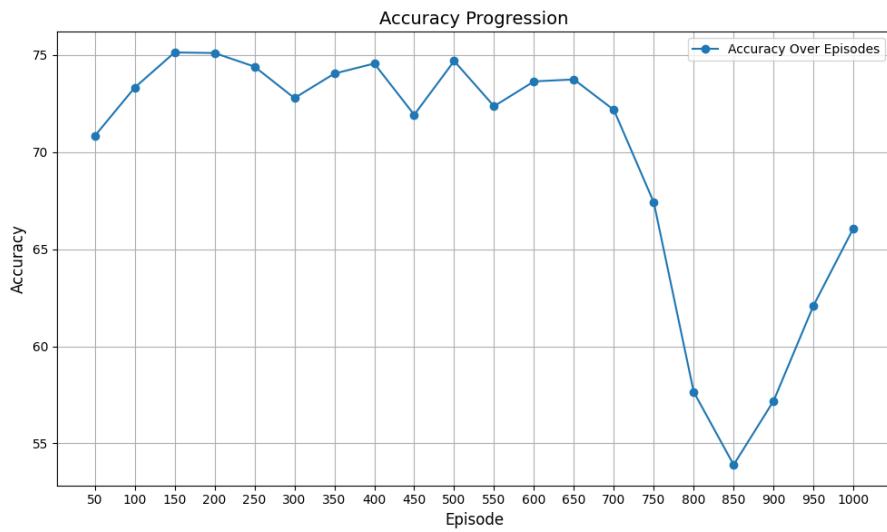


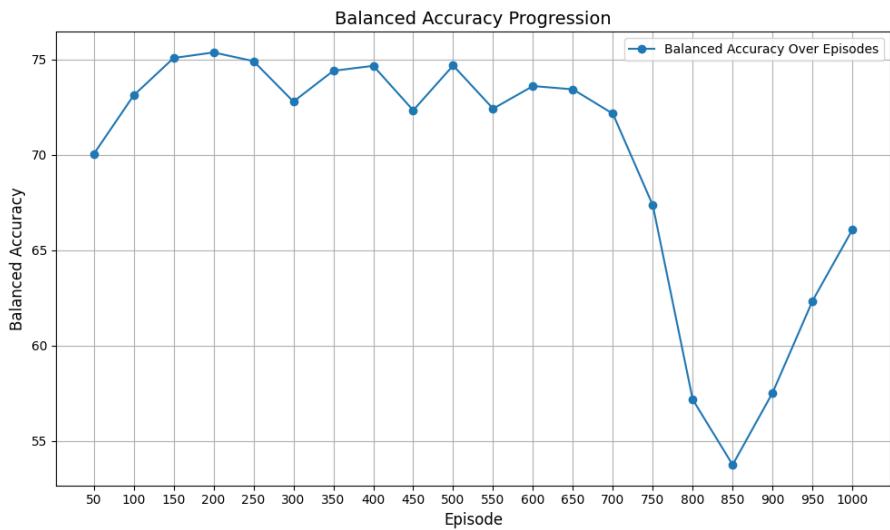
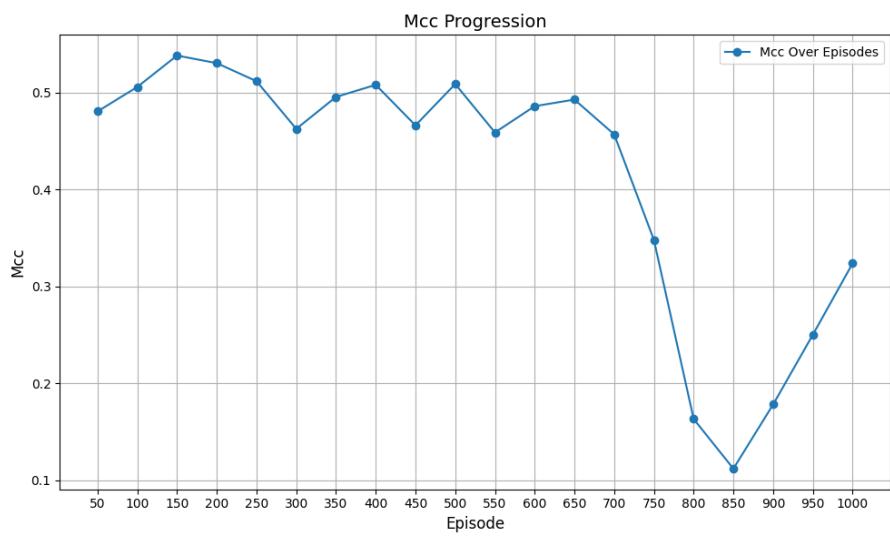
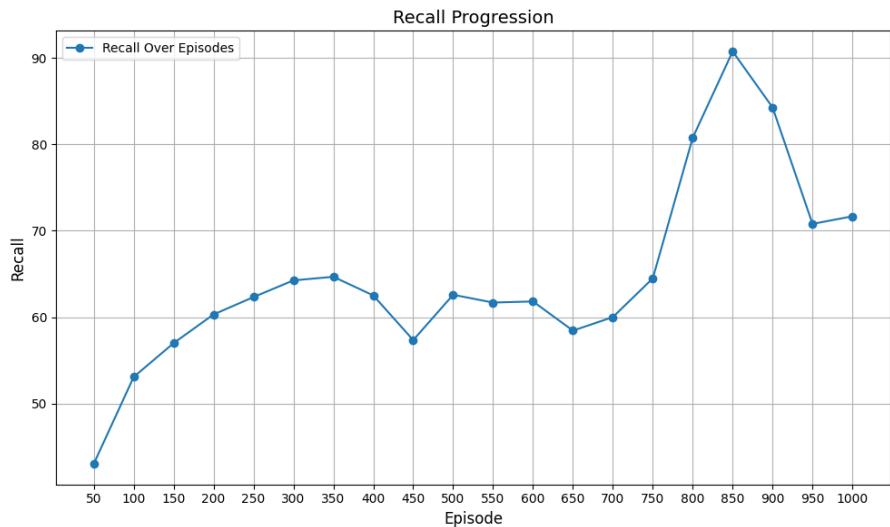


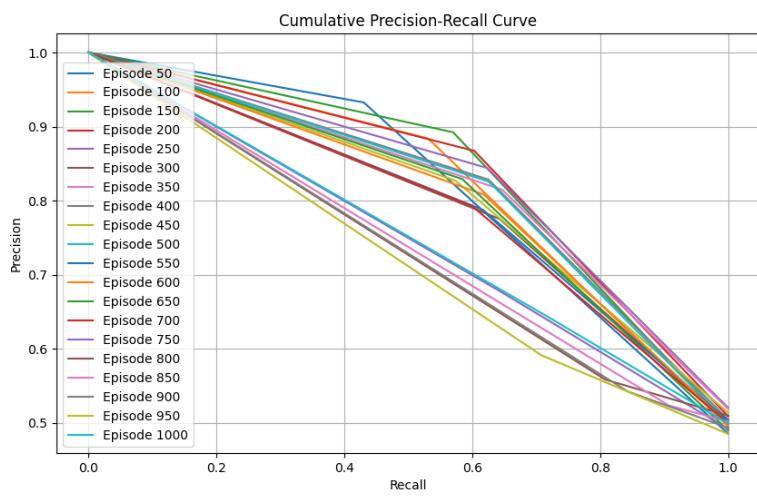
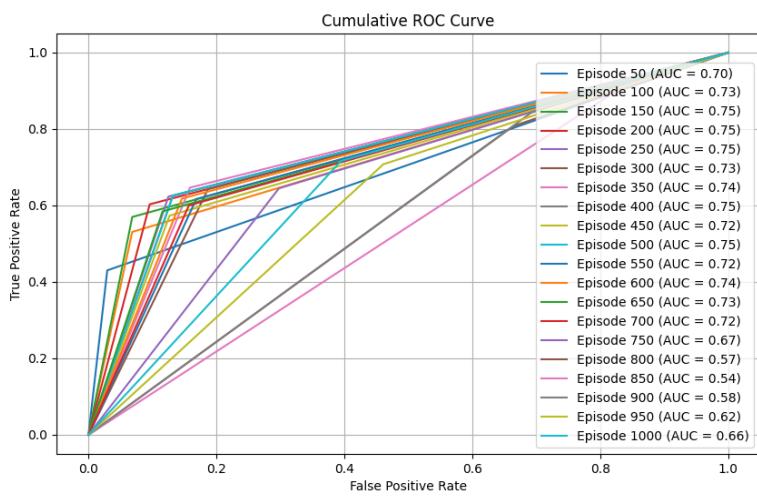
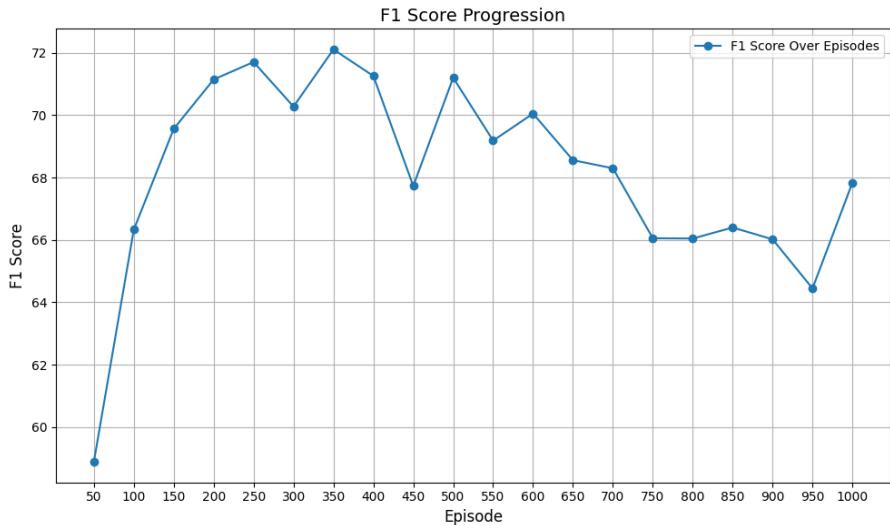


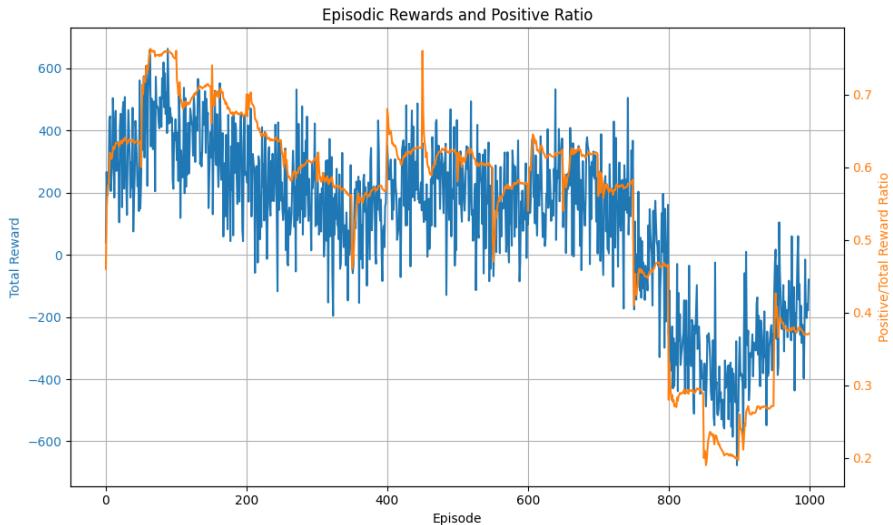


3.5.9 Q-Learning standard con lookup table senza normalizzazione ma con imbalance factor

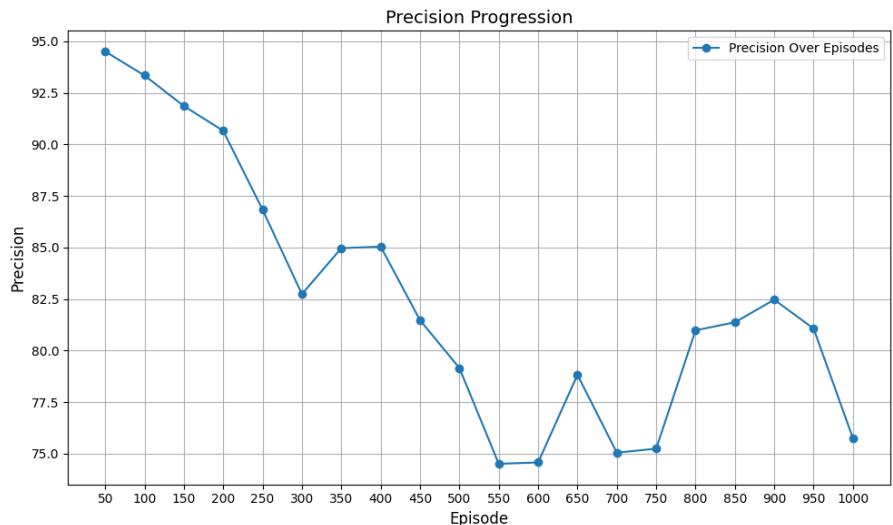
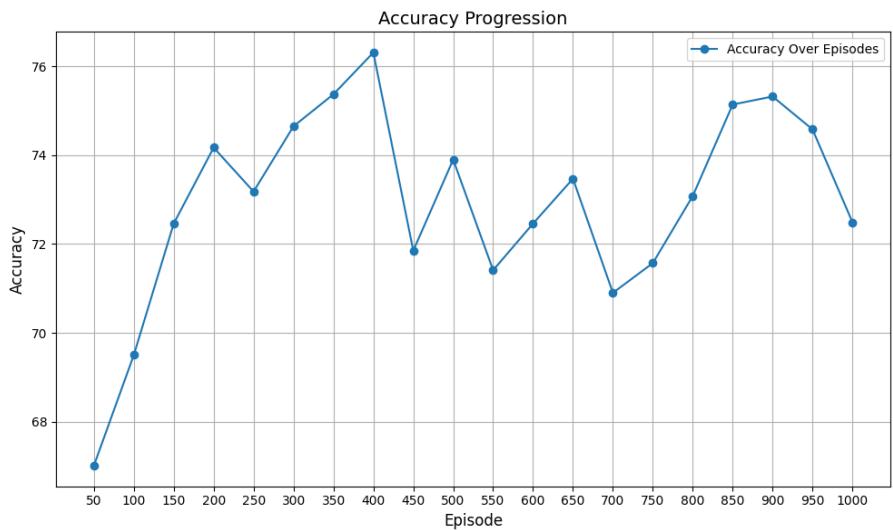


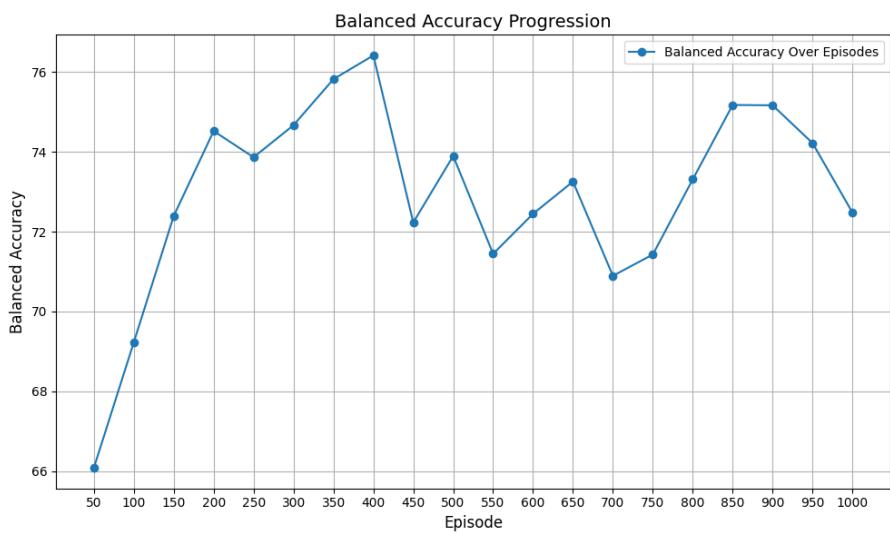
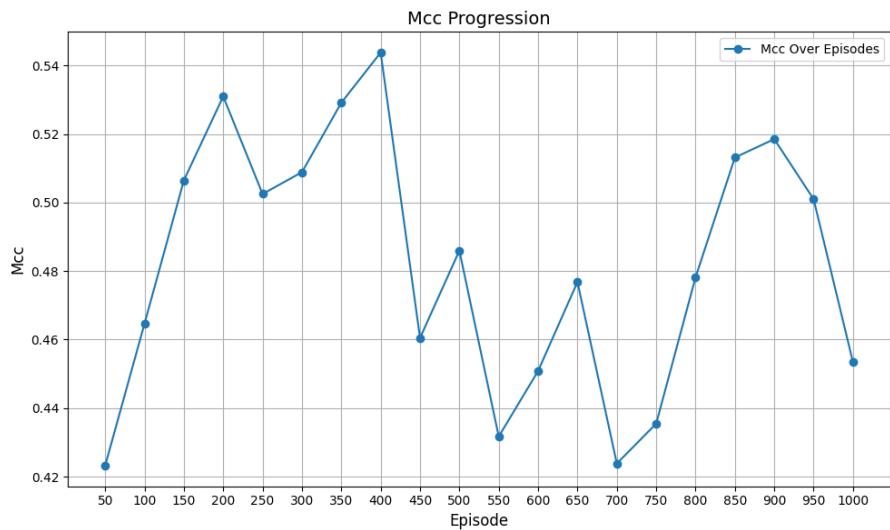
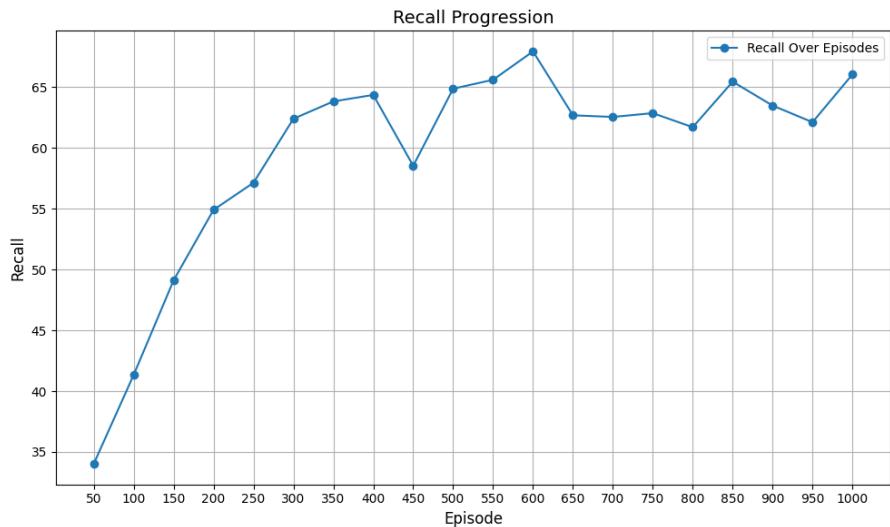


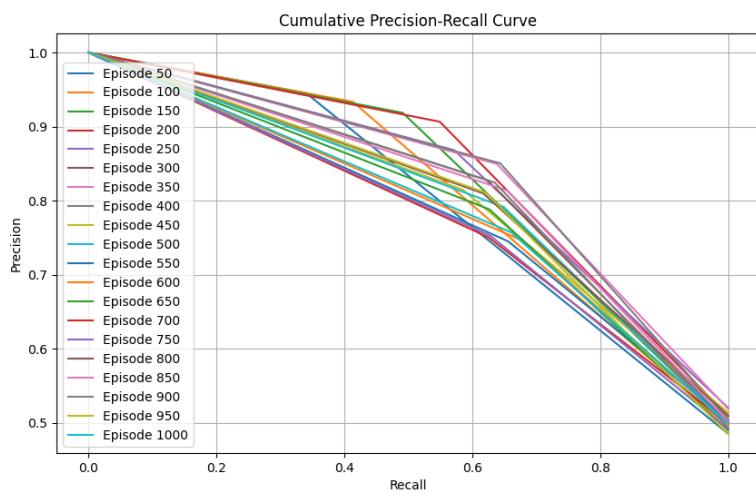
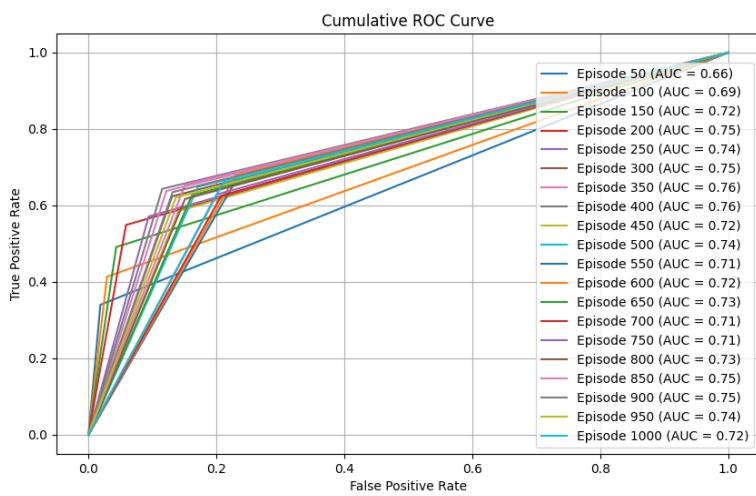
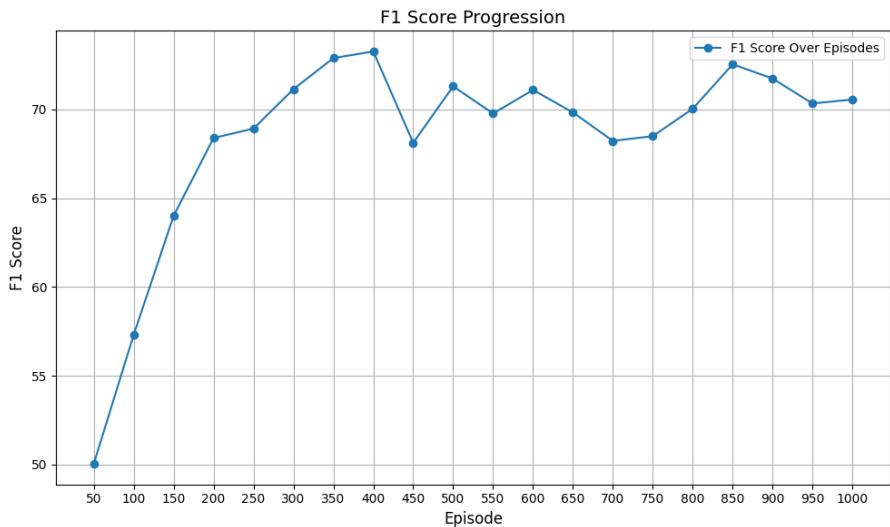


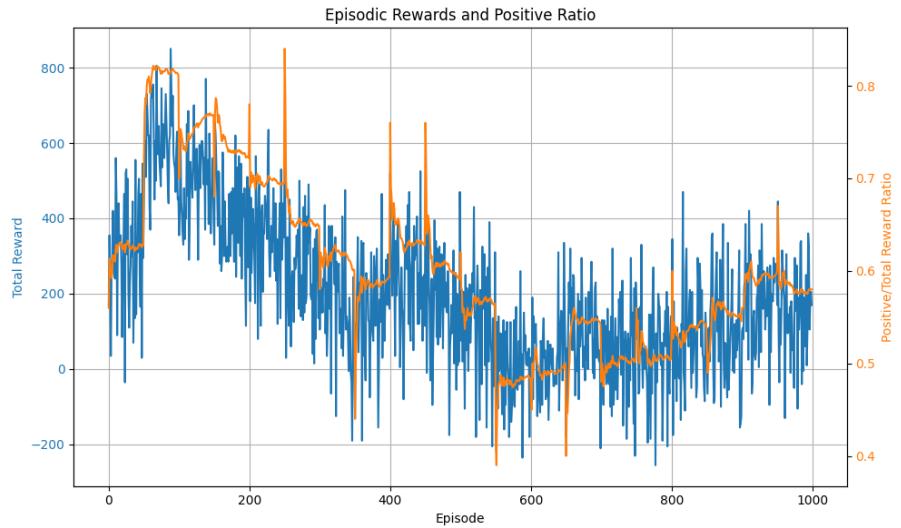


3.5.10 Q-Learning standard con lookup table con normalizzazione ma senza imbalance factor

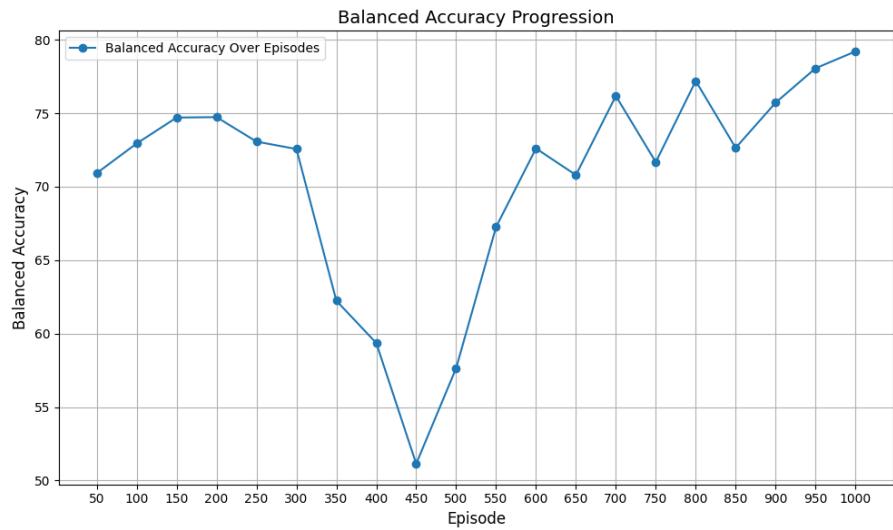
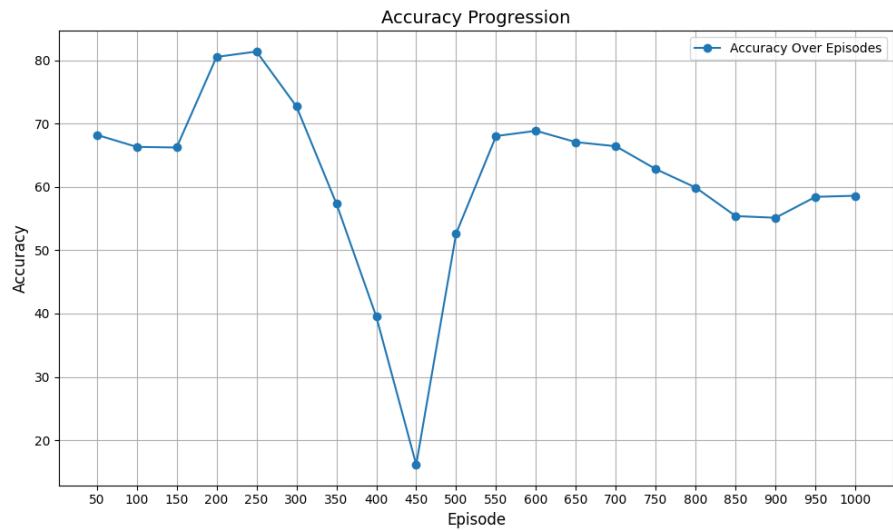


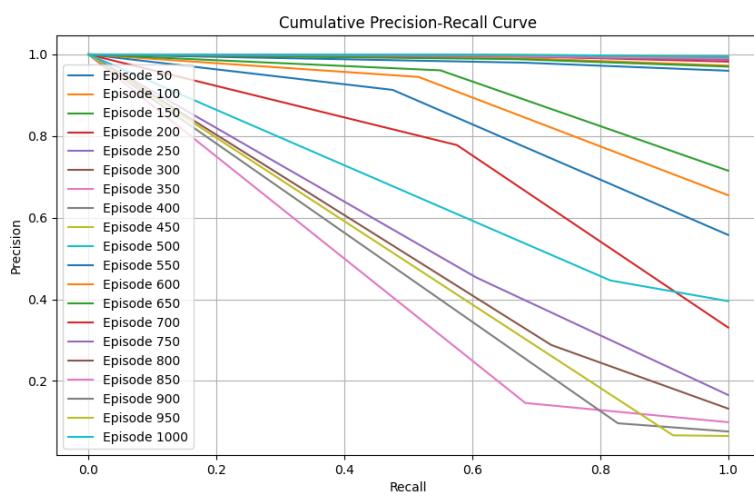
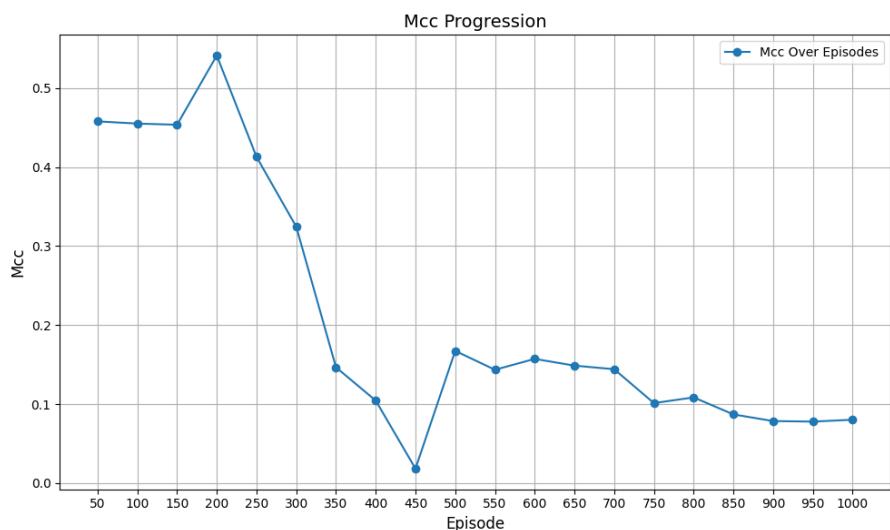
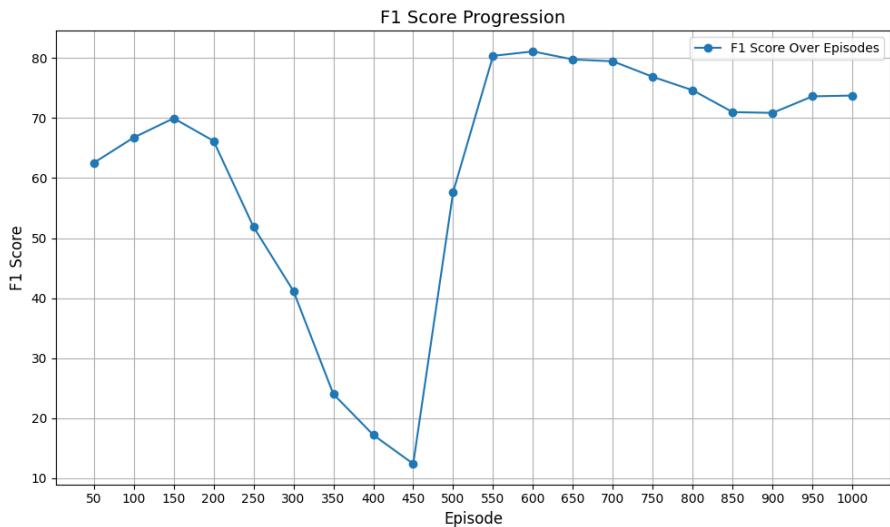


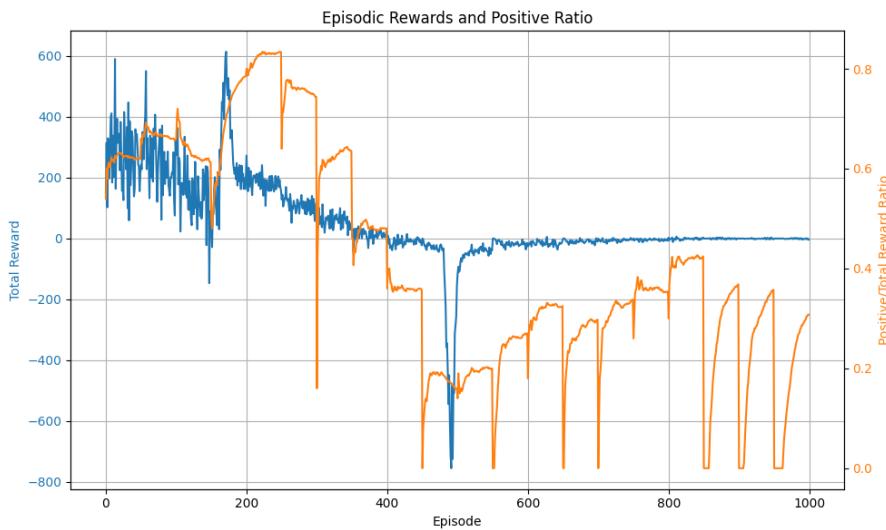
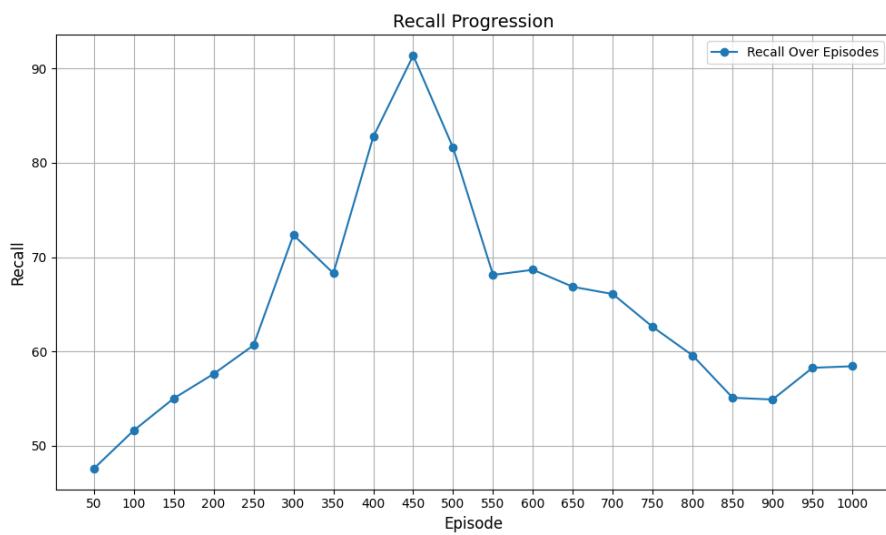
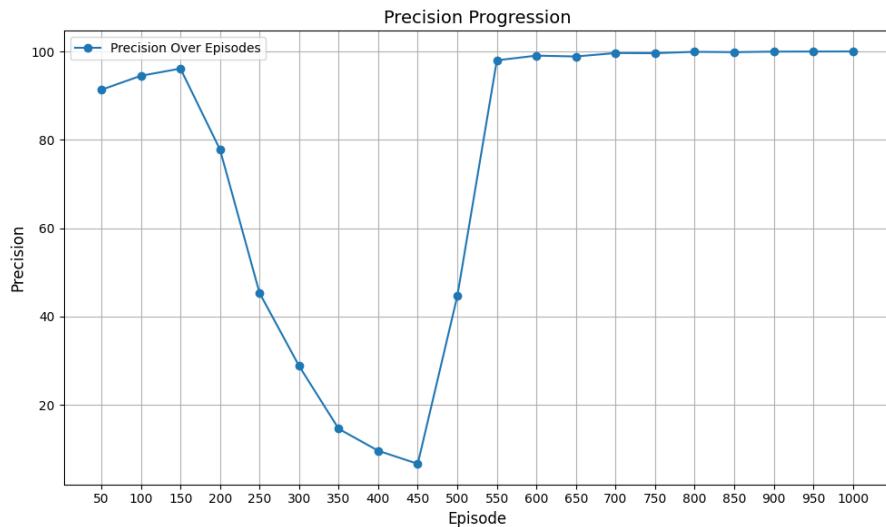


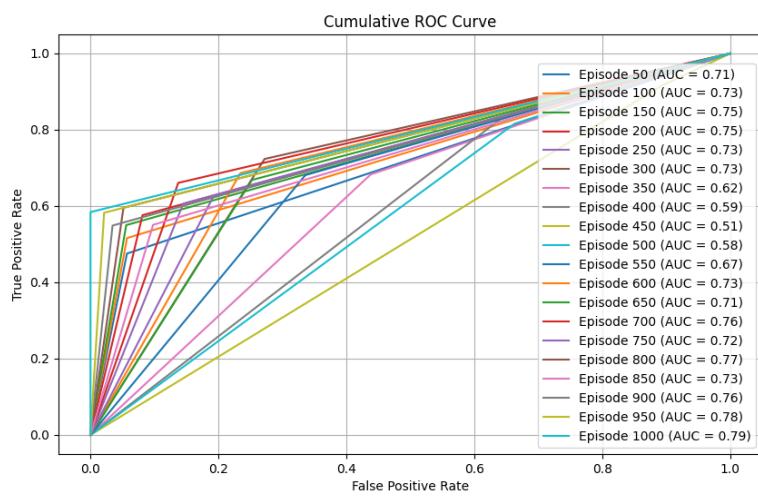


3.5.11 SARSA con normalizzazione ed imbalance factor





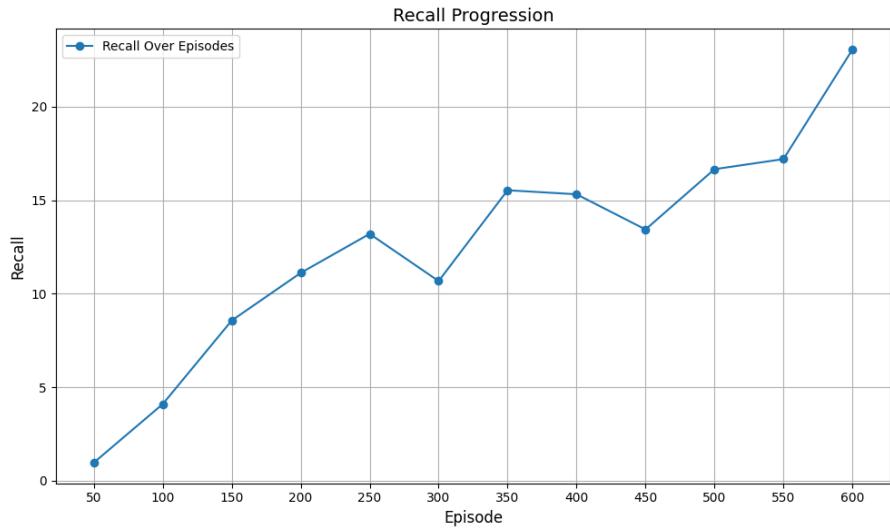
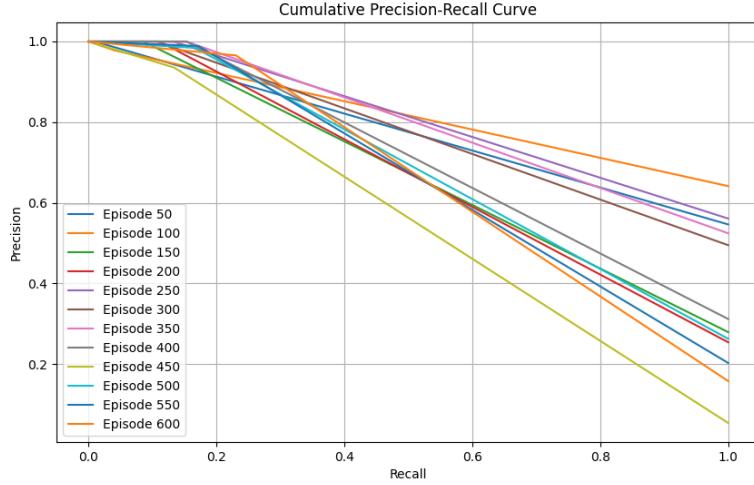


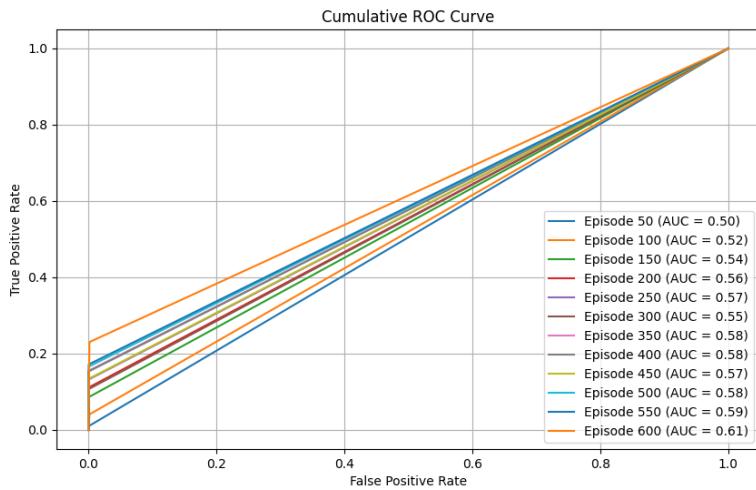
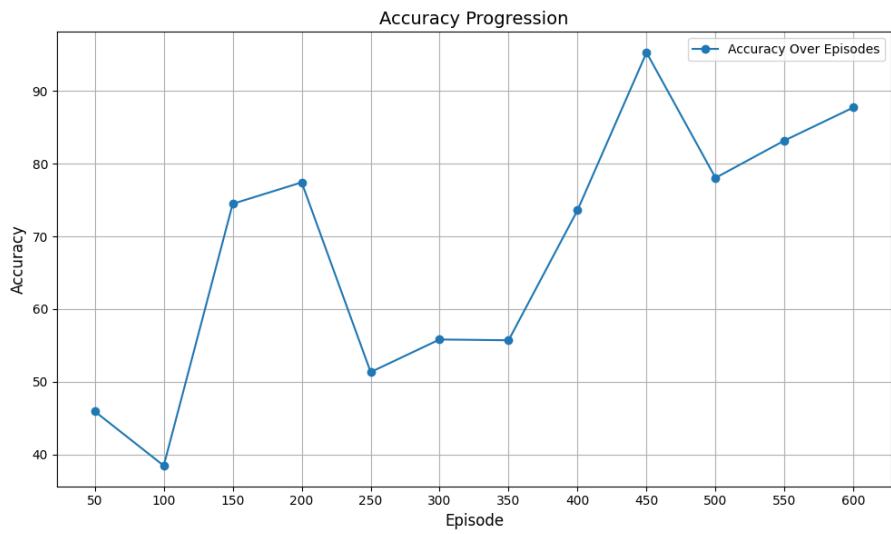
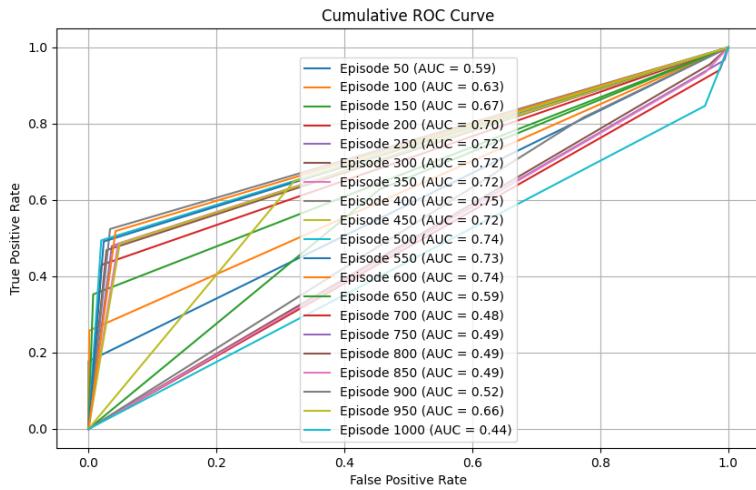


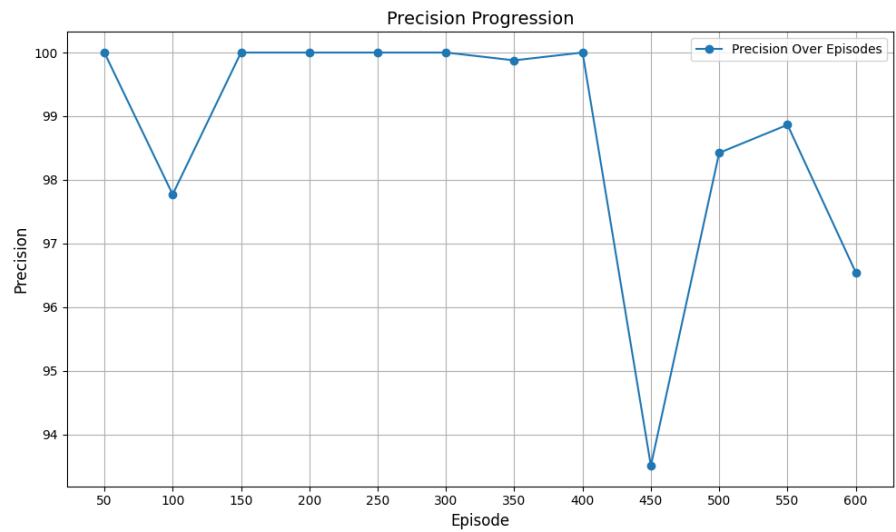
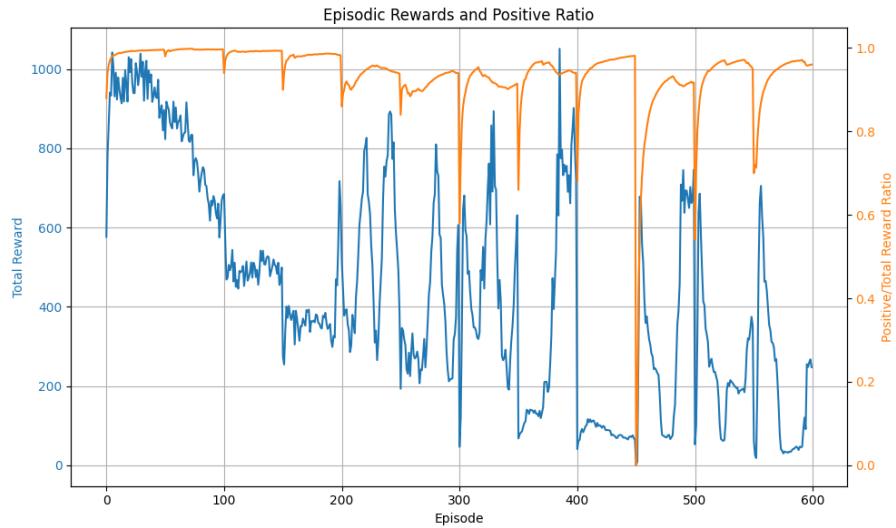
3.6 Risultati agenti ed IDS

3.6.1 GAT/DQN con normalizzazione delle reward ed imbalance factor

Con learning rate pari a 0.001 per l'agente e pari a 0.01 per la GNN. La GNN ha inoltre uno scheduler che ha step size pari a 4 e gamma pari a 0.99 per fare decrescere il learning rate col passare degli episodi.

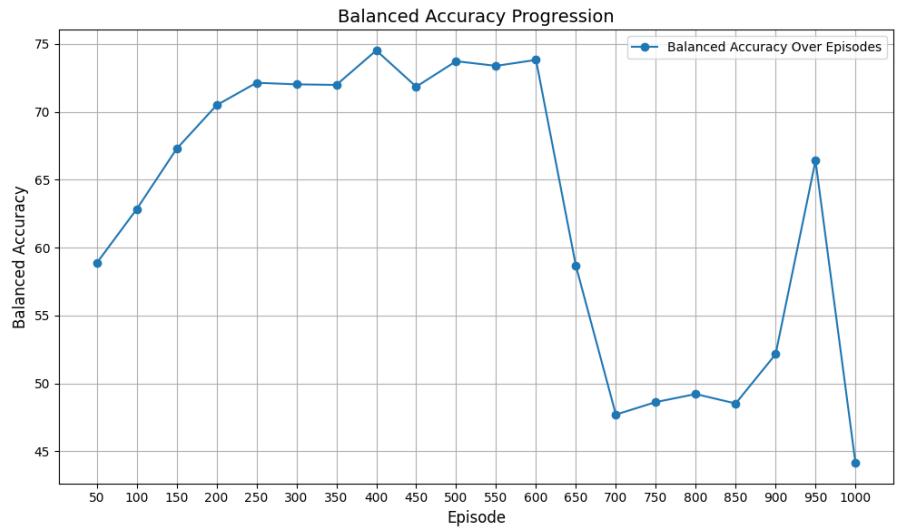
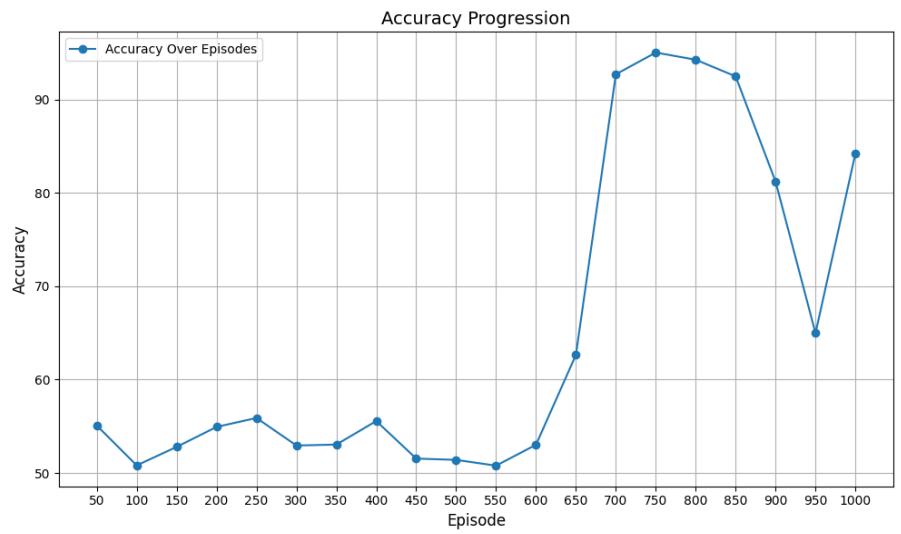
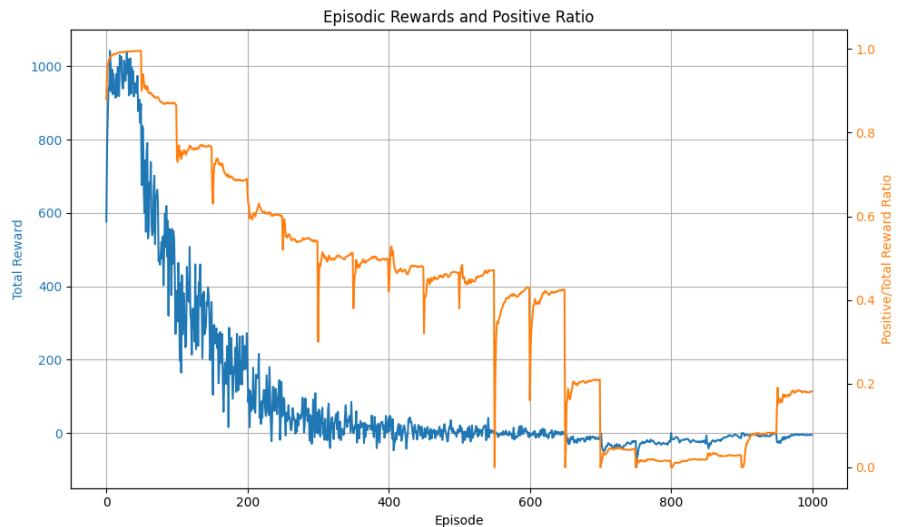


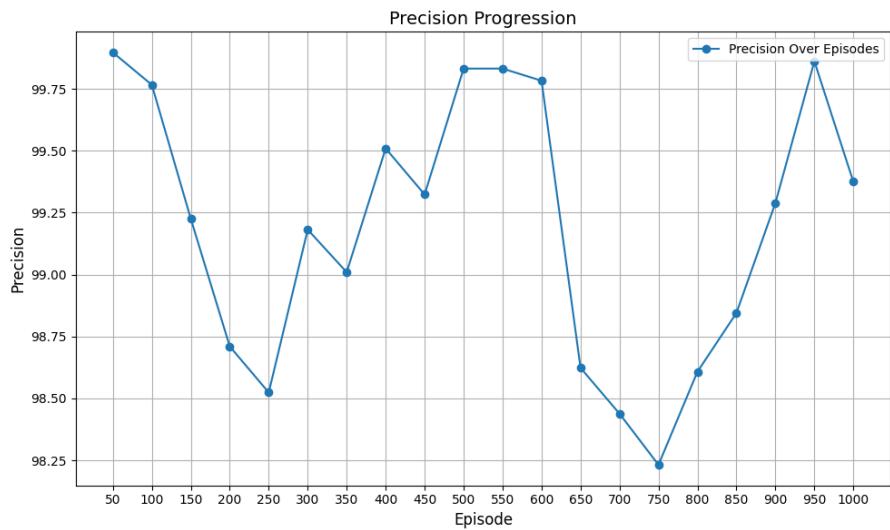
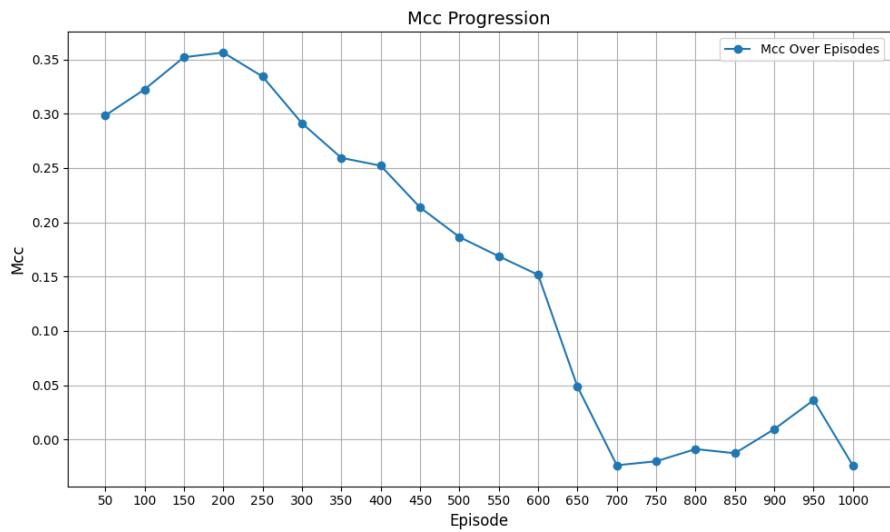
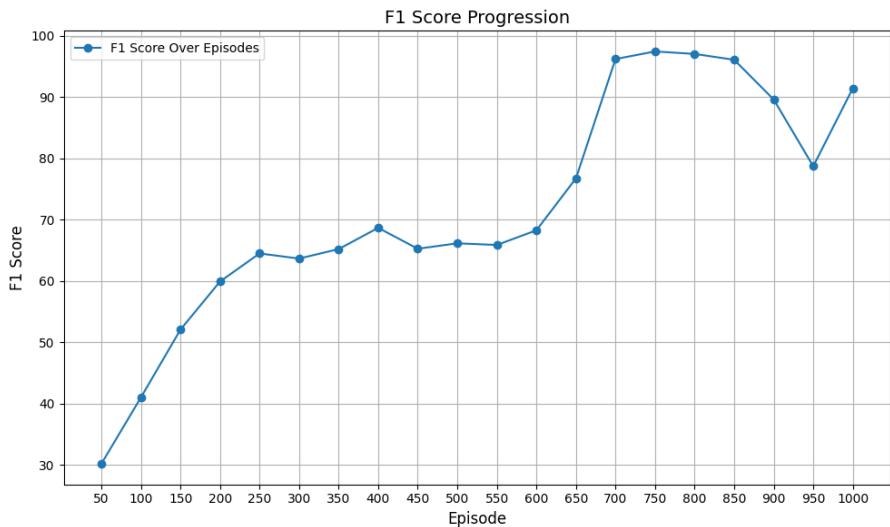


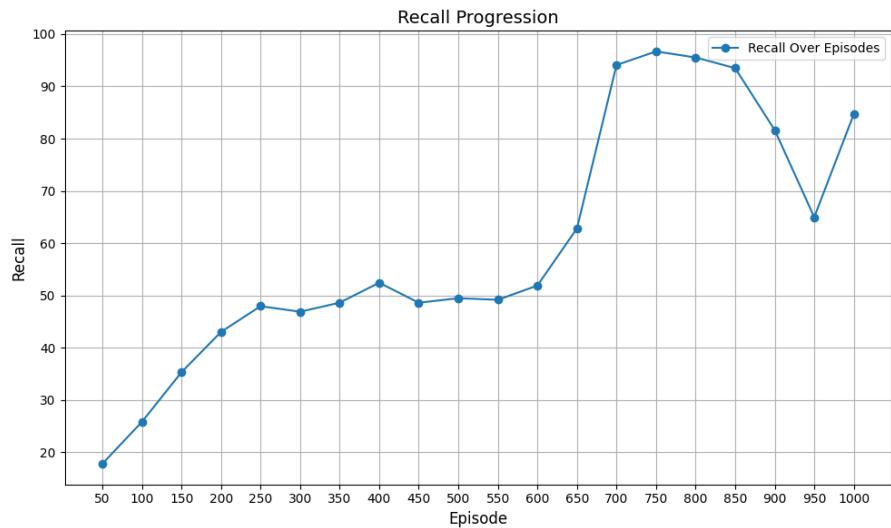
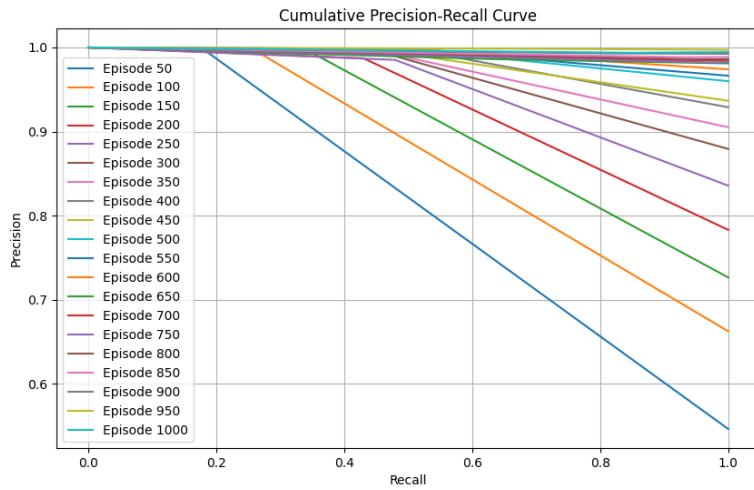


3.6.2 GAT/DQN con learning rate alto, normalizzazione delle reward ed imbalance factor

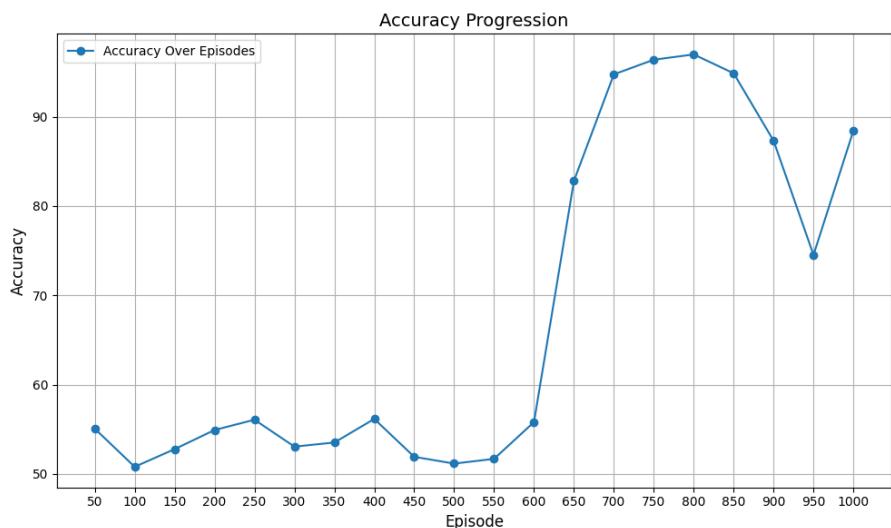
Questo test è stato fatto per mostrare come un learning rate maggiore di 0.001 per l'agente impatti fortemente sulle sue perfomance.

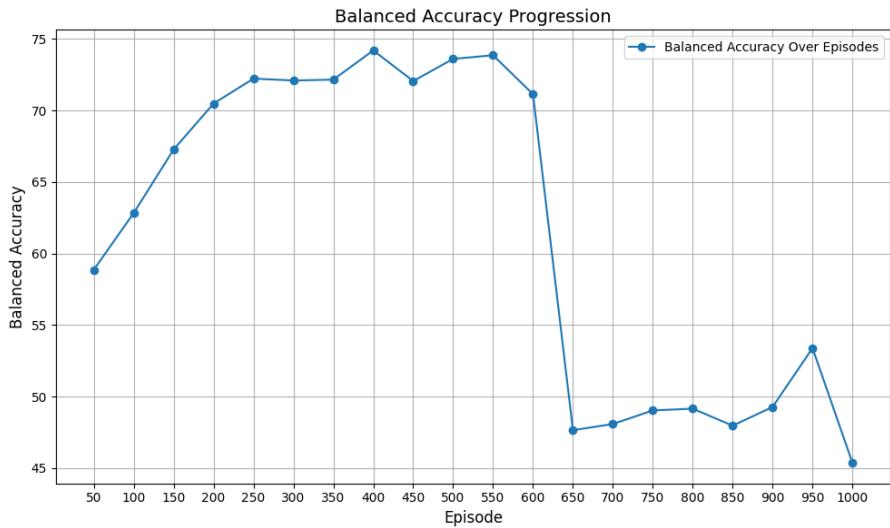
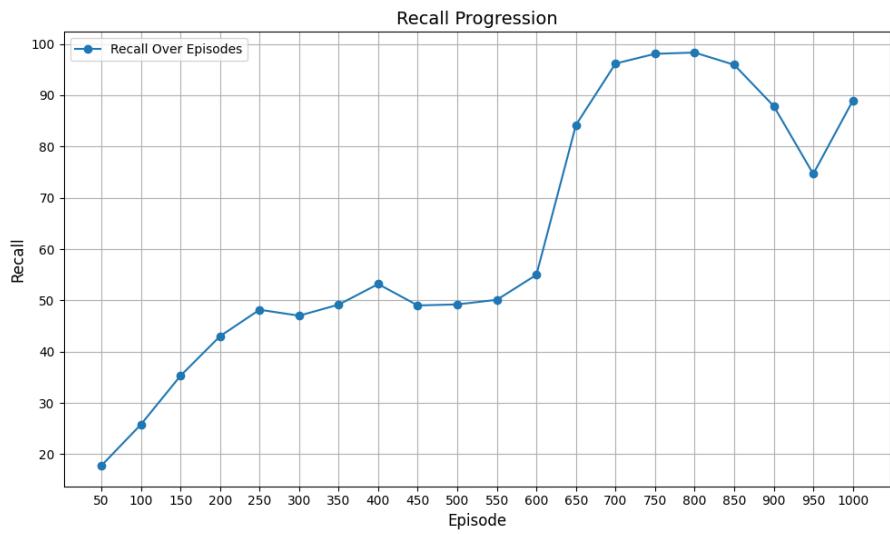
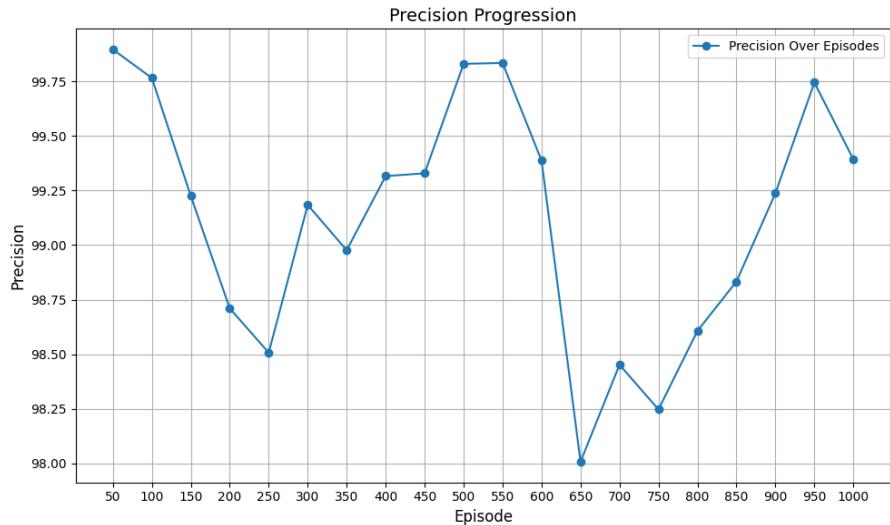


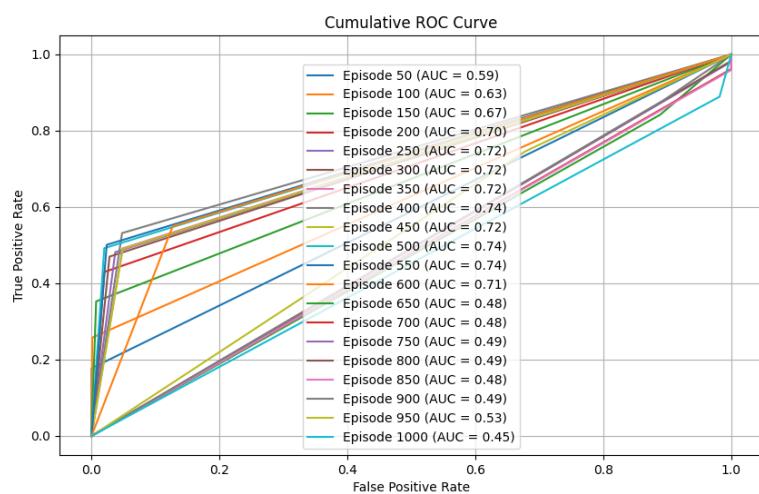
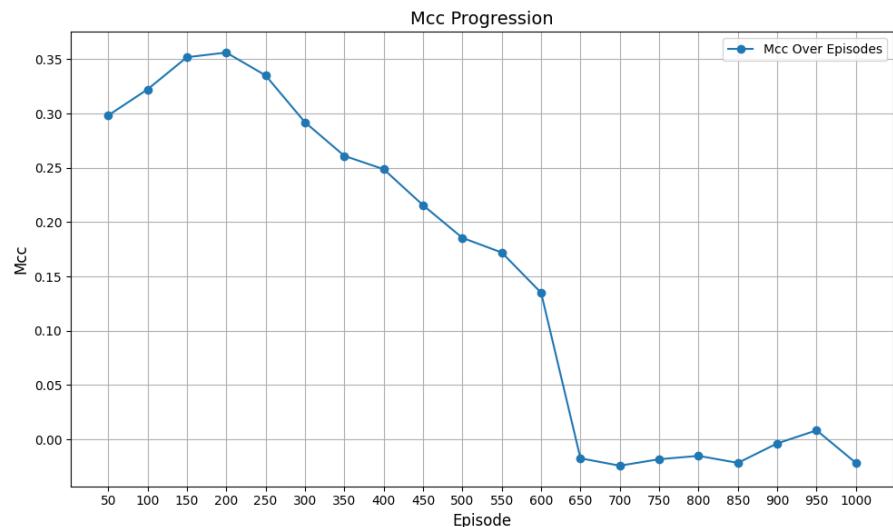
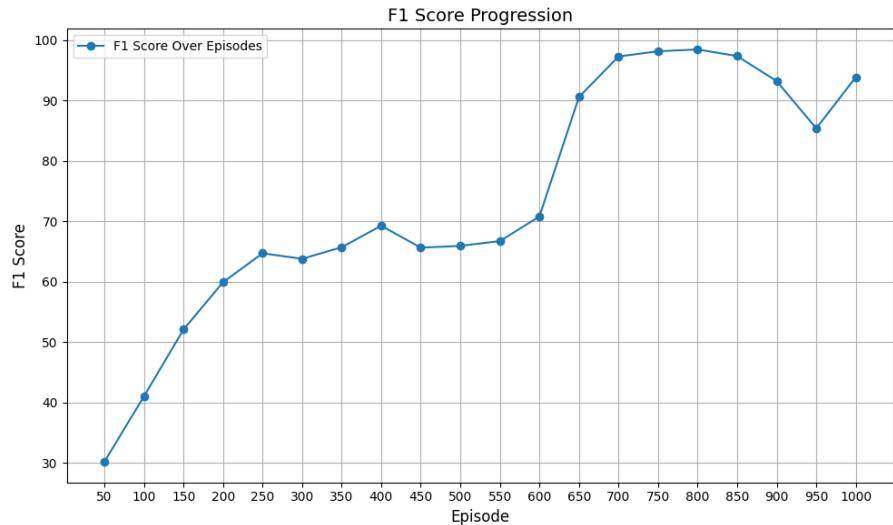


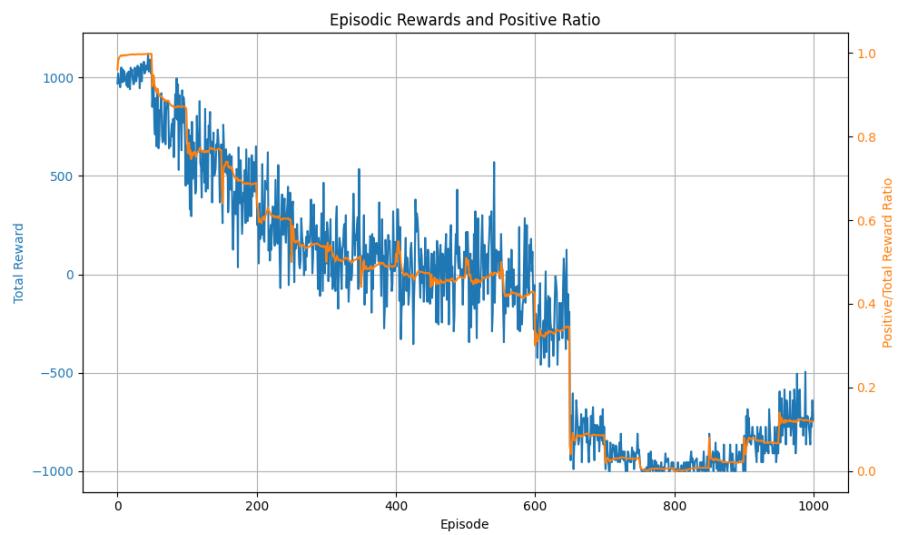
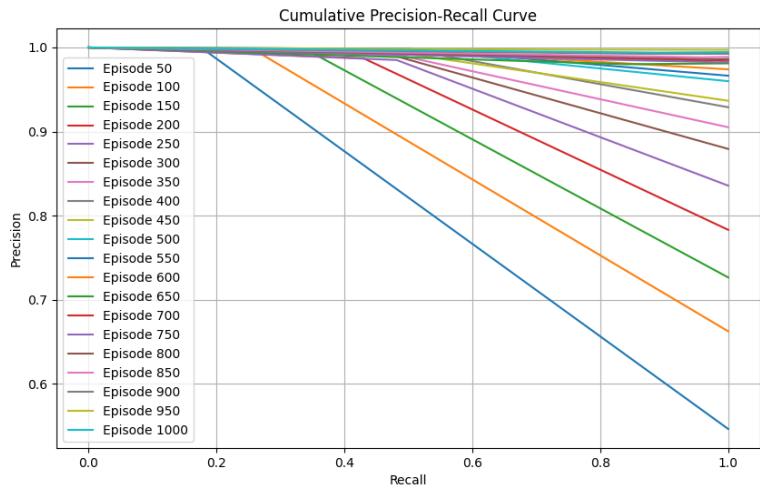


3.6.3 GAT/DQN con normalizzazione delle reward ma senza imbalance factor

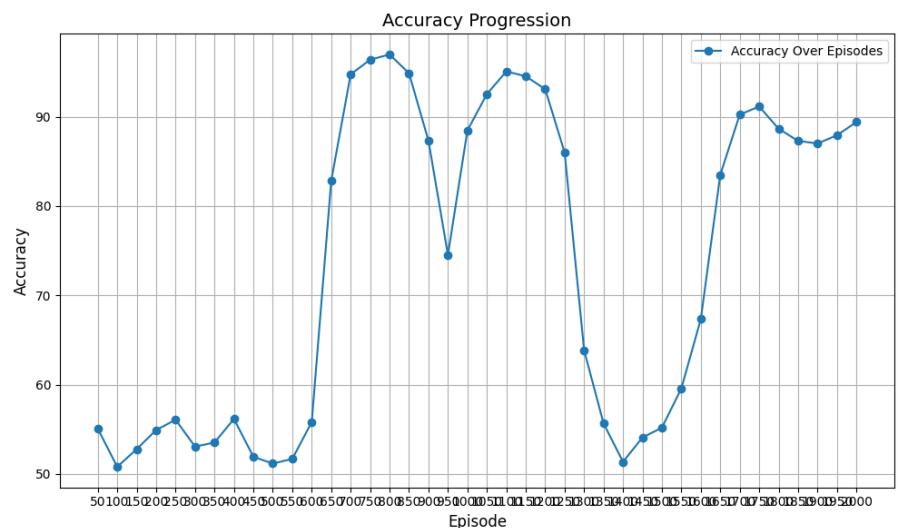


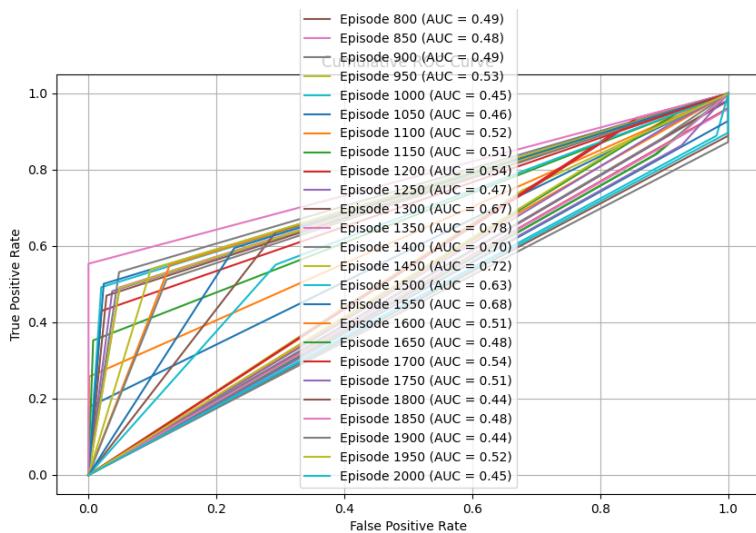
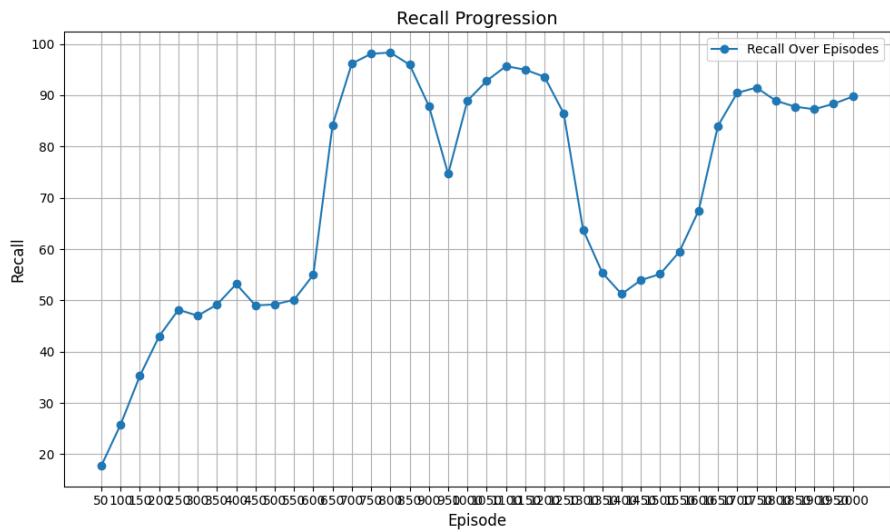
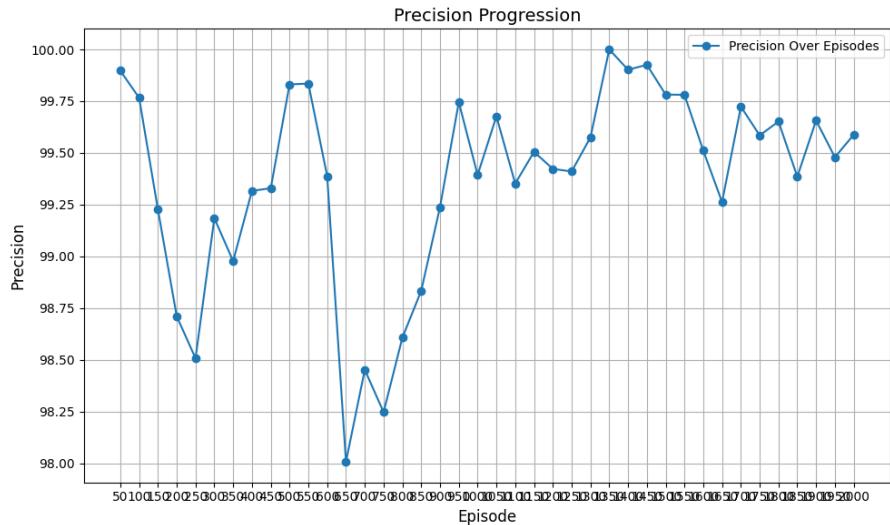


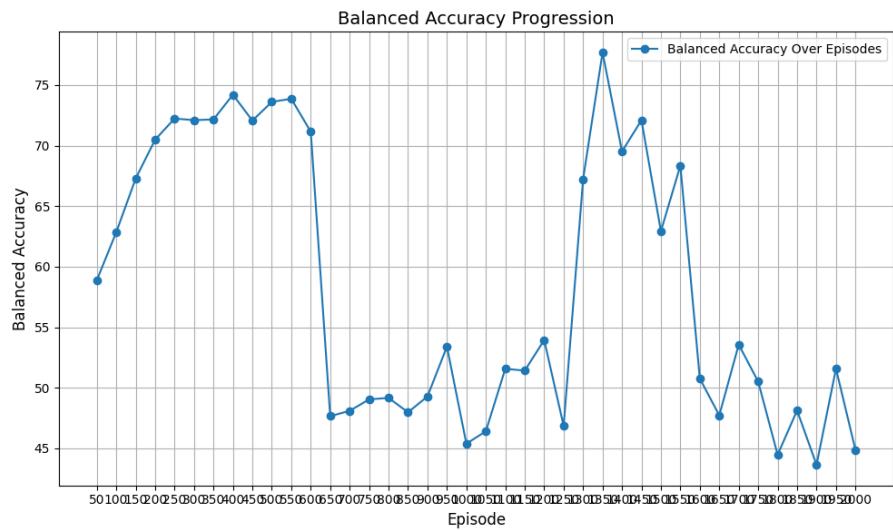
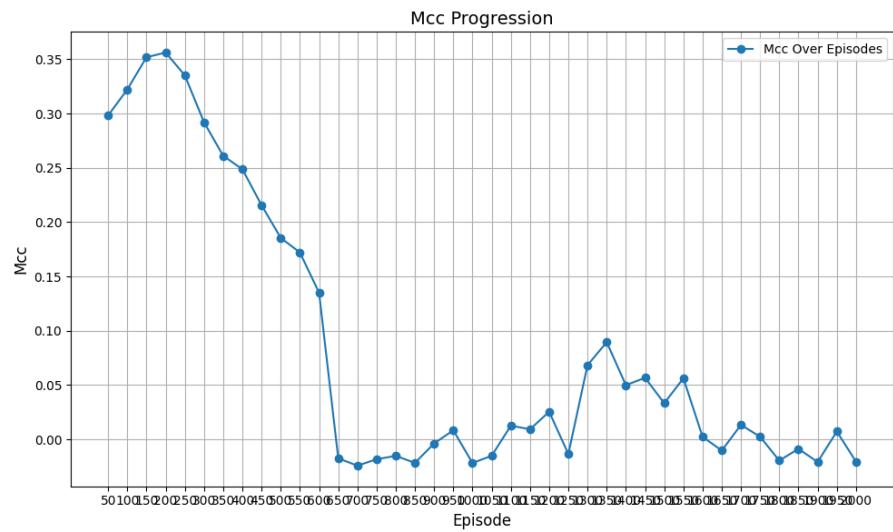
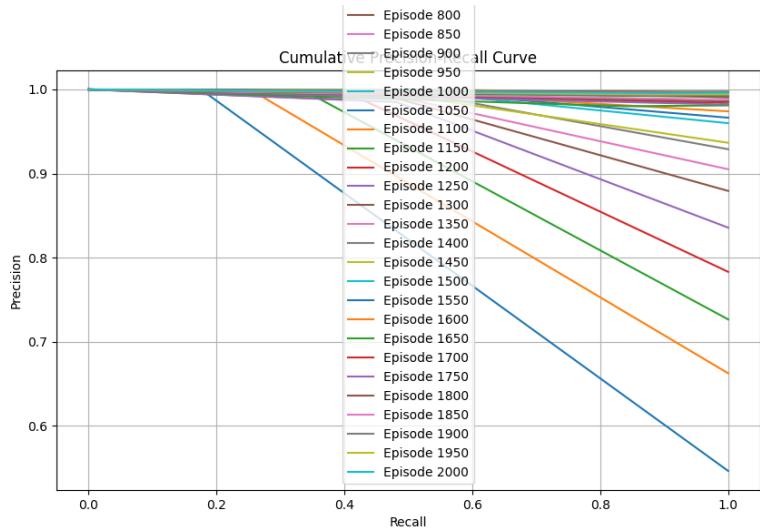


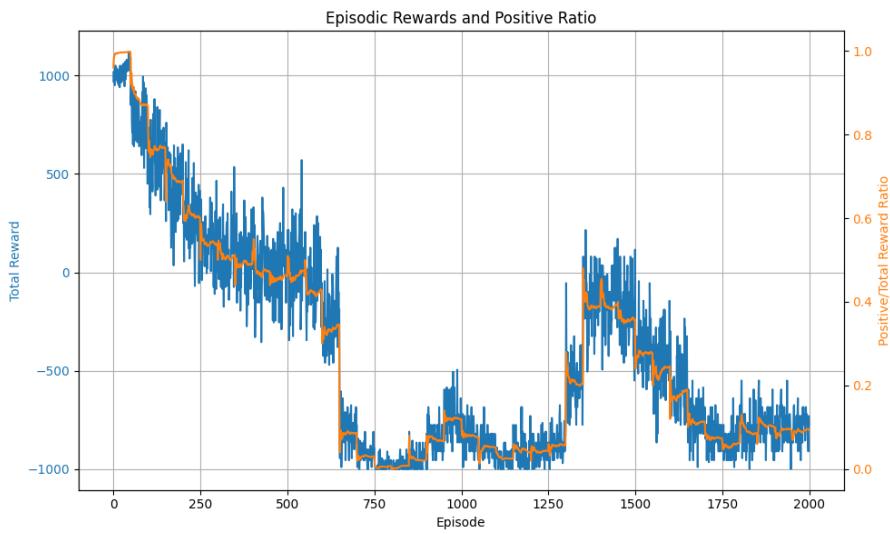
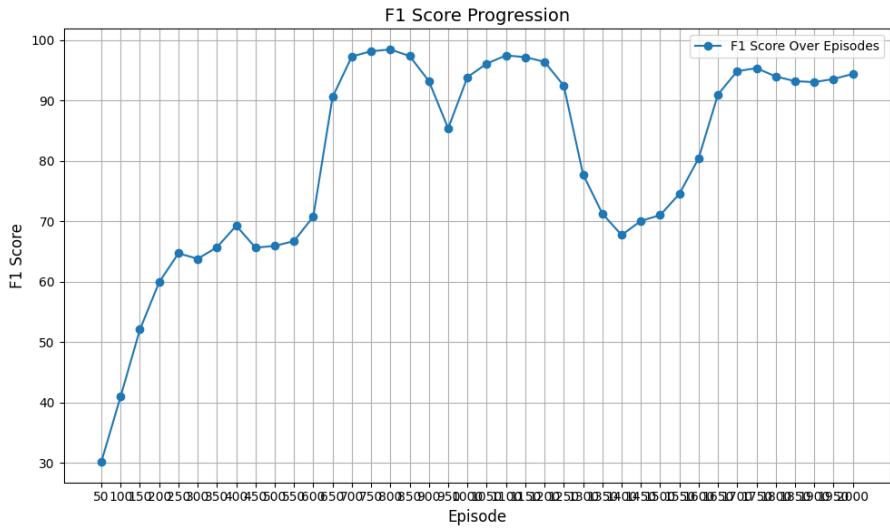


3.6.4 GAT/DDQN con normalizzazione ma senza imbalance factor e con 2000 episodi

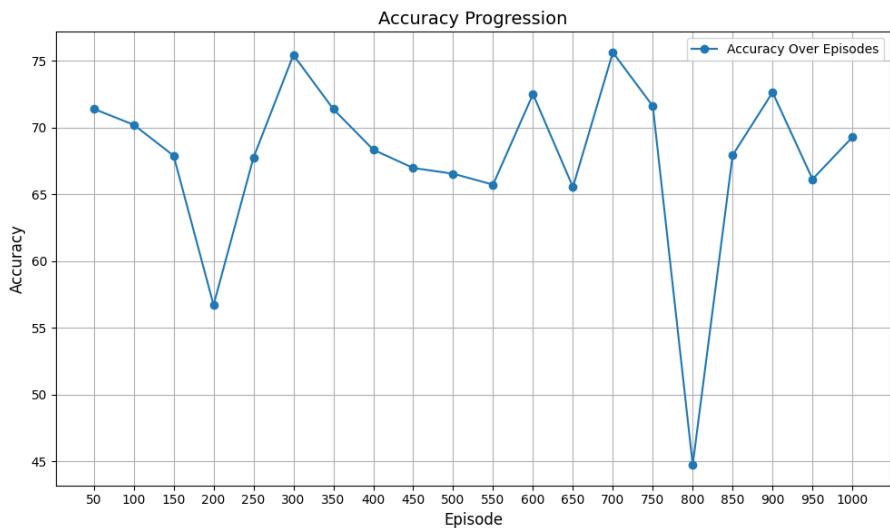


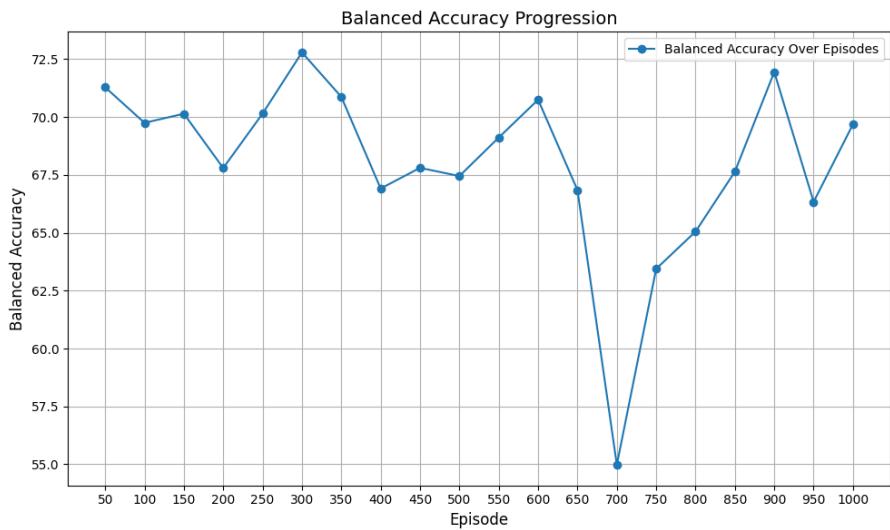
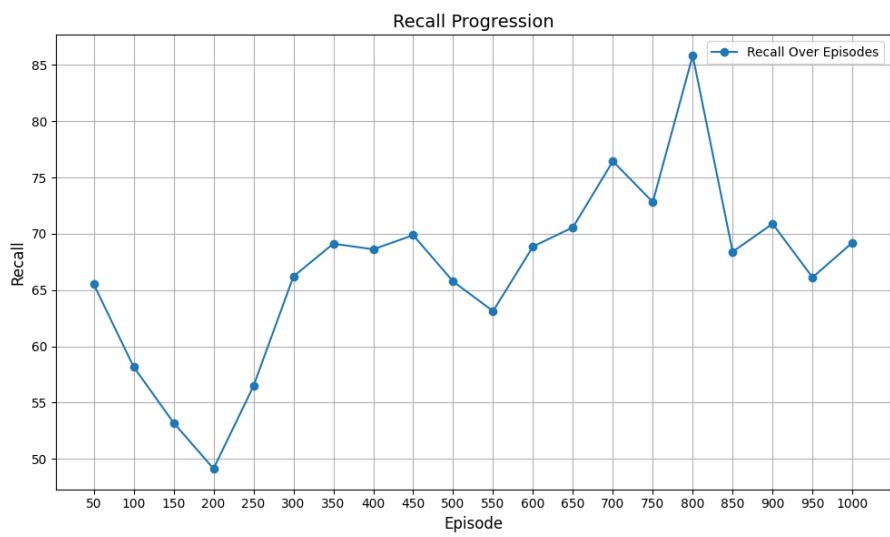
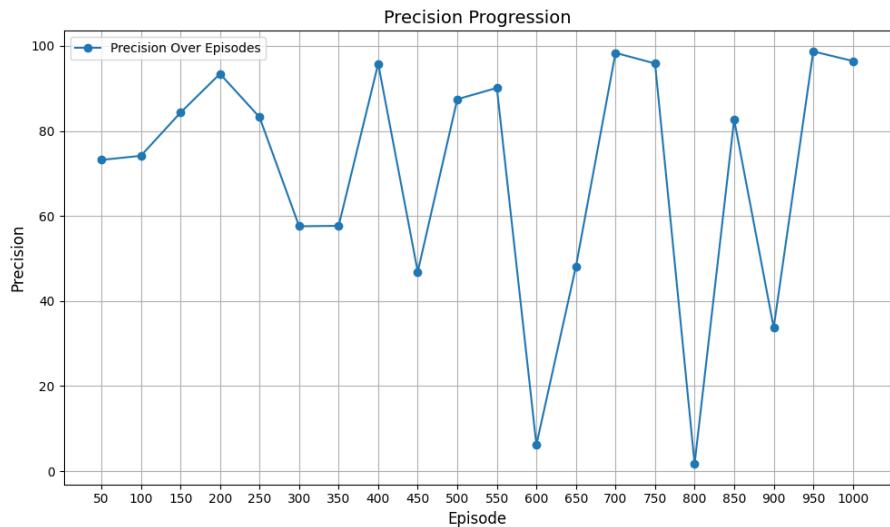


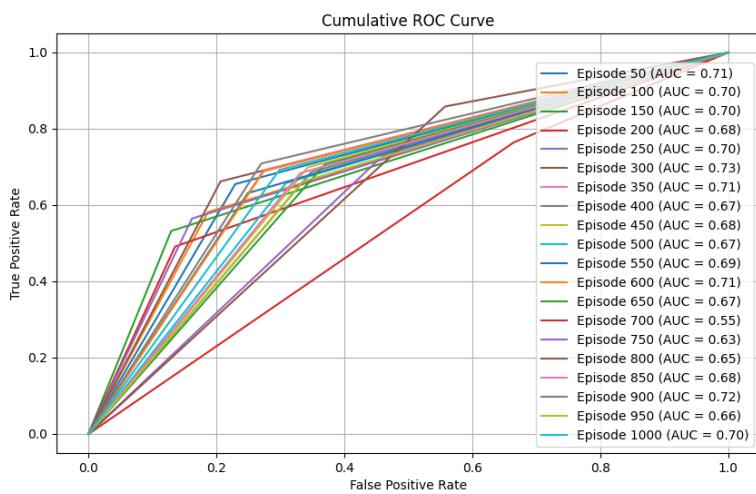
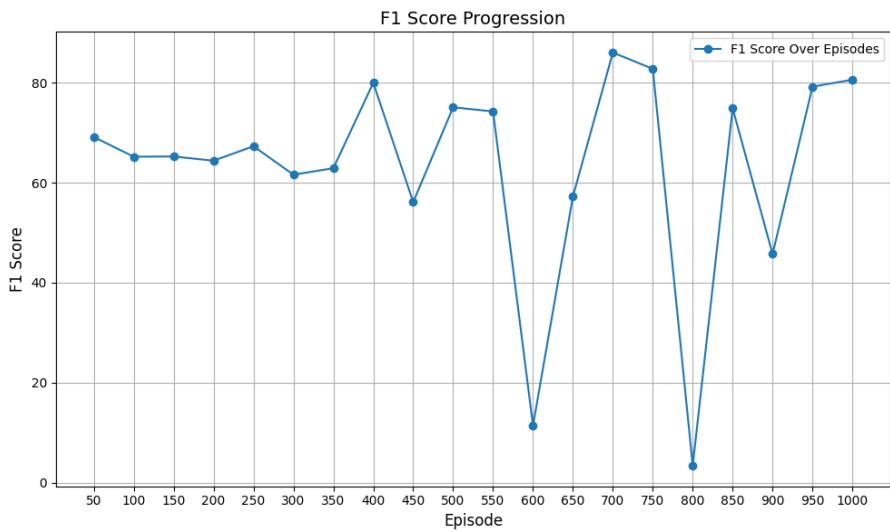
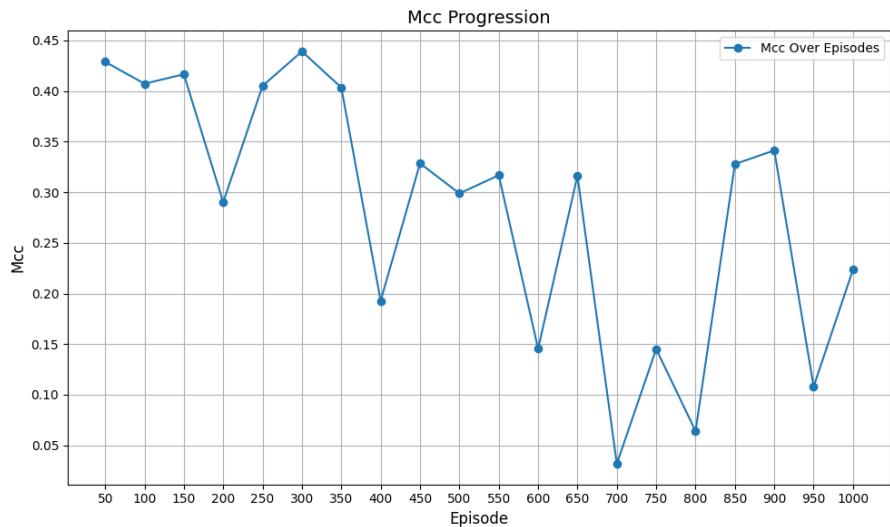


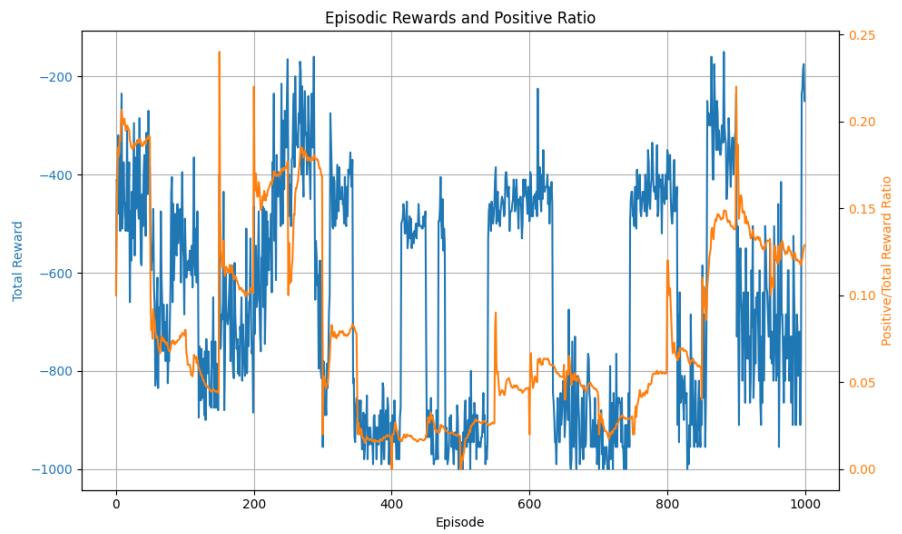
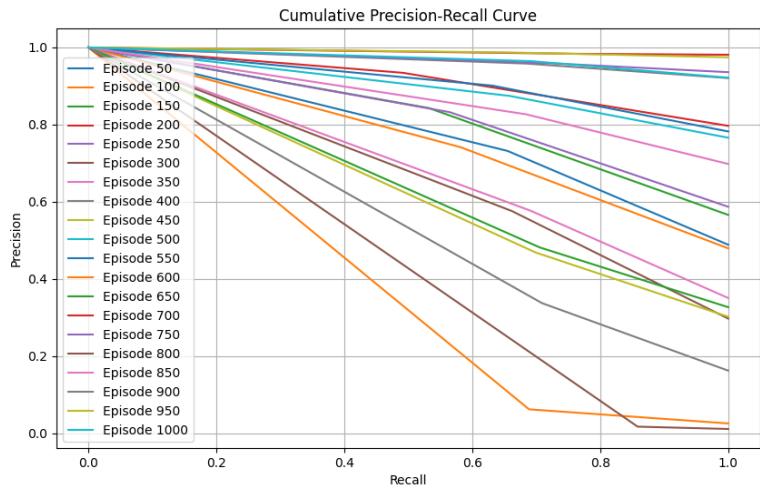


3.6.5 GraphSage/DDQN con normalizzazione ma senza imbalance factor

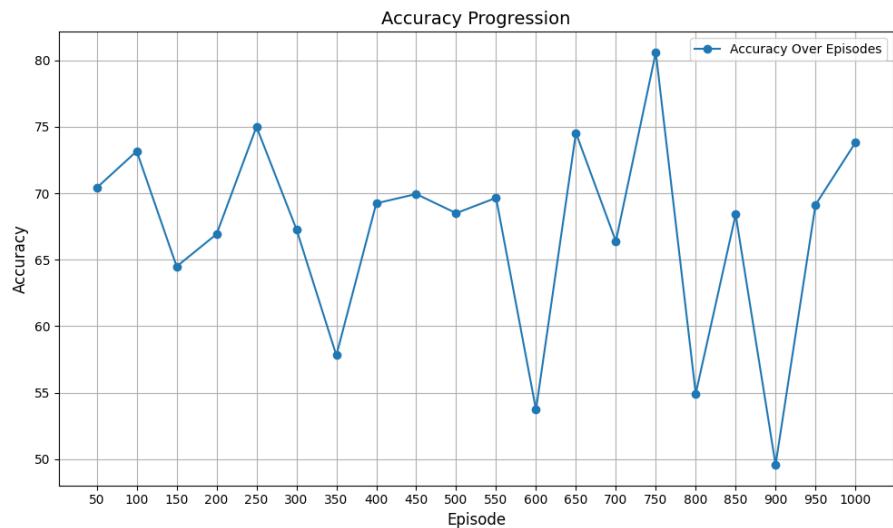


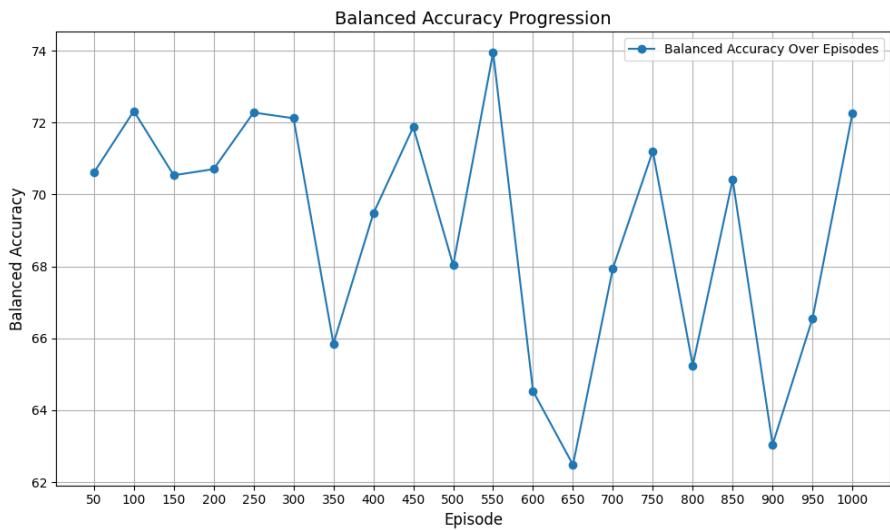
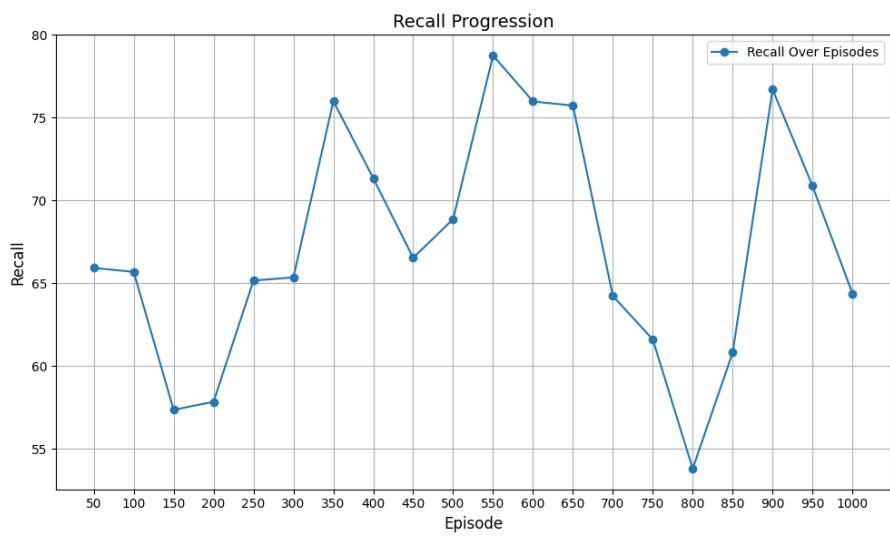
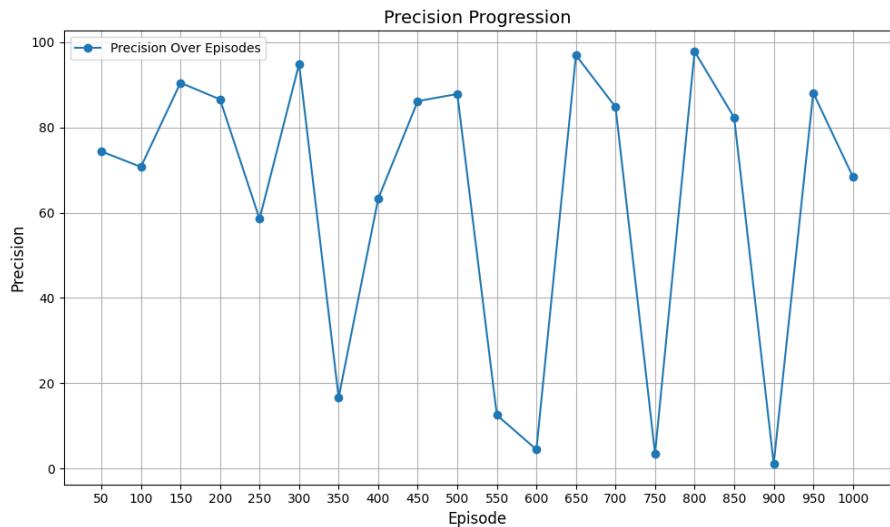


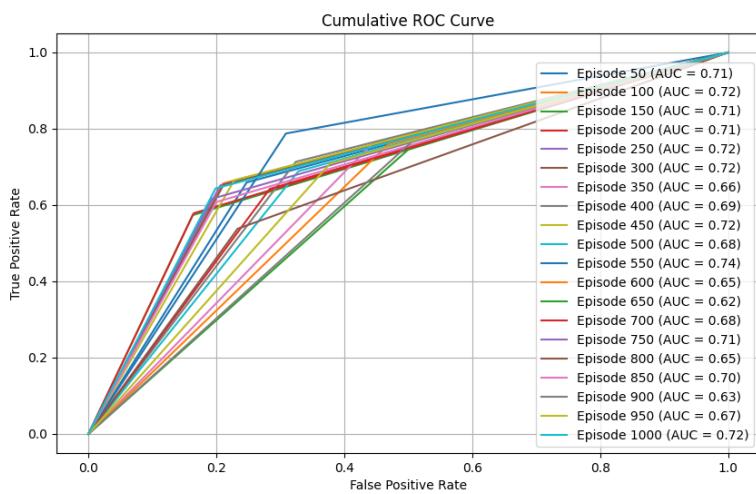
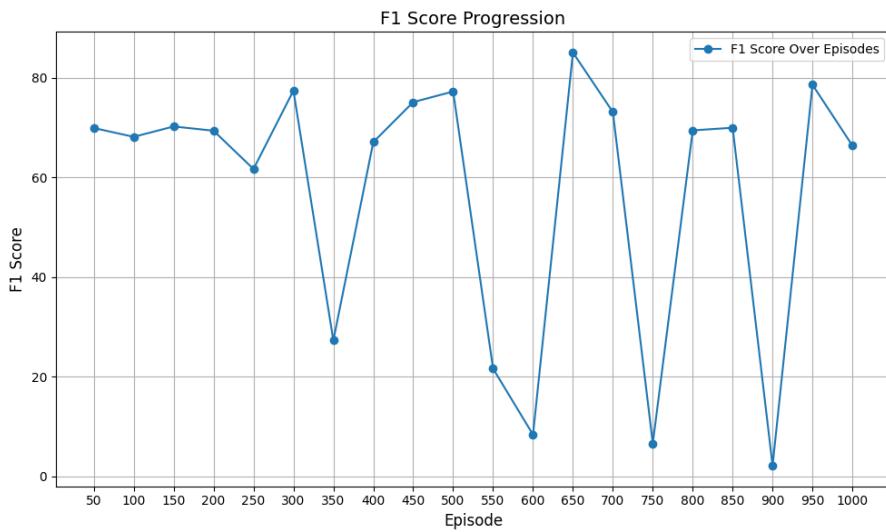
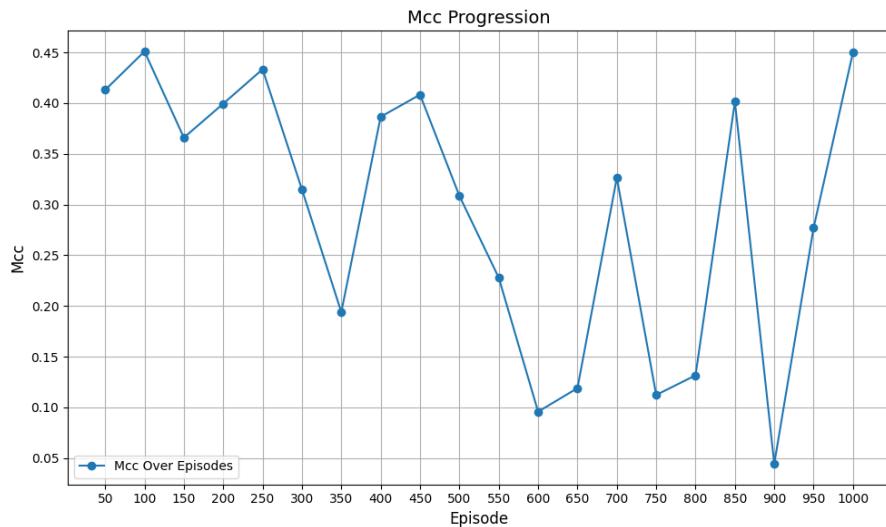


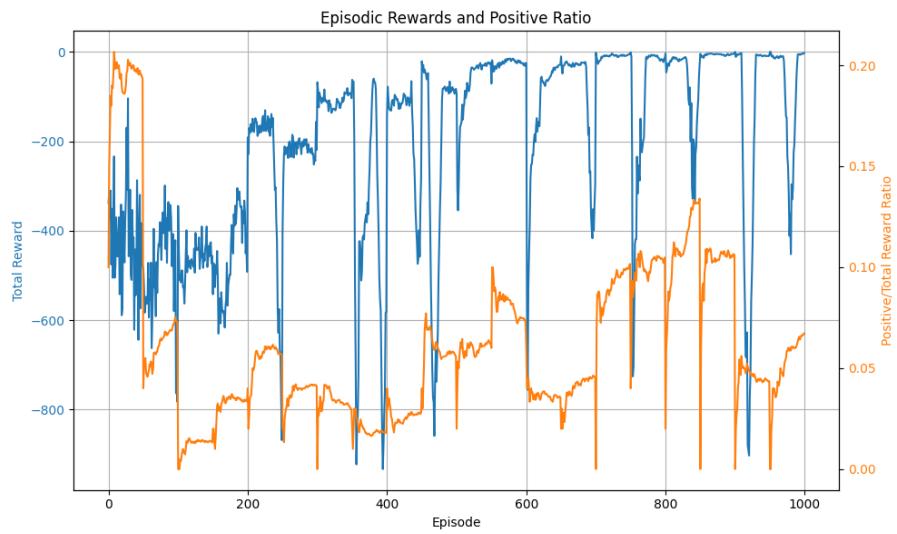
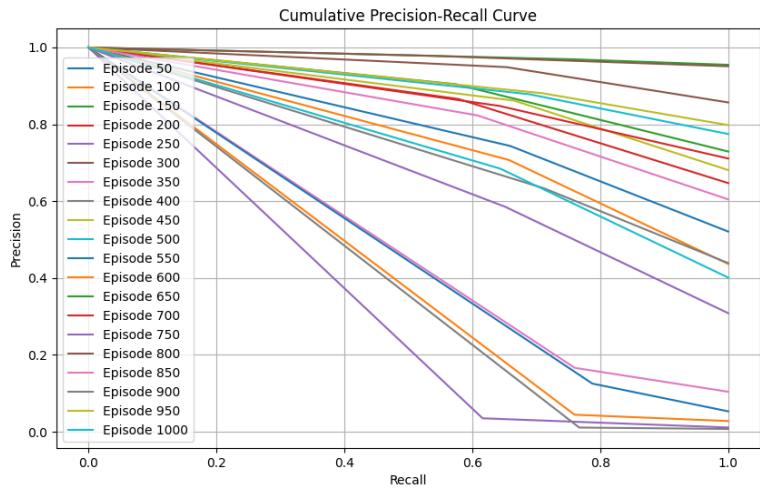


3.6.6 GraphSage/DDQN con normalizzazione e con imbalance factor



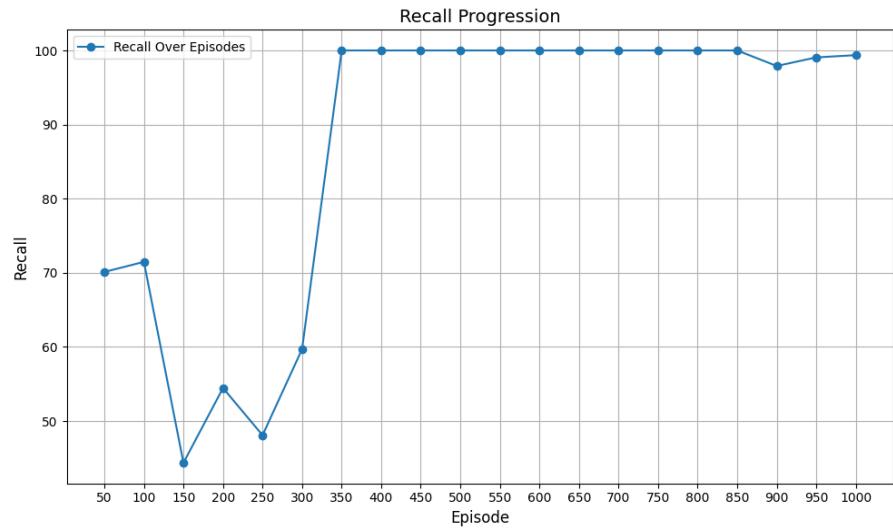
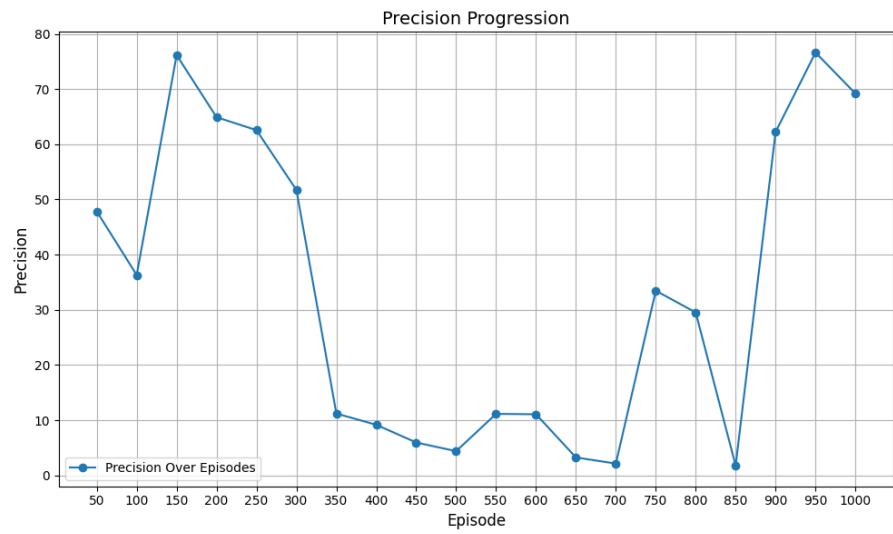
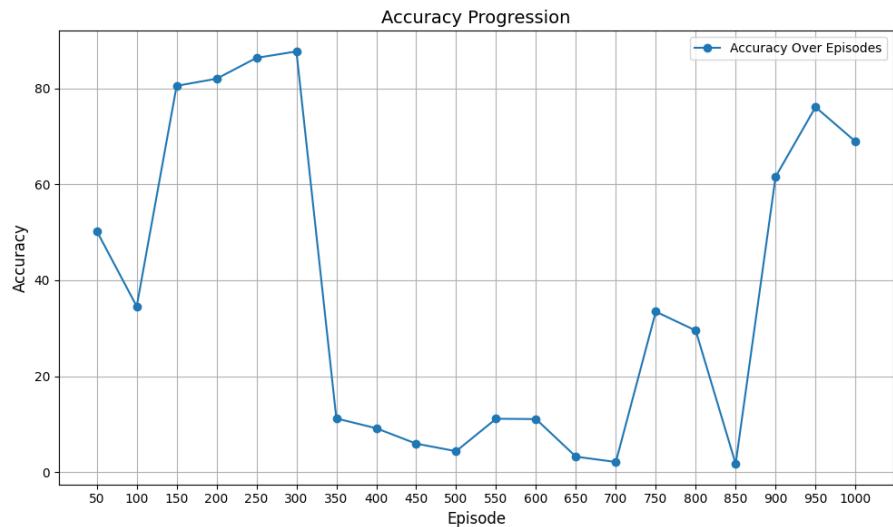


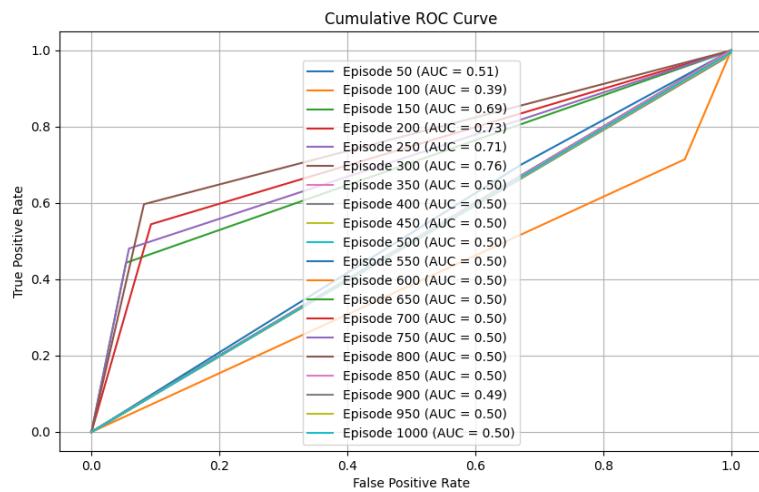
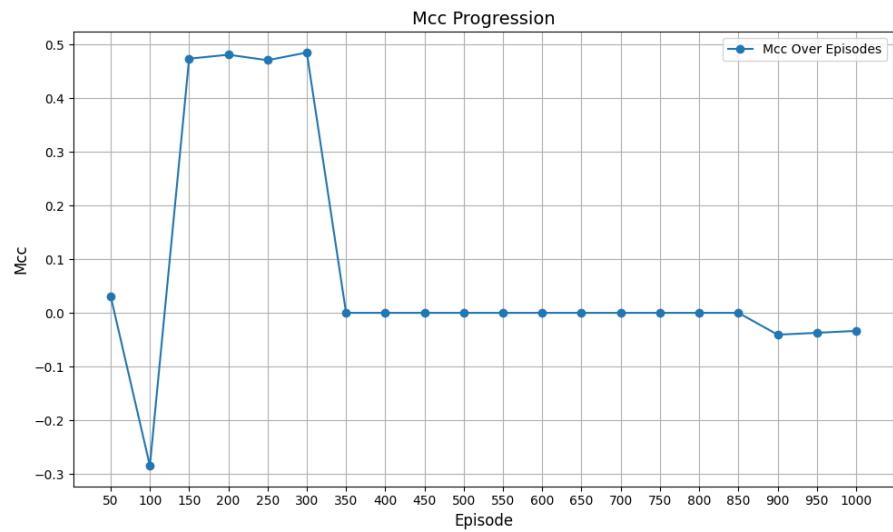
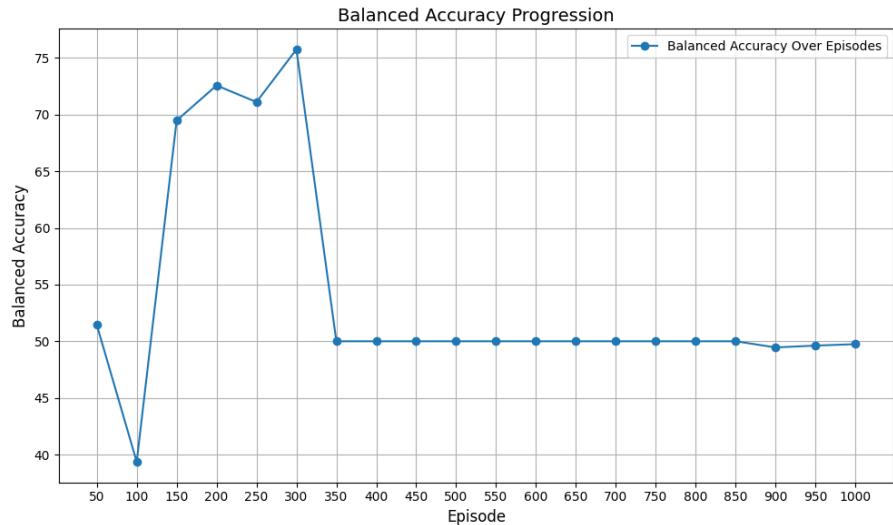


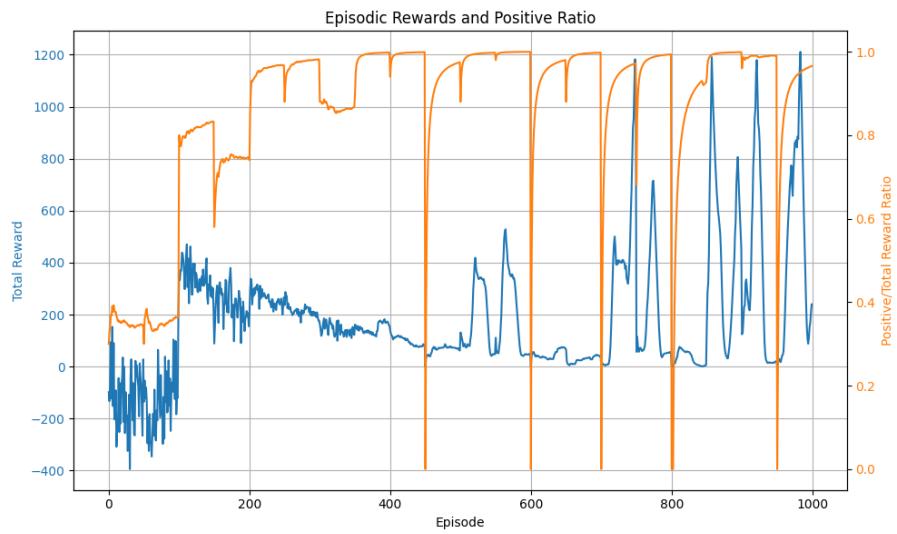
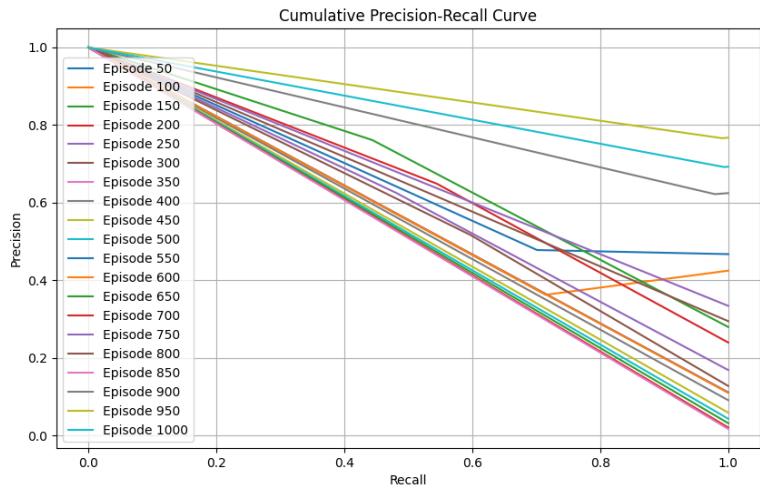


3.6.7 GCN-multiclasse/DDQN

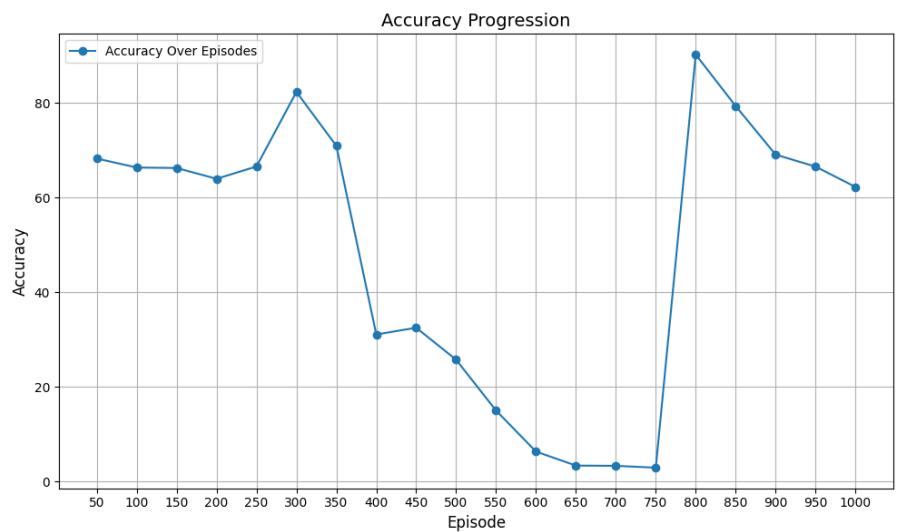
Questo test è stato fatto per mostrare come il caso in cui la GNN classifichi in maniera non binaria (dove quindi ogni tipo di attacco viene distinto dall'altro) la renda poco performante considerando l'agente che ha un costantemente alto Positive/Total ratio.

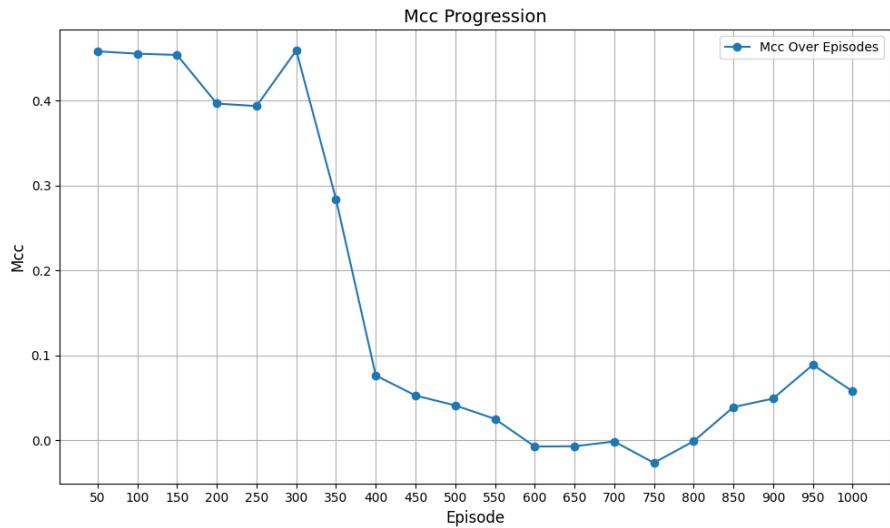
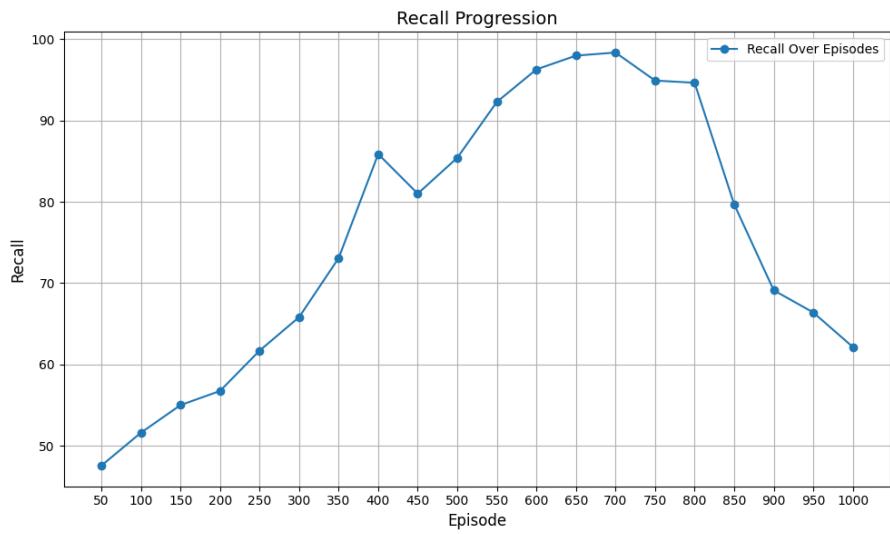
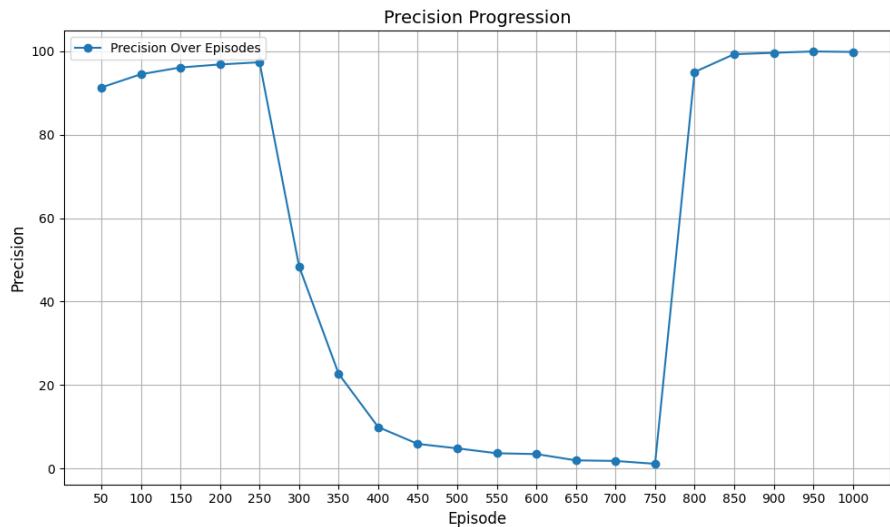


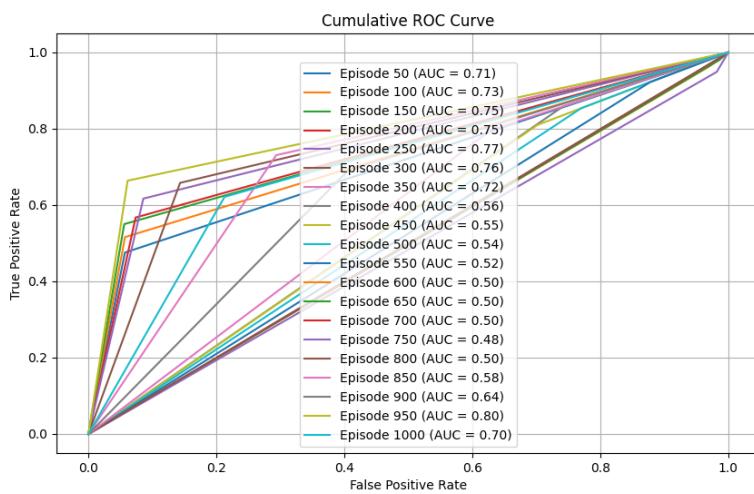
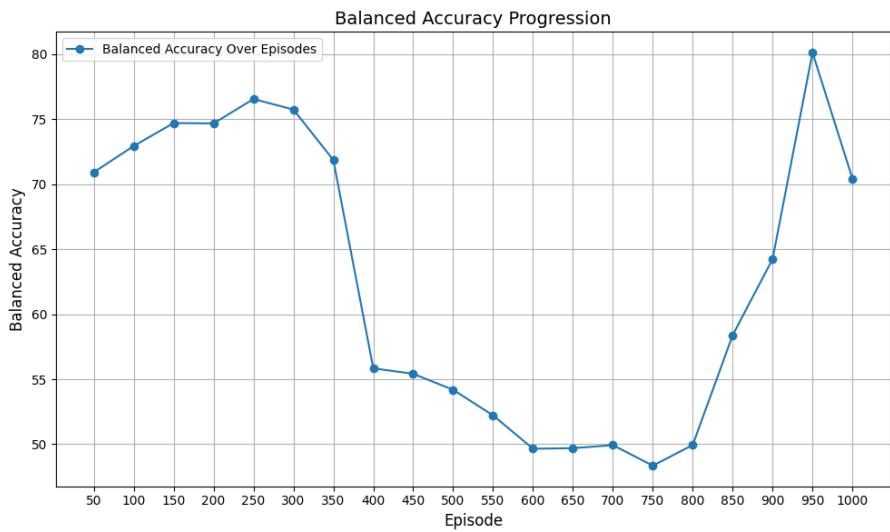
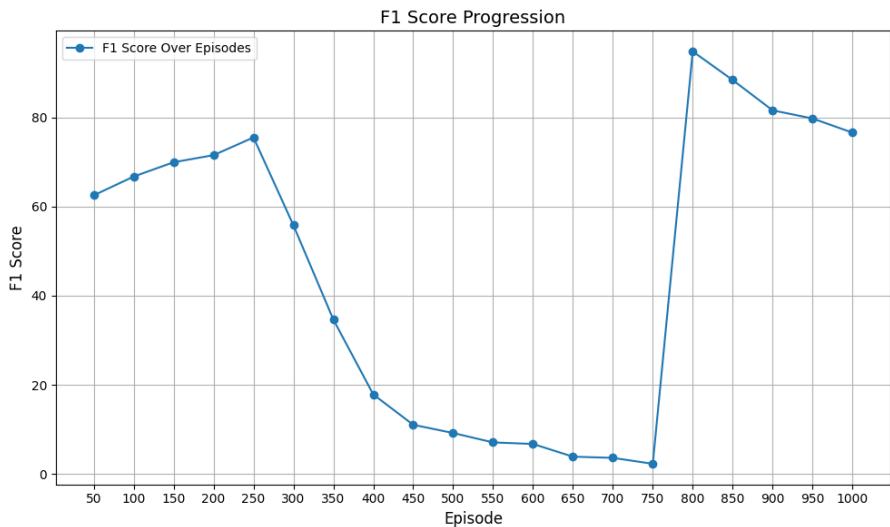


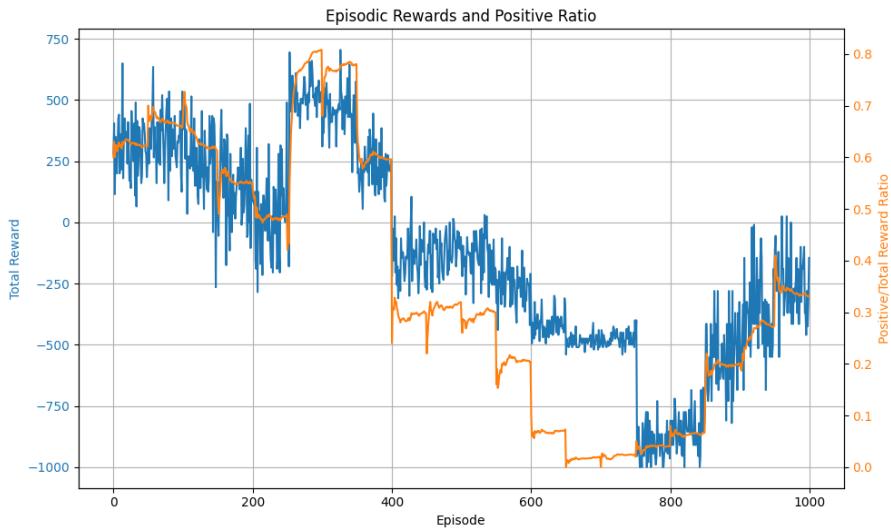
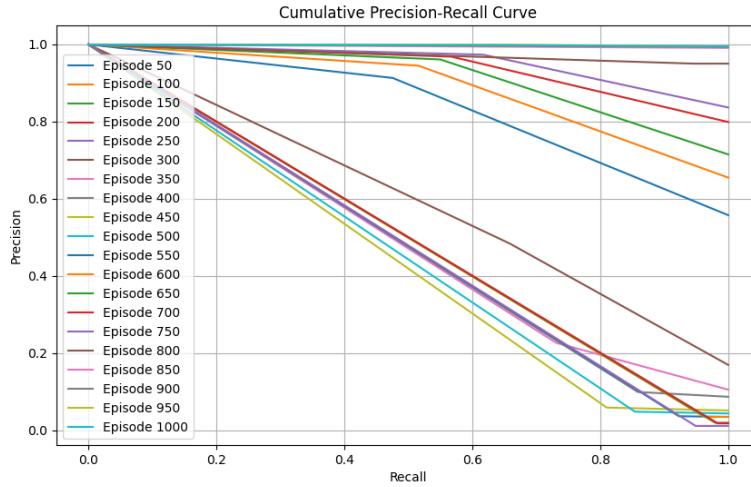


3.6.8 GCN/SARSA con normalizzazione ma senza imbalance factor









4 Conclusioni

Questo progetto si prefiggeva di utilizzare nuove tecnologie per creare un agente ed un IDS che riuscissero a sfidarsi a vicenda. Da questa sfida poi ognuno avrebbe imparato l'uno dall'altro. Utilizzando tecnologie come RL e GNN è stato possibile creare una versione funzionante con performance accettabili ma non ottime. La simbiosi tra l'agente e l'IDS è presente ma non sempre ed a volte c'è uno sbilanciamento/confusione nella loro iterazione.

Abbiamo visto che l'agente che sembra performare meglio è DDQN mentre la GNN che risulta adattarsi meglio agli attacchi dell'agente risulta essere la GCN. Inoltre la normalizzazione delle reward sembra essere una buona aggiunta mentre il calcolo del reward imbalance factor sembra essere un'aggiunta eccessiva e poco utile.

Infine, considerando la natura "sperimentale" del progetto, siamo soddisfatti del nostro tentativo.

Riferimenti bibliografici

[1] Paper per GNN:

- [2] A. Venturi, D. Stabili, and M. Marchetti, "Problem space structural adversarial attacks for Network Intrusion Detection Systems based on Graph Neural Networks," University of Modena and Reggio Emilia, University of Bologna, arXiv:2403.11830, Apr. 2024.
- [3] Z. Sun, A. M. H. Teixeira, and S. Toor, "GNN-IDS: Graph Neural Network based Intrusion Detection System," in Proc. 19th Int. Conf. on Availability, Reliability and Security (ARES 2024), Vienna, Austria, Jul.–Aug. 2024, pp. 1–12.
- [4] D.-H. Tran and M. Park, "FN-GNN: A Novel Graph Embedding Approach for Enhancing Graph Neural Networks in Network Intrusion Detection Systems," Appl. Sci., vol. 14, no. 6932, pp. 1–23, Aug. 2024, doi:10.3390/app14166932.
- [5] T. Bilot, N. El Madhoun, K. Al Agha, and A. Zouaoui, "Graph Neural Networks for Intrusion Detection: A Survey," IEEE Access, vol. 11, pp. 1–25, May 2023, doi:10.1109/ACCESS.2023.3275789.
- [6] D. Pujol-Perich, J. Suárez-Varela, A. Cabellos-Aparicio, and P. Barlet-Ros, "Unveiling the Potential of Graph Neural Networks for Robust Intrusion Detection," in Proc. Workshop on AI in Networks and Distributed Systems (WAIN), Milan, Italy, Mar. 2024, pp. 1–10.
- [7] E. Caville, W. W. Lo, S. Layeghy, and M. Portmann, "Anomal-E: A Self-Supervised Network Intrusion Detection System Based on Graph Neural Networks," Knowl.-Based Syst., vol. 258, pp. 1–12, Oct. 2022, doi:10.1016/j.knosys.2022.110030.
- [8] R. Xu, G. Wu, W. Wang, X. Gao, A. He, and Z. Zhang, "Applying Self-Supervised Learning to Network Intrusion Detection for Network Flows with Graph Neural Networks," Comput. Netw., vol. 248, pp. 1–15, May 2024, doi:10.1016/j.comnet.2024.110495.
- [9] M. Zhong, M. Lin, C. Zhang, and Z. Xu, "A Survey on Graph Neural Networks for Intrusion Detection Systems: Methods, Trends, and Challenges," Comput. Secur., vol. 141, pp. 1–19, Mar. 2024, doi:10.1016/j.cose.2024.103821.
- [10] P. Deng and Y. Huang, "Edge-Featured Multi-Hop Attention Graph Neural Network for Intrusion Detection System," Comput. Secur., vol. 148, pp. 1–15, Sep. 2025, doi:10.1016/j.cose.2024.104132.

[11] Paper per RL:

- [12] M. Lopez-Martin, B. Carro, and A. Sanchez-Esguevillas, "Application of deep reinforcement learning to intrusion detection for supervised problems," *Expert Systems With Applications*, vol. 141, pp. 112963, 2020, doi:10.1016/j.eswa.2019.112963.

- [13] T. T. Nguyen and V. J. Reddi, "Deep Reinforcement Learning for Cyber Security," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 8, pp. 3779–3794, 2023, doi:10.1109/TNNLS.2021.3121870.
- [14] M. Mouyart, G. Medeiros Machado, and J.-Y. Jun, "A Multi-Agent Intrusion Detection System Optimized by a Deep Reinforcement Learning Approach with a Dataset Enlarged Using a Generative Model to Reduce the Bias Effect," *Journal of Sensor and Actuator Networks*, vol. 12, no. 68, pp. 1–29, 2023, doi:10.3390/jsan12050068.
- [15] W. Yang, A. Acuto, Y. Zhou, and D. Wojtczak, "A Survey for Deep Reinforcement Learning Based Network Intrusion Detection," *arXiv preprint*, vol. 2410.07612, Sep. 2024.
- [16] G. Caminero, M. Lopez-Martin, and B. Carro, "Adversarial environment reinforcement learning algorithm for intrusion detection," *Computer Networks*, vol. 159, pp. 96–109, 2019, doi:10.1016/j.comnet.2019.05.013.
- [17] K. Sethi, E. S. Rupesh, R. Kumar, P. Bera, and Y. V. Madhav, "A context-aware robust intrusion detection system: a reinforcement learning-based approach," *International Journal of Information Security*, vol. 19, pp. 657–678, 2020, doi:10.1007/s10207-019-00482-7.
- [18] F. Louati, F. B. Ktata, and I. Amous, "Enhancing Intrusion Detection Systems with Reinforcement Learning: A Comprehensive Survey of RL-based Approaches and Techniques," *SN Computer Science*, vol. 5, no. 665, 2024, doi:10.1007/s42979-024-03001-1.
- [19] A. Tellache, A. Mokhtari, A. A. Korba, and Y. Ghamri-Doudane, "Multi-agent Reinforcement Learning-based Network Intrusion Detection System," *arXiv preprint*, vol. 2407.05766, Jul. 2024.
- [20] Yuankai Luo, Lei Shi, Xiao-Ming Wu "Classic GNNs are Strong Baselines: Reassessing GNNs for Node Classification",*arXiv preprint*, Jan. 2024