UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Miglioramento delle performance di addestramento di un Vision Transformer attraverso approcci di Reinforcement Learning

Improving the training performance of a Vision Transformer through Reinforcement Learning approaches

Relatore

Prof. Domenico Ursino

Candidato

Davide Traini

Correlatori

Dott. Francesco Cauteruccio

Dott. Luca Virgili

ANNO ACCADEMICO 2022-2023

Sommario

Negli ultimi anni si è assistito ad un'enorme evoluzione dell'Intelligenza Artificiale grazie all'introduzione dei transformer. Tale rete neurale è nata nell'ambito del Natural Language Processing ed è stata, poi, utilizzata anche nella Computer Vision, con il nome di Vision Transformer (ViT). Una delle principali problematiche di questa architettura riguarda la complessità temporale, la quale cresce quadraticamente rispetto al numero di patch in cui viene divisa l'immagine in input. In questo progetto di tesi si vuole proporre un approccio basato sul Deep Q-Learning capace di diminuire la complessità temporale dei ViT mantenendone inalterate le prestazioni. Il framework da noi proposto fa uso di un agente che si interfaccia con un environment, il quale fornisce l'attention score associato alle immagini in input. L'agente, una volta osservato lo stato corrente, restituisce una lista di patch che vengono utilizzate per l'allenamento del ViT. Al termine dell'allenamento, l'agente riceve un reward basato su una combinazione di training loss e numero di patch scelte.

Keyword: Transformer, Vision Transformer, Computer Vision, Reinforcement Learning, Deep Q-learning, Ottimizzazione, CIFAR10, Python, Pytorch.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Negli ultimi anni, grazie all'utilizzo delle reti neurali profonde, l'Intelligenza Artificiale ha vissuto una straordinaria espansione, portando a notevoli progressi in diversi ambiti, tra cui troviamo il Natural Language Processing (NLP) e la Computer Vision (CV). In particolare, l'anno di svolta è il 2018, anno nel quale venne introdotta l'architettura dei *transformer* per l'NLP. Tale rete neurale è costituita da blocchi *attention*, i quali riescono a individuare le relazioni semantiche tra le parole, indipendentemente dalla loro posizione all'interno della frase. Ciò risultava essere un grande vantaggio rispetto alle architetture precedenti, basate su Recurrent Neural Network e Convolutional Neural Network, le cui prestazioni degradavano all'aumentare della lunghezza della frase. I transformer si sono da subito rivelati superiori rispetto agli altri approcci e sono stati utilizzati in moltissime applicazioni. Nel 2018 Google ha sviluppato BERT (Bidirectional Encoder Representations from Transformers), un modello di Machine Learning basato sui transformer. Già nel 2020 tutte le richieste in lingua inglese venivano analizzate da Google tramite questa rete neurale, garantendo una maggiore capacità di interpretazione delle query. Inoltre, verso la fine del 2022, abbiamo assistito ad un'evoluzione dei chatbot basati su Generative Pre-trained Transformer (GPT), i quali si sono imposti nel mercato grazie alla loro capacità di rielaborare una grande mole di testi in maniera efficiente. Un'altra applicazione dei transformer riguarda la traduzione automatica; il loro utilizzo, infatti, ha consentito a DeepL, uno dei servizi di traduzione automatica più in voga, di garantire traduzioni ad altissima qualità.

Le innovazioni apportate dai transformer nel mondo dell'Intelligenza Artificiale non sono limitate all'NLP; infatti nel 2021 è stato introdotto il Vision Transformer (ViT). Tale architettura deriva direttamente dal transformer utilizzato nei task di NLP; tuttavia, anziché dividere la frase in parole, divide l'immagine in patch rettangolari non sovrapposte e cerca le correlazioni semantiche tra di esse. I ViT hanno dimostrato di essere altamente competitivi e, in alcuni casi, superiori alle reti neurali convoluzionali. In particolare, essi tendono a superare le CNN su dataset di piccole dimensioni, poiché il meccanismo di attention permette loro di catturare relazioni complesse anche con un numero limitato di esempi di addestramento. Inoltre, i ViT possono gestire immagini in alta risoluzione in modo più efficiente rispetto alle CNN.

Inizialmente l'architettura del ViT poteva essere utilizzata unicamente per la classificazione. Successivamente, sono state proposte architetture alternative per la risoluzione di problematiche specifiche, come la segmentazione o l'object recognition. Sono state, anche, proposte delle modifiche all'architettura originale al fine di migliorarne le prestazioni. In particolare, la principale problematica dei ViT riguarda il tempo di train, in quanto, per ogni strato, deve essere calcolata l'attention di ogni token rispetto a tutti gli altri. Per questo

motivo sono nate alcune varianti che cercano di ridurre il tempo di train, come, ad esempio, la *SimpleViT*, la quale semplifica l'architettura diminuendo il numero di parametri.

Nell'era attuale dell'Intelligenza Artificiale, anche il Reinforcement Learning (RL) si erge come uno dei pilastri più affascinanti e promettenti. Negli ultimi anni, questa disciplina ha conosciuto un'accelerazione vertiginosa, catalizzata dalla sua applicabilità in una vasta gamma di contesti, i quali spaziano dai videogiochi alla guida autonoma. Il Reinforcement Learning cerca di simulare il modo attraverso cui gli esseri viventi si approcciano all'ambiente circostante ed imparano attraverso l'esperienza. L'esplorazione dell'ambiente permette di acquisire informazioni cruciali, che vengono, poi, utilizzate per scegliere la migliore azione da compiere, con l'obiettivo di massimizzare la ricompensa nel lungo periodo. Questo paradigma di apprendimento automatico offre un'enorme versatilità, permettendo alle macchine di affrontare sfide complesse e adattarsi a contesti mutevoli. Negli anni sono stati proposti diversi algoritmi per la risoluzione di tali problemi; tuttavia, solo grazie all'utilizzo delle reti neurali è stato possibile allenare agenti capaci di adattarsi ai contesti più vari, con prestazioni spesso superiori rispetto agli esseri umani. Ad oggi il Deep Reinforcement Learning aiuta diverse aziende nel prendere decisioni che massimizzano il rendimento in vari settori, come la gestione delle risorse energetiche e la finanza, o in contesti complessi, dove le interazioni tra variabili sono intricate e dinamiche.

Il presente progetto di tesi si colloca nei contesti applicativi menzionati in precedenza. In particolare, il suo obiettivo consiste nel proporre un approccio basato sul Reinforcement Learning per ridurre la complessità temporale dell'allenamento di un Vision Transformer. Anziché ridurre i parametri della rete neurale, come generalmente accade negli approcci precedentemente presentati in letteratura, si vuole proporre un approccio attraverso il quale diminuire il numero di operazioni eseguite dagli strati di attention, lasciando invariata l'architettura. L'agente è rappresentato da una rete Deep Q-learning costituita da due strati e il cui output è una lista di valori booleani. Tale rete si interfaccia con un environment codificato secondo lo standard proposto da OpenAI. L'agente ha il compito di osservare una batch di immagini, ovvero un sottoinsieme del dataset di training da dare in input al ViT e, conseguentemente, scegliere un sottoinsieme delle patch, in modo da diminuire in maniera significativa la complessità degli strati di attention. Per ogni singola batch di training, perciò, l'agente si interfaccia con l'environment osservando i valori di attention associati alle immagini in input e restituendo una lista di lunghezza pari al numero di patch; ad ogni elemento della lista corrisponde un valore booleano che indica se la patch deve essere utilizzata o meno durante il training. In seguito viene allenato il Vision Transformer, il quale osserva solo le patch scelte. Al termine dell'allenamento l'agente riceve un reward basato su una combinazione di loss di training e tempo di addestramento del ViT. L'utente può decidere quanto peso dare ad un parametro, piuttosto che all'altro, in modo che l'agente possa privilegiare un insieme di patch che garantisca un basso tempo di training o una bassa loss. Con un buon bilanciamento è possibile ottenere una forte riduzione del tempo di train, mantenendo pressoché invariate le performance del ViT. Al fine di migliorare la stabilità e la generabilità dell'agente, si è fatto uso di una replay memory in cui inserire i dati osservati ed utilizzarli successivamente durante l'addestramento. Inoltre, l'agente tende a scegliere un'azione casuale con probabilità pari a ϵ; tale valore decade esponenzialmente andando avanti con l'addestramento; ciò garantisce una buona capacità esplorativa nelle fasi iniziali, in modo da evitare di ricadere in un ottimo locale. È interessante rilevare che l'agente sceglie in maniera totalmente autonoma le patch da utilizzare, perciò non è necessario specificarne precedentemente il numero; ciò garantisce flessibilità.

La presente tesi è composta da sette capitoli strutturati come di seguito specificato:

- Nel Capitolo 1 si discuterà del Natural Language Processing e della nascita dei transformer; in seguito, si parlerà della storia della Computer Vision e delle principali

innovazioni apportate dai Vision Transformer rispetto agli approcci precedentemente utilizzati.

- Nel Capitolo 2 si introdurrà il Reinforcement Learning. In prima istanza verranno definiti i fondamenti psicologici su cui esso si basa. In seguito, sarà fornita una panoramica riguardo agli algoritmi più utilizzati, ponendo massima attenzione sul Deep Q-learning, il quale verrà adottato nel presente progetto di tesi.
- Nel Capitolo 3 verranno descritti i problemi che affliggono i ViT, con particolare attenzione per quanto riguarda la complessità temporale. In seguito si parlerà dei miglioramenti apportati dall'architettura SimpleViT; infine, sarà descritto il nostro approccio basato sul Reinforcement Learning.
- Nel Capitolo 4 saranno presentati i moduli che compongono il framework da noi proposto. In particolare, saranno descritti l'environment ed i meccanismi attraverso cui esso si interfaccia con il ViT; in seguito, sarà presentata l'implementazione dell'algoritmo di Deep Q-learning con replay memory e target network.
- Nel Capitolo 5 saranno riportati i risultati ottenuti, in modo da valutare l'efficienza e l'efficacia dell'approccio. In particolare, si prenderanno in esame i tempi di training e le metriche di performance del modello.
- Nel Capitolo 6 saranno descritti in maniera più accurata gli approcci già presenti in letteratura. In seguito sarà effettuato un confronto qualitativo con il framework da noi proposto.
- Nel Capitolo 7 verranno tratte le conclusioni e saranno delineati alcuni possibili sviluppi futuri. In particolare, si proporranno degli algoritmi di Reinforcement Learning che potrebbero risultare più promettenti per la risoluzione di tale problema.

CAPITOLO 1

I vision transformer

In questo capitolo sarà fornita una panoramica riguardo la nascita dei transformer e la loro applicazione nel Natural Language Processing e nella Computer Vision. Non verranno descritte tutte le architetture presenti in letteratura, bensì verranno introdotte le principali caratteristiche distintive rispetto alle reti neurali precedentemente utilizzate.

1.1 I Transformer nel Natural Language Processing

In questa sezione illustreremo il ruolo dei transformer nel Natural Language Processing. In particolare nella Sottosezione 1.1.1 verranno descritti brevemente la storia e i task di Natural Language Processing. In seguito, nella Sottosezione 1.1.2, saranno introdotti i transformer e le loro principali innovazioni rispetto allo stato dell'arte.

1.1.1 Il Natural Language Processing

Dopo la seconda guerra mondiale si sentì la necessità di sviluppare algoritmi per la traduzione automatica di testi. Inizialmente la traduzione avveniva parola per parola, ma con scarsi risultati. Nel 1949, Warren Weaver, uno scienziato statunitense, propose nuovi approcci per affrontare tale problema; queste idee pionieristiche gettarono le basi per la nascita del Natural Language Processing (NLP). Con questo termine si intende l'ambito di ricerca riguardante la creazione di algoritmi di Intelligenza Artificiale in grado di analizzare, rappresentare e comprendere il linguaggio naturale.

I principali task dell'NLP riguardano:

- *Part-of-Speech Tagging (POS Tagging)*: consiste nell'assegnare a ciascuna parola di una frase una specifica etichetta grammaticale (verbo, nome, aggettivo, pronome, etc.). Questo è essenziale per comprendere la struttura sintattica della frase e può essere utilizzato come input per altri task di NLP.
- *Named Entity Recognition (NER)*: mira a identificare e classificare le entità di interesse in un testo (nomi di persone, luoghi, organizzazioni, date, valute, etc.). Ciò è utile per estrarre informazioni strutturate da testi non strutturati e per comprenderne il contenuto. Viene spesso utilizzato per individuare le componenti più importanti in un documento.

- *Information Retrieval*: utilizzato dai motori di ricerca per estrarre i documenti che rispecchiano le query effettuate dagli utenti.
- *Sentiment Analysis*: si concentra sull'identificazione del sentimento o dell'emozione espressa in un testo. Questo task è ampiamente utilizzato nell'analisi delle opinioni dei clienti e nell'analisi dei sentimenti degli utenti sui social network.
- *Text Classification*: ha come obbiettivo l'assegnazione di una o più etichette ad un testo in base al suo contenuto. Può essere utilizzata per scopi diversi, come il rilevamento di spam, la categorizzazione di documenti, la classificazione di recensioni di prodotti etc. Le tecniche utilizzate per tale scopo spaziano dai semplici algoritmi di Machine Learning (KNN, Naive Bayes, etc.) alle più evolute reti profonde (LSTM, Transformers, etc.).
- *Machine Translation*: si occupa della traduzione di testo da una lingua sorgente a una lingua target. Questo task ha lo scopo di rendere il contenuto multilingue accessibile e comprensibile a un pubblico più vasto.
- *Text Generation*: riguarda la creazione automatica di frasi, paragrafi o testi completi. Negli ultimi anni è uno dei campi maggiormente in sviluppo grazie all'evoluzione dei Chatbot. Tale task è uno dei più complessi e richiede un enorme quantitativo di dati per l'allenamento.
- *Text Summarization*: ha come obiettivo la creazione di un breve riassunto di un testo più lungo. Può essere di tipo estrattivo, cioè estrarre frasi chiave, o di tipo astrattivo, creando nuove frasi per sintetizzare il contenuto.

Inizialmente l'attenzione era posta su sistemi di traduzione automatica e in tale contesto la sfida principale riguardava la traduzione tra lingue con regole grammaticali e strutture sintattiche molto diverse. Nel 1957, il celebre linguista americano Noam Chomsky fornì un importante contributo per risolvere questa sfida con la pubblicazione del suo libro "Syntactic Structures" (Chomsky [1957]). In questa opera introdusse la teoria della grammatica generativa, ovvero un insieme formale e ricorsivo di regole per descrivere le strutture sintattiche di un linguaggio. Questa teoria fornì una solida base teorica per lo sviluppo dei sistemi di NLP. Tuttavia, negli anni '60 e '70, l'Automatic Language Processing Advisory Committee (ALPAC) pubblicò un report (AA.VV. [1966]) che esaminava i progressi nella traduzione automatica e rilevava le limitazioni dei sistemi esistenti. La causa di tale "inverno dell'IA" era l'eccessiva rigidità delle regole su cui si basavano i sistemi di traduzione automatica.

Solo a fine anni '80 ci fu una rinascita dell'NLP grazie al passaggio da tecniche "rule-based" a tecniche "corpus-based"; infatti, grazie alla crescita della potenza computazionale, fu possibile allenare i modelli statistici su dei corpora di grandi dimensioni, al fine di indurre in maniera automatica la semantica delle parole cercando anche di tener conto del contesto. I corpora non sono altro che insiemi di testi provvisti di metadati utili per l'allenamento.

L'approccio statistico risultava, però, poco flessibile, poiché incapace di interpretare i modi di dire, gli errori di battitura o le parole non presenti nel corpus. Per questo motivo Tomáš Mikolov, nel 2013, introdusse l'approccio basato su vettorizzazione (Mikolov *et al.* [2013]) In tale approccio, tramite una fase di addestramento su un corpus vengono ricavate le rappresentazioni vettoriali delle singole parole. Tali rappresentazioni riescono a carpire la semantica e il contesto, ottenendo dei risultati molto più accurati rispetto alle tecniche statistiche. La rappresentazione vettoriale ha aperto la strada all'utilizzo delle reti neurali; in particolare, sono risultate di notevole interesse le reti neurali convoluzionali (Convolutional Neural Network-CNN) e ricorrenti (Recurrent Neural Network-RNN).

1.1.2 Nascita e funzionamento dei Transformer

Nonostante le reti neurali ricorrenti risultassero superiori rispetto agli altri approcci proposti, si è notato che, all'aumentare della lunghezza delle frasi, le performance tendevano a degradare in modo significativo. Per migliorare tali architetture si è fatto uso di un encoder e un decoder, legati da un meccanismo di "attention", il quale consentiva alla rete neurale di individuare i legami tra le parole indipendentemente dalla loro distanza. Ciò ha migliorato le performance, ma l'utilizzo di RNN e CNN continuava a essere un collo di bottiglia per i tempi di addestramento, data l'impossibilità di parallelizzare le operazioni. Il punto di svolta in tal senso è l'articolo "Attention is all you need." (Vaswani *et al.* [2017]), il quale ha introdotto i *transformer*, una nuova rete neurale interamente basata sul meccanismo di attention. Nelle successive sottosezioni sarà discusso il funzionamento di ogni singola componente dell'architettura, la quale è rappresentata in Figura 1.1.

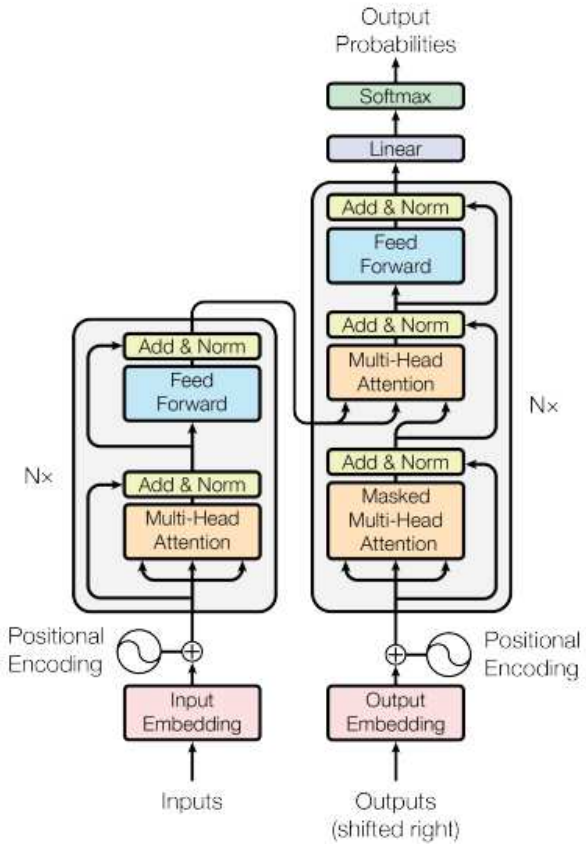


Figura 1.1: Architettura di un transformer

Input Embedding e Positional Encoding

Il testo in input viene inizialmente elaborato tramite un layer di embedding, il quale trasforma ogni parola del testo in un vettore numerico. I pesi di questo strato di embedding possono essere inizializzati casualmente e perfezionati durante il training, oppure possono essere pre-allenati usando librerie come Word2Vec, GloVe o FastText. A differenza delle RNN, i transformer non mantengono uno stato interno sequenziale e non conoscono l'ordine delle parole nella frase. Perciò il modello non è capace di distinguere se due parole con lo stesso embedding ma posizionate in punti diversi della frase abbiano significati diversi. Per questo motivo a ogni embedding viene aggiunta l'informazione riguardante il posizionamento

all'interno della frase. Tale operazione prende il nome di "positional encoding" ed è eseguito tramite la seguente formula:

$$\mathrm{PE}(t, 2i) = \sin\left(\frac{t}{10000^{\left(\frac{2i}{d_{\text{model}}}\right)}}\right)$$

$$\mathrm{PE}(t, 2i+1) = \cos\left(\frac{t}{10000^{\left(\frac{2i}{d_{\text{model}}}\right)}}\right)$$

dove:

- $\mathrm{PE}(t, 2i)$ e $\mathrm{PE}(t, 2i+1)$ sono, rispettivamente, i valori del vettore di posizione per la parola t e la dimensione i.
- d_{model} è la dimensione dell'embedding word (ad esempio, 512, 1024, ecc.).
- t è l'indice della parola nella sequenza, da 0 a $N-1$, dove N è la lunghezza massima della sequenza.
- i è la dimensione del vettore di posizione, che varia da 0 a $d_{\text{model}} - 1$.

Dopo aver eseguito queste due fasi, il modello ha accesso alla rappresentazione vettoriale delle singole parole (*Input Embedding*) e alla loro posizione all'interno della frase (*Positional Encoding*). L'output passa attraverso due componenti principali, l'Encoder e il Decoder, entrambi composti da più strati di blocchi di "attention".

Encoder

L'Encoder riceve il testo di input e lo elabora per estrarre informazioni utili. Ogni strato è composto da: Multi-Head Attention (MHA), Feed-Forward Neural Networks e Layer Normalization. Più specificatamente, gli strati dell'Encoder sono i seguenti:

- *Multi-Head Attention (MHA)*: è responsabile del meccanismo di attention, esso calcola l'importanza di ciascuna parola in base alla sua relazione con tutte le altre parole nella frase. Per farlo, crea diverse "teste" di attenzione (head) per acquisire diverse rappresentazioni delle parole, tale operazione è suddivisa in tre fasi principali:
  - *Creazione delle Query, delle Key e dei Value*: ogni parola del testo di input viene utilizzata per creare tre nuovi vettori chiamati "query", "key" e "value". Tali vettori sono ottenuti attraverso proiezioni lineari dell'embedding word originale.
  - *Calcolo dell'Attention Score*: per ciascuna parola, vengono calcolati gli attention score rispetto alle altre parole della frase. Ciò avviene attraverso il prodotto scalare normalizzato tra le query e le key. Il risultato è una distribuzione di pesi che indica quanto ciascuna parola è rilevante rispetto alle altre.
  - *Aggregazione dei Valori in base agli attention score*: utilizzando gli attention score calcolati nel passaggio precedente, le parole vengono pesate e combinate per formarne una nuova rappresentazione. Questo processo genera un nuovo embedding word per ogni parola, basato sulla rappresentazione vettoriale della parola e sul suo valore di attention.
- *Feed-Forward Neural Networks (FFN)*: riceve in ingresso l'output dell'MHA. Questo layer è composto da due strati completamente connessi separati da una funzione di attivazione GELU. Il ruolo della Feed-Forward Neural Network è quello di combinare

informazioni provenienti da diverse parole, integrandole insieme in una rappresentazione più complessa e significativa. Questo strato introduce un'ulteriore componente non-lineare alla rappresentazione, aumentando la capacità del modello di apprendere relazioni complesse nei dati di input. In questo modo, il transformer può integrare l'informazione proveniente da ciascun canale di attenzione (le diverse teste di attenzione) e produrre una rappresentazione globale significativa della frase.

- *Layer Normalization (LN)*: si trova a valle degli strati MHA e FNN; esso aiuta a bilanciare la distribuzione dei valori nei diversi canali, evitando fenomeni di "vanishing gradient" e "exploding gradient", che possono rallentare o impedire la convergenza del modello durante l'addestramento. La formula utilizzata in questo layer è la seguente:

$$LayerNorm(x) = \frac{x - \text{mean}(x)}{\sqrt{\text{var}(x) + \epsilon}} \times \gamma + \beta$$

dove:

  - x è l'input del blocco di normalizzazione (l'output della MHA o dell'FNN);
  - $\text{mean}(x)$ è la media degli elementi dell'input x;
  - $\text{var}(x)$ è la varianza degli elementi dell'input x;
  - ϵ è un termine di stabilizzazione per evitare divisioni per zero;
  - γ e β sono i parametri di scala e spostamento, che vengono appresi durante l'addestramento del modello.

Decoder

Il Decoder riceve in input l'output dell'Encoder, cioè la rappresentazione testuale avanzata della frase di input. Questa rappresentazione contiene informazioni sia sulle singole parole che sulle relazioni tra di esse. Inoltre, il Decoder riceve anche la sequenza di output desiderata, cioè la sequenza di parole che si desidera generare come risultato del processo di decodifica. Questa sequenza di output desiderata è una sequenza di parole che termina con un token speciale. Anche il Decoder è composto da strati di Multi-Head Attention e Feed-Forward Neural Networks, ma con alcune modifiche significative. Più specificatamente gli strati del Decoder sono i seguenti:

- *Self-Attention (SA)*: durante il processo di decodifica, il Decoder utilizza un meccanismo di attenzione speciale chiamato self-attention. Per generare la nuova parola della sequenza, il modello tiene conto di quelle generate precedentemente; in questo modo è possibile tenere conto del contesto della frase, al fine di guidare la generazione del testo in modo più efficace. Il meccanismo di self-attention nel Decoder si differenzia da quello di attenzione standard nell'Encoder poiché dipende esclusivamente dalla sequenza di output che il Decoder ha generato fino a quel momento, senza coinvolgere l'input originale della frase.

- *Feed-Forward Neural Networks (FFN)*: esegue un compito analogo a quello svolto nell'Encoder.

- *Layer Normalization (LN)*: esegue un compito analogo a quello svolto nell'Encoder.

- *Softmax*: a valle dell'FNN, viene applicata la funzione Softmax a ciascun elemento dell'output. Questa funzione converte i valori numerici in probabilità, associando a ciascuna parola della sequenza di output una probabilità di essere la prossima parola

da generare. La parola con la probabilità più alta viene inserita nella sequenza di output e il processo di generazione del testo continua iterativamente. La sequenza di parole generate viene utilizzata come input per il passaggio successivo di self-attention, permettendo al modello di continuare a generare parole coerenti e significative, sfruttando sia le informazioni provenienti dall'Encoder che il contesto generato durante il processo di decodifica.

L'architettura descritta consente di avere un alto grado di parallelismo, il che porta a tempi di training molto più bassi rispetto alle architetture basate su RNN e CNN.

A partire dai transformer sono nate diverse architetture utilizzate nello stato dell'arte, le quali però non saranno qui descritte in quanto esulano dallo scopo della tesi.

1.2 Dall'NLP alla Computer Vision

In questa sezione descriveremo l'utilizzo dei transformer, inizialmente pensati per l'NLP, nel contesto della Computer Vision. In particolare, nella Sottosezione 1.2.1 verranno descritti brevemente la storia e i task di Computer Vision. Inoltre, nella Sottosezione 1.2.2 si parlerà dell'utilizzo dei transformer in questo campo dell'Intelligenza Artificiale .

1.2.1 La Computer Vision

La Computer Vision (CV) è un campo dell'Intelligenza Artificiale che mira a sviluppare algoritmi che consentano ai computer di comprendere e interpretare il mondo visivo in modo analogo rispetto agli esseri umani. Le principali applicazioni della Computer Vision riguardano:

- *Elaborazione delle Immagini*: si occupa di sviluppare algoritmi per l'elaborazione, il filtraggio e la manipolazione delle immagini. Include operazioni di pre-processing, come filtraggio e riduzione del rumore, miglioramento della qualità dell'immagine, correzione dei colori e segmentazione.
- *Riconoscimento degli Oggetti*: è una delle sfide principali della Computer Vision. Si tratta di identificare e classificare gli oggetti presenti in un'immagine o in un video. Gran parte delle tecniche fa uso di bounding-box, cioè di rettangoli disegnati all'interno dei quali sono presenti gli oggetti da riconoscere e classificare.
- *Tracciamento e Localizzazione degli Oggetti*: riguarda il monitoraggio e il tracciamento degli oggetti nel tempo e nello spazio. Ciò è particolarmente utile nella sorveglianza, nella guida autonoma e nella robotica.
- *Classificazione delle Immagini*: consiste nell'assegnare un'etichetta o una categoria a un'immagine. È uno dei compiti fondamentali della Computer Vision e, in genere, viene realizzato utilizzando CNN addestrate su grandi dataset di immagini annotate.
- *Segmentazione delle Immagini*: è il processo di suddivisione di un'immagine digitale in più insiemi di pixel. Esistono due tipi principali di segmentazione delle immagini: la segmentazione semantica e la segmentazione di istanza. Nella segmentazione semantica, tutti gli oggetti dello stesso tipo sono contrassegnati da un'etichetta di classe, mentre nella segmentazione di istanza gli oggetti simili ricevono etichette separate.
- *Generazione e Modifica di Immagini*: alcuni approcci di Computer Vision consentono di generare nuove immagini o modificare immagini esistenti in modo creativo e realistico. Negli ultimi anni tale ambito risulta essere uno dei più promettenti nel mondo dell'Intelligenza Artificiale.

La Computer Vision prende forma negli anni '60, quando i ricercatori iniziarono a esplorare metodi per l'elaborazione delle immagini e il riconoscimento di forme; contemporaneamente, nel 1958, Frank Rosenblatt stava definendo il percettrone, cioè l'antenato delle moderne reti neurali. Negli anni '70, la Computer Vision ha iniziato a fare progressi con l'introduzione di algoritmi per l'estrazione di feature dalle immagini. In questi anni nasce il filtro di Sobel, il quale consente di individuare bordi nelle immagini. Viene anche introdotta la "Trasformata di Hough" proposta da Richard Duda e Peter Hart nel 1972; essa permette di rilevare linee e altre forme geometriche nelle immagini ed è tutt'ora utilizzata in diverse applicazioni.

Negli anni '80 sono nati altri approcci per l'estrazione di feature, come il "Canny Edge Detector" di John Canny; tale algoritmo consente di individuare bordi e contorni con maggior accuratezza e indipendentemente dalla forma dell'oggetto. Solo negli anni '90, grazie all'aumento della potenza computazionale, la Computer Vision ha iniziato a ottenere risultati ottimi. Sono state introdotte le Convolutional Neural Network (CNN), un tipo di rete neurale specificamente progettato per il riconoscimento delle immagini. Nel nuovo millennio, la creazione di database di immagini annotate, come ImageNet, ha consentito l'addestramento di reti neurali profonde, portando a risultati sorprendenti nel riconoscimento di oggetti e il superamento di prestazioni umane in alcune applicazioni. Negli ultimi anni la Computer Vision ha raggiunto livelli di precisione e complessità mai visti prima. Le CNN hanno dimostrato di essere particolarmente efficaci nei task di Computer Vision, anche se stanno nascendo altri approcci con risultati comparabili; tra questi troviamo i Vision Transformer.

1.2.2 I Vision Transformer

I Vision Transformer (ViT) derivano dall'applicazione dei transformer alla Computer Vision. Di seguito discuteremo il principio generale senza soffermarci su una specifica architettura, in quanto tali reti neurali sono un argomento molto discusso in letteratura, e perciò in continua evoluzione.

I Vision transformer sono stati introdotti nell'articolo "An Image is Worth 16x16 Words: transformer for Image Recognition at Scale" (Dosovitskiy *et al.* [2021]). Precedentemente esistevano architetture basate su CNN al cui interno venivano utilizzati blocchi "attention", ma non esisteva alcuna rete interamente basata su tale tipologia di layer. Nella Figura 1.2 è rappresentata l'architettura completa. Le componenti aggiuntive rispetto all'architettura già vista per l'NLP riguardano l'embedding dell'immagine. Un'altra differenza riguarda l'assenza dell'encoder, il quale non è necessario nei task di classificazione. Nelle sottosezioni successive discuteremo tali differenze e i miglioramenti apportati da questa nuova architettura rispetto alle classiche reti convoluzionali.

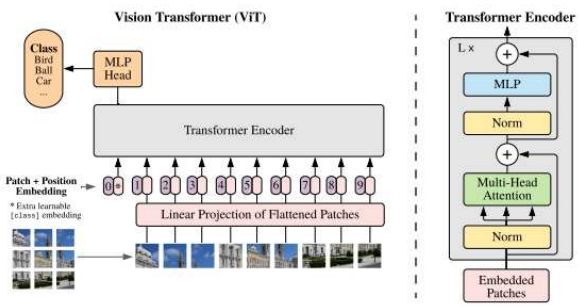


Figura 1.2: Architettura di un Vision Transformer

Input Embedding e Positional Encoding

A differenza delle reti neurali convoluzionali che lavorano direttamente sull'immagine, i ViT trattano l'immagine come una griglia di patch rettangolari non sovrapposte. Le patch, rappresentate come vettori, sono proiettate linearmente in uno spazio ad alta dimensionalità, detto embedding space. Come nell'NLP il modello non conosce la posizione delle patch all'interno dell'immagine, perciò ad ogni embedding viene sommato il Positional Encoding, il quale viene calcolato con la seguente formula:

$$\text{Positional Embedding}_{(i,j)} = \sin\left(\frac{i}{10000^{\frac{2j}{d}}}\right)$$

$$\text{Positional Embedding}_{(i,j)} = \cos\left(\frac{i}{10000^{\frac{2j}{d}}}\right)$$

Dove:

- i rappresenta la posizione della patch all'interno dell'immagine (l'indice della patch);
- j rappresenta la dimensione del vettore di embedding di posizione;
- d rappresenta la dimensione totale degli embedding.

Encoder

L'output della fase precedente viene utilizzato come input a un classico encoder; non c'è alcuna differenza rispetto all'encoder descritto nella Sottosezione 1.1.2, infatti anche qui troviamo una serie di attention layer, Feed-Forward Neural Network e Normalization Layer impilati. A valle dell'Encoder è presente un Multi Layer Perceptron per la classificazione multiclasse. Come già detto non è presente un Decoder poiché nei task di classificazione non è necessario alcun meccanismo di ricostruzione dell'informazione.

ViT vs CNN

A differenza delle CNN, le quali estraggono informazioni dall'immagine tramite operazioni locali, i ViT riescono a catturare relazioni a lungo raggio tra le diverse parti di un'immagine grazie al meccanismo di attention. Inoltre, analogamente a quanto accade nell'NLP, i transformer consentono una maggior parallelizzazione rispetto alle reti convoluzionali, richiedendo un tempo minore per l'addestramento. Un altro vantaggio riguarda l'adattamento a diverse risoluzioni; per poter adattare una CNN ad immagini con risoluzioni diverse è necessario agire a livello di architettura, oppure in fase di preprocessing modificando l'immagine; nel caso dei ViT, invece, basta modificare la dimensione delle patch.

CAPITOLO 2

Il reinforcement learning

In questo capitolo verrà introdotto il Reinforcement Learning descrivendone i fondamenti neuroscientifici per poi passare ad analizzare l'utilizzo di tali concetti nel mondo informatico. Discuteremo gli algoritmi principali, soffermandoci sul Deep Q-Learning, il quale verrà utilizzato durante lo sviluppo del progetto di tesi.

2.1 Fondamenti psicologici: la teoria del rinforzo

A inizio del secolo scorso John Watson fonda una corrente psicologica detta Comportamentismo; essa si basava sull'assunto che il comportamento esplicito dell'individuo sia l'unica unità di analisi scientificamente studiabile dalla psicologia. Secondo tale visione lo scienziato deve somministrare stimoli al soggetto e valutarne le risposte direttamente osservabili. Questa corrente è stata particolarmente utile per la conoscenza dei meccanismi di apprendimento, consentendo la definizione di teorie che oggi sono state verificate dalle neuroscienze. Nella Sottosezione 2.1.1 si parlerà dell'apprendimento e della prima fase del Comportamentismo; mentre nella Sottosezione 2.1.2 si parlerà della seconda fase.

2.1.1 Il condizionamento classico

Per apprendimento si intende un mutamento relativamente permanente nel comportamento in seguito all'acquisizione di esperienza. Esso può essere declinato in due tipologie:

- *Apprendimento Associativo*: in questa categoria rientrano il condizionamento classico e il condizionamento operante. Accomuna gli esseri umani e le altre specie animali.
- *Apprendimento Cognitivo*: tipico degli esseri umani, comporta attività di comprensione, conoscenza, anticipazione ed è, quindi, fondato su processi mentali superiori, come attenzione, memoria, linguaggio, etc.

Nel Reinforcement Learning l'apprendimento è di tipo associativo; per questo motivo, in questa sede, non sarà trattato l'apprendimento cognitivo, dato che esula dagli scopi della tesi. Con l'apprendimento associativo la semplice ripetizione di una risposta non produce necessariamente un apprendimento. Ciò che governa l'apprendimento associativo è il rinforzo, ovvero qualunque evento in grado di far aumentare la probabilità che, dato uno stimolo, si verifichi una determinata risposta.

A inizio '900 il fisiologo russo Ivan Pavlov era intento nello studio dei meccanismi di salivazione nei cani. Ben presto si accorse che ogniqualvolta si presentasse del cibo di fronte a un cane, questo cominciava a salivare. Pavlov chiamò questo fenomeno "riflesso di salivazione". Eseguendo ripetutamente l'esperimento, osservò che il cane iniziava a secernere la saliva alla semplice visione dell'inserviente addetto alla somministrazione del cibo, anche in mancanza di quest'ultimo. Il cane aveva appreso che, quando l'inserviente si avvicinava alla gabbia, lo faceva per portare il cibo. Per approfondire e confermare ciò che stava studiando, Pavlov cominciò ad applicare uno stimolo uditivo un attimo prima di servire il cibo. Inizialmente questi stimoli risultavano neutrali; tuttavia, dopo varie applicazioni, si venne a creare un'associazione: quando i cani udivano il suono di un campanello essi iniziavano a salivare, indipendentemente dal fatto che il cibo venisse effettivamente servito. Pavlov definì "riflesso condizionato" la salivazione che si verificava dopo questa associazione. Lo schema generale individuato da Pavlov è composto dalle seguenti componenti:

- *Stimolo Incondizionato (SI)*: stimolo che provoca automaticamente una risposta nell'organismo.
- *Risposta Incondizionata (RI)*: risposta automatica dell'organismo a uno stimolo incondizionato. Ad esempio, la quantità di saliva secreta da un cane in risposta al cibo.
- *Stimolo Neutro (SN)*: stimolo che, quando presente, non causa risposta nell'organismo.
- *Stimolo Condizionato (SC)*: stimolo neutro che, dopo essere stato associato temporaneamente a uno stimolo incondizionato, provoca una risposta simile a quella dell'RI.
- *Risposta Condizionata (RC)*: risposta che appare quando si presenta solo lo stimolo condizionato. Per esempio, la quantità di saliva secreta dai cani in risposta a uno stimolo uditivo.

Pavlov osservò che il condizionamento era più rapido se lo stimolo incondizionato seguiva immediatamente lo stimolo neutro. Inoltre possono essere definite delle "catene di condizionamento": uno stimolo condizionato ben appreso può essere utilizzato per la creazione di un ulteriore apprendimento, ma solo se lo stimolo condizionato è diventato abbastanza forte da poter essere utilizzato come se fosse uno stimolo incondizionato. Negli ultimi anni, grazie allo sviluppo delle neuroscienze, è stato possibile comprendere le motivazioni alla base di questi meccanismi: il substrato neurale del condizionamento classico è stato individuato nel circuito talamo-amigdala. Tale struttura accomuna gli umani alle altre specie; infatti, sia gli animali che gli uomini tendono a creare associazioni tra gli eventi della realtà circostante e le risposte fornite. Un fenomeno rilevante è l'estinzione, la quale si verifica quando il rapporto tra lo stimolo condizionato è quello incondizionato si indebolisce o viene eliminato del tutto; al contrario, si parla di recupero spontaneo quando una risposta appresa ricompare dopo un'apparente estinzione. Inoltre è importante rilevare il fenomeno della generalizzazione, cioè la capacità del nostro cervello di adattarsi a stimoli simili, ma non identici allo stimolo condizionato, generando una risposta simile a quella condizionata; ciò è dovuto al nostro meccanismo evolutivo, che ci ha resi capaci di adattarci a situazioni analoghe, ma non identiche, a quelle che fanno parte della nostra esperienza. Tali scoperte valsero a Pavlov il Nobel per la medicina nel 1904.

Circa 10 anni dopo le scoperte di Pavlov, nel 1913, John Watson pubblica "La psicologia secondo il punto di vista del comportamentista" (Watson [1913]). Egli propose il processo di condizionamento classico come spiegazione dell'apprendimento negli umani; infatti, riteneva che le differenze nel comportamento tra gli individui fossero causate dalle diverse esperienze vissute da ciascuno. Per dimostrare tale tesi, John Watson e Rosalie Rayner effettuarono

il famoso "Esperimento del piccolo Albert", con lo scopo di condizionare una fobia in un bambino emotivamente stabile. Durante l'esperimento fu consentito ad Albert di giocare con un topo da laboratorio bianco. In seguito Watson e Rayner iniziarono a provocare un forte suono ogniqualvolta il bambino toccava il topo. Albert rispondeva al rumore piangendo e mostrando paura. Dopo diverse ripetizioni, alla sola visione del topo bianco, il bimbo piangeva e cercava di scappare. Apparentemente, il bambino associava il ratto bianco al rumore, cioè era stato condizionato a provare paura.

2.1.2 Il condizionamento operante

La seconda fase del Comportamentismo ebbe origine negli Stati Uniti nella prima metà del XX secolo. Il suo principale esponente è Burrhus Frederic Skinner, uno psicologo di rilievo che sviluppò la teoria del condizionamento operante. La principale differenza con il condizionamento classico sta nel fatto che la risposta voluta non è più un comportamento involontario (ad esempio, la salivazione nell'esperimento di Pavlov), ma è un'attivazione di muscoli volontari. L'apprendimento avviene, quindi, a livello di operazioni motorie complesse che operano attivamente sull'ambiente. Skinner postulò che il comportamento è determinato dalle sue conseguenze immediate. Se un comportamento viene seguito da un rinforzo positivo, è più probabile che si ripeta. Al contrario, se un comportamento viene seguito da una punizione, è meno probabile che si ripeta.

Per approfondire lo studio del condizionamento operante, Skinner inventò la *skinner box*: una gabbia apposita per studiare questo tipo di comportamento negli animali. Nella gabbia sono presenti solo una leva e una ciotola vuota di mangime, entrambi collegati a un distributore. Inizialmente l'animale esplora l'ambiente in maniera casuale; nel momento in cui preme la leva viene attivato il distributore di mangime. La somministrazione del rinforzo positivo (mangime) altera la frequenza con la quale l'animale attua il comportamento richiesto (azionare la leva). Il rinforzo positivo fa si che l'animale capisca che l'azione presa è corretta e, perciò, è da ripetere. Come nel condizionamento classico, anche qui la risposta condizionata può estinguersi e può subire recupero spontaneo. Il rinforzo negativo si verifica quando un comportamento ha come conseguenza la diminuzione o l'eliminazione di qualcosa di spiacevole. Gli psicologi hanno poi definito delle diverse categorizzazioni del rinforzo, in particolare esso può essere:

- Continuo: segue ogni risposta corretta; inizialmente è molto utile per l'apprendimento di risposte nuove, ma genera risposte che tendono a estinguersi velocemente.
- Parziale: non segue tutte le risposte corrette; questo tipo di rinforzo mantiene più elevata l'attesa e la motivazione, quindi l'apprendimento sarà più lento ma le risposte saranno molto più resistenti all'estinzione; tale meccanismo è utilizzato nel gioco d'azzardo per mantenere alte le aspettative del giocatore.

I rinforzi positivi e negativi aumentano la probabilità che venga fornita una determinata risposta. Per diminuire la probabilità di una risposta possono essere utilizzate le punizioni; ad esempio, nel caso della skinner box, la punizione è rappresentata da una scossa elettrica che colpisce il topo ogniqualvolta eseguirà un'azione ritenuta errata. L'efficacia di una punizione dipende da tre variabili: tempo, coerenza e intensità. Infatti, affinché la punizione ottenga l'effetto desiderato, deve essere somministrata subito dopo la risposta e ogni volta che si verifica quella stessa risposta; inoltre, la punizione deve essere proporzionata rispetto alla risposta, tenendo conto degli effetti avversi che essa può generare: paura, evitamento o aggressività.

Da un punto di vista neuroscientifico il condizionamento operante coinvolge una serie di circuiti neurali che collegano diverse regioni del cervello. In particolare le componenti maggiormente interessate sono:

- *L'amigdala*: si tratta di una struttura interna del sistema limbico, focalizzata sull'elaborazione emotiva e sulla valutazione delle conseguenze delle azioni. Nel contesto del condizionamento operante, l'amigdala può svolgere un ruolo chiave nella valutazione emotiva delle conseguenze di un'azione. Ad esempio, se un comportamento porta a conseguenze piacevoli, l'amigdala potrebbe contribuire a riconoscere la gratificazione associata a quelle conseguenze. Viceversa, se un comportamento ha conseguenze sgradevoli, l'amigdala potrebbe essere coinvolta nell'elaborare le implicazioni negative. Questa valutazione può influenzare la probabilità che il comportamento venga ripetuto in futuro.
- *Il nucleo accumbens*: situato nella parte centrale del cervello, il nucleo accumbens fa parte del sistema di ricompensa. Questa regione è connessa all'elaborazione delle sensazioni di piacere e gratificazione. Durante il condizionamento operante, il nucleo accumbens può essere coinvolto quando un individuo riceve una ricompensa positiva in risposta a un determinato comportamento. Questo processo rafforza il legame tra il comportamento e la gratificazione, aumentando la probabilità che il comportamento venga ripetuto. Inoltre, il nucleo accumbens è coinvolto anche nell'elaborazione delle ricompense attese, contribuendo, così, al processo di pianificazione delle azioni basate su previsioni di risultati positivi.
- *La dopamina*: la dopamina è un neurotrasmettitore chiave nel sistema di reward. Il cervello cerca di massimizzare le ricompense future attraverso un aumento del rilascio di dopamina quando si prevede o si ottiene un risultato positivo. Questo meccanismo incentiva l'adozione di comportamenti che conducono a risultati gratificanti. Nel condizionamento operante, quando un individuo compie un'azione ritenuta come positiva, la dopamina viene rilasciata nelle aree del cervello coinvolte nel sistema di reward, rafforzando così l'associazione tra il comportamento e la ricompensa. Come detto, il cervello cerca di massimizzare le ricompense future; ciò implica che esso potrebbe prendere decisioni non promettenti nel breve termine al fine di raggiungere un alto rilascio di dopamina nel lungo termine. Tale meccanismo governa gran parte delle nostre azioni e consente a molte specie animali di eseguire previsioni a lungo termine (un esempio banale riguarda la decisione di iscriversi all'università: tale azione non genera un reward positivo nel breve termine, ma assicura una ricompensa molto elevata nel lungo periodo).

2.2 Il Reinforcement Learning

In questa sezione parleremo diffusamente del Reinforcement Learning. In particolare, nella Sottosezione 2.2.1 verranno applicate al mondo informatico le conoscenze acquisite nella sezione precedente, definendo il quadro teorico su cui si basano gli algoritmi di Reinforcement Learning. Nelle sottosezioni successive saranno discusse le principali tecniche per la risoluzione di problemi di Reinforcement Learning.

2.2.1 L'apprendimento associativo nell'informatica

Il Reinforcement Learning (RL) è una tecnica di Intelligenza Artificiale che mira ad addestrare agenti autonomi in grado di scegliere azioni da compiere per il conseguimento di

determinati obiettivi tramite l'interazione con l'ambiente in cui sono immersi. A differenza di quanto accade nell'apprendimento supervisionato o non supervisionato, questo paradigma si occupa di problemi di decisione sequenziali, in cui l'azione da compiere dipende dallo stato attuale del sistema e ne determina quello futuro. L'approccio utilizzato per la formalizzazione dei problemi di RL rispecchia esattamente il paradigma esposto nella sottosezione relativa al condizionamento operante. Le componenti principali sono:

- *Agent*: agente che prende decisioni (topo).
- *Environment*: ambiente in cui è immerso l'agente (skinner box)
- *State*: stato corrente osservato dall'agente.
- *Reward*: feedback fornito dall'ambiente (cibo o scossa elettrica, in base alla bontà dell'azione).
- *Policy*: mapping tra lo stato corrente e l'azione da intraprendere (ciò è rappresentato dagli schemi mentali del topo che lo portano ad associare una specifica azione allo stato corrente).

Analogamente a ciò che avviene nel nostro cervello tramite il rilascio di neurotrasmettitori, l'agente cercherà di eseguire le azioni che massimizzeranno le ricompense future cumulative; perciò potrebbero essere prese decisioni che portano ad un reward nel lungo termine, piuttosto che ad un reward immediato. Il meccanismo di interazione tra agente e ambiente è descritto in Figura 2.1: l'agente osserva uno stato ed esegue un'azione, spostandosi nello stato successivo e ricevendo un reward dall'environment; il ciclo si ripete fino al termine dell'addestramento. I problemi di Reinforcement Learning si dividono in due tipologie:

- *Episodici*: dopo un certo numero di interazioni con l'ambiente, o dopo aver raggiunto lo stato finale, l'agente torna nella posizione iniziale e comincia l'episodio successivo. L'obiettivo dell'agente è, tipicamente, massimizzare la somma delle ricompense all'interno di ciascun episodio, indipendentemente dagli episodi precedenti. (ad esempio, il gioco degli scacchi, l'esplorazione di un labirinto, etc.).
- *Non episodici*: l'agente interagisce con l'ambiente in un flusso continuo senza una distinzione chiara tra episodi separati. L'obiettivo dell'agente è la massimizzazione delle ricompense nel lungo periodo. (ad esempio, il trading finanziario, il controllo automatico della temperatura, etc.).

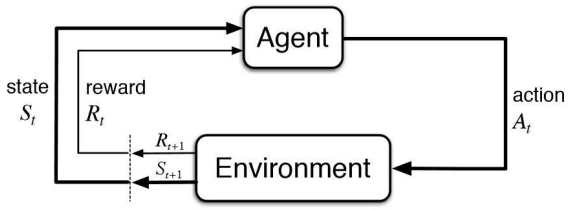


Figura 2.1: Schema di funzionamento del Reinforcement Learning

2.2.2 Exploration vs Exploitation

In questa tipologia di problemi una scelta errata può compromettere i risultati ottenuti successivamente; per questo motivo, l'agente è spronato a scegliere sempre l'azione più

vantaggiosa. D'altra parte, se l'agente scegliesse sempre l'azione con reward cumulativo più alto (scelta *greedy*), ciò precluderebbe la possibilità di trovare nuove strade più vantaggiose per il raggiungimento dell'obiettivo. Immaginiamo di dover allenare un agente a risolvere il problema rappresentato nella Figura 2.2: partendo dalla posizione S bisogna arrivare alla posizione E con il minor numero di passi, evitando di cadere nelle buche contrassegnate dalla X. Tra le infinite traiettorie supponiamo di valutare il percorso in verde e quello in rosso, entrambi consentono all'agente di terminare il task con successo, ma con un numero diverso di passi. Se l'agente esplorando l'ambiente arrivasse allo stato finale seguendo il percorso verde, egli tenderebbe a scegliere sempre quel percorso come migliore, nonostante sia evidente che il migliore è quello rosso. Ciò è dovuto al fatto che l'agente sceglie sempre le azioni con un approccio greedy. Negli algoritmi di RL è necessario bilanciare in maniera accurata la fase di esplorazione (exploration) e di sfruttamento della conoscenza acquisita (exploitation). Per questo motivo gran parte degli algoritmi sfruttano un meccanismo definito ϵ-*greedy*: l'agente esegue con probabilità pari ad ϵ un'azione casuale, e con probabilità 1-ϵ l'azione più promettente. Esistono diverse tecniche per definire il valore di ϵ; una delle più usate consiste nello scegliere un valore iniziale elevato e diminuirlo linearmente o esponenzialmente all'aumentare del numero di iterazioni, in modo da diminuire la probabilità di esplorazioni in favore dello sfruttamento della conoscenza.

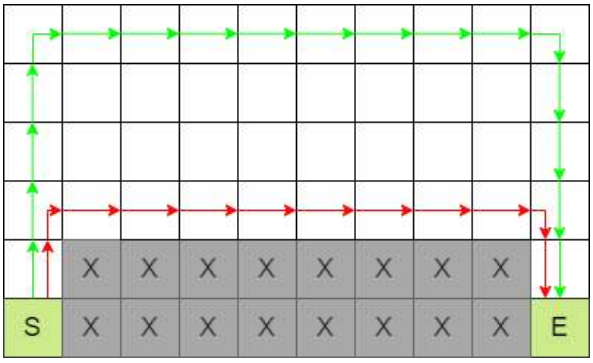


Figura 2.2: GridWorld: confronto tra due traiettorie

2.2.3 Markov Decision Process

Gran parte dei problemi di Reinforcement Learning possono essere modellati matematicamente attraverso i *Processi Decisionali di Markov* (MDP). Un MDP è rappresentato attraverso un insieme finito di stati rappresentati tramite dei nodi. Ogni stato contiene uno o più archi uscenti, i quali rappresentano le azioni che possono essere eseguite dall'agente; ogni azione perciò è rappresentata attraverso una transizione di stato e può essere etichettata con un valore compreso tra 0 ed 1. Tale valore indica qual è la probabilità che un l'agente esegua quella determinata azione partendo da quello specifico stato. Poiché l'agente deve sempre prendere una decisione, la somma delle probabilità delle azioni disponibili per ogni stato deve essere uguale a 1. La proprietà più rilevante degli MDP è che, dato uno stato, la probabilità di scegliere uno specifico stato successivo dipende solo dallo stato corrente e dall'azione intrapresa, indipendentemente dalle tappe precedenti.

Nella maggior parte dei problemi non è possibile conoscere esattamente le probabilità di transizione da uno stato all'altro, per questo motivo sono stati sviluppati algoritmi detti model-free, i quali sono più flessibili rispetto a quelli "model-based", che dipendono da una rappresentazione accurata del modello di transizione. Di conseguenza, nelle prossime sezioni, affronteremo in dettaglio i meccanismi e le caratteristiche degli algoritmi model-free, poiché più generali.

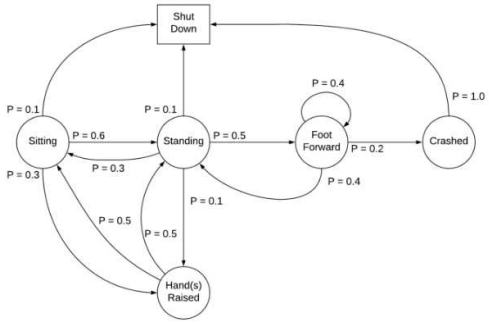


Figura 2.3: Esempio di funzionamento dell'MDP

2.2.4 Policy ed Equazione di Bellman

Come già espresso in precedenza, lo scopo dell'allenamento consiste nel trovare la policy ottima, cioè il mapping tra stato e azione tale da garantire il massimo reward cumulativo. Per far ciò è necessario un meccanismo per valutare ogni stato e ogni azione, in modo da scegliere quella migliore. Di seguito sono descritte le funzioni attraverso le quali si stimano i valori delle ricompense future e un algoritmo per il loro calcolo.

State-Value Function

La State-Value Function, denotata come $V(s)$, assegna a ciascuno stato s un valore reale, il quale riflette la somma attesa delle ricompense future che l'agente può accumulare intraprendendo azioni seguendo la policy π. Ciò consente di stimare quanto sia vantaggioso per l'agente trovarsi in uno specifico stato e seguire una data policy.

La State-Value Function è formalmente espressa come:

$$V(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right]$$

Dove:

- $\mathbb{E}_\pi$ rappresenta l'aspettativa rispetto alle azioni intraprese seguendo la politica π.
- R_{t+k+1} è la ricompensa ottenuta all'istante $t + k + 1$.
- γ è il fattore di sconto che determina l'importanza delle ricompense future rispetto a quelle immediate. È compreso tra 0 e 1; quando γ è vicino a 0 l'agente attribuisce molta importanza alle ricompense immediate e tende a prendere decisioni che massimizzano le ricompense a breve termine. Quando γ è vicino a 1 l'agente dà maggior peso alle ricompense future rispetto a quelle immediate. Ciò spinge l'agente a prendere decisioni che massimizzano il ritorno complessivo nel lungo periodo, anche se ciò potrebbe richiedere rinunce o sacrifici immediati.

La funzione $V(s)$ riflette l'utilità o la bontà di trovarsi nello stato s secondo la politica π. Essa tiene conto sia delle ricompense istantanee che di quelle future, ponderate dal fattore di sconto γ. Un valore elevato di $V(s)$ indica che lo stato s è altamente vantaggioso per l'agente nel lungo periodo.

La State-Value Function guida le decisioni dell'agente, poiché fornisce una stima dell'importanza e dell'utilità dei vari stati. Quando l'agente si trova nello stato s, può scegliere le azioni che lo portano nello stato s' che massimizza il valore di $V(s)$ al fine di conseguire ricompense elevate.

Action-Value Function

L'Action-Value Function, denotata come $Q(s, a)$, stima quanto sia vantaggioso intraprendere un'azione specifica a nello stato s secondo una politica specifica π.

Questa funzione assegna a ciascuna coppia stato-azione (s, a) un valore reale che riflette la somma attesa delle ricompense future che l'agente può accumulare intraprendendo l'azione a nello stato s e seguendo la politica π. Formalmente:

$$Q(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right]$$

Dove:

- $\mathbb{E}_\pi$ rappresenta l'aspettativa rispetto alle azioni intraprese seguendo la politica π.
- R_{t+k+1} è la ricompensa ottenuta all'istante $t + k + 1$.
- γ è il fattore di sconto che determina l'importanza delle ricompense future rispetto a quelle immediate.

La funzione $Q(s, a)$ fornisce all'agente informazioni sulla bontà di intraprendere un'azione specifica a nello stato s. Essa tiene conto sia delle ricompense istantanee che di quelle future, consentendo all'agente di giudicare il valore a lungo termine di un'azione in una data situazione.

Ad esempio, se $Q(s, a)$ è alto, significa che intraprendere l'azione a nello stato s è probabilmente vantaggioso nel lungo periodo, poiché si prevede un accumulo significativo di ricompense positive.

La funzione $Q(s, a)$ è essenziale nella selezione delle azioni ottimali. Quando l'agente si trova nello stato s, può scegliere l'azione a che massimizza il valore di $Q(s, a)$, garantendo una maggiore probabilità di ottenere ricompense elevate nel lungo periodo.

Equazione di Bellman

L'Equazione di Bellman esprime il valore di uno stato (o azione) in termini dei valori dei successivi stati (o azioni). Essa svolge un ruolo cruciale nell'apprendimento e nell'aggiornamento delle value function, consentendo all'agente di stimare i valori degli stati o delle azioni attraverso le interazioni con l'ambiente.

Per la State-Value Function $V(s)$, l'Equazione di Bellman è espressa come:

$$V(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V(s') \right]$$

Dove:

- $\pi(a|s)$ è la probabilità di intraprendere l'azione a nello stato s secondo la politica π.
- $p(s', r|s, a)$ è la probabilità di transizione allo stato s' con ricompensa r quando si intraprende l'azione a nello stato s.
- γ è il fattore di sconto che pesa le ricompense future.

Per l'Action-Value Function $Q(s, a)$, l'Equazione di Bellman è espressa come:

$$Q(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') Q(s', a') \right]$$

Dove:

- $\pi(a'|s')$ è la probabilità di intraprendere l'azione a' nello stato s' secondo la politica π.
- γ è il fattore di sconto che riflette quanto l'agente valuta le ricompense future rispetto a quelle immediate.

L'Equazione di Bellman riflette il principio di decomposizione dei valori, il quale afferma che il valore di uno stato o azione è determinato dalla ricompensa immediata e dal valore dei successivi stati o azioni. Questo principio è essenziale nell'apprendimento dei valori attraverso le interazioni con l'ambiente, poiché consente di aggiornare gradualmente le stime dei valori sulla base delle informazioni ricevute.

2.2.5 Algoritmi on-policy

Gli algoritmi di Reinforcement Learning possono essere suddivisi in due categorie principali in base a come gestiscono l'esperienza, ovvero algoritmi "on-policy" e "off-policy". Questa distinzione si basa sulla relazione tra le azioni che vengono prese dall'agente e i dati di esperienza che vengono utilizzati per l'apprendimento. Gli algoritmi on-policy si concentrano su come l'agente dovrebbe agire in base alla sua policy corrente, mentre contemporaneamente impara da nuovi dati di esperienza generati seguendo questa politica. In altre parole, l'agente esplora l'ambiente e raccoglie dati utilizzando la policy attuale e, in seguito, utilizza l'esperienza raccolta per aggiornare la propria policy. Esistono vari algoritmi appartenenti a tale famiglia; di seguito sono discussi i più rilevanti.

Monte Carlo Policy Gradient (REINFORCE)

Tale algoritmo è uno dei più semplici e può essere utilizzato soltanto nel caso in cui il problema di RL sia episodico. Le fasi che lo compongono sono le seguenti:

1. *Inizializzazione*: la policy viene inizializzata in maniera casuale.
2. *Svolgimento dell'episodio*: l'agente segue la policy fino al termine dell'episodio; esso interagisce con l'ambiente, osserva lo stato corrente, esegue un'azione, si sposta nello stato successivo ed ottiene un reward.
3. *Calcolo dei valori attesi*: terminato l'episodio, per ciascun passo effettuato, si calcola il ritorno cumulativo, cioè la somma delle ricompense future a partire dal passo corrente fino alla fine dell'episodio.
4. *Calcolo del gradiente*: per ciascun passo dell'episodio, si calcola il gradiente della log-probabilità dell'azione rispetto ai parametri della policy, in modo da misurare la variazione della probabilità di un'azione rispetto ai cambiamenti nei parametri della policy. Il gradiente viene, poi, moltiplicato per il ritorno cumulativo ottenuto dall'azione, in modo da tener conto di quanto quella particolare azione ha contribuito al successo dell'episodio.
5. *Aggiornamento dei parametri*: dopo aver calcolato i gradienti per tutti i passi se ne calcola la media ponderata. I parametri della policy vengono poi modificati in base al valore della media ponderata moltiplicata per un learning rate α compreso tra 0 ed 1.
6. *Loop*: dopo aver migliorato la policy si torna alla Fase 2 iterando fino alla convergenza.

Proximal Policy Optimization (PPO)

L'obiettivo di PPO è di migliorare una policy in modo che possa massimizzare le ricompense cumulative. In genere la policy è rappresentata da una rete neurale che riceve in input lo stato osservato e produce una distribuzione di probabilità sulle possibili azioni. Inoltre, PPO utilizza anche una value-function, che stima la ricompensa attesa data un certo stato, ciò velocizza la convergenza all'ottimo. I passi per l'addestramento sono i seguenti:

1. *Inizializzazione*: i valori della policy e della value-function vengono scelti casualmente
2. *Interazione*: l'agente interagisce con l'ambiente fino al termine dell'episodio raccogliendo tuple nel formato (stato, azione, ricompensa).
3. *Calcolo degli Advantage*: calcola gli "avanzamenti" (advantage) delle azioni in base alle ricompense ottenute e alle stime della value-function. Gli avanzamenti rappresentano quanto un'azione sia positiva o negativa rispetto alla media delle azioni in uno specifico stato.
4. *Calcolo della funzione obiettivo*: si misura quanto cambiano le probabilità delle azioni tra la policy attuale e quella precedente.
5. *Clipping del Rapporto di Probabilità*: viene effettuato il clipping della funzione obiettivo, in modo che gli aggiornamenti della policy siano limitati e controllati, evitando cambiamenti troppo drastici che potrebbero rendere l'apprendimento instabile.
6. *Ottimizzazione*: utilizzando Adam si ottimizza la policy utilizzando la loss calcolata e pesando la variazione in base ad un learning rate compreso tra 0 ed 1. Inoltre, viene anche ottimizzata la value-function calcolando la differenza tra stime delle ricompense e ricompense effettive ottenute.
7. *Loop*: dopo aver migliorato la policy si torna alla Fase 2 iterando fino alla convergenza.

SARSA

SARSA (State-Action-Reward-State-Action) è un algoritmo che mira a trovare una strategia ottimale per un agente che prende decisioni in un ambiente interagendo con esso nel tempo. L'obiettivo è imparare una politica che massimizzi le ricompense cumulative nel lungo termine. Esso fa uso della Q-function, la quale stima il valore atteso delle ricompense future per ogni coppia di stato-azione. I passi per l'addestramento sono i seguenti:

1. *Inizializzazione*: il valore della Q-value function viene inizializzato casualmente.
2. *Esplorazione*: l'agente interagisce con l'ambiente eseguendo azioni in base a una politica epsilon-greedy (già discussa nella Sottosezione 2.2.2). Esso osserva lo stato, sceglie un'azione, ottiene una ricompensa, osserva un nuovo stato e sceglie una nuova azione.
3. *Aggiornamento della Q-value function*: le informazioni raccolte vengono utilizzate per ottimizzare la funzione Q tramite la seguente formula:

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

   - $Q(s,a)$ è la stima attuale di Q per lo stato s e l'azione a.
   - α è il tasso di apprendimento; esso controlla quanto l'apprendimento viene aggiornato in base a nuove informazioni.

- r è la ricompensa ricevuta dall'agente dopo l'azione a nello stato s.
- γ è il fattore di sconto che rappresenta quanto l'agente considera importanti le ricompense future rispetto alle ricompense immediate.
- $Q(s', a')$ è la stima di Q per lo stato successivo s' e l'azione successiva a' secondo la politica di esplorazione.

4. *Loop*: dopo aver ottimizzato la Q-value function si ricomincia dalla Fase 2 fino alla convergenza.

SARSA è noto per essere stabile e appropriato per ambienti in cui l'esplorazione è importante. Tuttavia, a causa della sua natura on-policy, può essere meno efficiente nell'apprendimento rispetto agli algoritmi off-policy, come il Q-learning.

2.2.6 Algoritmi off-policy

Gli algoritmi off-policy consentono all'agente di apprendere da esperienze che non sono state generate seguendo la policy attuale; ciò migliora la generalizzazione.

Q-learning

L'obiettivo del Q-learning è apprendere la Q-value function $Q(s, a)$ utilizzando le conoscenze apprese durante l'esplorazione. Tale funzione associa ad ogni coppia stato-azione un ricompensa attesa; perciò, può essere facilmente rappresentata attraverso una tabella, detta Q-table, in cui le righe rappresentano gli stati e le colonne le azioni. L'algoritmo si compone delle seguenti fasi:

1. *Inizializzazione*: i valori della Q-table vengono inizializzati casualmente.
2. *Exploration ed Exploitation*: durante l'addestramento, l'agente interagise con l'ambiente scegliedo un'azione sulla base di una politica epsilon-greedy.
3. *Aggiornamento della Q-table*: dopo aver effettuato l'azione, l'agente osserva la ricompensa ottenuta e il nuovo stato in cui si trova. L'aggiornamento della Q-table avviene utilizzando la seguente formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [R + \gamma \cdot \max(Q(s', a')) - Q(s, a)]$$

   Dove:

   - $Q(s, a)$ è il valore Q corrente per lo stato s e l'azione a.
   - α è il tasso di apprendimento, che controlla la velocità di aggiornamento.
   - R è la ricompensa ottenuta prendendo l'azione a nello stato s.
   - γ è il fattore di sconto, che pondera l'importanza delle ricompense future.
   - $\max(Q(s', a'))$ rappresenta il valore Q massimo per tutte le possibili azioni nello stato successivo s'.

4. *Loop*: dopo aver aggiornato la Q-table l'agente torna alla Fase 2 iterando fino alla convergenza.

Con il tempo, i valori della Q-value function convergono ai valori ottimali, i quali rappresentano la massima ricompensa attesa che l'agente può ottenere partendo da uno stato specifico e seguendo la policy ottima. Tale algoritmo può essere utilizzato sia per problemi episodici che non.

Deep Q-learning

Il Deep Q-Network (DQN) è un algoritmo di apprendimento automatico che combina l'algoritmo Q-learning con le reti neurali profonde. Il principale problema nell'utilizzo di una Q-table consiste nell'impossibilità di rappresentare osservazioni continue. L'utilizzo delle reti neurali risolve tale problema consentendo, inoltre, una maggiore capacità di generalizzazione.

La rete neurale riceve in input lo stato corrente e restituisce come output i Q-value associati ad ogni azione. Inoltre, viene utilizzata una replay memory che contiene tuple (s, a, r, s'), dove s è lo stato corrente, a è l'azione scelta, r è la ricompensa ottenuta e s' è lo stato successivo. L'agente viene allenato utilizzando i dati nella replay memory, ciò riduce la correlazione tra le esperienze e rende l'apprendimento più stabile. Per rendere l'apprendimento più stabile, il DQN utilizza una seconda rete neurale (nota come "target network") per calcolare il termine $\max_a Q(s', a')$ nella formula dell'aggiornamento. Questa rete viene aggiornata meno frequentemente rispetto alla rete principale. L'algoritmo segue le seguenti fasi:

1. *Inizializzazione*: la Q-network e la target network vengono inizializzate casualmente.
2. *Esplorazione e Sfruttamento*: l'agente esplora l'ambiente eseguendo un'azione sulla base di una politica epsilon-greedy.
3. *Aggiornamento della Q-network*: durante l'addestramento, l'agente campiona un minibatch dalla replay memory e utilizza la seguente formula per aggiornare la rete neurale:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \Big(r + \gamma \cdot \max_a Q(s', a') - Q(s, a) \Big)$$

4. *Aggiornamento della Target Network*: la rete target non viene aggiornata tramite backpropagation, bensì, i pesi della Q-network vengono copiati nella target network periodicamente.
5. *Loop*: dopo aver aggiornato le reti neurali, l'agente torna alla Fase 2 e continua fino alla convergenza.

Il DQN ha dimostrato di essere efficace in molte applicazioni, soprattutto per quanto riguarda i giochi Atari, dove ha raggiunto o superato le prestazioni umane. L'utilizzo di reti neurali profonde consente di affrontare problemi più complessi e di generalizzare meglio tra diverse situazioni; per questo motivo, nello sviluppo del progetto di tesi, sarà utilizzato tale approccio.

CAPITOLO 3

Definizione dell'approccio

In questo capitolo verranno presentate le ottimizzazioni proposte dall'architettura SimpleViT. Successivamente, sarà illustrato il nostro approccio basato sul Reinforcement Learning.

3.1 Definizione del problema teorico

Come già detto nelle sezioni precedenti, lo scopo del presente elaborato consiste nell'applicare un algoritmo di Reinforcement Learning ad un vision transformer, al fine di diminuirne i tempi di addestramento. In particolare, si è deciso di utilizzare l'architettura SimpleViT (Beyer *et al.* [2022]), la quale ha delle prestazioni analoghe all'architettura originale, ma riduce i tempi di addestramento. Tale implementazione fa uso di un positional embedding 2D per la codifica dell'informazione relativa alla posizione delle patch e non utilizza meccanismi di dropout per la regolarizzazione dell'apprendimento. Inoltre, nel modello ViT originale, l'ultimo layer era un Multi Layer Perceptron n (MLP) il quale produceva le predizioni finali. Beyer et al. hanno dimostrato che un semplice strato lineare può funzionare in modo simile, senza una significativa perdita di prestazioni.

Nonostante la sostanziale diminuzione dei tempi di addestramento apportata dall'architettura SimpleViT, se le immagini vengono divise in un numero piuttosto elevato di patch può risultare necessario attendere diverse ore per portare a termine la fase di training. Infatti, come già discusso nella Sezione 1.2.2, per ogni strato di attention è necessario calcolare il coefficiente di attention di ogni token rispetto agli altri. Dato il numero di patch (N) e la dimensione dello spazio delle feature definito dall'utente (d), come si può osservare nella Figura 3.1, il layer di attention ha una complessità pari a $O(N^2)$.

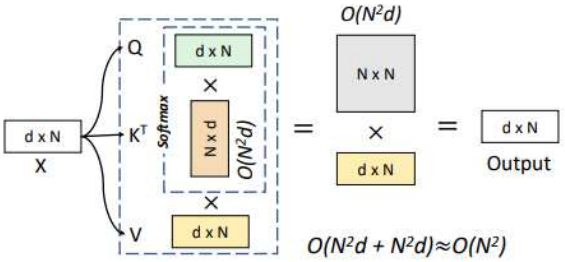


Figura 3.1: Complessità dello strato di attention

Un possibile metodo per diminuire la complessità dei ViT, perciò, consiste nel diminuire il numero di patch che la rete deve processare. Infatti, nel caso in cui il numero di patch è molto elevato, è probabile che molte di esse siano inutili per la corretta classificazione dell'immagine e causino un incremento della complessità computazionale. Negli anni sono stati sviluppati diversi approcci che hanno cercato di rendere più efficiente l'allenamento dei transformer diminuendo la complessità del calcolo dell'attention. Alcune tecniche sviluppate nell'ambito dell'NLP possono essere adattate anche ai ViT. Ad esempio, Child *et al.* [2019] hanno proposto un approccio per ridurre la complessità a $\mathcal{O}(n\sqrt{n})$ tramite il calcolo di valori di attention locali, i quali, combinati tra loro, approssimano il valore dell'attention globale. Esistono, poi, approcci sviluppati appositamente per l'utilizzo nell'ambito della Computer Vision (CV). Fayyaz *et al.* [2021] hanno proposto un modulo capace di ridurre progressivamente la complessità computazionale adattandosi al batch di immagini in input. In particolare, a valle di ogni strato che costituisce il transformer, è posizionato un layer capace di selezionare un sottoinsieme dei token in base al loro valore di attention. Renggli *et al.* [2022] hanno, invece, proposto un modulo che riduce il numero di patch che la rete deve elaborare, eseguendone il merge.

3.2 Definizione dell'approccio

L'approccio che si vuole proporre consiste nell'utilizzare una rete neurale a monte del ViT, al fine di selezionare un sottoinsieme delle patch durante l'addestramento. L'algoritmo che si vuole usare è il Deep Q-learning (DQL). Nella Figura 3.2 è rappresentato il ciclo di funzionamento dell'algoritmo applicato al problema in esame. Nell'algoritmo 1, invece, ne è stata data una rappresentazione in pseudocodice. Per ogni epoca e per ogni batch nel dataset di training, l'agente ha il compito di osservare lo stato corrente dell'environment e restituire un'azione, rappresentata attraverso una lista di N valori, dove N è il numero di patch in cui sono divise le immagini di input. Tale lista indica quali patch selezionare per allenare il SimpleViT; al termine del training, l'environment restituisce il nuovo stato ed il reward associato all'azione svolta dall'agente. In ultima istanza, la tupla $(state, action, new\_state, reward)$ viene utilizzata per migliorare le performance dell'agente. Tale processo va avanti fino al termine delle epoche di addestramento.

Nelle sottosezioni successive si discuterà approfonditamente riguardo la scelta dello stato, le possibili azioni e la formula per il calcolo del reward.

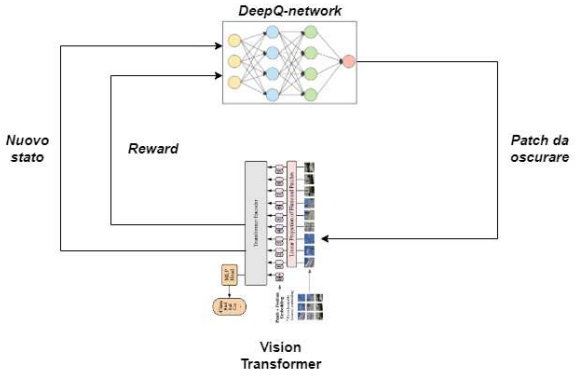


Figura 3.2: Schema di Reinforcement Learning con ViT

Algorithm 1: Pseudocodice relativo all'approccio proposto

```
foreach epoch do
    foreach batch in training dataset do
        state ← agent.get_state();
        action ← agent.get_action();
        vit.train(action);
        reward ← agent.get_reward();
        new_state ← env.get_new_state();
        agent.train(state, action, new_state, reward);
```

3.2.1 Stato

Nell'Reinforcement Learning (RL) lo stato osservato dall'agente è, in genere, codificato come un vettore di valori continui, che rappresentano la condizione in cui si trova l'environment. Nel nostro caso, inizialmente, avevamo utilizzato come stato il batch di immagini da dare in input al SimpleViT. Per questo motivo, l'agente doveva essere implementato attraverso una rete neurale convoluzionale, capace di osservare il batch di immagini e restituire la giusta azione. Tale scelta si è poi rilevata infruttuosa, in quanto una rete convoluzionale con pochi strati non restituiva risultati soddisfacenti, mentre una rete a più strati forniva buoni risultati, ma aumentava la complessità computazionale, vanificando l'utilizzo del framework da noi proposto. Per tale motivo, si è deciso di utilizzare come stato il valore di attention ottenuto facendo processare il batch di immagini al primo layer del SimpleViT. Data un'immagine divisa in N patch, l'ouptut del primo layer di attention è una matrice $N \times d$, dove d è la dimensione del vettore di feature definito dall'utente. Tale matrice viene, poi, trasformata in un vettore di N elementi, calcolando la media dei valori lungo la dimensione d. Il risultato di questa operazione è lo stato osservato dall'agente e rappresenta il valor medio dell'attention associata ad ogni patch.

Abbiamo deciso di utilizzare tale vettore come stato poiché fornisce all'agente una rappresentazione compatta, ma molto significativa, dell'ambiente. Infatti, l'attention associata al primo layer del SimpleViT comprende le informazioni relative alle immagini in input e allo stato di addestramento del transformer. Inoltre, avendo codificato lo stato con un singolo vettore di valori continui, possiamo rappresentare l'agente attraverso una semplice rete MLP, diminuendo anche il costo computazionale del framework proposto.

3.2.2 Azione

L'agente ha il compito di osservare lo stato e restituire una lista di dimensione pari al numero di patch in cui sono divise le immagini utilizzate per addestrare il SimpleViT. Il vettore restituito è costituito da valori continui, i quali, in affinità con l'algoritmo DQL, rappresentano il reward cumulativo stimato dall'agente. Perciò, gli indici in corrispondenza dei quali osserviamo i valori più alti, rappresentano le patch più significative. Nel classico algoritmo di Q-learning l'agente sceglie l'indice associato al reward cumulativo più alto ed aggiorna il valore della Q-function utilizzando la seguente formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [R + \gamma \cdot \max(Q(s',a')) - Q(s,a)]$$

Nel nostro caso, invece, vengono selezionati tutti gli elementi del vettore il cui valore è maggiore della media. In tal modo sono selezionate tutte le azioni con un reward cumulativo più alto, cioè tutte le patch più promettenti. La formula utilizzata per l'aggiornamento è

identica a quella mostrata in precedenza, ma l'agente viene ottimizzato una volta per ogni indice selezionato.

Per l'aggiornamento dei parametri dell'agente è stata utilizzata la SmoothL1Loss, la quale calcola la distanza tra $\max(Q(s', a'))$ e $Q(s, a)$. La formula della SmoothL1Loss messa a disposizione da pytorch è la seguente:

$$\mathcal{L}(x, y) = L = \{\ell_1, \ldots, \ell_N\}^T \tag{3.2.1}$$

$$\ell(x, y) = \begin{cases} 0.5\left(|x_n - y_n|^2/\beta^2\right), & \text{if } |x_n - y_n| < \beta \\ |x_n - y_n| - 0.5 \cdot \beta, & \text{otherwise} \end{cases} \tag{3.2.2}$$

$$\ell(\mathcal{L}) = \text{mean}(\mathcal{L}) \tag{3.2.3}$$

Si è scelto di utilizzare questa loss, in quanto soffre in maniera minore la presenza di outlier rispetto all'MSE e, in alcuni casi, previene problemi di esplosione del gradiente.

3.2.3 Reward

Il reward deve necessariamente tenere conto dell'andamento dell'allenamento del SimpleViT e del tempo di training. Si è deciso, perciò, di calcolarlo come una somma pesata di due quantità che tengono conto della loss di training e del numero di patch selezionate dall'agente. Il calcolo del reward è effettuato come definito di seguito:

$$\text{loss\_reward} = \frac{\text{initial\_train\_loss}}{\text{train\_loss}} \tag{3.2.4}$$

$$\text{patches\_reward} = -\frac{|\text{num\_zeros} - \text{ideal\_zeros}|}{\text{ideal\_zeros}} \tag{3.2.5}$$

$$\text{reward} = \text{loss\_reward} \times \text{loss\_weight} + \text{patches\_reward} \times \text{patch\_weight} \tag{3.2.6}$$

dove:

- *initial_train_loss* è il valore della loss di training della prima iterazione;
- *train_loss* è la loss di training dell'iterazione corrente;
- *num_zeros* è il numero di patch non selezionate dall'agente per l'iterazione corrente;
- *ideal_zeros* è il numero ideale di patch non selezionate; può essere definito dall'utente, ma di default è valorizzato ad 1;
- *loss_weight e patch_weight* sono due parametri definiti dall'utente.

Da questa formula si può osservare che l'agente è incentivato a scegliere un numero di patch vicino al valore definito dall'utente, oppure un numero molto piccolo, se l'utente non l'ha specificato. Inoltre, esso è anche incentivato a scegliere un sottoinsieme delle patch tale da minimizzare la loss di training. Entrambi i valori sono pesati, in modo che l'utente possa specificare quanta importanza dare a ciascuna di queste quantità. Ovviamente, un alto valore di loss_weight fa si che l'agente scelga un numero di patch abbastanza elevato; invece, un valore di patch_weight alto fa si che l'agente scelga un numero molto ridotto di patch, oppure, se specificato dall'utente, un numero di patch vicino al valore definito.

3.3 Tecnologie utilizzate

Per la risoluzione del problema proposto si è deciso di far uso del linguaggio di programmazione python. Le principali librerie utilizzate sono:

- `torch`: utilizzata per creare la rete neurale che rappresenta l'agente.
- `vit_pytorch`: fornisce le implementazioni delle varie architetture di tipo vision transformer.
- `gym`: mette a disposizione metodi e classi per la gestione dell'environment.
- `numpy`: libreria per il calcolo scientifico.
- `matplotlib`: utilizzata per analizzare, tramite grafici, i risultati ottenuti.

L'ambiente di programmazione utilizzato è Colab, in quanto esso fornisce risorse utili per l'addestramento intensivo di reti neurali ed, inoltre, consente di monitorare in maniera sufficientemente accurata i tempi di esecuzione dei programmi.

CAPITOLO 4

Implementazione del prototipo

In questo capitolo verranno descritti i moduli software che compongono il framework da noi proposto.

4.1 Vision Transformer

Come già detto nel capitolo precedente, l'architettura utilizzata è la SimpleViT, poiché garantisce dei tempi di training piuttosto bassi. Tale architettura deve, però, essere modificata al fine di adattarla ai nostri scopi. Nelle sottosezioni successive saranno discusse tali modifiche.

4.1.1 Metodo Set_patches()

Tale metodo ha il compito di valorizzare il parametro `patches` del ViT. In particolare, esso riceve una lista di N valori continui, i quali rappresentano, per ogni patch, il reward cumulativo. A questo punto calcola la media e, successivamente, crea una nuova lista di N valori binari in cui nella posizione i_esima è presente 0 se il valore nella lista in input è minore della media, 1 altrimenti. In questo modo le patch selezionate saranno soltanto quelle con reward cumulativo superiore alla media, cioè quelle più significative. Lo pseudocodice relativo al metodo `set_patches(patch)` viene riportato nell'Algoritmo 2.

Algorithm 2: set_patches(patch)

```
1 mean = torch.mean(patch);
2 selected = [0 if elem < mean else 1 foreach elem in patch];
3 self.patches = selected;
```

4.1.2 Metodo Get_att()

Tale metodo verrà utilizzato all'interno dell'environment per ottenere lo stato corrente, il quale è rappresentato dal valore di attention del batch di immagini di training. Il metodo, perciò, riceve il batch di immagini e restituisce il valore di attention del primo strato del ViT. Il vettore restituito dallo strato di attention ha dimensione pari a $N \times d$, dove N è il numero di patch e d è un parametro definito dall'utente. Si è deciso di calcolarne la media al fine di ottenere un vettore di dimensione N, che rappresenta, per ogni patch, il valore di attention

medio. Lo pseudocodice relativo al metodo `get_att(img)` viene riportato nell'Algoritmo 3.

Algorithm 3: get_att(img)

```
*_, h, w, dtype = *img.shape, img.dtype ;
x = self.to_patch_embedding(img) ;
pe = posemb_sincos_2d(x) ;
x = rearrange(x, b' ... d') + pe ;
attn, ff = self.transformer.layers[0] ;
x = attn(x) ;
x = torch.mean(x, dim=2) ;
return x ;
```

4.1.3 Metodo Forward()

Nella logica di `pytorch`, il metodo `forward()` è chiamato quando la rete neurale viene utilizzata in fase di previsione. Nel nostro caso è stato necessario modificare tale metodo in modo da oscurare le patch non significative. Per far ciò è stata creata una maschera a partire dalla lista di patch definita tramite il metodo `set_patches(patch)` discusso in precedenza. Lo pseudocodice relativo alla maschera viene riportato nell'Algoritmo 4.

Algorithm 4: Pseudocodice relativo alla maschera

```
// codice precedente
mask = torch.tensor(self.patches, dtype=torch.bool) ;
x = x[:, mask, :] ;
// codice successivo
```

4.2 Environment

L'environment costituisce una delle componenti fondamentali di un problema di Reinforcement Learning. L'agente, infatti, si interfaccia con esso per osservare gli stati e ricevere i reward. L'environment da noi proposto è un'implementazione dell'interfaccia fornita da OpenAI tramite la libreria `gym` di Python.

4.2.1 Classe MultiContinue

Tale classe è stata utilizzata per definire lo spazio delle azioni. Infatti, essa mette a disposizioni il metodo `sample`, il quale fornisce una lista di N valori continui, dove N è il numero di patch in cui sono divise le immagini di training. Lo pseudocodice relativo al metodo `sample()` viene riportato nell'Algoritmo 5.

Algorithm 5: sample()

```
action = np.random.rand(self.n_patch);
return torch.tensor(action, dtype=torch.float);
```

4.2.2 Classe ViTEnv

Tale classe rappresenta l'environment vero e proprio. Al suo interno è definito lo spazio delle azioni come istanza della classe `MultiContinue`. L'environment deve occuparsi di gestire la successione degli stati e i reward ottenuti in seguito alle azioni eseguite dall'agente. Per far ciò, esso mette a disposizione i seguenti metodi:

- `step_train()`: riceve il batch di immagini e allena il ViT.
- `step_test()`: riceve il batch di immagini, allena il ViT e calcola il reward associato. Il calcolo del reward è eseguito come mostrato nell'Algoritmo 6.

Algorithm 6: Calcolo reward

```
1 loss_reward = (self.train_loss_list[0]/train_loss);
2 ideal_zeros = len(action) - self.n_patch_selected;
3 patches_reward = (- abs(num_zeros - ideal_zeros) / ideal_zeros);
4 reward = loss_reward × self.loss_weight + patches_reward × self.time_weight;
```

  Il reward dipende da due quantità, una rappresenta il guadagno in termini di loss di training, l'altra il guadagno in termini di patch selezionate. Per quanto riguarda la `loss_reward` essa è data dal rapporto tra la loss della prima epoca e quella nell'epoca corrente. La `patch_reward` viene calcolata come il rapporto tra il numero di patch non selezionate ed il valore ideale di zeri. Tale valore può essere deciso dall'utente in sede di inizializzazione. Il reward finale è dato da una loro somma ponderata, i cui pesi sono definiti dall'utente. Utilizzando un reward di questo tipo, l'agente tenderà a privilegiare un sottoinsieme di patch il cui numero si avvicini il più possibile al valore definito dall'utente, tenendo però conto anche del guadagno di loss durante le epoche.

- `get_state()`: metodo che riceve un batch di immagini e ne ottiene il valore di attention chiamando il metodo `get_att()` del ViT.

È stato necessario definire i metodi `step_train()` e `step_test()`, poiché, come vedremo in seguito, l'allenamento dell'agente non avverrà ad ogni iterazione. Infatti sarà possibile definire ogni quante batch di training del ViT eseguire il fine tuning dell'agente, e solo in questo caso sarà necessario calcolare il reward. Il motivo di tale scelta è che un allenamento continuo dell'agente potrebbe appesantire il framework, aumentando i tempi di addestramento.

4.3 Agente

L'agente che si è deciso di utilizzare, come già detto più volte, è basato sul Deep Q-learning. Sono state introdotte alcune ottimizzazioni tipiche di questa tipologia di algoritmo, come, ad esempio, una *replay memory* ed una *target network*. Nelle sottosezioni seguenti saranno descritte le corrispettive implementazioni.

4.3.1 Classe QNetwork

L'agente è rappresentato da una rete neurale con due strati, uno di 1024 neuroni e l'altro di 256. L'input della rete neurale è un vettore di valori di attention di N elementi, mentre l'output è anch'esso un vettore di N elementi, i quali rappresentano il reward cumulativo associato ad ogni patch.

4.3.2 Classe Replay Memory

L'obiettivo della replay memory consiste nell'immagazzinare una serie di tuple con formato $(state, new\_state, reward)$. In tal modo, quando l'agente deve essere allenato, viene estratto un batch casuale da questa memoria e viene utilizzato per il training; ciò consente all'agente di allenarsi anche su esperienze passate, migliorando la generalità. La replay memory è stata implementata come un vettore di dimensione finita ad accesso ed eliminazione casuali. In tal modo, quando un nuovo elemento deve essere inserito, se non ci sono posizioni libere, viene selezionato un elemento casuale, il quale viene rimpiazzato dal nuovo elemento.

4.3.3 Target Network

Per migliorare la stabilità del modello si è deciso di utilizzare una target network, cioè una rete neurale identica all'agente. Essa, però, non viene allenata durante l'avanzare delle epoche, bensì i suoi pesi vengono periodicamente copiati dall'agent-network. La copia è di tipo *soft*, cioè i pesi non vengono totalmente sostituiti, bensì l'adattamento dipende da un parametro definito dall'utente. Il codice utilizzato per l'update è rappresentato nella sottosezione successiva.

4.3.4 Classe DQNAgent

Tale classe rappresenta l'agente vero e proprio; essa, infatti, istanzia due reti neurali, le quali rappresentano l'agent-network e la target-network. Inoltre, mette a disposizione una serie di metodi che consentono di scegliere le azioni e ottimizzare l'agente. Tali metodi sono:

- `select_action()`: riceve in input uno stato e restituisce una lista di N valori continui, i quali rappresentano il reward cumulativo stimato dall'agente. Tale metodo è stato implementato utilizzando la logica ϵ-greedy: l'agente sceglie con probabilità pari ad ϵ una lista di N valori casuali, altrimenti lo stato viene passato all'agent network e viene restituito l'output fornito dalla rete.
- `optimize_model()`: estrae un batch casuale dalla replay memory. L'insieme di stati correnti viene passato attraverso l'agent-network, mentre gli stati successivi passano attraverso la target-network. In seguito viene calcolato il valore dell'`expected_state_action_values`. La loss è calcolata come la distanza tra `state_action_values` ed `expected_state_action_values`. Al termine, se si è raggiunto il numero opportuno di iterazioni, viene effettuato il soft update della target-network.

Lo pseudocodice relativo all'aggiornamento dell'agente viene riportato nell'Algoritmo 7, mentre quello relativo al *soft update* della target network viene mostrato nell'Algoritmo 8.

Algorithm 7: Ottimizzazione dell'agente

```
1 state_action_values = self.q_network(state_batch);
2 next_state_values = self.target_network(new_state_batch);
3 expected_state_action_values = (next_state_values ×γ) + reward_batch;
4 criterion = nn.SmoothL1Loss();
5 loss = criterion(state_action_values, expected_state_action_values);
```

Algorithm 8: Soft update della target network

```
target_network_state_dict = self.target_network.state_dict();
q_network_state_dict = self.q_network.state_dict();
for key in q_network_state_dict do
    target_network_state_dict [key] = q_network_state_dict [key] × self.tau +
     target_network_state_dict [key] × (1 - self.tau);
self.target_network.load_state_dict(target_network_state_dict);
```

4.4 Classe TrainingTestingAgent

Tale classe gestisce il ciclo di training del ViT e restituisce le informazioni relative all'andamento dell'allenamento durante le epoche. Essa fornisce i seguenti metodi pubblici:

- `train_info()`: fornisce informazioni relative alla loss e al tempo di train.
- `test_info()`: fornisce le informazioni relative alla loss e all'accuracy di test.
- `train_test()`: allena il ViT per un numero di epoche definito dall'utente. Come si può osservare nell'Algoritmo 9, per ogni epoca si effettua un ciclo `for` sulle batch nel `train_loader`. Viene chiamato il metodo `get_attention()` della classe ViT, ottenendo il valore di attention associato al batch di immagini. Viene invocato il metodo `select_action` dell'agente, il quale, osservato il valore di attention, restituisce l'azione opportuna. Se non è stato raggiunto il numero opportuno di iterazioni viene invocato il metodo step_train; altrimenti, viene invocato il metodo `step_test`, sono inserite le tuple $(state, next\_state, reward)$ nella replay memory e si ottimizza l'agente. In ultima istanza, viene aggiornato il valore di epsilon con un decadimento esponenziale tenendo conto del valore iniziale, di quello finale e del coefficiente di decadimento definito dall'utente.

Algorithm 9: Ciclo di esecuzione

```
while epoca < epoche do
    epoca++;
    iteration++;
    for (data, target) in train_loader do
        state = viT.get_attention(data);
        action = agent.select_action(state, eps);
        if iteration mod get_reward_every  != 0 then
            env.step_train(action, data, target);
        else
            new_state, reward = env.step_test(action, data, target, validation_loader);
            replay_memory.push(state, new_state, reward);
            agent.optimize_model();
        eps = eps_end + (eps_start - eps_end) * math.exp(-1. * iteration / eps_decay);
```

CAPITOLO 5

Campagna sperimentale

In questo capitolo saranno presentati i risultati ottenuti tramite l'utilizzo del nostro approccio. In particolare verranno effettuate due tipologie di analisi. La prima analisi è quantitativa e si sofferma sulla durata delle epoche e sulle metriche di performance ottenute dal nostro approccio rispetto ad altri modelli, come il SimpleViT, l'ATSViT ed il ViT originale. In seguito sarà effettuata un'analisi qualitativa osservando le immagini originali e quelle mappate dall'agente; l'obiettivo consiste nel valutare le scelte effettuate dall'agente ed individuare eventuali criticità.

5.1 Esperimenti sul dataset CIFAR10

In questa sezione verrà effettuato un confronto tra il framework da noi proposto ed una serie di altri modelli. Il dataset utilizzato è CIFAR10 (Krizhevsky *et al.* [2009]), il quale è composto da un totale di 60.000 immagini RGB di dimensione 32×32, divise in 10 classi. Tale dataset è suddiviso in due insiemi: uno di addestramento ed uno di test. Il set di addestramento contiene 50.000 immagini, mentre il set di test ne contiene 10.000. CIFAR10 è stato progettato in modo che ogni categoria abbia lo stesso numero di immagini nel set di addestramento, quindi ogni classe contiene esattamente 5.000 immagini. Si è deciso di utilizzare tale dataset poiché la bassa risoluzione delle immagini rappresenta una sfida per molti algoritmi di Computer Vision (CV). Inoltre, CIFAR10 è un dataset con una numerosità estremamente minore rispetto ad altri (ad esempio, ImageNet); ciò consente di effettuare il testing dei modelli anche senza avere un intero cluster a disposizione.

5.1.1 Definizione dei modelli utilizzati

I modelli utilizzati sono i seguenti:

- SimpleViT64, i cui parametri utilizzati sono:
  - `patch_size = 4×4;`
  - `dim = 128;`
  - `depth = 6;`
  - `heads = 16;`
  - `mlp_dim = 512;`

- SimpleViT16: come il precedente, ma è stato allenato con delle patch di dimensione 8×8, dividendo l'immagine in 16 regioni, anziché 64.
- ATSViT: implementazione dell'approccio proposto da Fayyaz *et al.* [2021]; i parametri utilizzati sono:
  - `patch_size = 4×4;`
  - `dim = 128;`
  - `depth = 6;`
  - `max_tokens_per_depth = (64, 32, 16, 8, 4, 2);`
  - `heads = 16;`
  - `mlp_dim = 512;`
  - `dropout = 0.1;`
  - `emb_dropout = 0.1;`
- ViT: implementazione originale proposta da Dosovitskiy *et al.* [2021]; in questo caso sono stati utilizzati i seguenti parametri:
  - `patch_size = 4×4;`
  - `dim = 128;`
  - `depth = 6;`
  - `heads = 16;`
  - `mlp_dim = 512;`
  - `dropout = 0.1;`
  - `emb_dropout = 0.1;`
- AgentViT: implementazione del nostro approccio basato su RL; fa uso di un modello analogo al SimpleViT64 al quale sono aggiunti i parametri utili a regolare il comportamento dell'agente:
  - `buffer_batch_size = 8;`
  - `buffer_size = 64;`
  - `gamma = 0.95;`
  - `eps_start = 1;`
  - `eps_end = 0.01;`
  - `eps_decay = 20000;`
  - `lr = 0.01;`
  - `tau = 0.1;`
  - `update_every = 2;`
  - `get_reward_every = 50;`
  - `n_patch_selected = 30;`
  - `time_weight = 20;`
  - `loss_weight = 1;`

- AgentViT_V2: modello analogo al precedente; in esso sono stati utilizzati gli stessi parametri eccetto i seguenti:
  - `buffer_batch_size = 32;`
  - `buffer_size = 256;`
  - `get_reward_every = 10.`

  Questa seconda versione rappresenta un agente che viene aggiornato più spesso, poiché il ciclo di ottimizzazione avviene una volta ogni 10 batch (get_reward_every), anziché una ogni 50. Inoltre, è stata incrementata la dimensione della replay memory (buffer_size) e del numero di elementi (buffer_batch_size) che l'agente estrae da essa durante l'ottimizzazione. Ciò dovrebbe far si che l'agente sia più capace di generalizzare, a discapito di un lieve incremento dei tempi di esecuzione.

Tutti i modelli sono stati allenati per 100 epoche, con un learning_rate di 0.001 e con ottimizzatore Adam.

5.1.2 Confronto tra i modelli

Nella Figura 5.1 sono rappresentati gli andamenti dell'accuracy e della durata delle epoche per i modelli AgentViT e SimpleViT64. Le performance sono pressoché comparabili, ma il nostro approccio tende a migliorare moderatamente i tempi di addestramento; ciò è dovuto al fatto che AgentViT deve processare solo un sottoinsieme delle patch. In media, nelle ultime 20 epoche, AgentViT tenda a selezionare 30 ± 5 patch, garantendo un risparmio considerevole. Da tale analisi si è pensato di confrontare le performance del nostro modello con quelle di un SimpleViT allenato con un numero minore di patch. Nella Figura 5.2 è rappresentato il confronto tra il nostro modello e il SimpleViT16, il quale è stato allenato con 16 patch di dimensione 8×8. Si può notare che, nonostante i tempi di addestramento siano di poco inferiori, l'accuratezza risulta essere molto degradata rispetto all'approccio da noi proposto.

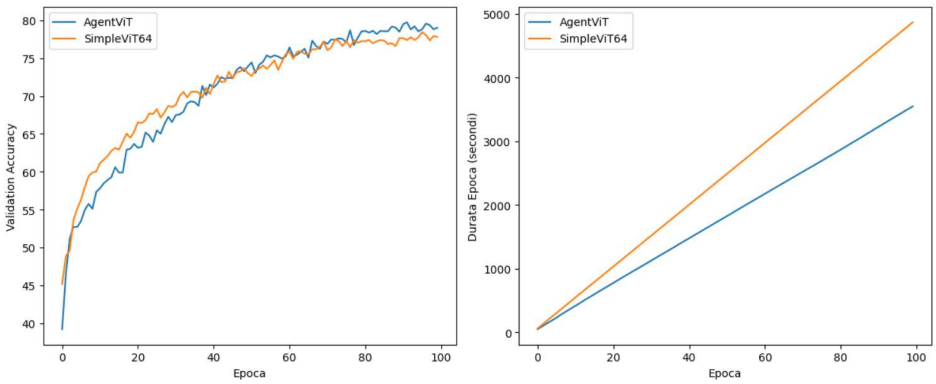


Figura 5.1: Confronto tra AgentViT e SimpleViT64

Nella Figura 5.3 è possibile osservare che il nostro approccio risulta superiore a quello proposto da Fayyaz *et al.* [2021]. Un miglioramento così netto dipende, probabilmente, dal fatto che ATSViT è indicato per l'utilizzo con modelli di grandi dimensioni e con un alto numero di parametri. Non è stato effettuato un test su una rete di questo tipo poiché l'allenamento necessiterebbe di un gran numero di risorse, le quali, al momento della stesura del presente progetto, non sono disponibili.

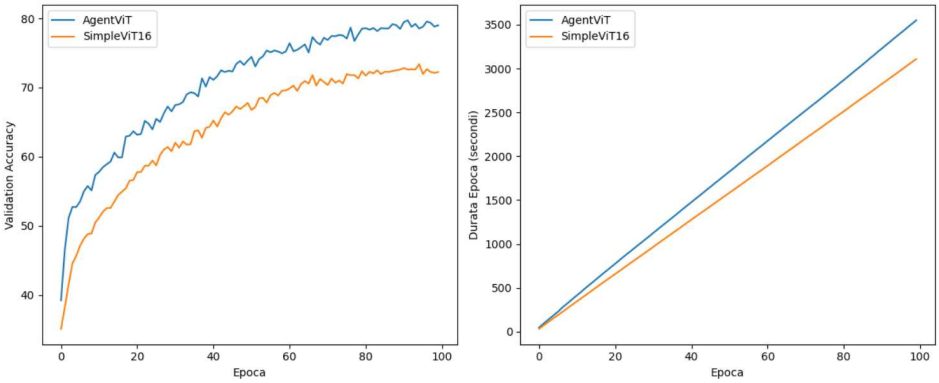


Figura 5.2: Confronto tra AgentViT e SimpleViT16

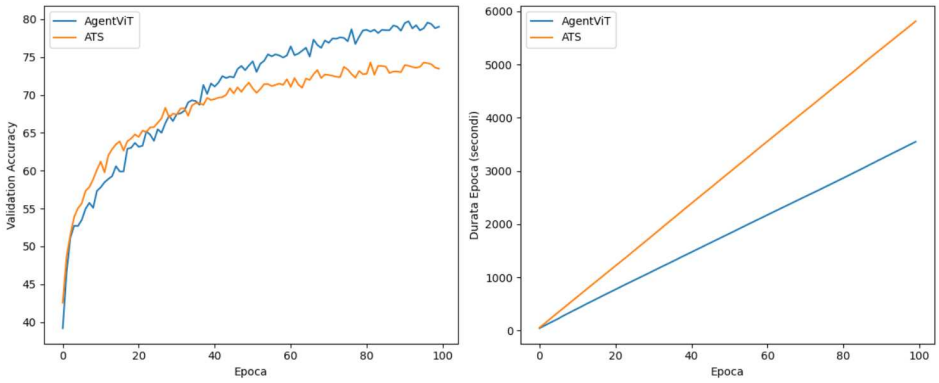


Figura 5.3: Confronto tra AgentViT e ATSViT

Nella Figura 5.4 si può osservare il confronto tra il nostro approccio e l'architettura ViT originale, proposta da Vaswani et al. Ovviamente il nostro approccio ha prestazioni peggiori, ma richiede dei tempi di addestramento quasi dimezzati. Sarebbe interessante applicare il nostro approccio direttamente all'architettura ViT originale, per valutare il guadagno in termini di tempo di esecuzione. Tale analisi, però, esula dagli obiettivi del presente progetto di tesi, in quanto necessiterebbe di modificare, anche se lievemente, l'architettura ViT.

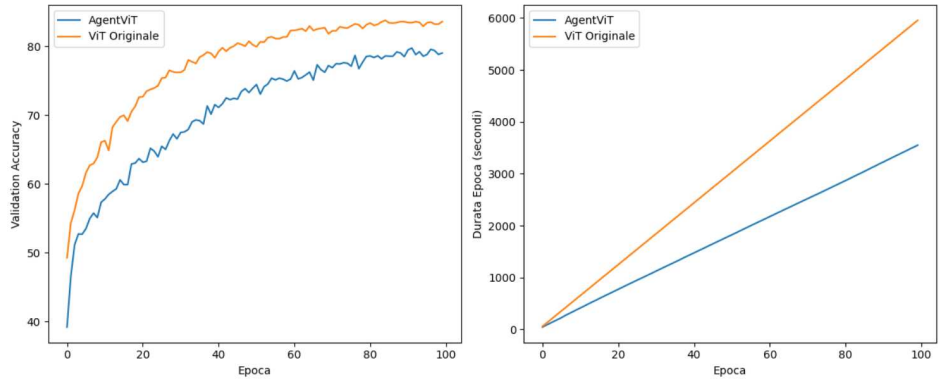


Figura 5.4: Confronto tra AgentViT e ViT

Nella Figura 5.5 sono messi a confronto gli andamenti di accuracy e durata delle epoche per quanto riguarda i modelli AgentViT e AgentViT_V2. Come già detto, AgentViT_V2 fa uso di una replay memory di dimensioni maggiori e allena l'agente più frequentemente; ciò si ripercuote sui tempi di addestramento, i quali risultano di poco maggiori, ma, in compenso,

il modello risulta lievemente più performante; infatti, esso converge più velocemente ad un'accuratezza che si attesta intorno al 79%.

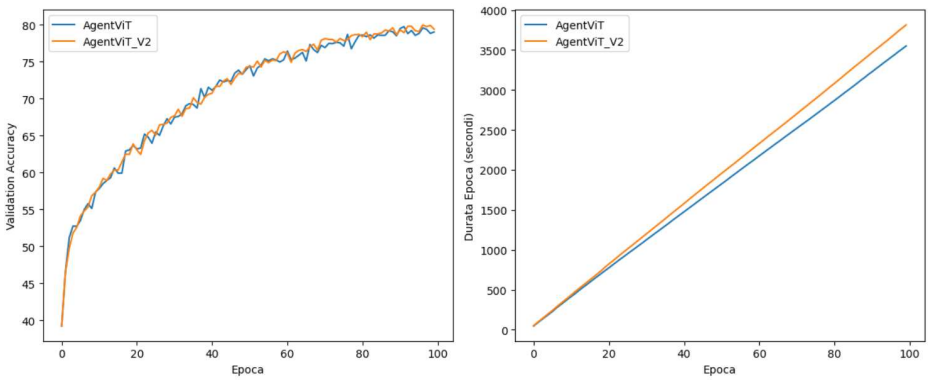


Figura 5.5: Confronto tra AgentViT e AgentViT_V2

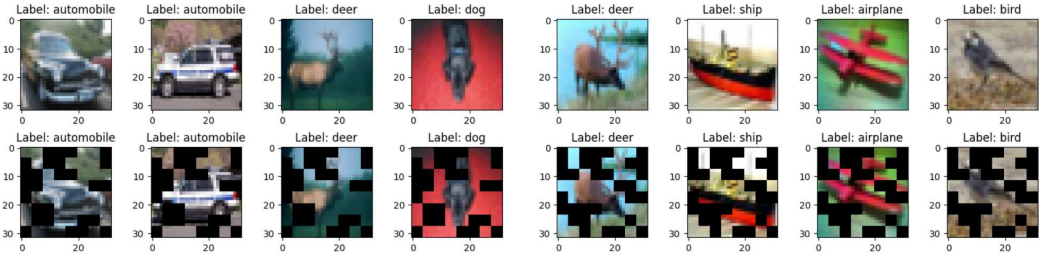
Infine, nella Tabella 5.1 sono rappresentati, per ogni modello, i tempi medi per epoca e le metriche di performance ottenute sul dataset di validation. Si può facilmente notare che il nostro approccio garantisce il miglior compromesso tra la durata delle epoche di addestramento e la correttezza delle previsioni. Inoltre l'utilizzo di un agente più complesso (AgentViT_V2) non apporta alcun vantaggio significativo rispetto ad un agente più semplice (AgentViT). Infatti, come già detto, il lieve guadagno di accuratezza ed f1_score non giustifica l'aumento dei tempi di training.

	ViT	ATSViT	SimpleViT64	AgentViT_V2	AgentViT	SimpleViT16
Tempo	59.55	58.13	48.70	38.14	33.73	31.08
Accuratezza	83.77	74.29	78.44	79.96	79.74	73.38
Precision	81.36	73.24	80.17	79.71	81.04	73.55
Recall	85.71	75.44	78.57	81.09	79.53	73.17
F1	83.48	74.32	79.36	80.39	80.27	73.35

Tabella 5.1: Tempo di esecuzione medio e valore delle metriche per ogni modello.

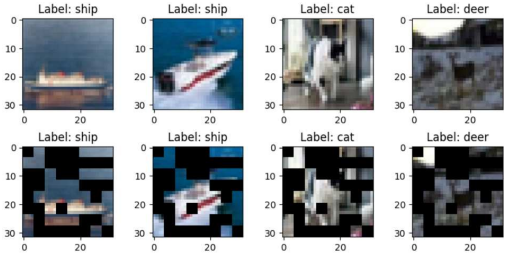
5.1.3 Analisi qualitativa

Nella Figura 5.6 sono rappresentate alcune delle immagini del dataset CIFAR10, raggruppate in batch di dimensione 4 e alle quali è stata applicata la maschera da parte dell'agente. È rilevante ricordare che l'agente riceve una batch di immagini e restituisce una maschera da applicare a tutte le immagini della batch. Ciò implica che l'agente cerca di selezionare le patch che risultano irrilevanti per gran parte delle immagini in input. Perciò è possibile che, se la batch in input contiene molte immagini, l'agente selezioni un insieme di patch che si adattano bene a molte delle immagini, ma meno a un sottoinsieme ridotto di esse. Ad esempio, se l'agente osserva una batch di 4 immagini in cui una patch rappresenta lo sfondo per tre di esse, probabilmente tenderà ad eliminarla, anche se dovesse risultare rilevante per la quarta immagine. Ciò accade, ad esempio, nella Sottofigura 5.6f, in cui vengono eliminate delle patch non significative per tre delle immagini, ma tale mapping rende difficile la classificazione della seconda immagine, poiché il cervo viene quasi totalmente oscurato. Perciò il nostro approccio, come anche gli altri descritti in questo capitolo, è consigliato nel caso in cui si abbia una batch_size ridotta.

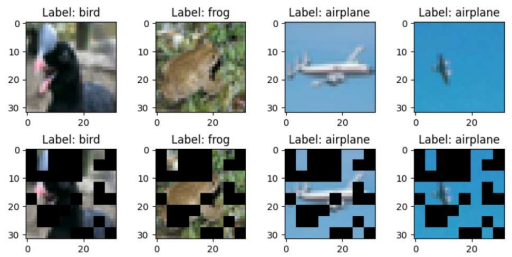


(a) Batch 1

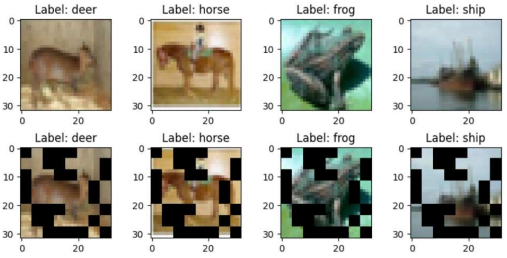
(b) Batch 2



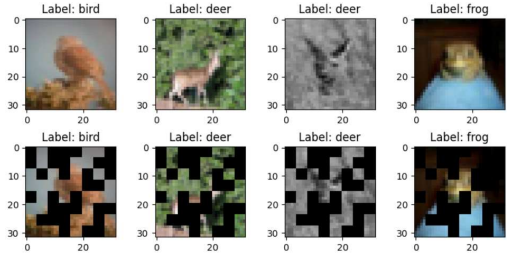
(c) Batch 3



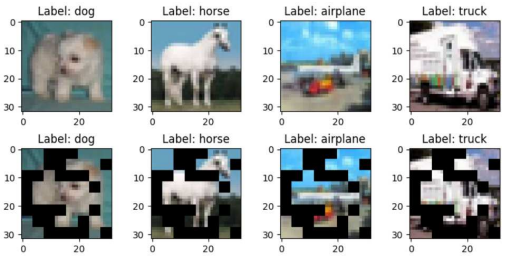
(d) Batch 4



(e) Batch 5



(f) Batch 6



(g) Batch 7

Figura 5.6: Esempio di mapping da parte dell'agente

CAPITOLO 6

Discussione in merito al lavoro svolto

In questo capitolo saranno descritti in maniera accurata gli approcci già presenti in letteratura. Inoltre, sarà effettuato un confronto qualitativo con il framework da noi proposto al fine di valutarne il livello di innovatività e di utilizzabilità.

6.1 Stato dell'arte

Come già discusso nella Sezione 3.1, in letteratura esiste un ampio numero di approcci che cercano di diminuire i tempi di addestramento dei modelli Vision Transformer (ViT). Queste tecniche posso essere divise in due macrogruppi: quelle nate nel mondo del Natural Language Processing (NLP) e quelle sviluppate specificatamente per l'ottimizzazione dei ViT. Nel primo gruppo rientrano le tecniche di quantizzazione (Wang *et al.* [2018], Gong *et al.* [2014]), il pruning (He *et al.* [2017], Rao *et al.* [2018]), la fattorizzazione a basso rango (Yu *et al.* [2017], Jaderberg *et al.* [2014]) e la distillazione della conoscenza (Hinton *et al.* [2015], Liu *et al.* [2020], Wang *et al.* [2020]). Ad esempio, l'approccio proposto da Guo *et al.* [2022] consiste nel sostituire la struttura completamente connessa, tipica del classico meccanismo di attention, con una topologia a stella, in cui ogni coppia di nodi non adiacenti è collegata attraverso un nodo condiviso. TinyBERT (Jiao *et al.* [2019]) ha introdotto un meccanismo di distillazione della conoscenza, attraverso il quale un modello di dimensione minore apprende cercando di replicare le predizioni di un modello preaddestrato e di dimensioni maggiori. Queste tecniche possono essere applicate anche al mondo dei ViT, garantendo una diminuzione della complessità; tuttavia, non essendo state progettate specificatamente per la Computer Vision necessitano di una serie di modifiche.

Nel secondo gruppo ricadono gli approcci sviluppati direttamente per i ViT. Rao *et al.* [2021] hanno proposto DynamicViT (DyViT), una particolare versione di vision transformer alla quale sono stati aggiunti dei blocchi capaci di ridurre il numero di token da processare. Tale approccio fa uso di una serie di strati, i quali vengono allenati durante i cicli di training del ViT. Questi strati sono posti a valle degli attention layer e vanno a selezionare un sottoinsieme dei token da utilizzare negli strati più in profondità. Il numero di token deve essere fissato dall'utente, ciò può risultare svantaggioso, poiché si rischia di utilizzare pochi token in immagini complesse, o troppi token in immagini particolarmente semplici. Expediting Vision Transformer (EViT) è un approccio proposto da Liang *et al.* [2022]; esso utilizza gli attention score per valutare l'importanza di ogni singolo token. Per ridurre la complessità, i token a cui è associato un valore basso sono eliminati o fusi tra loro. L'Adaptive Token Sampling

Vision Transformer (ATSViT) è una variante dei vision transformer proposta da Fayyaz *et al.* [2022]. Essa sfrutta l'importanza dei token per diminuire la complessità computazionale. In particolare, dopo aver calcolato una metrica specifica basata sul valore di attention utilizzano un campionamento con rimpiazzo in cui ogni token ha una probabilità di essere scelto pari al valore della metrica. La principale differenza rispetto agli approcci descritti precedentemente sta nel fatto che i layer che riducono il numero di token sono privi di iperparametri; perciò non è necessario allenarli. Inoltre, il numero di token utilizzati dipende dal batch di immagini in input e non è staticamente definito dall'utente. Ciò garantisce una maggiore flessibilità.

6.2 Confronto qualitativo

Nella Tabella 6.1 sono rappresentate le principali caratteristiche del nostro approccio e di quelli descritti nella sezione precedente. AgentViT, a differenza degli altri modelli, è stato sviluppato per migliorare i tempi di addestramento, ma non è applicabile durante la fase di inferenza. Ciò potrebbe essere implementato in uno sviluppo futuro, ma necessiterebbe di alcuni miglioramenti per quanto riguarda l'algoritmo di Reinforcement Leaning (RL) scelto; queste proposte saranno ben illustrate nella Sezione 6.2.

Il nostro approccio, così come ATSViT, utilizza un numero di token variabili in base alla batch di immagini in input. Ciò consente di adattarsi meglio alle immagini fornite in input, a differenza di quanto avviene con DyViT e EViT. In particolare ATSViT consente all'utente di specificare il numero massimo di token da utilizzare, affinché, se dopo il resampling il numero di token è superiore al valore definito dall'utente, vengono selezionati solo i più promettenti. Nel nostro caso, invece, l'utente può indicare il numero di token desiderati e l'agente viene allenato in modo tale da selezionare un numero di token che si avvicini il più possibile a quello indicato. In tal modo, se l'agente seleziona un numero troppo elevato (o troppo piccolo) di token, esso viene punito. Nel caso in cui, però, l'agente ottiene una loss di training particolarmente bassa, il reward complessivo farà si che l'agente capisca che per quella particolare batch di immagini è vantaggioso utilizzare un alto (o basso) numero di token.

Gli approcci presentati in letteratura fanno uso di una serie di layer posti a valle degli strati di attention, in modo che i token vengano eliminati in modo gerarchico. Nel nostro caso, invece, il modulo è posto a monte del ViT. Ciò è una limitazione dovuta all'algoritmo di RL utilizzato. Infatti, nel Q-learning è necessario conoscere il valore dello stato corrente e dello stato successivo. Nel nostro caso lo stato è rappresentato dai valori di attention; perciò, per ottenere lo stato corrente e successivo, è necessario calcolare il valore dell'attention del primo strato prima e dopo l'allenamento. Ciò risulta problematico nel caso in cui è necessario accedere ai valori di attention degli strati più profondi, poiché aggiunge dell'overhead. Anche in questo caso il problema può essere risolto utilizzando altri algoritmi.

	DyViT	EViT	ATSViT	AgentViT
Utilizzabile in fase di inferenza	Si	Si	Si	No
Numero di token variabili	No	No	Si	Si
I layer sono applicati su più livelli	Si	Si	Si	No
I layer hanno iperparametri da ottimizzare	Si	Si	No	Si

Tabella 6.1: Confronto tra gli approcci

ATSViT, come già detto, fa uso di una tecnica di sampling dei token al fine di selezionarne soltanto un sottoinsieme. In tale approccio, perciò, non ci sono iperparametri da ottimizzare. Ciò risulta essere un vantaggio poiché non viene aggiunto alcun overhead durante l'adde-

stramento. D'altra parte, l'utilizzo di layer ottimizzabili fa si che il modello si adatti meglio ai dati ottenendo risultati più soddisfacenti.

L'architettura AgentViT, perciò, risulta un ibrido tra gli approcci precedentemente descritti. Le principali limitazioni sono dovute alla tipologia di algoritmo di RL utilizzato. Ciò, però, risulta essere un punto di forza, poiché implica che tale approccio può essere ulteriormente sviluppato utilizzando algoritmi diversi.

Conclusioni

Nel presente progetto di tesi abbiamo presentato un approccio basato sull'algoritmo Deep Q-learning (DQL) per migliorare i tempi di addestramento di un Vision Transformer. L'obiettivo, infatti, è di diminuire il numero di operazioni eseguite dagli strati di attention modificando lievemente l'architettura SimpleViT. L'agente è rappresentato da una rete Deep Q-learning costituita da due strati e il cui output è una lista di valori booleani. Tale rete si interfaccia con un environment codificato secondo lo standard proposto da OpenAI. L'agente, dopo aver osservato i valori di attention associati alle immagini del training set, sceglie un sottoinsieme delle patch da utilizzare per l'allenamento del ViT. Al termine dell'allenamento l'agente riceve un reward basato su una combinazione di loss di training e tempo di addestramento del ViT. Al fine di migliorare la stabilità e la generabilità dell'agente, si è fatto uso di una replay memory in cui inserire i dati osservati ed utilizzarli successivamente durante l'addestramento. Inoltre, l'agente tende a scegliere un'azione casuale con probabilità pari a ϵ; tale valore decade esponenzialmente andando avanti con l'addestramento. Ciò garantisce una buona capacità esplorativa nelle fasi iniziali, in modo da evitare di ricadere in un ottimo locale.

Nei Capitoli 5 e 6 abbiamo effettuato un confronto quantitativo e qualitativo rispetto ad alcune delle architetture presenti in letteratura. Abbiamo osservato che l'agente tende a migliorare di molto la complessità temporale dell'allenamento, senza intaccare le performance del ViT. L'approccio presentato, quindi, si è rivelato soddisfacente e fortemente innovativo. Nonostante ciò, il nostro framework può essere ulteriormente migliorato. Di seguito riportiamo alcune delle modifiche suggerite:

- *Modifiche all'environment*: potrebbe essere utile modificare il reward sostituendo il `patch_reward` con un coefficiente che tenga conto dell'effettivo tempo di addestramento del ViT, anziché del numero di patch utilizzate. Inoltre, si potrebbe modificare il `loss_reward`, in modo che venga considerata la loss di validation, anziché quella di training.

- *Modifiche all'algoritmo di Reinforcement Learning*: nel futuro pensiamo di focalizzarci su algoritmi specializzati nella previsione di azioni rappresentate sotto forma di lista di valori. Nel nostro caso, infatti, si è deciso di utilizzare il DQL, il quale è stato sviluppato per scegliere la migliore azione su una lista di N possibili azioni. Per questo motivo è stato necessario adattare l'algoritmo al nostro problema in esame, al fine di selezionare un insieme di azioni ottime, anziché una singola. Perciò pensiamo di esplorare nuovi approcci facendo uso di un algoritmo di tipo *Actor Critic*, come, ad esempio, il *Proximal Policy Optimization* (PPO).

- *Applicazione su diverse architetture di tipo ViT*: il nostro approccio è stato applicato alla sola architettura *SimpleViT*, ma potrebbe essere rilevante valutare gli effetti in una delle altre architetture dello stato dell'arte. Inoltre, potrebbe essere interessante applicare contemporaneamente il nostro approccio e uno di quelli proposti nella Sezione 6.1.

In conclusione, ci aspettiamo che questo primo approccio proposto possa dare il via ad una serie di ricerche che cerchino di applicare le più disparate tecniche di RL al fine di ottimizzare la complessità computazione dei ViT. Tale approccio, infatti, anche se presenta delle criticità rispetto ai migliori approcci in letteratura, risulta fortemente innovativo e potrebbe aprire la strada ad una serie di nuove architetture.

Bibliografia

AA.VV. (1966), *Language and Machines: Computers in Translation and Linguistics*, The National Academies Press, Washington, DC, URL https://nap.nationalacademies.org/catalog/9547/language-and-machines-computers-in-translation-and-linguistics. (Cited at page 5)

BEYER, L., ZHAI, X. e KOLESNIKOV, A. (2022), «Better plain ViT baselines for ImageNet-1k», *arXiv preprint arXiv:2205.01580*. (Cited at page 24)

CHILD, R., GRAY, S., RADFORD, A. e SUTSKEVER, I. (2019), «Generating Long Sequences with Sparse Transformers», *CoRR*, vol. abs/1904.10509, URL http://arxiv.org/abs/1904.10509. (Cited at page 25)

CHOMSKY, N. (1957), *Syntactic Structures*, Mouton and Co., The Hague. (Cited at page 5)

DOSOVITSKIY, A., BEYER, L., KOLESNIKOV, A., WEISSENBORN, D., ZHAI, X., UNTERTHINER, T., DEHGHANI, M., MINDERER, M., HEIGOLD, G., GELLY, S., USZKOREIT, J. e HOULSBY, N. (2021), «An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale», . (Cited at pages 10 e 35)

FAYYAZ, M., KOOHPAYEGANI, S. A., JAFARI, F. R., SOMMERLADE, E., JOZE, H. R. V., PIRSIAVASH, H. e GALL, J. (2021), «ATS: Adaptive Token Sampling For Efficient Vision Transformers», *CoRR*, vol. abs/2111.15667, URL https://arxiv.org/abs/2111.15667. (Cited at pages 25, 35 e 36)

FAYYAZ, M., KOOHPAYEGANI, S. A., JAFARI, F. R., SENGUPTA, S., JOZE, H. R. V., SOMMERLADE, E., PIRSIAVASH, H. e GALL, J. (2022), «Adaptive Token Sampling For Efficient Vision Transformers», . (Cited at page 41)

GONG, Y., LIU, L., YANG, M. e BOURDEV, L. D. (2014), «Compressing Deep Convolutional Networks using Vector Quantization», *CoRR*, vol. abs/1412.6115, URL http://arxiv.org/abs/1412.6115. (Cited at page 40)

GUO, Q., QIU, X., LIU, P., SHAO, Y., XUE, X. e ZHANG, Z. (2022), «Star-Transformer», . (Cited at page 40)

HE, Y., ZHANG, X. e SUN, J. (2017), «Channel Pruning for Accelerating Very Deep Neural Networks», in «Proceedings of the IEEE International Conference on Computer Vision (ICCV)», . (Cited at page 40)

HINTON, G., VINYALS, O. e DEAN, J. (2015), «Distilling the knowledge in a neural network», *arXiv preprint arXiv:1503.02531*. (Cited at page 40)

JADERBERG, M., VEDALDI, A. e ZISSERMAN, A. (2014), «Speeding up Convolutional Neural Networks with Low Rank Expansions», *CoRR*, vol. abs/1405.3866, URL http://arxiv.org/abs/1405.3866. (Cited at page 40)

JIAO, X., YIN, Y., SHANG, L., JIANG, X., CHEN, X., LI, L., WANG, F. e LIU, Q. (2019), «TinyBERT: Distilling BERT for Natural Language Understanding», *CoRR*, vol. abs/1909.10351, URL http://arxiv.org/abs/1909.10351. (Cited at page 40)

KRIZHEVSKY, A., HINTON, G. e OTHERS (2009), «Learning multiple layers of features from tiny images», . (Cited at page 34)

LIANG, Y., GE, C., TONG, Z., SONG, Y., WANG, J. e XIE, P. (2022), «Not All Patches are What You Need: Expediting Vision Transformers via Token Reorganizations», . (Cited at page 40)

LIU, B., RAO, Y., LU, J., ZHOU, J. e HSIEH, C. (2020), «MetaDistiller: Network Self-Boosting via Meta-Learned Top-Down Distillation», *CoRR*, vol. abs/2008.12094, URL https://arxiv.org/abs/2008.12094. (Cited at page 40)

MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S. e DEAN, J. (2013), «Distributed representations of words and phrases and their compositionality», *Advances in neural information processing systems*, vol. 26. (Cited at page 5)

RAO, Y., LU, J., LIN, J. e ZHOU, J. (2018), «Runtime network routing for efficient image classification», *IEEE transactions on pattern analysis and machine intelligence*, vol. 41 (10), p. 2291–2304. (Cited at page 40)

RAO, Y., ZHAO, W., LIU, B., LU, J., ZHOU, J. e HSIEH, C. (2021), «DynamicViT: Efficient Vision Transformers with Dynamic Token Sparsification», *CoRR*, vol. abs/2106.02034, URL https://arxiv.org/abs/2106.02034. (Cited at page 40)

RENGGLI, C., PINTO, A. S., HOULSBY, N., MUSTAFA, B., PUIGCERVER, J. e RIQUELME, C. (2022), «Learning to Merge Tokens in Vision Transformers», . (Cited at page 25)

VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L. e POLOSUKHIN, I. (2017), «Attention Is All You Need», *CoRR*, vol. abs/1706.03762, URL http://arxiv.org/abs/1706.03762. (Cited at page 6)

WANG, K., LIU, Z., LIN, Y., LIN, J. e HAN, S. (2018), «HAQ: Hardware-Aware Automated Quantization», *CoRR*, vol. abs/1811.08886, URL http://arxiv.org/abs/1811.08886. (Cited at page 40)

WANG, W., WEI, F., DONG, L., BAO, H., YANG, N. e ZHOU, M. (2020), «MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers», in LAROCHELLE, H., RANZATO, M., HADSELL, R., BALCAN, M. e LIN, H., curatori, «Advances in Neural Information Processing Systems», vol. 33, p. 5776–5788, Curran Associates, Inc., URL https://proceedings.neurips.cc/paper_files/paper/2020/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. (Cited at page 40)

WATSON, J. B. (1913), «Psychology as the behaviorist views it.», *Psychological review*, vol. 20 (2), p. 158. (Cited at page 13)

YU, X., LIU, T., WANG, X. e TAO, D. (2017), «On compressing deep models by low rank and sparse decomposition», in «Proceedings of the IEEE conference on computer vision and pattern recognition», p. 7370–7379. (Cited at page 40)

Siti web consultati

- Repository GitHub contenente le implementazioni di gran parte delle architetture basate su vision transformer – `https://github.com/lucidrains/vit-pytorch`
- Medium – `https://medium.com/`
- Pytorch – `https://pytorch.org/`
- Springer – `https://link.springer.com/`
- Wikipedia – `https://www.wikipedia.org/`