ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

END-TO-END DEEP METRIC LEARNING CON VISION-LANGUAGE MODEL PER IL FASHION IMAGE CAPTIONING

Elaborato in
Programmazione Di Applicazioni Data Intensive

Relatore
Prof. Gianluca Moro

Co-relatore
Dott. Stefano Salvatori

Presentata da
Riccardo Gennari

Terza Sessione di Laurea
Anno Accademico 2021 – 2022

PAROLE CHIAVE

Image Captioning

Machine Learning

Deep Neural Networks

Transformers

Python

A chi mi è stato vicino,
e mi ha aiutato a raggiungere questo traguardo.

Introduzione

Negli ultimi decenni si è assistito ad una rapidissima crescita della mole di informazioni memorizzate in forma digitale. Dagli importanti archivi e database aziendali, fino alle vastissime quantità di dati presenti nei social network, sempre più enti e servizi dipendono dalla gestione di grandi quantità di dati per poter svolgere le proprie funzioni.

Di conseguenza, anche le tecniche utilizzate per gestire, recuperare e categorizzare queste informazioni si sono evolute di pari passo. In particolare, gli ultimi anni hanno visto sviluppare rapidamente nuove tecniche di machine learning e di deep learning, grazie al lavoro della comunità scientifica e agli investimenti delle grandi imprese del settore informatico.

Queste tecniche sono state applicate con ottimi risultati, e hanno avanzato lo stato dell'arte in numerosi campi, come speech recognition, natural language processing, e computer vision, aprendo la strada a nuovi prodotti, servizi e tecnologie.

Un caso d'uso delle tecniche di deep learning consiste nell'image captioning, cioè la generazione di una didascalia, o caption, che descriva le caratteristiche di un'immagine data in input, ad esempio per migliorare l'accessibilità di un sito web. Pensiamo, ad esempio, a come potremmo generare una caption che descriva in dettaglio i prodotti in vendita su un sito di e-commerce, per permetterne un acquisto più consapevole ai clienti con difficoltà visive.

La generazione di descrizioni accurate per gli articoli di moda online è importante non solo per migliorare le esperienze di acquisto dei clienti, ma anche per aumentare le vendite online. Oltre alla necessità di presentare correttamente gli attributi degli articoli, descrivere i propri prodotti con un linguaggio ricco e vivace può contribuire a catturare l'attenzione dei clienti.

In questa tesi, ci poniamo l'obiettivo di sviluppare un sistema in grado di generare una caption che descriva in modo dettagliato l'immagine di un prodotto dell'industria della moda dato in input, sia esso un capo di vestiario o un qualche tipo di accessorio.

Per i problemi di image captioning, negli ultimi anni molti studi hanno proposto soluzioni basate su reti convoluzionali e LSTM [1, 2, 3]. In questo

progetto proponiamo un'architettura encoder-decoder full-transformer, al fine di testare questa recente tecnologia nel campo della visione artificiale.

Indice

Elenco delle figure

Capitolo 1

Analisi dei dati forniti

Per potere applicare tecniche di machine learning, ci serve prima un insieme di dati su cui lavorare. Introduciamo quindi il dataset utilizzato in questa tesi, e la struttura dei dati che contiene. Definiamo poi con più precisione gli obiettivi del progetto.

1.1 Il dataset Fashiongen

Ai fini di questo progetto, utilizziamo il dataset Fashiongen, pubblicato insieme al paper *Fashion-Gen: The Generative Fashion Dataset and Challenge* [5] nel 2018. Questo dataset contiene 293,008 immagini di dimensione 256x256 pixels, raffiguranti prodotti dell'industria della moda, inclusi capi di vestiario come t-shirt, jeans, e accessori come collane e orologi. Per ogni prodotto sono disponibili da 1 a 6 fotografie, ciascuna scattata da una diversa angolazione. Inoltre, per ogni prodotto sono fornite le seguenti informazioni:

- Il nominativo associato al prodotto;
- Una didascalia in lingua inglese, che ne descrive in modo molto dettagliato le caratteristiche;
- La categoria di abbigliamento a cui il prodotto appartiene;
- La sotto-categoria di abbigliamento a cui il prodotto appartiene;
- I materiali di cui il prodotto è composto;
- Se il capo è indirizzato al pubblico maschile, femminile oppure se è unisex;
- Il brand dell'azienda produttrice;

- Un identificativo che indica l'angolazione da cui il prodotto è stato ritratto;

I prodotti sono divisi in 48 categorie, elencate in dettaglio in Figura 1.1, e 121 sotto-categorie più specifiche. La figura 1.2 mostra un esempio completo di prodotto contenuto nel dataset. Il dati sono suddivisi in due parti:

- training set, contenente 260480 prodotti
- validation set, contenente 32524 prodotti

Le caption fornite descrivono in maniera molto dettagliata il prodotto. Il seguente è un esempio di didascalia presente nel dataset:

```
Hooded windbreaker in navy sheen. Tonal jacquard skull print
throughout. Zip closure at front. Zippered welt pockets at
waist. Bungee drawstring at hood and hem. Ribbed knit sleeve
cuffs in deep navy. Tonal partial mesh lining. Tonal stitching.
```

Come mostrato in Figura 1.3, la maggior parte delle caption sono lunghe tra 15 e 40 parole. Le caption consistono in più frasi separate da un punto, ciascuna delle quali si sofferma a descrivere un particolare dettaglio del relativo prodotto.

1.2 Obiettivi

L'obiettivo di questa tesi è quello di sviluppare un sistema in grado di generare una caption che descriva al meglio l'immagine di un prodotto dell'industria della moda dato in input, sia esso un capo di vestiario o un qualche tipo di accessorio.

A questo scopo, ci serviremo di tecniche di machine learning e più nello specifico deep learning, per ottenere un modello che soddisfi questi requisiti.

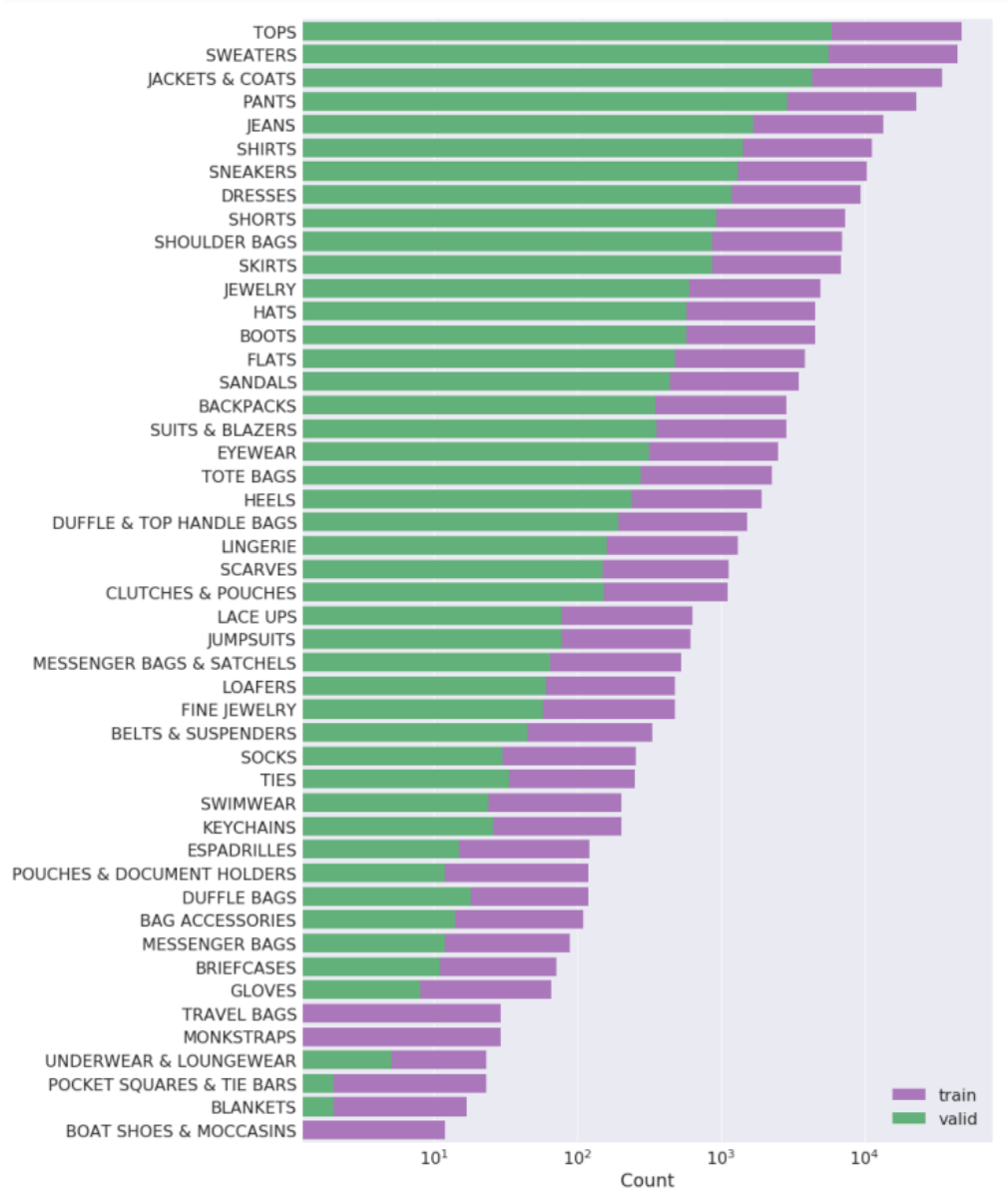


Figura 1.1: Categorie di prodotti nel dataset. Nel grafico, le linee viola e verdi indicano il numero di prodotti appartenenti alla categoria indicata contenuti rispettivamente nel training e validation set.

Fonte: Fashiongen paper [5]

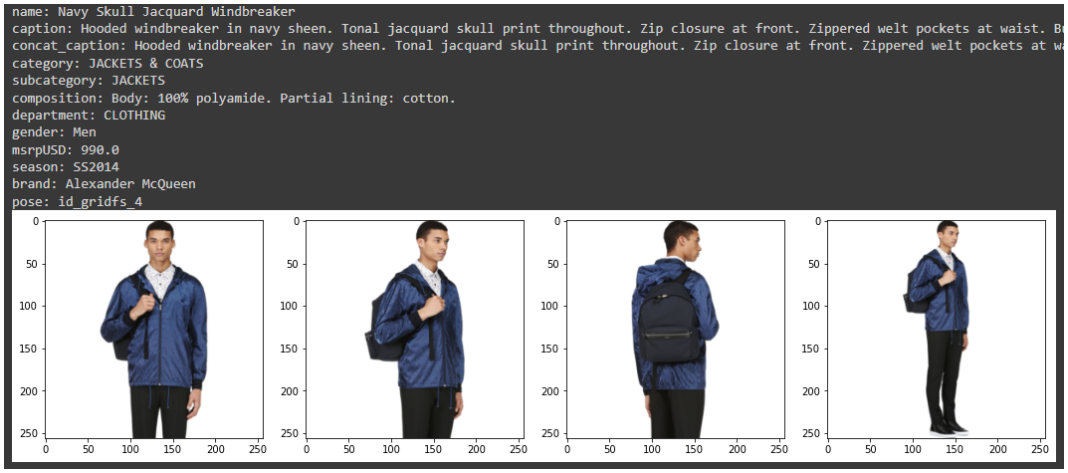


Figura 1.2: Esempio di un prodotto contenuto nel dataset.

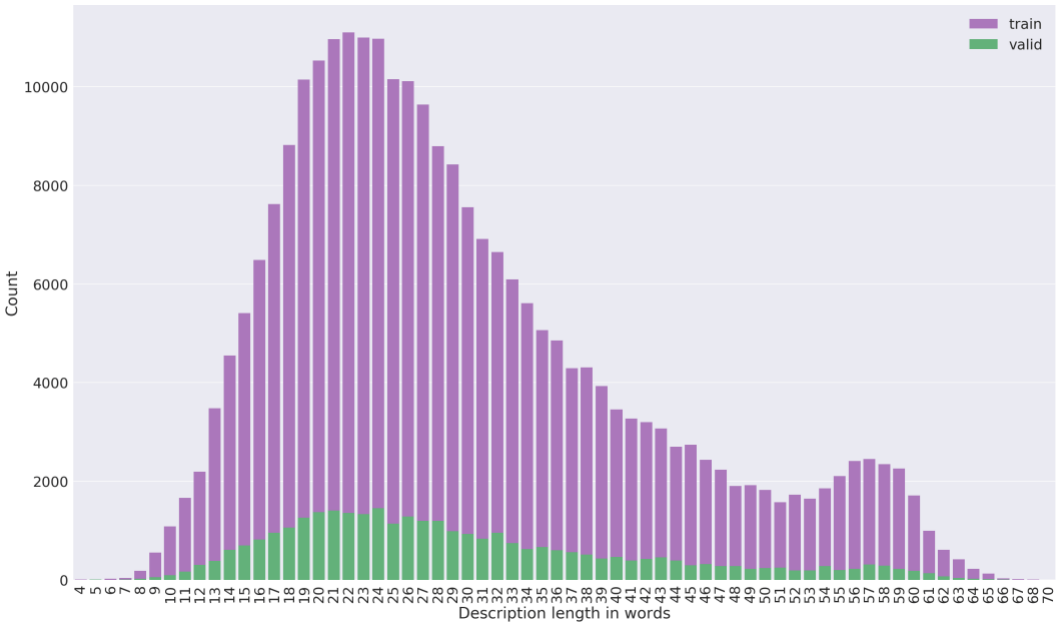


Figura 1.3: Lunghezza delle caption contenute nel dataset. Nel grafico, le linee viola e verdi indicano il numero di caption contenuti rispettivamente nel training e validation set.

Fonte: Fashiongen paper [5]

Capitolo 2

Le tecnologie disponibili

In questo progetto andremo ad affrontare un task di image captioning. Per poter raggiungere buoni risultati, dovremo utilizzare tecniche di machine learning. Andiamo quindi ad analizzare le tecnologie che abbiamo a disposizione.

2.1 Machine Learning

Il machine learning è un ramo dell'intelligenza artificiale, che utilizza metodi statistici per identificare pattern presenti nei dati. Le relazioni esistenti tra i dati vengono apprese dagli algoritmi di learning durante la fase di addestramento, e la conoscenza così acquisita viene successivamente sfruttata da questi algoritmi per effettuare predizioni più accurate e risolvere problemi di classificazione.

Riportando il pensiero del professor Tom Mitchell in [6], il campo del machine learning tenta di rispondere alla seguente domanda:

> *"Come possiamo costruire sistemi informatici che migliorano automaticamente con l'esperienza e quali sono le leggi fondamentali che regolano tutti i processi di apprendimento?"*

Le tecniche di machine learning hanno ottenuto importanti successi quando applicate su casi reali, tra i quali:

- Visione artificiale
- Riconoscimento vocale
- Traduzione automatica e generazione di linguaggio naturale
- Raccomandazioni di prodotti
- Filtraggio di email e malware detection

Il machine learning può quindi essere applicato per risolvere svariati tipi di task. Inoltre, l'apprendimento automatico può essere effettuato in più modalità. Le tre tipologie principali sono:

- Apprendimento supervisionato
- Apprendimento non supervisionato
- Apprendimento per rinforzo

Le tre sono distinte principalmente dal tipo di segnale, o feedback, che il sistema ha a disposizione durante la fase di addestramento.

2.1.1 Apprendimento supervisionato

L'apprendimento supervisionato viene definito in [7] come

> *"Il task di apprendimento automatico che consiste nel dedurre una funzione da dati di training etichettati"*

Un algoritmo di learning supervisionato, infatti, analizza i dati di training per estrapolare una funzione utilizzabile per effettuare predizioni anche su dati mai visti durante l'addestramento. I dati di training devono essere etichettati, ovvero devono comprendere l'output desiderato, e il sistema sfrutterà queste informazioni durante la fase di addestramento per migliorare la propria accuratezza.
Questa modalità di learning si presta bene a risolvere problemi di regressione e di classificazione.

Regressione

La regressione nel machine learning è una tecnica usata per estrapolare le relazioni esistenti tra una variabile dipendente e una o più variabili indipendenti. La funzione con la quale si cerca di rappresentare le relazioni fra le variabili in gioco può presentarsi in molteplici varianti. Nella regressione, i valori ottenuti in output assumono valori numerici continui.

Classificazione

Nei problemi di classificazione, i dati di output vengono divisi in due o più classi di appartenenza. Il compito del sistema è quello di apprendere il pattern che determina a quale classe assegnare i dati, in modo da essere in grado di determinare anche per dati non visti durante la fase di training

la corretta classe di appartenenza. Nella classificazione, i valori ottenuti in output corrispondono alle classi di appartenenza, che essendo in numero finito assumono valori discreti.

2.1.2 Apprendimento non supervisionato

Nell'apprendimento non supervisionato, la fase di addestramento avviene utilizzando dati non etichettati. Questo comporta alcuni vantaggi, come la maggior facilità di acquisizione dei dati. Chiaramente la mancanza di etichette di riferimento comporta anche svantaggi, come la mancanza di un'indicazione sulla bontà della previsione. Di conseguenza diventa molto difficile trovare un metodo generale di valutare l'errore del modello. L'apprendimento non supervisionato si presta bene a risolvere problemi di clustering, dove il sistema deve classificare un insieme di dati, senza conoscere a priori le possibili categorie di appartenenza. I dati di input vengono quindi suddivisi in più gruppi, in modo che che i dati appartenenti allo stesso gruppo siano il più possibile simili tra loro, e quelli appartenenti a gruppi diversi siano il più possibile dissimili.

2.1.3 Apprendimento per rinforzo

Il reinforcement learning, o apprendimento per rinforzo, è una tipologia di apprendimento automatico adatta a problemi in cui è necessario prendere una sequenza di decisioni, dove ogni decisione dipende dallo stato corrente del sistema, e ne influenza quello futuro.

Il sistema sceglie tra le decisioni possibili basandosi su un valore numerico chiamato ricompensa, associato a ciascuna scelta. Generalmente, la decisione con ricompensa maggiore è considerata la migliore.

Il punto di forza del reinforcement learning sta nella sua dinamicità: il sistema non si limita a ricercare pattern in un insieme di dati, ma piuttosto impara a relazionarsi con l'ambiente con cui interagisce, attraverso i feedback ricevuti tramite le ricompense.

Il processo di addestramento consiste solitamente nel ripetere iterativamente un certo task. Grazie all'uso delle ricompense, eventualmente il sistema trova la sequenza di decisioni ottimale, anche se questo può richiedere anche milioni di iterazioni.

2.1.4 La discesa del gradiente

Tipicamente, la fase di addestramento di un modello di learning è centrata sulla ricerca dei parametri, o pesi, che quando applicati alla funzione con cui il modello approssima i dati reali, minimizzano un certo errore. Questo errore

consiste in una funzione, la cui scelta dipende dal problema affrontato e dal modello utilizzato. I parametri che rendono minima la funzione d'errore scelta vengono ricercati durante la fase d'addestramento del modello, generalmente utilizzando la tecnica della discesa del gradiente.

Il gradiente di una funzione è il vettore delle derivate parziali della funzione stessa, e se calcolato in un punto ci dà informazioni sull'inclinazione che la curva della funzione assume in quel punto. In particolare, utilizzando il gradiente possiamo determinare in quale direzione una funzione decresce più rapidamente. Possiamo quindi applicare il metodo di discesa del gradiente, per seguire la direzione in cui la nostra funzione d'errore decresce più rapidamente ed eventualmente raggiungere un minimo locale. In particolare:

1. Scegliamo un punto di partenza casuale x_k
2. Calcoliamo il gradiente ∇f della funzione d'errore f in x_k
3. Sottraiamo ad x_k un vettore proporzionale al gradiente ∇f per ottenere un valore x_{k+1} tale che $f(x_{k+1}) < f(x_k)$
4. Incrementiamo di 1 l'indice k e ripetiamo i passi 1-4 finchè non si converge ad un valore minimo

Più precisamente, al passo k dell'algoritmo, determiniamo x_{k+1} come:

$$x_{k+1} = x_k - \eta * \nabla f(x_k)$$

Dove η è un iperparametro, chiamato step size o learning rate. Il learning rate determina di quanto l'algoritmo si sposta lungo la curva della funzione d'errore tra un'iterazione e la successiva. Uno step size troppo basso può portare l'algoritmo a convergere molto lentamente ad un minimo, mentre un valore troppo alto può causare problemi quando l'algoritmo è prossimo a convergere, a causa del movimento a tratti troppo ampi lungo la curva. Una rappresentazione del procedimento descritto è visibile in Figura 2.1.

In alcuni metodi, il learning rate viene fatto variare tra iterazioni. Solitamente a questo viene assegnato un valore relativamente elevato alla prima iterazione dell'algoritmo, che viene poi gradualmente ridotto man mano che l'algoritmo converge.

2.2 Deep Learning

Il deep learning è un ramo dell'intelligenza artificiale, basato sull'utilizzo di reti neurali artificiali, ovvero modelli computazionali la cui struttura è

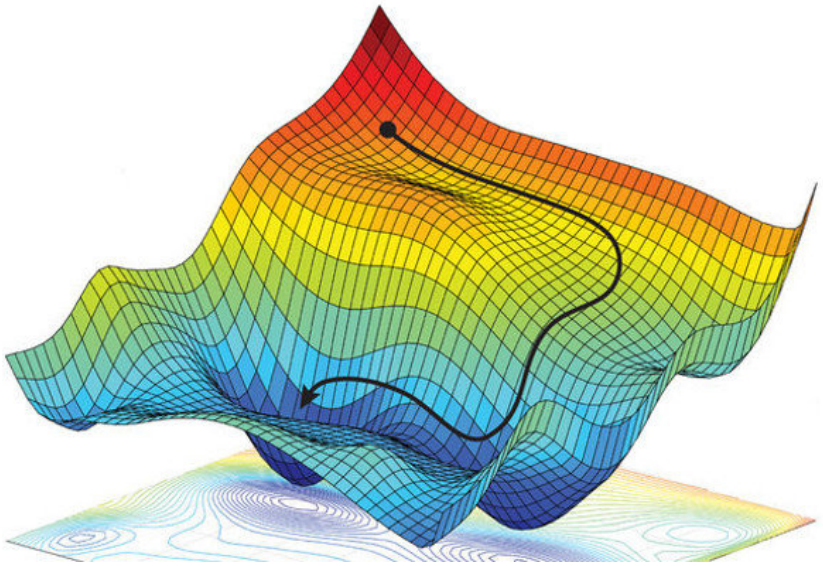


Figura 2.1: Un'illustrazione grafica della discesa del gradiente, applicata ad una funzione tridimensionale

Fonte: www.researchgate.net

vagamente ispirata a quella delle reti neurali biologiche. Riportando il pensiero di Heaton et al. in [8], l'uso corrente del termine "deep learning" va oltre la classica prospettiva neuroscientifica. Si riferisce, infatti, ad un più generale principio di apprendimento a più livelli, che viene applicato ai framework di machine learning e non necessariamente è ispirato alla struttura neuronale.

Le tecniche del deep learning hanno destato uno spiccato interesse nella comunità scientifica grazie alla loro capacità di sopperire ad alcune mancanze dei convenzionali algoritmi di apprendimento automatico. Come sostenuto da LeCun et al. in [9], le tecniche di learning convenzionali erano limitate nella loro abilità di processare dati naturali nella loro forma grezza. L'estrazione di feature dai pixel di un'immagine, ad esempio, era un'operazione ardua che richiedeva vasta esperienza nel dominio applicativo.

Gli algoritmi di deep learning, invece, sono risultati molto efficaci nello scoprire le intricate relazioni esistenti tra dati ad elevata dimensionalità, come linguaggio naturale ed immagini. Per questa ragione, le reti neurali artificiali sono state ampiamente utilizzate per risolvere problemi di natural language processing, speech recognition e image recognition, ottenendo ottimi risultati e in molti casi avanzando lo stato dell'arte.

2.2.1 Reti neurali feed-forward

Le reti feed-forward sono reti neurali artificiali la cui struttura è vagamente ispirata a quella delle reti neurali biologiche. Questa architettura, la prima ad essere teorizzata, si basa su insiemi di nodi di calcolo tra loro interconnessi, chiamati neuroni per analogia con l'equivalente biologico. I neuroni sono disposti su diversi strati, e le informazioni circolano dal primo strato verso i successivi, senza formare cicli (da cui il nome feed-forward). Questa struttura è illustrata nella Figura 2.2. Le reti neurali feed-forward non mantengono informazioni sugli input precedenti, quindi l'output dipende solamente dall'input corrente. Per questo motivo, le reti feed-forward sono poco adatte a risolvere task in cui è importante considerare le inter-dipendenze presenti all'interno di sequenze di input, come ad esempio i problemi di natural language processing.

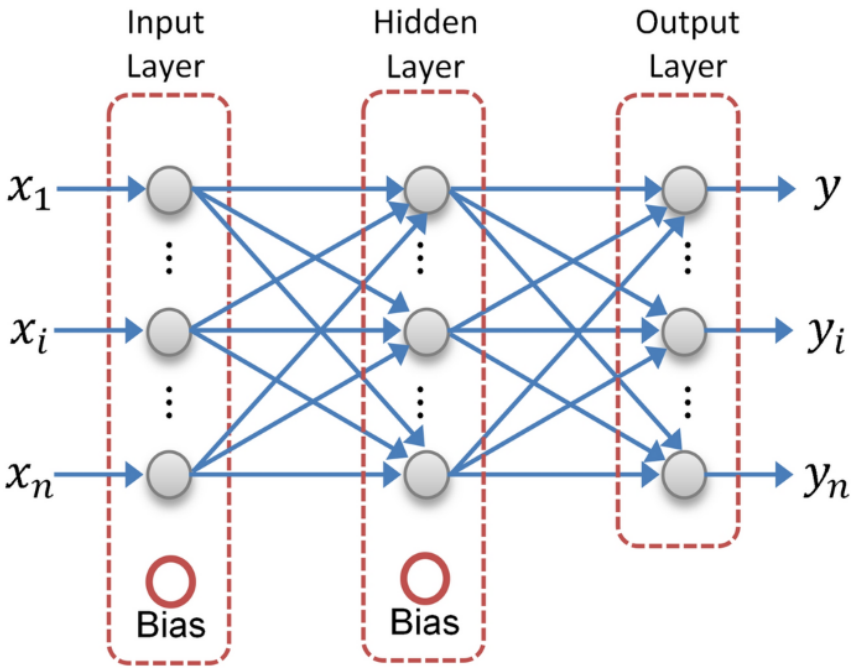


Figura 2.2: Architettura di rete neurale feed-forward.

Fonte: `https://doi.org/10.1007/s13042-020-01252-x`

Il neurone

La rete neurale accetta input multipli. All'interno di ogni singolo nodo della rete, o neurone, ciascun input viene moltiplicato per un peso. I prodotti così ottenuti vengono sommati tra loro, e alla loro somma viene aggiunto un bias. Il risultato viene passato ad una funzione di attivazione, che simula l'accensione

o meno del neurone. Sommare un bias permette, in pratica, di effettuare un operazione di shifting sulla funzione di attivazione. La struttura del neurone artificiale viene illustrata in Figura 2.3.

I vari pesi associati ad ogni input vengono aggiustati durante la fase di addestramento, e simulano il ruolo che le sinapsi assumono nei neuroni biologici, ovvero l'amplificazione o riduzione di un segnale. I pesi nei neuroni vengono scelti in modo da minimizzare l'errore nella rete, attraverso la tecnica della retro-propagazione dell'errore, o backpropagation [10], che sfrutta la discesa del gradiente (2.1.4).

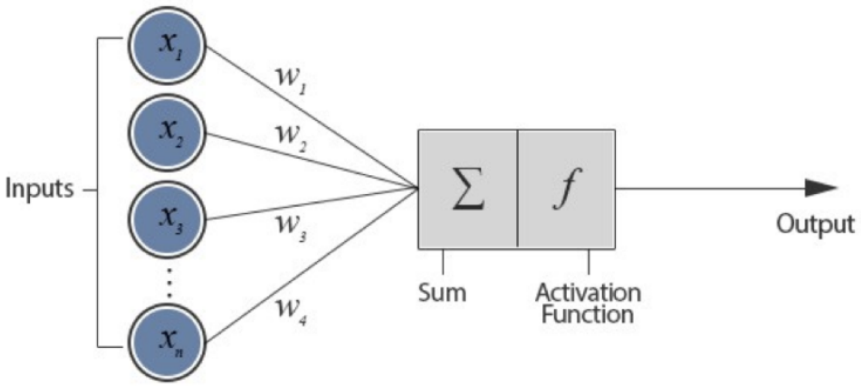


Figura 2.3: Architettura del neurale artificiale in una rete feed-forward.

Fonte: domsoria.com

Funzione sigmoidea

La funzione sigmoidea (sigmoid) può essere utilizzata come funzione di attivazione. Essa è continua, derivabile, e assume valori nel range [0, 1]. Se calcolata in un punto x, la funzione ha valore:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Come si può notare dalla formula sopra mostrata, utilizzare questa funzione di attivazione equivale ad effettuare una regressione logistica in ogni neurone.

Rectified linear unit (ReLU)

La funzione ReLU è un'altra funzione di attivazione comunemente utilizzata nelle reti neurali. Essa è continua, derivabile, e assume valori nel range [0, ∞]. Se calcolata in un punto x, la funzione ha valore:

$$ReLU(x) = \begin{cases} 0 & \text{if x<=0} \\ x & \text{if x>0} \end{cases}$$

2.2.2 Reti neurali ricorrenti (RNN)

Una rete neurale ricorrente, a differenza di una rete feed-forward, contiene cicli tra le connessioni neuronali. Grazie a questa caratteristica, uno degli strati può essere utilizzato per memorizzare lo stato della rete, rendendo così possibile l'utilizzo delle informazioni relative agli input precedenti durante l'elaborazione dell'output. Per questo motivo, le reti neurali ricorrenti risultano molto più adatte, rispetto alle reti feed-forward, a risolvere problemi in cui è necessario considerare le relazioni presenti all'interno di sequenze di input, come ad esempio nel caso del natural language processing, e nei task di riconoscimento vocale. L'architettura di questo tipo di rete è illustrata in Figura 2.4.

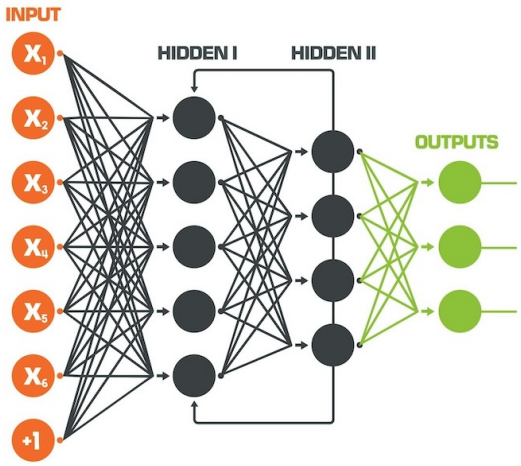


Figura 2.4: Struttura di una rete neurale ricorrente.

Fonte: gosmar.eu

2.2.3 Reti Long Short-Term Memory (LSTM)

Le reti long short-term memory, o LSTM, sono reti neurali ricorrenti capaci di filtrare le informazioni ricevute ad ogni passo, scartando quelle irrilevanti. Questo processo avviene all'interno dei singoli neuroni tramite l'uso di appositi gates.

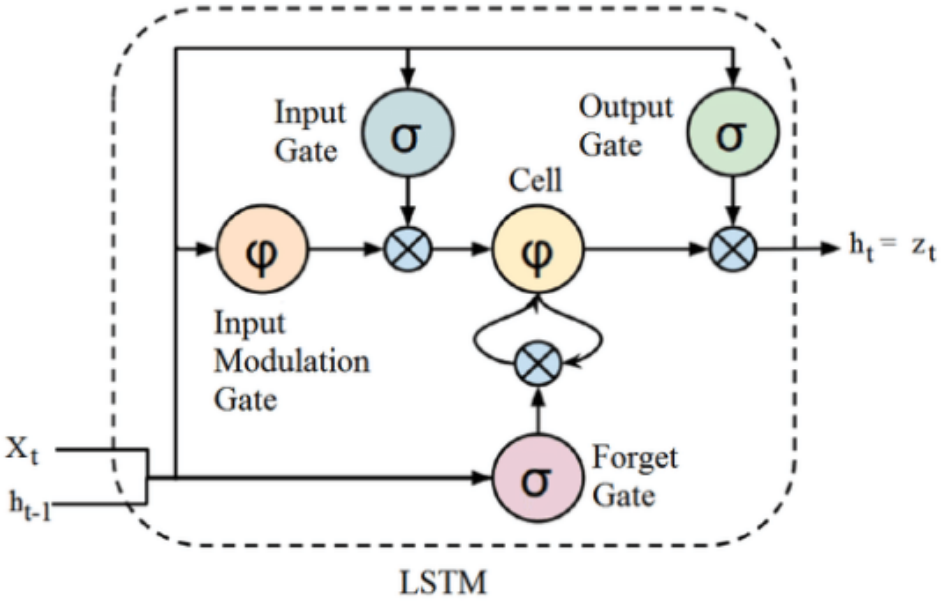


Figura 2.5: Struttura di un nodo all'interno di una rete LSTM.

Fonte: medium.com

Le unità neuronali di una rete LSTM contengono diversi gates, come mostrato in Figura 2.5. Ogni gate svolge una diversa funzione, in particolare:

- Il forget gate scarta le informazioni irrilevanti memorizzate durante i passi precedenti
- L'input gate filtra le nuove informazioni, ricevute al passo corrente
- L'output gate aggiorna lo stato interno del neurone con le informazioni filtrate dal forget gate e dall'input gate

La scomparsa del gradiente

Le LSTM furono introdotte per risolvere un problema presente nelle classiche reti neurali ricorrenti, ovvero il fenomeno della scomparsa del gradiente, o vanishing gradient. Questo fenomeno consiste nella riduzione del valore del gradiente, che se troppo piccolo può compromettere il corretto addestramento della rete. La causa della scomparsa del gradiente è da ricercare nelle ripetute moltiplicazioni effettuate tra la matrice dei pesi e la matrice degli input: infatti, se almeno una delle matrici assume valori minori di 1, il prodotto ottenuto avrà valore più basso dopo ogni step.

Il risultato di questo fenomeno, è che la rete tende ad attribuire pesi troppo bassi alle informazioni meno recenti, di fatto dimenticandole. Anche se una vera soluzione alla scomparsa del gradiente è stata trovata solo con l'introduzione dei meccanismi di attention [11], le LSTM riescono comunque ad alleviare questo problema grazie alla struttura delle loro unità neuronali, che permettono di filtrare le informazioni irrilevanti e mantenere memorizzate quelle rilevanti, anche se poco recenti.

LSTM nell'image captioning

Come affermato da Aneja et al. in *Convolutional Image Captioning* [12], negli ultimi anni sono stati fatti importanti progressi nel campo della visione artificiale, usando reti LSTM. Infatti, per diverso tempo queste reti sono state considerate uno standard de-facto per l'image captioning [13, 14, 15].

Anche se le reti LSTM mitigano il problema della scomparsa del gradiente e memorizzano le dipendenze tra input con efficacia, queste sono complesse e devono necessariamente elaborare i dati in maniera sequenziale. Per superare questi ostacoli, studi recenti hanno mostrato i vantaggi delle reti convoluzionali per task di image captioning e generazione di immagini.

2.2.4 Reti neurali convoluzionali

Le reti neurali convoluzionali sono reti neurali di tipo feed-forward che contengono layer convoluzionali. Queste reti trovano largo utilizzo nei task di visione artificiale, e sfruttano il meccanismo della convoluzione per estrarre feature che descrivono i dati di input a diversi livelli di astrazione. I primi strati della rete identificano feature di basso livello, mentre gli strati successivi catturano caratteristiche con livello d'astrazione progressivamente più elevato. La struttura di questo tipo di rete è mostrata in Figura 2.6, e consiste in una serie di strati di convoluzione, ciascuno seguito da una funzione di attivazione (come ReLU) e da un layer di pooling. Gli strati di pooling effettuano operazioni di riduzione sulla matrice dei dati, aggregandone le informazioni e creando feature map di dimensioni più piccole. L'ultimo strato consiste in un fully-connected layer, che svolge la classificazione di immagini vera e propria.

Le convoluzioni in matematica

In matematica, una convoluzione è un'operazione tra due funzioni di una variabile che consiste nell'integrare il prodotto tra la prima e la seconda traslata di un certo valore. Se indichiamo con f la prima funzione e con g la seconda, avremo:

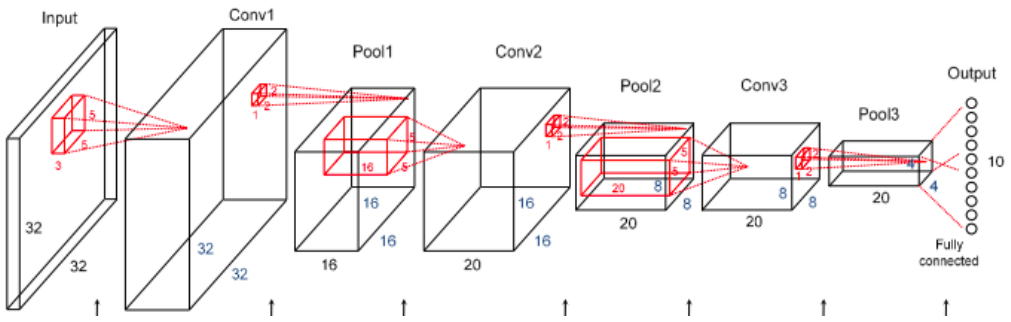


Figura 2.6: Architettura di una rete convoluzionale.

Fonte: domsoria.com

$$(f \cdot g)(x) = \int f(y) \cdot g(x-y)\, dy = \int f(x-y) \cdot g(y)\, dy$$

A livello intuitivo, una convoluzione è un integrale che quantifica la sovrapposizione tra una funzione g e una funzione f, quando g viene traslata su f. In altre parole, le due funzioni vengono fuse tra loro.

Le convoluzioni nella visione artificiale

Se applicata in un problema di visione artificiale, la funzione f potrebbe rappresentare un'immagine di input, mentre la funzione g rappresenta un filtro che identifica un particolare pattern o struttura dell'immagine.

Ogni strato di convoluzione presente nella rete neurale potrebbe contenere un filtro diverso, che identifica feature di livello più o meno alto dell'immagine di input. Ad esempio, in un primo layer il filtro potrebbe riconoscere linee orizzontali, verticali, o diagonali. Un filtro successivo potrebbe invece concentrarsi su porzioni più ampie dell'immagine, e tra gli ultimi strati della rete potremmo trovare un filtro che ci permette di identificare l'oggetto nella sua interezza. In Figura 2.7 viene mostrato un esempio di quale aspetto possono assumere le feature estratte da una rete convoluzionale.

Le reti neurali convoluzionali trovano largo impiego nel campo della visione artificiale grazie alla loro capacità di catturare dettagli di diverso livello astrattivo all'interno delle immagini. Questa capacità permette di analizzare le immagini scomponendole in molte feature più dettagliate, e così facendo coglie dettagli più significativi e distintivi rispetto a una classica rete feed-forward, che invece può solo analizzare l'immagine nel suo complesso.

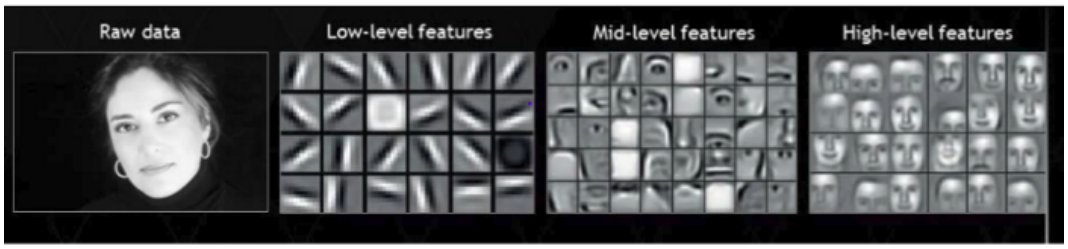


Figura 2.7: Architettura del neurale artificiale in una rete feed-forward.

Fonte: medium.com

2.2.5 Reti ad architettura Transformer

Le reti Transformer sono reti neurali dotate di un'architettura innovativa, basata su un modello encoder-decoder. Questa architettura è stata introdotta nel 2017 con il paper *Attention is All you Need* [16]. Come le classiche reti ricorrenti e LSTM, i modelli Transformer sono adatti a risolvere problemi di natural language processing e visione artificiale grazie alla loro capacità di modellare le relazioni presenti in lunghe sequenze di input.

A differenza delle architetture tradizionali, però, i modelli Transformer non devono necessariamente processare i dati in ordine. Ad esempio, in un problema di NLP non sarà necessario processare una frase ordinatamente dalla prima parola fino all'ultima, perchè l'architettura Transformer sfrutta meccanismi di self-attention e cross-attention per identificare il contesto che dà senso alla parola all'interno di una frase. Non dovendo processare gli input in maniera sequenziale, l'addestramento di questi modelli si presta molto bene a venire parallelizzato su più macchine, riducendo i tempi di training rispetto alle classiche RNN, senza sacrificare accuratezza nelle predizioni.

La struttura encoder-decoder di un Transformer, come descritta nel paper originale [16], è illustrata in Figura 2.8.

Il modello si divide in due macro-componenti, ovvero encoder e decoder. Ciascuno dei due moduli è formato da più istanze identiche dello stesso blocco, tra loro interconnesse. Nel paper originale, encoder e decoder si compongono di sei blocchi ciascuno. In Figura 2.9 ne è mostrato un esempio. I singoli blocchi hanno la stessa identica struttura, ma non condividono i pesi.

Il modulo encoder

Un singolo modulo encoder è composto da due strati: un self-attention layer e una rete neurale feed-forward. Consideriamo ad esempio un problema di natural language processing. Una sequenza di input, in questo caso, potrebbe corrispondere ad una frase in linguaggio naturale. Questa frase viene prima

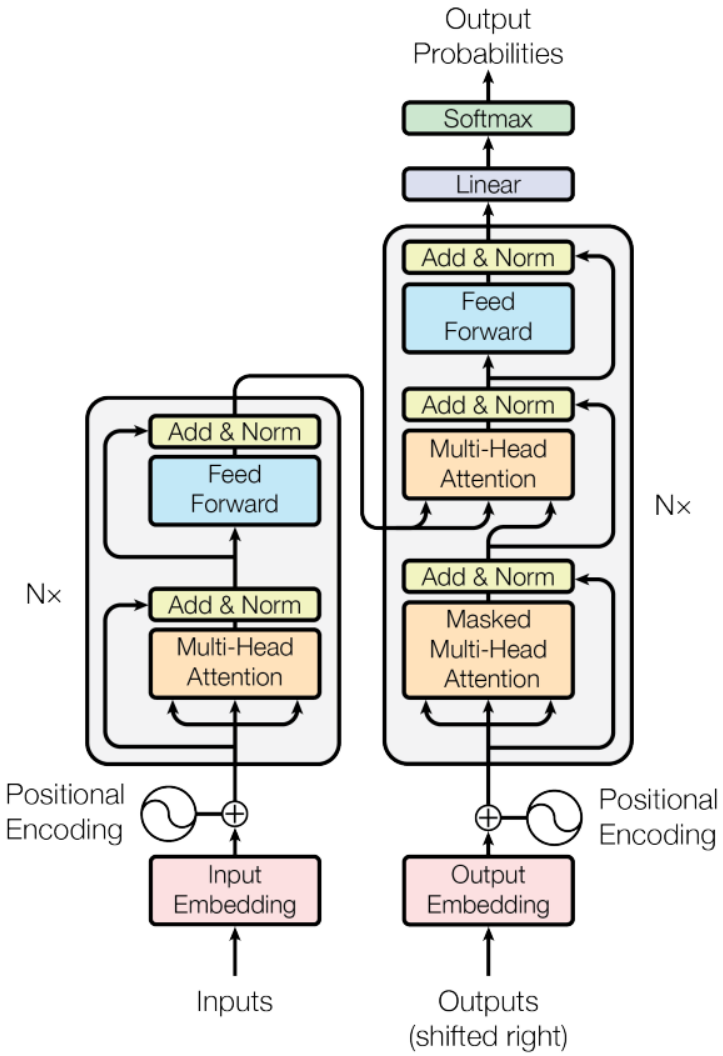


Figura 2.8: Architettura della struttura Encoder-Decoder in un modello Transformer.

Fonte: Attention is All you Need, [16]

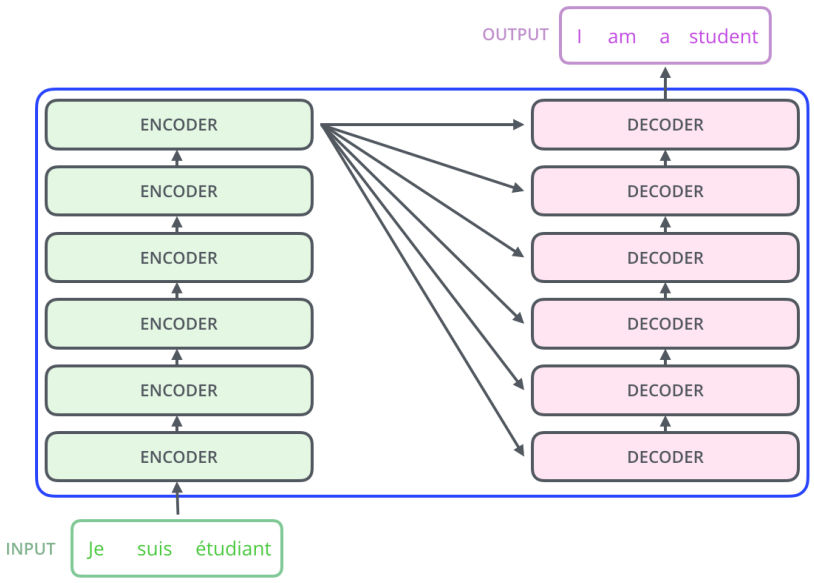


Figura 2.9: Un esempio di stack encoder-decoder, formato da sei componenti per modulo. Qui il transformer viene usato come traduttore da francese a inglese.

Fonte: The illustrated Transformer

elaborata dallo strato di self-attention, che arricchisce la codifica di ogni parola nella sequenza di input con una parte di informazioni relative alle altre parole contenute nella frase da processare.
Questo processo consente di rappresentare ogni parola unitamente al suo contesto nel resto della frase. La codifica così ottenuta viene poi passata in input al secondo strato, ovvero una classica rete neurale feed forward.

Il modulo decoder

Anche il modulo decoder contiene gli stessi due strati dell'encoder, ovvero self-attention layer e rete neurale feed-forward. In aggiunta a questi, il modulo decoder si compone anche di un terzo strato, posto tra self-attention layer e rete feed-forward. Questo strato è chiamato encoder-decoder attention layer, e ha lo scopo di migliorare la capacità del decoder di concentrarsi su particolari punti della frase processata. La Figura 2.10 mostra graficamente le componenti interne di encoder e decoder.
Una caratteristica interessante del decoder, è che il suo strato di self-attention può solo considerare le posizioni precedenti rispetto al dato corrente, e non quelle successive. Questo la differenzia dalla self-attention nell'encoder, che invece

funziona in maniera bidirezionale. Per permettere al decoder di focalizzarsi solo sulle posizioni precedenti, a quelle successive viene applicato un processo di masking, che effettivamente setta il loro valore a $-\infty$ durante il calcolo della self-attention.

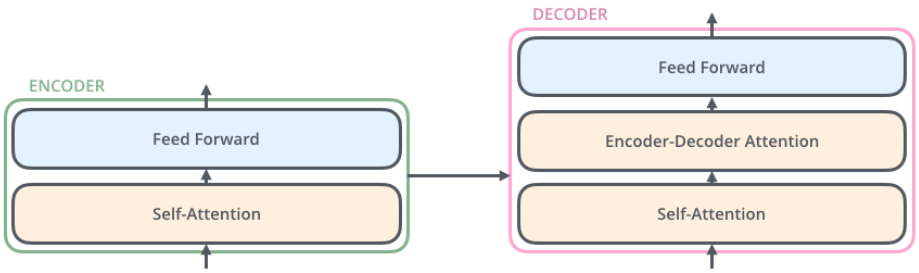


Figura 2.10: Rappresentazione delle componenti interne di encoder e decoder.

Fonte: The illustrated Transformer

Lo strato di self-attention

Il self-attention layer permette all'encoder di migliorare la qualità della codifica di una parola, incorporando parti della codifica di altre parole presenti nella sequenza di input nella rappresentazione della parola corrente. Questo permette di aggiungere informazioni all'embedding di una parola che riguardano il contesto in cui questa è utilizzata nella frase.
Questo meccanismo, utilizzato nelle reti ad architettura Transformer, svolge una funzione molto simile all'hidden state che le reti neurali ricorrenti standard utilizzano come memoria di stato. Infatti, in entrambi i casi viene incorporata una parte della rappresentazione delle parole precedenti nella codifica della parola corrente.
Grazie allo strato di self-attention, è possibile rappresentare il contesto di una parola all'interno della frase senza bisogno di memorizzare altre informazioni di stato. La Figura 2.11 mostra graficamente come questo meccanismo sia in grado di catturare la semantica di un termine in relazione al suo contesto nella frase. In questo esempio, vediamo come il termine "*it*" contiene nella sua codifica una parte dell'embedding dei termini "*the*" ed "*animal*" in quanto, nella frase, il pronome "*it*" si riferisce al soggetto "*the animal*".
Un'altra caratteristica interessante dello strato di self-attention, è che le dipendenze tra le varie parole della sequenza di input sono considerate solo all'interno di questo layer. Infatti, l'output della self-attention consiste nella codifica delle parole già comprendenti il loro contesto nella frase. Questo significa che queste

codifiche possono venire elaborate dal successivo strato, ovvero la rete feed-forward, in maniera indipendente tra di loro. Questa caratteristica rende il processo di training facilmente parallelizzabile, cosa che permette di tagliare in modo considerevole i tempi di addestramento, sfruttando il calcolo distribuito.

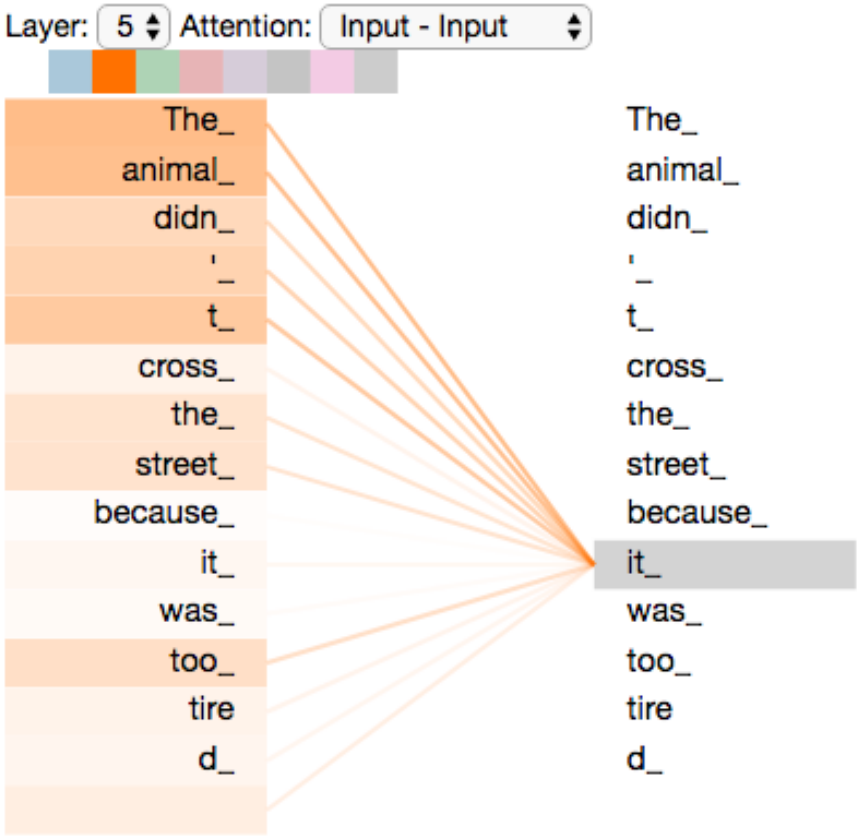


Figura 2.11: Un esempio di self-attention.

Fonte: The illustrated Transformer

La codifica posizionale

Le reti ricorrenti classiche e LSTM elaborano le frasi di input in ordine sequenziale. La prima parola di una frase sarà quindi la prima ad essere processata, e così via. Invece, come accennato in precedenza, i modelli Transformer non necessariamente elaborano le parole seguendo l'ordine che queste avevano nella frase. Questo non permette di conservare un importante informazione:

infatti, la posizione che una parola assume all'interno della frase ci dà informazioni importanti sul suo contesto e sulla sua semantica.
Per risolvere questo problema l'architettura Transformer si serve di una tecnica chiamata codifica posizionale, o positional encoding. Alla codifica di ogni parola, viene aggiunto un ulteriore vettore che segue uno specifico pattern, che permette al modello di determinare la posizione della parola nella frase, o più nello specifico la sua distanza dalle altre parole nella sequenza. Il pattern in questione non è conosciuto a priori dal modello, ma verrà imparato durante la fase di addestramento.

2.2.6 Vision Transformer

I modelli Transformer, nella loro forma originale, hanno trovato applicazione soprattutto nel campo del natural language processing. Nei task di visione artificiale, la tecnologie predominanti hanno continuato a basarsi su reti neurali convoluzionali (2.2.4). Nel 2020, con il paper *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* [4], viene introdotto il Vision Transformer, che altera l'architettura originale del Transformer per renderla efficace nell'elaborazione di immagini.
La ragione per cui i classici modelli Tranformer sono inadatti a risolvere task di visione artificiale, è da ricercare nel meccanismo di attention. Il calcolo della attention viene effettuato per ogni token, e il costo di questo processo aumenta esponenzialmente con il numero di token contenute nella sequenza di input. Nei problemi di natural language processing, un token corrisponde ad una parola, e una sequenza di input ad una frase. In un task di visione artificiale, invece, un token corrisponderebbe ad un singolo pixel, parte dell'immagine che costituisce la sequenza di input. Il numero di pixel che compongono un immagine può essere elevatissimo, e quindi di conseguenza anche il costo computazionale legato al calcolo dell'attention aumenterebbe a dismisura. Questo problema rende i modelli Transformer inadatti ad affrontare task di computer vision.
Per superare questo ostacolo, in [4] viene proposta una nuova architettura basata sul Transformer originale, il Vision Transformer. La Figura 2.12 ne illustra la struttura.

Il Vision Transformer, invece di utilizzare i singoli pixel come token, utilizza delle sezioni di dimensione variabile, 16x16 pixel nella versione base introdotta nel paper. Queste sezioni vengono poi disposte in una sequenza. A questo punto possiamo fare un parallelo con i classici Transformer, applicati ad un problema di natural language processing. Nel caso del Transformer standard, avremmo avuto come input una sequenza di token, dove ogni singolo token rappresenta una parola, e l'intera sequenza una frase. Nel Vision Transformer, abbiamo in input sempre una sequenza di token, ma ogni token corrisponde

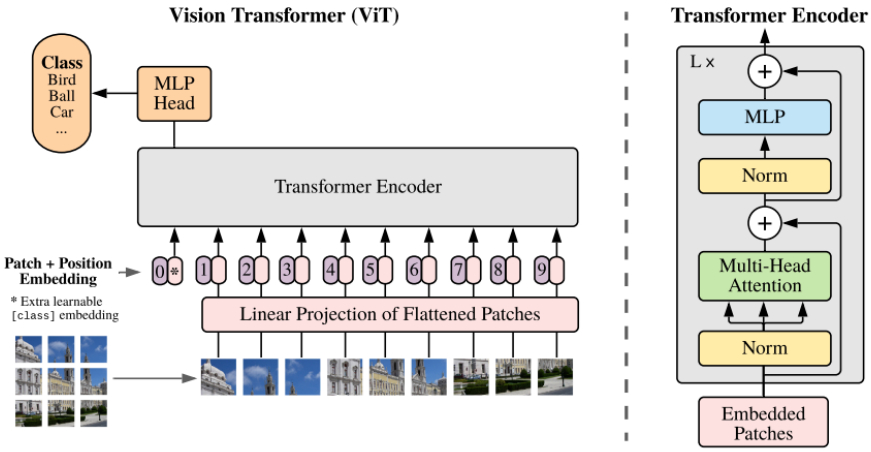


Figura 2.12: Struttura di un Vision Transformer.

Fonte: An Image is Worth 16x16 Words, [4]

invece ad una sezione dimensionata 16x16 pixel dell'immagine di input. L'intera sequenza di input, invece, rappresenta l'immagine completa.
Le sequenze di input, ottenute come descritto sopra, vengono poi codificate dal Vision Transformer, che aggiunge anche la codifica posizionale ad ogni singola sezione di immagine. Il risultato di questo processo può quindi essere passato in input ad un Transformer encoder, identico a quello proposto nel paper originale [16]. All'interno dell'encoder, il meccanismo di self-attention viene applicato come se i token di input rappresentassero parole, e nonostante ciò produce ottimi risultati anche su immagini. La Figura 2.13 mostra un esempio grafico, fornito dagli autori di Vision Transformer nel relativo paper, di come la self-attention riesce a focalizzarsi su aree rilevanti dell'immagine in input.

Nell'esempio mostrato in Figura 2.12, l'output dell'encoder viene passato ad uno strato finale composto da un muli-layer perceptron, una semplice rete feed-forward. Questo risolve un task di image classification sui token ricevuti dal Transformer, e restituisce come output finale la classe d'appartenenza. Questa parte finale è soltanto un possibile esempio di utilizzo: l'output dell'encoder può essere sfruttato anche in altri modi, ad esempio può essere passato in input ad un Transformer decoder unitamente ad altri dati per risolvere task più complessi.
I modelli Vision Transformer sono stati testati sui più importanti benchmark di image classification, tra cui ImageNet. I risultati ottenuti sono migliori rispetto

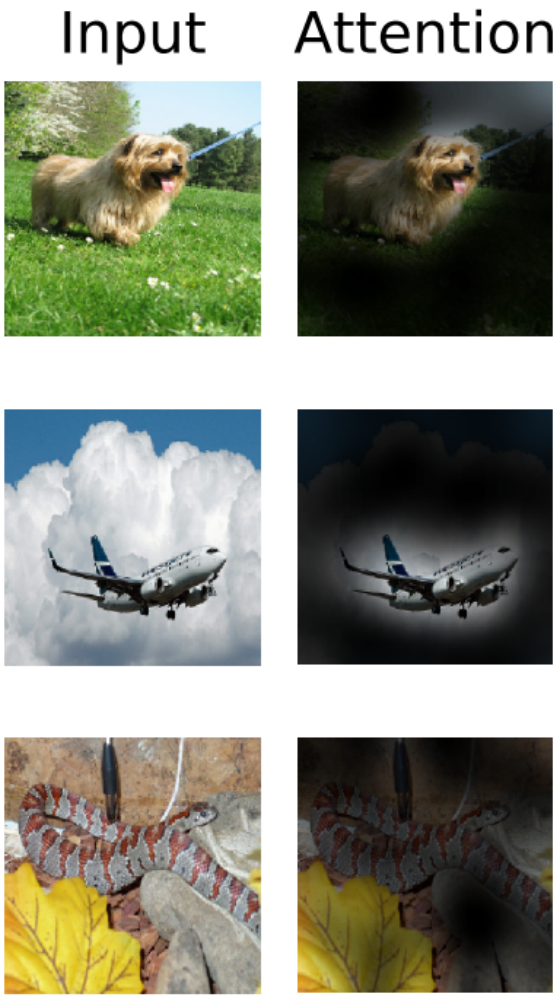


Figura 2.13: Esempio di self-attention applicata in un Vision Transformer.

Fonte: An Image is Worth 16x16 Words, [4]

a quelli raggiunti dalle classiche reti convoluzionali, come mostrato in Figura 2.14. Anche se Vision Transformer supera di poco i record precedenti, gli autori mostrano che a parità di budget computazionale, ViT raggiunge un'accuratezza migliore della famosa rete ResNet [17] sul benchmark ImageNet.

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21k (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	**88.55** ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02	88.4/88.5*
ImageNet ReaL	**90.72** ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54	90.55
CIFAR-10	**99.50** ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06	–
CIFAR-100	**94.55** ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08	–
Oxford-IIIT Pets	**97.56** ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23	–
Oxford Flowers-102	99.68 ± 0.02	**99.74** ± 0.00	99.61 ± 0.02	99.63 ± 0.03	–
VTAB (19 tasks)	**77.63** ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70	–
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k	12.3k

Figura 2.14: Risultati ottenuti da Vision Transformer su task di image classification, come pubblicati in [4].

Fonte: An Image is Worth 16x16 Words, [4]

2.2.7 Bidirectional Encoder Representations from Transformers (BERT)

Bidirectional Encoder Representations from Transformers, o in breve BERT, è un modello basato sull'architettura Transformer sviluppato e rilasciato da Google nel 2018. Questo modello, introdotto con il paper *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* [18], ha superato i record raggiunti dalle classiche reti ricorrenti e LSTM quando applicato a problemi di natural language processing.

L'architettura di BERT è basata su quella del Transformer, ed utilizza meccanismi di self-attention e cross-attention per modellare le relazioni presenti all'interno di una sequenza di input, senza utilizzare convoluzioni, layer di memoria di stato o i gates presenti nelle LSTM.

In particolare, BERT è costituito da uno stack di Transformer encoder, come mostrato in Figura 2.15. Il numero di moduli encoder dipende dalla versione del modello, quella di base ne contiene dodici. Ogni encoder ha la stessa struttura illustrata in Figura 2.8, cioè quella del Transformer originale.

BERT è un modello bidirezionale, perchè riesce a rappresentare ogni parola unitamente al suo contesto all'interno della frase, reperendo le informazioni sul contesto sia a sinistra che a destra della parola stessa. In questo modo, il modello ha a disposizione un quadro più chiaro e completo della semantica assunta dalla frase, portando così benefici alla qualità degli embedding.

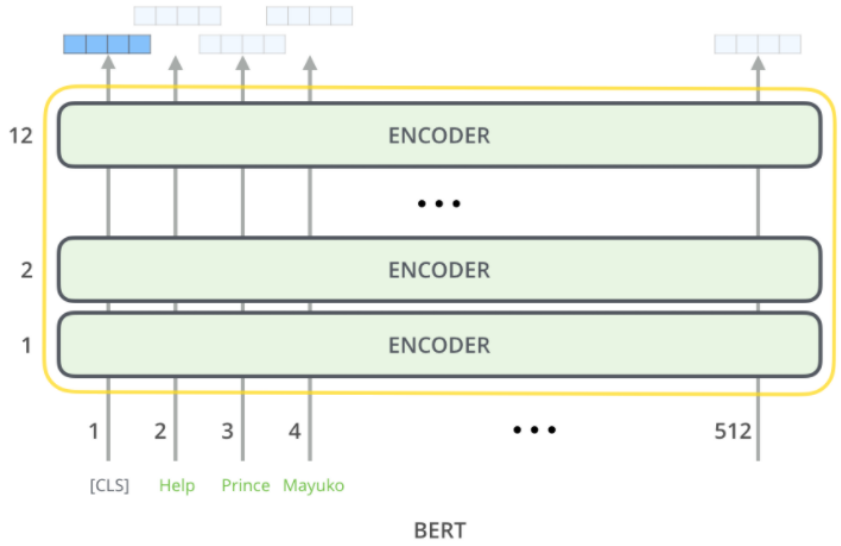


Figura 2.15: Architettura ad alto livello di BERT.

Fonte: The illustrated Bert

Una importante caratteristica di BERT, che ha contribuito a determinarne la fama, consiste nel modo in cui la fase di addestramento è organizzata. Possiamo dividere il training in due fasi: pre-training e fine-tuning.

La fase di pre-training consiste nell'addestramento del modello su certi specifici task, effettuato dagli autori del modello stesso su grandi quantità di dati. Questa fase comporta costi computazionali molto elevati, ma è necessario effettuarla solamente una volta: il modello pre-addestrato, infatti, può essere riapplicato a task diversi o più specifici di quelli oggetto del pre-training. Per ottenere buoni risultati anche su task diversi, il modello deve solitamente venire riaddestrato, ma è sufficiente un più breve addestramento su una quantità di dati molto inferiore a quella iniziale. Questa seconda fase è chiamata fine-tuning, e ottimizza le performance del modello su un task specifico sfruttando il transfer learning.

Nel caso specifico di BERT, la fase di pre-training è stata effettuata su due task, masked language modeling, illustrato in Figura 2.16, e next sentence prediction. Nel masked language modeling, il 15% dei token viene mascherato, e il modello deve dedurre i token nascosti sfruttandone il restante 85%. Nel next sentence prediction, al modello viene fornita una frase. Il task consiste, presentata una seconda frase, nel determinare se è probabile che la seconda frase segua realmente la prima.

BERT è stato pubblicato dopo questa frase di pre-training, e altri utenti possono sfruttare il transfer learning per applicare il modello ai propri task,

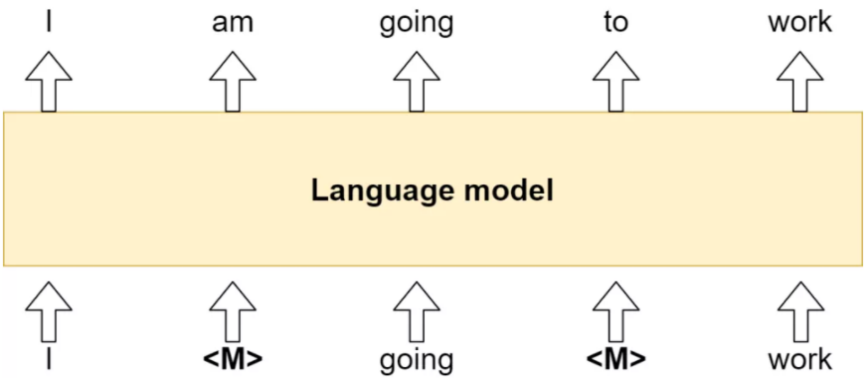


Figura 2.16: Illustrazione del task di masked language modeling.

Fonte: machinecurve.com

effettuando solo una molto meno costosa fase di fine-tuning.

2.2.8 Generative Pre-trained Transformer 2 (GPT-2)

Generative Pre-trained Transformer 2, o in breve GPT-2, è un modello Transformer causale, o unidirezionale, a differenza di altri modelli Transformer bidirezionali come BERT [18]. GPT-2 è stato introdotto da OpenAi nel 2019, con il paper *Language Models are Unsupervised Multitask Learners* [19], e beneficia di una fase di pre-training su un grande corpus testuale di circa 40GB ricavato da circa otto milioni di pagine web.

Il modello GPT-2 è addestrato con l'obiettivo di generare testo in linguaggio naturale. In particolare, dato un insieme di parole, il modello deve predire la successiva. GPT-2 ha dimensioni notevoli, considerati gli 1.5 miliardi di parametri di cui si compone. La spiccata diversità del dataset di addestramento, costituito da pagine web, consente al modello di raggiungere buoni risultati con input provenienti da molti diversi contesti.

GPT-2 è adatto a risolvere molti task, tra cui la traduzione di testi in linguaggio naturale, il question answering, e la generazione di riassunti, ma fatica a predire output più lunghi di qualche frase. In questi casi, spesso l'output diventa ripetitivo e poco coerente.

Il modello GPT-2, come illustrato in Figura 2.18, è composto da uno stack di Transformer Decoder, in diretta contrapposizione a BERT, che è invece costituito da uno stack di Encoder. Una conseguenza di questa scelta, è che GPT-2 è in grado di generare un solo token per volta. Ogni token generato viene poi aggiunto in coda alla sequenza di input, che viene poi

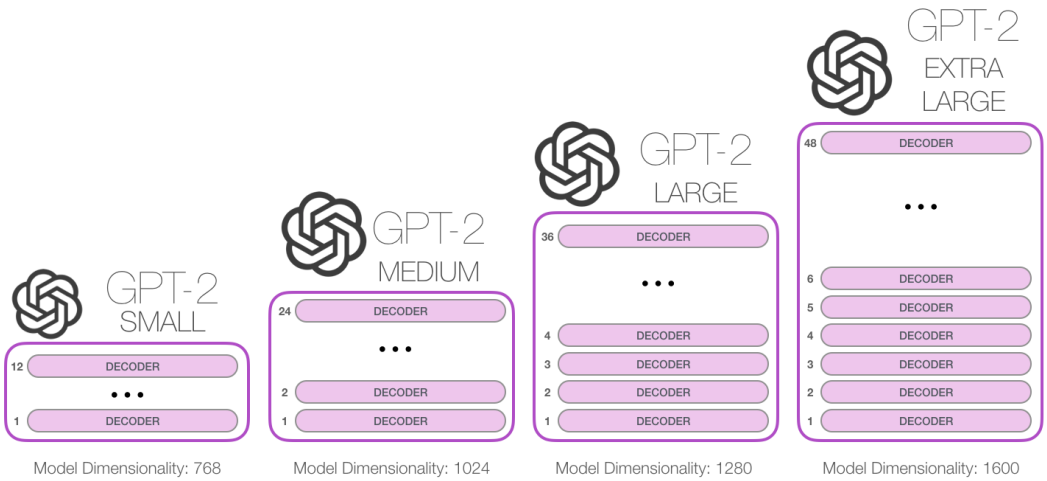


Figura 2.17: Le dimensioni delle diverse versioni di GPT-2

Fonte: The illustrated GPT-2

usata per la generazione al passo successivo. Questo meccanismo è denominato auto-regressione.

L'auto-regressione in GPT-2 è molto simile al meccanismo realizzato nelle reti ricorrenti classiche tramite il layer di memoria di stato, e comporta notevoli benefici nei task generativi. Il compromesso, è che per questo motivo GPT-2 non è in grado di utilizzare il contesto in modo bidirezionale, come invece può fare BERT. Infatti, GPT-2 usa il contesto in modo unidirezionale, left-to-right più precisamente, come le classiche reti ricorrenti e LSTM.

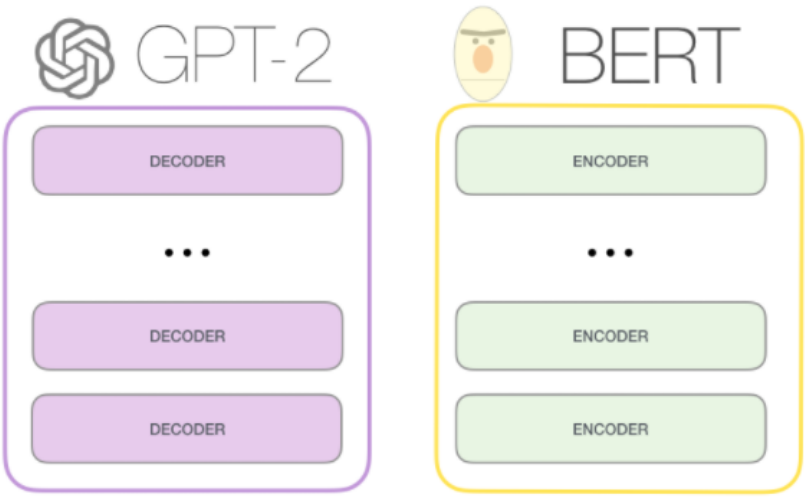


Figura 2.18: Architettura ad alto livello di GPT-2, posta in contrasto con quella di Bert

Fonte: The illustrated GPT-2

Capitolo 3

Modellazione del progetto

Valutiamo, tra le tecnologie discusse nel capitolo precedente, quali siano le più adatte a risolvere il task che ci siamo posti. Dopo aver selezionato i dati più utili tra quelli disponibili, proponiamo l'architettura del modello sviluppato per questa tesi.

3.1 Analisi dei dataset

In questo progetto, ci poniamo l'obiettivo di generare una caption che descriva al meglio una data immagine raffigurante un prodotto dell'industria della moda. Le immagini sono parte del dataset Fashiongen, e come specificato nella sezione 1.1, hanno dimensione 256x256 pixel.

Per prima cosa, consideriamo quali campi del nostro dataset potranno esserci utili per generare una caption. Ricapitolando, ogni prodotto contenuto nel dataset riporta diverse informazioni:

- Quattro fotografie del prodotto, scattate da diverse angolazioni. Questo campo è fondamentale, e costituirà l'input del nostro sistema.

- Una caption in lingua inglese, che ne descrive in modo molto dettagliato le caratteristiche. Questo è esattamente l'output che vogliamo ottenere.

- Il nominativo associato al prodotto. Il nominativo non descrive abbastanza dettagliatamente il prodotto, e non fornisce informazioni aggiuntive rispetto alla caption. Per questo motivo, non utilizzeremo questo campo.

- La categoria di abbigliamento a cui il prodotto appartiene. Questa informazione è già contenuta nella caption, consideriamo quindi questo campo come ridondante.

- La sotto-categoria di abbigliamento a cui il prodotto appartiene. Similmente alla categoria, è un informazione ridondante.
- I materiali di cui il prodotto è composto. La caption descrive già il prodotto in maniera adeguata, incluso il materiale di cui questo si compone. Consideriamo quindi anche questo campo ridondante.
- Se il prodotto è rivolto al pubblico maschile, femminile oppure se il capo è unisex. Questa informazione potrebbe risultare utile a migliorare la qualità della caption generata, in quanto le didascalie fornite nel dataset generalmente non contengono questo dato.
- Il brand dell'azienda produttrice. Non siamo interessati ad includere questa informazione nella caption, perciò scartiamo il campo.
- Un identificativo che indica l'angolazione da cui il prodotto è stato ritratto. Questo dato potrebbe potenzialmente aiutare il sistema a distinguere più accuratamente le immagini.

Riassumendo, abbiamo due campi fondamentali, l'immagine e la caption. L'immagine, per definizione del problema, costituirà il nostro input. La caption, invece, rappresenta l'output desiderato ed è quindi il nostro target. In altre parole, possiamo considerare l'immagine come una variabile indipendente x, e la caption come una variabile y dipendente da x. Il nostro sistema, quindi, può essere pensato come una funzione f tale che:

$$f(x) = y$$

Abbiamo anche altri campi che contengono informazioni utili, come la categoria di appartenenza del prodotto e se questo è rivolto al pubblico maschile o femminile. Tuttavia, la caption fornisce già una descrizione adeguatamente precisa del prodotto. Per questa ragione, è probabile che i benefici portati dall'utilizzo di questi dati non compensino lo svantaggio causato dalla maggiore complessità del problema, che si otterrebbe utilizzando in input dati così eterogenei. Al momento, quindi, ci limiteremo ad usare immagine e caption.
Il problema così formulato si presta molto bene ad essere affrontato con l'uso di tecniche di machine learning. Il nostro sistema, quindi, si baserà su un modello che, presa in input un'immagine, restituirà la caption corrispondente.

Come abbiamo detto, oltre alle immagini di input, abbiamo a disposizione anche le caption che le descrivono. Essendo queste il nostro target, siamo in possesso di dati etichettati e possiamo quindi risolvere il problema tramite tecniche di learning supervisionato.
Essendo l'input costituito da immagini, per altro anche varie e dettagliate,

la fase di selezione delle features si rivelerebbe troppo ardua e complessa per essere svolta manualmente. Per questo motivo, ci avvarremo di tecniche di deep learning. In questo modo, le features verranno selezionate automaticamente dal modello, in quando considerate dei parametri, i quali pesi verranno ottimizzati durante la fase di addestramento.
Molti importanti studi nell'ambito dell'image captioning hanno utilizzato principalmente metodi di deep learning basati su reti convoluzionali e reti LSTM. Minore è invece il numero di studi che utilizzano architetture encoder-decoder basate su Transformers, anche per via del poco tempo trascorso dall'introduzione da questa tecnologia, specialmente nelle sue forme applicabili alla visione artificiale. In questa tesi, scegliamo di utilizzare una architettura encoder-decoder full-Transformer, che sfrutti il Transfer Learning. In particolare:

- Vision Transformer (ViT) viene scelto come modulo encoder;
- Generative Pre-trained Transformer 2 (GPT-2) viene utilizzato come decoder;

In Figura 3.1 è illustrata l'architettura del modello così ottenuto.

L'utilizzo di una architettura full-transformer in un task di visione artificiale non necessariamente porterà a risultati migliori rispetto ad una architettura basata su reti convoluzionali. Tuttavia, nel contesto di questa tesi è interessante effettuare questa prova, al fine di testare una così recente tecnologia ed analizzarne i risultati in questo complesso task.

Le immagini in input verranno quindi per prima cosa elaborate da Vision Transformer. Per prima cosa, queste devono venire pre-processate in modo che siano divise in sezioni della stessa dimensione, come mostrato in Figura 3.1. Successivamente, alle sezioni viene aggiunto il *positional encoding*. I dati così ottenuti costituiscono l'input di Vision Transformer, che li elaborerà svolgendo la stessa funzione che un Transformer encoder standard assume in un problema di natural language processing.

L'output di Vision Transformer è costituito da embedding, che contengono informazioni su ogni sezione di immagine. Le codifiche così ottenute prescindono dal tipo di dato originale, ovvero assumono la stessa forma sia che il dato di partenza sia una frase in linguaggio naturale oppure un immagine. Per questo motivo, possiamo utilizzare l'output di Vision Transformer come input per GPT-2, che tratterà gli embedding come prompt per la generazione di testo.

GPT-2 assume il ruolo di decoder nella nostra architettura. Questo modello, come riportato in 2.2.8, si compone di uno stack di Transformer decoder. L'output di GPT-2 consiste, dopo le dovute operazioni di decodifica, in una frase in linguaggio naturale che descrive l'immagine di input, ovvero la sua caption.

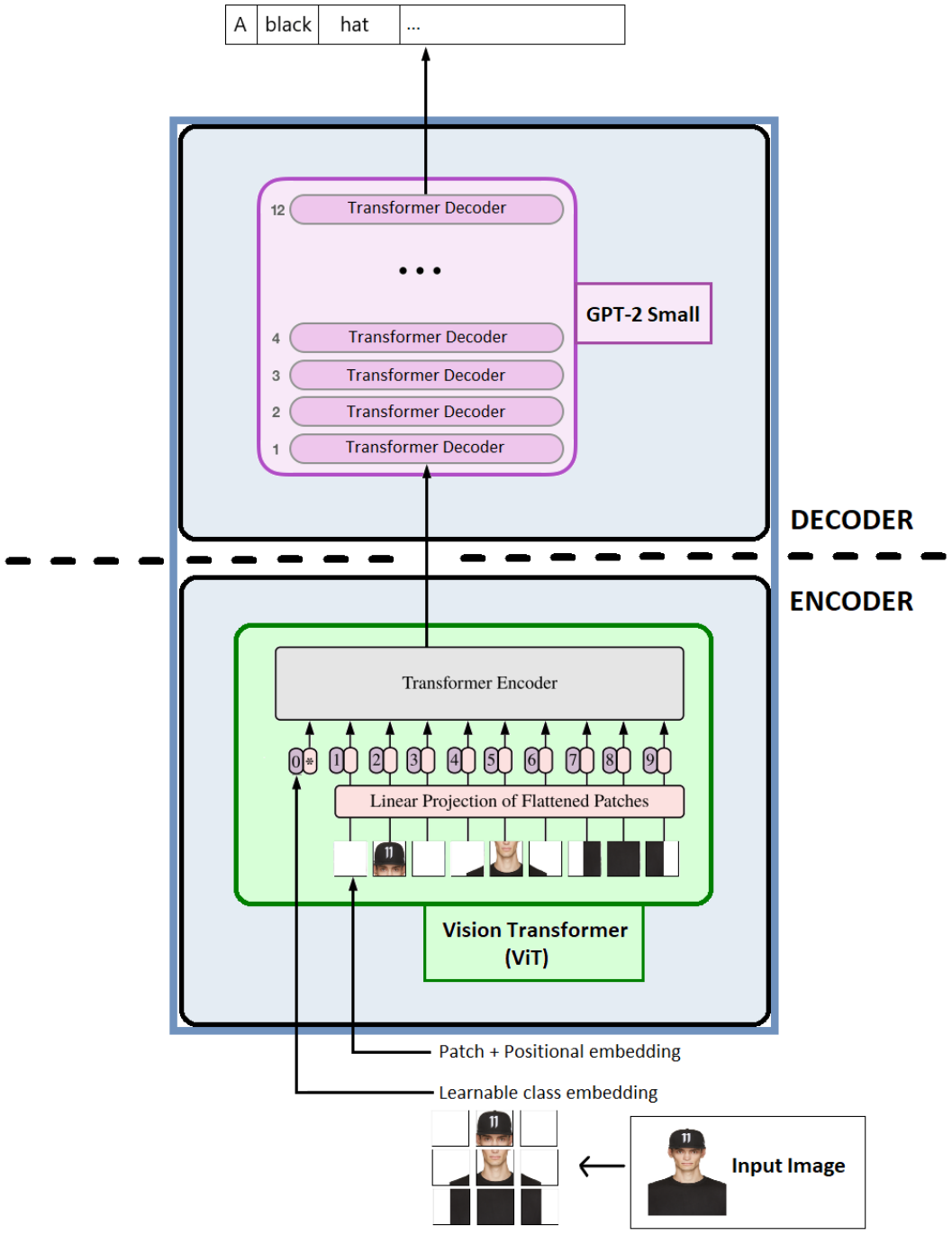


Figura 3.1: Architettura scelta per il nostro modello. Viene mostrato un caso di inferenza: durante l'addestramento, oltre all'immagine avremo in input anche la caption di riferimento, che il decoder utilizza per effettuare learning supervisionato.

Abbiamo quindi un'architettura adeguata a risolvere l'obiettivo di questa tesi. Resta ancora da determinare con quali modalità effettuare le fasi di pre-processing dei dati e di addestramento del modello, in modo da ottenere i risultati desiderati.

Capitolo 4

Sviluppo

Descriviamo la fase di preparazione del dataset. Illustriamo quindi in maggiore dettaglio il modello sviluppato, delineandone le scelte di progettazione e la fase di addestramento.

4.1 Configurazione del progetto

4.1.1 Librerie

Nello sviluppo di questo progetto ci serviremo di diverse librerie. Tra le principali, abbiamo:

- Huggingface - per la gestione del training e l'accesso a implementazioni di modelli di deep learning
- PyTorch - per la gestione del calcolo parallelo, l'API dei tensori e molto altro
- h5py - per il caricamento lazy da disco del dataset Fashiongen

4.1.2 Componenti del modello

La configurazione dei componenti del modello è contenuta in una Python class che definisce i vari moduli di cui il modello si compone. Le informazioni memorizzate includono quali siano l'encoder e il decoder utilizzati, specificando eventualmente il checkpoint di pre-training fornito da *Huggingface* che si desidera caricare, oltre al tokenizer e al pre-processore di immagini corrispondente. Questa classe ha lo scopo di semplificare la sostituzione dei componenti del modello al fine di svolgere operazioni di testing.

4.1.3 Dataset class

Ricordiamo che il dataset Fashiongen è suddiviso in due parti:

- Training set - contenente circa 260000 prodotti, utilizzato per addestrare il modello
- Validation set - contenente 32000 prodotti, utilizzato per valutare l'accuratezza del modello

Ciascuno dei due set viene utilizzato tramite una classe Python, che scelto un indice può restituire l'immagine del prodotto corrispondente e la sua caption, provvedendo eventualmente ad effettuare il pre-processamento. Utilizziamo queste classi per fornire i dati necessari al modello durante le fasi di training e validation.

4.2 Preprocessamento

4.2.1 Preparazione delle immagini

Per prima cosa, le immagini di input devono venire elaborate dall'encoder del modello, ovvero Vision Transformer. Come discusso nel capitolo precedente, è necessario preparare le immagini in una prima fase di pre-processing, perché possano essere utilizzate come input.

La prima fase di pre-processing consiste nella divisione di ogni immagine in più parti aventi le stesse dimensioni, che vengono poi disposte in sequenza. Le dimensioni assunte da queste sezioni di immagine sono fisse e stabilite a priori. Inoltre, è importante che non ci sia sovrapposizione tra queste.

Una volta suddivisa l'immagine in più parti, dobbiamo:

1. Ridurre la dimensionalità delle sezioni di immagine, trasformandole in *embedding lineari*
2. Aggiungere la *codifica posizionale*
3. Formare una *sequenza* dall'insieme delle sezioni, da fornire in input a un Transformer encoder standard
4. Effettuare la fase di *pre-training* supervisionato su una grandissima quantità di dati
5. Effettuare il *fine-tuning* sul dataset Fashiongen

La fase di pre-training richiederebbe una quantità vastissima di dati e altissimi costi computazionali. Sarebbe quindi una operazione costosissima, ma fortunatamente gli autori di Vision Transformer hanno reso pubblico il modello su cui è già stato effettuato il pre-training. Sfruttiamo quindi il Transfer learning per adattare il modello pre-addestrato ai nostri bisogni, effettuando solamente la fase di fine-tuning. La fase di fine-tuning consiste nel vero e proprio addestramento del nostro modello, e verrà descritta in seguito.

Tra le diverse versioni disponibili, scegliamo il modello Vision Transformer di dimensioni minori, che utilizza sezioni di immagine di dimensione 16x16, e di risoluzione di 224x224 pixel durante il fine-tuning. La ragione per cui si usa una risoluzione maggiore durante il fine-tuning, è che gli autori sostengono che questo permette di raggiungere risultati migliori. Dovremo quindi, durante il pre-processing, elaborare le immagini per ottenere immagini delle dimensioni sopra descritte.

Le immagini, vengono pre-processate tramite il feature extractor fornito dalla comunità *Huggingface* appositamente per l'uso con Visual Transformer. Il feature extractor è memorizzato come attributo nella classe del dataset, ed effettua il pre-processing per una data immagine solo quando questa viene richiesta in output alla classe del dataset. Inoltre, le immagini vengono normalizzate in modo che per ognuno dei tre canali media e deviazione standard siano pari a 0.5.

4.2.2 Preparazione delle caption

Per addestrare il modello con il learning supervisionato, dovremo affiancare alle immagini in input le loro caption corrispondenti, tra quelle fornite nel dataset. Vision Transformer non necessità di ricevere in input le caption, in quanto la sua funzione è di creare gli embedding delle immagini. Le caption devono invece essere passate in input al decoder, ovvero GPT-2, durante l'addestramento. Le caption contenute nel dataset sono in linguaggio naturale, mentre GPT-2 necessita che queste siano codificate in embedding con uno specifico formato. Quetso modello richiede che le frasi in input siano pre-processate tramite l'utilizzo di un Byte-Pair Encoding tokenizer.
Il Byte-Pair Encoding, o BPE, è un algoritmo di compressione dei dati introdotto già nel 1994 nel paper *A New Algorithm for Data Compression* [20]. La sua applicazione nel campo del natural language processing è invece stata proposta più recentemente, in particolare nel 2015 con il paper *Neural Machine Translation of Rare Words with Subword Units* [21].
Il Byte-Pair Encoding originale è un algoritmo di compressione dei dati in cui la coppia più ricorrente di bytes viene sostituita con un byte non presente all'interno di questi dati. Questo processo è illustrato in Figura 4.1, riportata

dal paper originale [20]. Nel natural language processing, ne viene utilizzata una variante in cui le parole più comuni nel vocabolario del modello vengono rappresentate con un singolo token, mentre le parole più rare, inteso come meno frequenti vengono suddivise in più token. Il risultato di questo processo, quindi, consiste nella trasformazione di una stringa di input (la nostra caption) in una lista di token, dove ogni token è un numero intero.

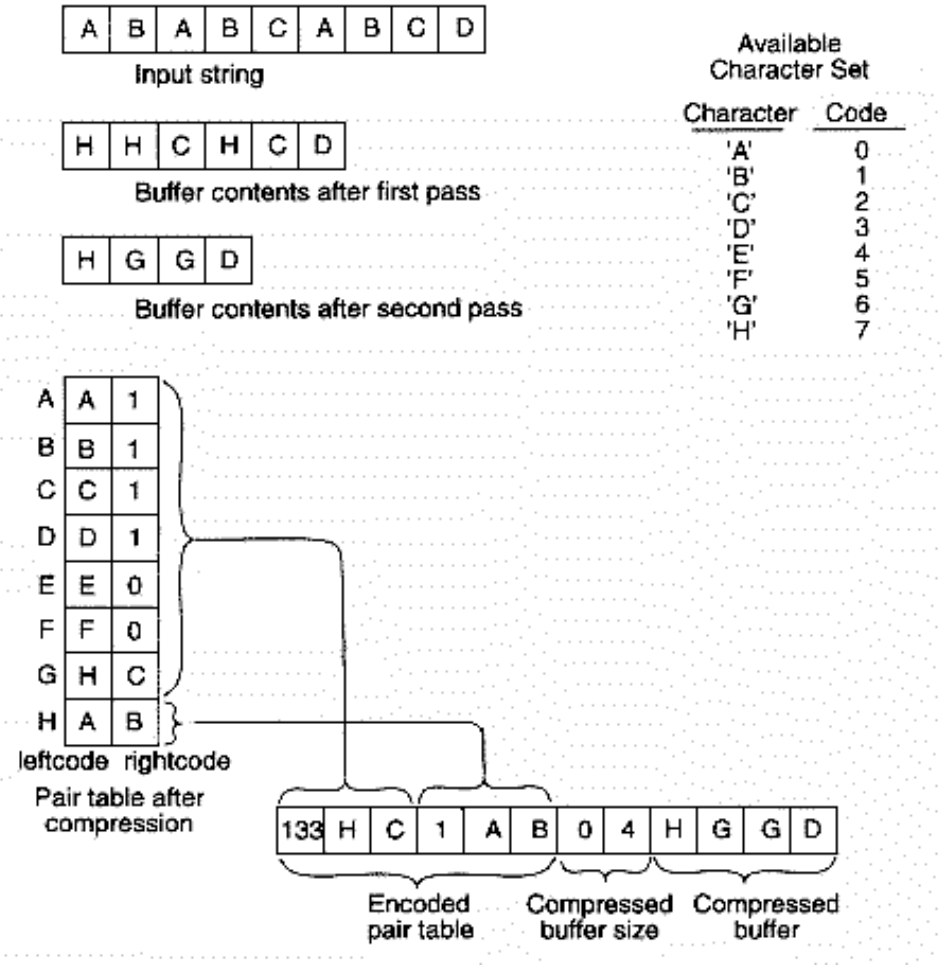


Figura 4.1: Illustrazione dell'algoritmo Byte-Pair Encoding (BPE).

Fonte: A New Algorithm for Data Compression *[20]*

Per effettuare il pre-processing delle caption, ci serviremo principalmente dell'implementazione *Huggingface* dell'apposito tokenizer per GPT-2. Per poter sfruttare al meglio l'API del tokenizer utilizzato, tutte le caption vengono pre-processate in blocco prima che inizi l'addestramento. Questo ci permette di

ottimizzare la fase di padding del testo, in modo da portare facilmente tutte le caption ad avere la stessa lunghezza di quella più lunga contenuta nel dataset.

Capitolo 5

Esperimenti

5.1 Addestramento

L'addestramento, come già discusso, avviene in modalità supervisionata utilizzando le immagini dei prodotti e le loro caption come input. Il modello viene addestrato per diverse epoche su questi dati, con l'obiettivo di minimizzare la funzione di loss scelta per il training. L'ottimizzazione dei parametri avviene tramite back-propagation, come di norma nelle reti neurali.

Il processo di addestramento viene gestito dalla classe CustomTrainer, che consiste in una sotto-classe del Trainer fornito da *Huggingface* per l'utilizzo con modelli sequence-to-sequence, adattato per poter lavorare con input costituito da immagini. I valori associati ai vari iper-parametri durante l'addestramento sono mostrati nella sezione 6.7, mentre la configurazione del modello è visibile in sezione 6.4. Tra i più importanti iper-parametri, abbiamo:

- learning rate iniziale pari a 4e-5
- weight decay pari a 0.01
- un totale di 500 step di riscaldamento per l'optimizer
- caption generate con beam search, con n-beams uguale a 3
- lunghezza massima per le caption generate di 64 token
- repetition-penalty, la penalità per le ripetizioni di token nella generazione, pari a 10.0

La scelta della funzione di Loss ha un impatto molto significativo sui risultati raggiunti. In questa tesi, testiamo due modelli diversi, con la stessa architettura ma addestrati con due funzioni di Loss diverse: Cross-Entropy e Cross-Entropy sommata a Triplet Loss.

5.1.1 Cross-Entropy Loss

Il concetto di entropia nella teoria dell'informazione deriva dal paper *A Mathematical Theory of Communication* [22] pubblicato nel 1948 da Claude E. Shannon. In questo paper, l'entropia di una variabile aleatoria è definita come la quantità media dell'informazione, sorpesa, o incertezza derivata dai possibili risultati della variabile stessa.

La cross-entropy si fonda su questo concetto di entropia, e misura la differenza esistente tra due distribuzioni di probabilità per una certa variabile aleatoria. Per calcolare la cross-entropy prendiamo in input due distribuzioni di valori, una reale, che indichiamo con $p(x)$, e una approssimata, indicata con $q(x)$. La cross-entropy si calcola quindi come:

$$H(p, q) = -\sum_{\forall x} p(x) \cdot log(q(x))$$

La Cross-Entropy è una funzione che si presta molto bene ad essere utilizzata come funzione di Loss in una rete neurale. Viene spesso applicata nei problemi di classificazione multi-classe, ma funziona bene anche se applicata ad altri task. Nel nostro caso, $p(x)$ corrisponde alla caption reale fornita dal dataset per il prodotto x, mentre $q(x)$ corrisponde alla caption generata dal nostro modello per il prodotto x, ovvero l'output del modello stesso. Essendo le due distribuzioni di valori dei vettori, possiamo calcolare la Cross-Entropy Loss come:

$$L = -y \cdot log(\hat{y})$$

Dove y contiene i valori reali, e $\hat{y}$ quelli predetti dal modello. E' importante notare che nella formula sopra il prodotto è un prodotto scalare (*dot product*) tra vettori.

5.1.2 Triplet Loss

La Triplet Loss è una funzione che prende in input tre parametri:

- Un valore di riferimento, chiamato àncora
- Un valore corrispondente a quello di riferimento, chiamato positivo
- Un valore che consideriamo diverso (o dissimile) dall'àncora, chiamato negativo

In un problema di classificazione, ad esempio, potremmo considerare un input x_1 che appartiene alla classe A. Assumiamo x_1 come valore di riferimento, o àncora. Preso un secondo input x_2, se anche questo appartiene alla classe A,

allora diciamo che corrisponde ad x_1 e quindi è un input positivo. Altrimenti, diciamo che è negativo. L'appartenenza ad una classe è solo un esempio di come si possono suddividere gli input in positivi e negativi, il criterio va scelto in base al problema affrontato.

Una volta suddivisi gli input, la Triplet Loss porterà a minimizzare la distanza tra l'àncora e l'input positivo, e massimizzare quella tra àncora e input negativo. In particolare, se indichiamo con a_i il valore àncora al passo i, con p_i il valore positivo, e con n_i il valore negativo, la Triplet Loss L sarà:

$$L(a, p, n) = max(d(a_i, p_i) - d(a_i, n_i) + margin, 0)$$

Dove $d(x_1, x_2)$ indica la distanza tra il valore x_1 e il valore x_2. La distanza può essere calcolata in diversi modi, ad esempio si può considerare la distanza euclidea. In questo caso:

$$d(x_1, x_2) = ||x_1 - x_2||_2$$

Il margine viene sommato alla Loss, ed è un iper-parametro stabilito a priori. In ogni caso, deve essere maggiore di zero.

Una prima formulazione della Triplet Loss è stata introdotta nel 2003 in *Learning a Distance Metric from Relative Comparisons* [23]. In questa prima versione non era ancora stato introdotto il concetto di àncora. La versione che utilizziamo in questa tesi è quella implementata in PyTorch, che fa riferimento al paper *Learning local feature descriptors with triplets and shallow convolutional neural networks* [24].

5.1.3 I due modelli

Come accennato in precedenza, andremo ad addestrare due modelli, in particolare:

1. Il primo modello, che consideriamo il modello base, verrà addestrato usando solo la Cross-Entropy Loss come obiettivo

2. Il secondo modello verrà addestrato usando una funzione di Loss composita, ottenuta sommando Cross-Entropy e Triplet Loss

Scegliamo di sommare la Triplet Loss alla Cross-Entropy per cercare di ottenere in output caption più specifiche, ovvero che riescano a descrivere più nel dettaglio le immagini di input, piuttosto che usare termini generici e poco discriminativi. Questo obiettivo può essere raggiunto scegliendo opportunamente gli input àncora, gli input positivi e quelli negativi nel calcolo della Triplet Loss.

Per calcolare la Triplet Loss per l'input x, effettuiamo le scelte in questo modo:

- Il valore àncora è l'output del nostro modello per l'input x, ovvero la caption generata
- Il valore positivo, è l'embedding dell'immagine in input, fornito da Visual Transformer (quindi l'output del nostro modulo encoder)
- Il valore negativo è l'embedding fornito dall'encoder (Visual Transformer) per un'immagine simile a quella di input x

Definire un concetto di similarità tra immagini è un problema di per sè non banale, ancora di più se bisogna distinguere i dettagli in capi di abbigliamento. In questa tesi, sfruttiamo la suddivisione dei prodotti in sottocategorie fornita con il dataset Fashiongen. Consideriamo quindi due prodotti simili se appartengono alla stessa sottocategoria (es. giacche a vento, orologi, t-shirt), e dissimili altrimenti.

Per un dato di input x, quindi, assumeremo la caption generata su x come valore àncora, l'embedding dell'encoder sull'immagine x come valore positivo, e come valore negativo l'embedding dell'encoder su una immagine scelta casualmente dal dataset, che appartenga alla stessa sottocategoria del prodotto x.

Lo scopo di questa scelta è il seguente: per minimizzare la Triplet Loss impostata in questo modo, il modello tenderà a minimizzare la distanza euclidea tra l'embedding dell'immagine x e quello della caption che la descrive, allo stesso tempo massimizzando la distanza tra caption generata ed un'immagine simile ad x. Così facendo, otterremo delle caption più discriminatorie, che descrivono l'immagine in input usando termini più specifici. In Figura 5.1 è illustrato questo processo.

Perchè abbia senso comparare le distanze tra i valori àncora, positivo e negativo, questi devono avere la stessa dimensionalità. Essendo sia il valore positivo che quello negativo ottenuti in output dallo stesso modulo encoder (Vision Transformer), entrambi avranno 768 dimensioni. Il valore àncora, però, è una frase in linguaggio naturale, e non può essere direttamente comparata con questi altri due valori. Per questo motivo, abbiamo bisogno di calcolare l'embedding del valore àncora, cioè la caption generata, in modo che abbia 768 dimensioni.

Non possiamo ottenere questo embedding usando Vision Transformer, perchè questo è pensato per elaborare immagini. Ci serviamo perciò di un altro modello: scegliamo di utilizzare Bert, che come discusso nella sezione 2.2.7, è adatto a questo tipo di task. In particolare, andremo ad usare il modello Bert nella sua

versione base, senza effettuare fine-tuning, e senza aggiornare il suo gradiente al momento di creare gli embedding. Infatti, vogliamo che la rappresentazione di Bert sia "statica", nel senso che non vogliamo che Bert venga addestrato insieme al nostro modello.

Le caption codificate con Bert avranno 768 dimensioni, e potranno essere comparate con gli altri embedding nel calcolo della Triplet Loss. Abbiamo deciso arbitrariamente di utilizzare Bert per questo incarico, ma anche altri modelli basati su architetture simili come RoBerta [25] avrebbero funzionato altrettanto bene.

5.2 Generazione delle caption

5.2.1 Greedy Search

GPT-2 restituisce in output una distribuzione di probabilità, che indica quali sono le parole che più probabilmente seguiranno la sequenza data in input. Per generare una caption, dobbiamo scegliere una parola per volta tra quelle in output, scegliendo quella di probabilità maggiore, e aggiungerla alla frase generata. Ogni volta che scegliamo una parola, la sequenza di input cambia, quindi questo modo di effettuare la generazione è detto *auto-regressivo*.

Se selezioniamo ad ogni passo la parola di probabilità maggiore, stiamo utilizzando la tecnica del *greedy-search*. Questo metodo, a causa della sua natura greedy, tende a mancare parole altamente probabili se queste vengono dopo termini con probabilità bassa. Per alleviare questo problema, viene introdotta la tecnica del *beam-search*.

5.2.2 Beam Search

La tecnica del beam-search memorizza un dato numero di ipotesi ad ogni passo, e alla fine sceglie quella più probabile. Il numero di ipotesi memorizzate è un parametro, e come mostrato in Figura 5.2, determina il numero di percorsi esplorati nell'albero della distribuzione ottenuta in output da GPT-2. In questo modo, la tecnica del beam-search riduce il rischio di mancare parole altamente probabili. Memorizzare un numero più alto di ipotesi rende aumenta la probabilità di trovare la soluzione ottimale, al costo di maggiori costi computazionali. In questa tesi, generiamo le caption utlizzando il beam-search per migliorare i risultati.

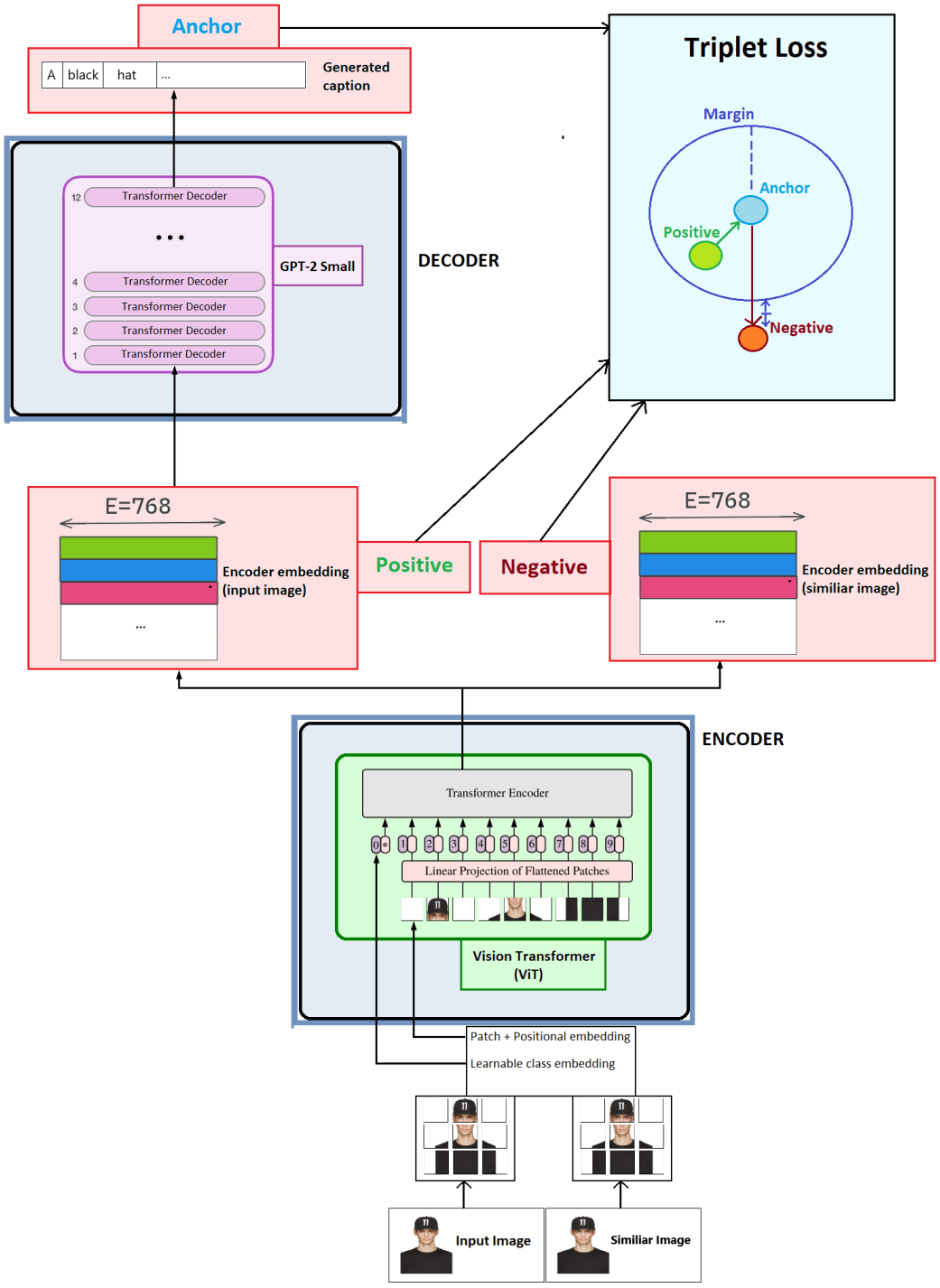


Figura 5.1: Calcolo della Triplet Loss nel nostro modello.

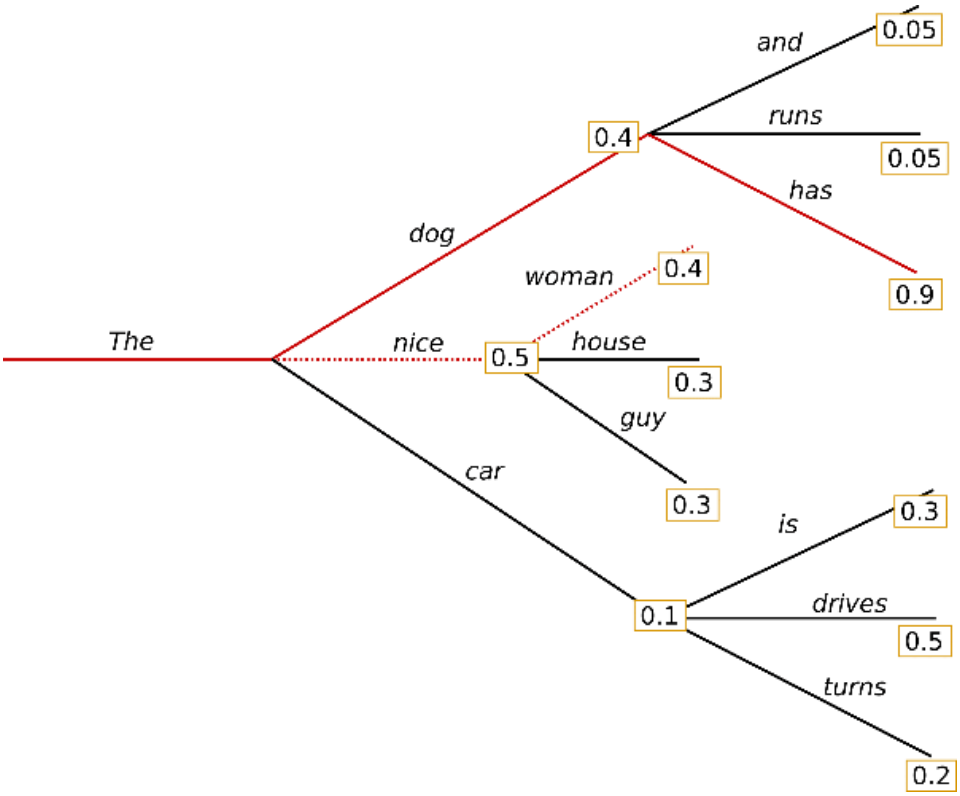


Figura 5.2: Esempio di beam-search, considerando due ipotesi. Questo comporta l'esplorazione di due rami dell'albero delle probabilità

5.3 Risultati

Dopo avere concluso il processo di addestramento discusso nella sezione precedente, analizziamo i risultati ottenuti e valutiamo la qualità delle caption generate.

5.3.1 Andamento della Loss

Per prima cosa, esaminiamo l'andamento delle funzioni di loss utilizzate durante il processo di addestramento, al fine di formare una prima idea sulla bontà dei due modelli.

Cross-Entropy Loss

Durante la fase di addestramento, abbiamo utilizzato la libreria Tensorboard per monitorare l'andamento della Loss. La Figura 5.3 mostra i valori assunti dalla cross-entropy loss al variare dello step di ottimizzazione. Entrambi i modelli sono stati addestrati per dodici epoche. Inizialmente, il primo modello con cross-entropy loss converge ad un minimo più rapidamente rispetto al secondo modello con cross-entropy e triplet loss. Dopo la prima epoca, però, il primo modello comincia a convergere molto lentamente, mentre la convergenza del secondo non rallenta. Dopo dodici epoche di addestramento, abbiamo che il secondo modello con cross-entropy e triplet loss raggiunge una loss pari a 0.28, contro i 0.48 del primo modello. Visto l'andamento della curva, è lecito supporre che continuando l'addestramento, la divergenza tra le curve della loss dei due modelli continuerebbe ad aumentare.

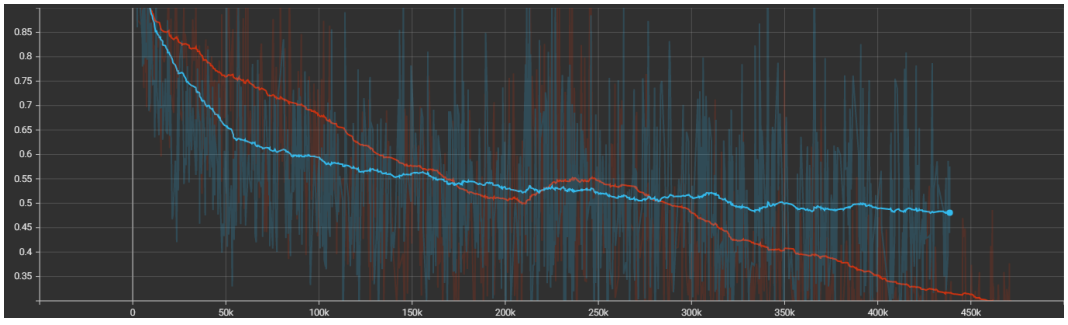


Figura 5.3: La curve della cross-entropy loss per i due modelli, al variare dello step di ottimizzazione. La curva blu rappresenta la funzione associata al primo modello, dove usiamo solo la cross-entropy per l'addestramento. La curva rossa indica la cross-entropy loss per il secondo modello, addestrato sommando cross-entropy e triplet loss

Triplet Loss

La triplet loss calcolata per il secondo modello decresce dapprima molto rapidamente, e quindi continua durante l'addestramento a convergere lentamente verso valori più bassi. Il range di valori assoluto assunto da questa funzione di loss dipende fortemente da un iper-parametro usato per calcolarla, ovvero il margine. Avendo scelto un valore per il margine pari a 0.1, otteniamo valori assoluti molto piccoli per la triplet loss, che dopo dodici epoche raggiunge quota 0.075. La Figura 5.4 mostra l'andamento di questa funzione lungo dodici epoche di addestramento.

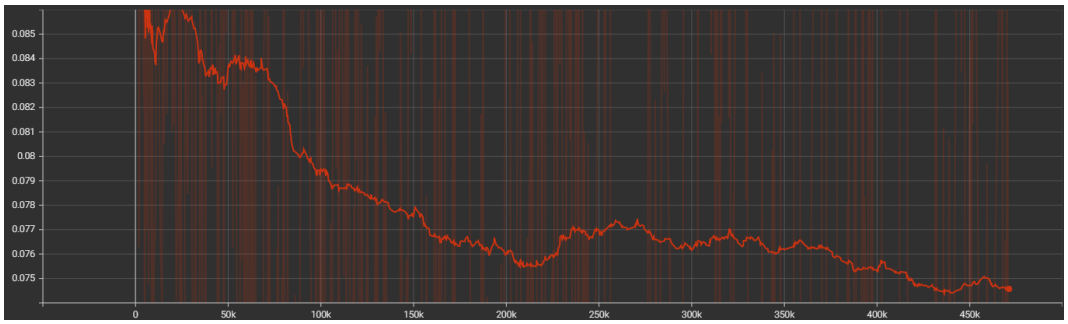


Figura 5.4: Andamento della triplet loss calcolata per il secondo modello, lungo dodici epoche di addestramento. Il range di valori assunto dalla funzione è determinato dal margine scelto come parametro

Loss finale

Vediamo come varia la funzione di loss finale, che usiamo per addestrare effettivamente i modelli. Il primo modello utilizza solamente la cross-entropy loss, perciò la loss finale è la stessa mostrata in precedenza. Il secondo modello, invece, utilizza la somma di cross-entropy e triplet-loss: nella Figura 5.5 vediamo che anche effettuando questa somma, dopo dodici epoche la loss del secondo modello raggiunge valori più bassi rispetto al primo. Questo lascia intendere che la triplet loss come utilizzata in questa tesi ha portato benefici al processo di addestramento.

5.3.2 Metriche di valutazione

Mentre l'andamento della funzione di loss può darci alcune informazioni sullo stato e progresso dell'addestramento, per poter effettivamente valutare la qualità delle caption generate dobbiamo affidarci ad altre metriche, che devono essere il più possibile accurate e correlate al giudizio umano. Valutiamo quindi

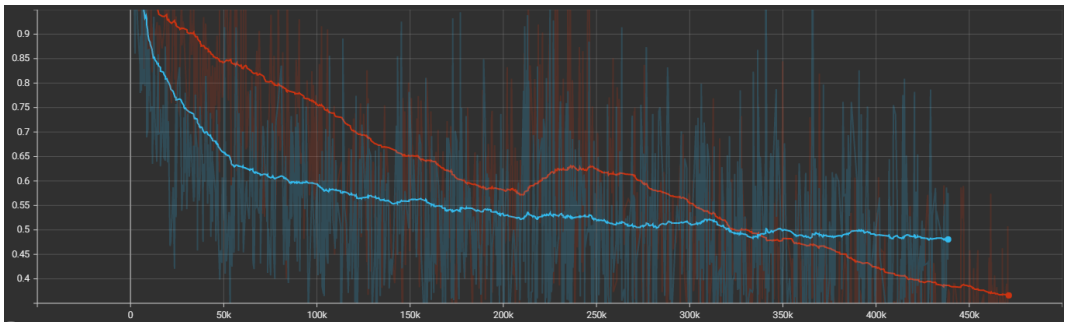


Figura 5.5: La loss effettivamente utilizzata per addestrare i due modelli. La curva blu è associata al primo modello, addestrato solo con cross-entropy. La curva rossa è associata al secondo modello, addestrato sommando cross-entropy e triplet loss

le caption generate dai nostri modelli tramite tre diverse metriche, tra le più utilizzate nel natural language processing.

BLEU

Bilingual Evaluation Understudy Score, o BLEU, è una metrica introdotta da Papineni et al. in *BLEU: a Method for Automatic Evaluation of Machine Translation* [26], sviluppata per valutare la qualità delle traduzioni di linguaggio naturale effettuate da sistemi automatici. Nonostante lo scopo originale, BLEU viene utilizzato anche per valutare le predizioni di modelli generativi contro frasi di riferimento. Questa metrica può assumere valori compresi tra 0 ed 1, dove 1 indica che frase predetta e reale sono identiche, e 0 indica che queste sono completamente diverse (nessun termine in comune). Anche se oggi esistono metriche più precise e complesse, BLEU è ancora largamente utilizzata perché ha un basso costo computazionale, è semplice da interpretare, è indipendente dal linguaggio naturale utilizzato, e tende ad avere una forte correlazione positiva con la valutazione umana.

Meteor

Metric for Evaluation of Translation with Explicit ORdering, o METEOR, è una metrica utilizzata per valutare la qualità di traduzioni automatiche, introdotta da Banerjee et al. in *METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments* [27]. METEOR si basa sul calcolo della media armonica tra precision e recall per ogni monogramma, dove al recall viene attribuito un peso maggiore. Inoltre, questa metrica utilizza lo stemming e il matching di sinonimi oltre al classico matching di parole esatte.

Grazie a queste caratteristiche, METEOR si dimostra una metrica più affidabile rispetto a BLEU.

RougeL

Recall-Oriented Understudy for Gisting Evaluation, o ROUGE, è una collezione di metriche utilizzata per valutare la qualità di riassunti e traduzioni automatiche, introdotta da Chin-Yew Lin in *ROUGE: A Package for Automatic Evaluation of Summaries* [28]. ROUGE comprende cinque diverse metriche, in questa tesi ci serviremo di una di queste, ROUGE-L. ROUGE-L è basato sul calcolo di statistiche LCS, ovvero *Longest Common Subsequence*. Le statistiche LCS prendono in considerazione la similarità degli input a livello di frase, e identificano il più lungo n-gram che compare sia nella frase testata che in quella di riferimento.

BERTScore

BERTScore è una metrica di valutazione automatica introdotta con il paper *BERTScore: Evaluating Text Generation with BERT* [29]. In questa metrica viene calcolato un punteggio di similarità con ogni token della frase di riferimento, ma invece di ricercare un match esatto tra token, la similarità si ottiene facendo riferimento agli embedding ottenuti in output da un modello. Il modello in questione può essere scelto arbitrariamente tra quelli supportati, in questa tesi utilizziamo RoBerta.

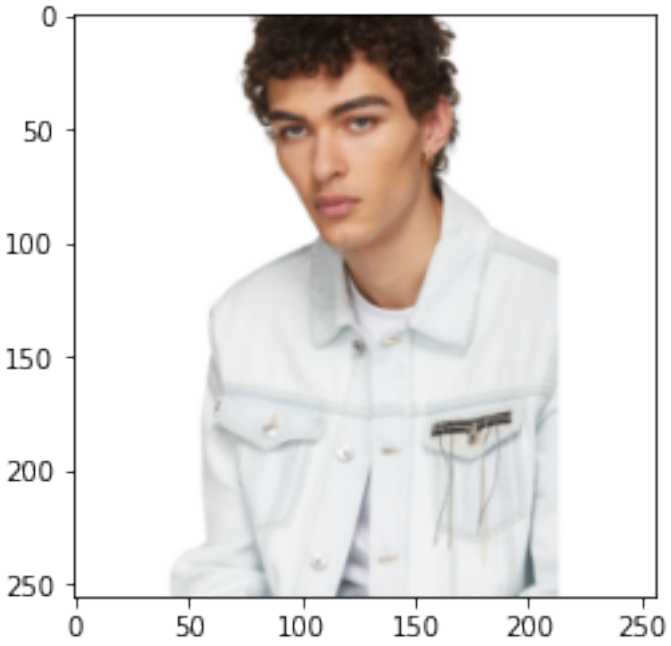
Secondo gli autori, BERTScore ha una migliore correlazione con il giudizio umano rispetto alle metriche preesistenti, ed è più robusto nel valutare campioni impegnativi.

5.3.3 Valutazione a campione

Diamo un giudizio sulla qualità delle caption generate, dapprima esaminandone un campione casuale, e quindi calcolando le metriche BLEU, Meteor, RougeL e BertScore su tutto il set di validazione. Valutiamo per prima cosa le performance dei due modelli su alcune immagini campione, mostrando le caption generate e le metriche raggiunte.

Esempio 1

1. **Immagine di input**



2. **Caption di riferimento**
   Long sleeve denim jacket in'super light' blue. Fading and distressing throughout. Spread collar. Button closure at front. Flap pockets featuring logo flag and deconstructed topstitching in black and off-white at chest. Welt pockets at waist. Single-button barrel cuffs. Open rip at back

3. **Caption del modello 1 (Cross-Entropy)**
   Long sleeve denim jacket in 'bleach white' blue. Fading and distressing throughout. Spread collar. Button closure at front. Flap pockets at chest. Welt pockets at waist. Single-button barrel cuffs. Adjustable buttoned tabs at back hem. Silver-tone hardware. Tonal stitching.

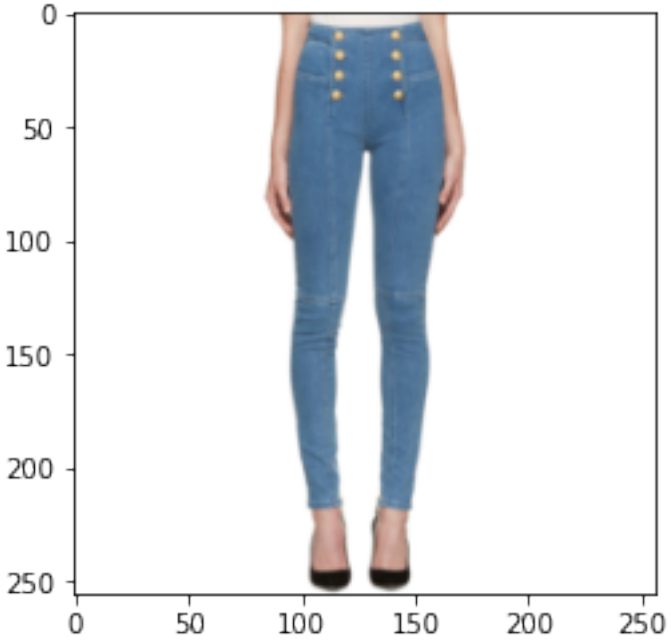
4. **Caption del modello 2 (Cross-Entropy + Triplet)**
   Long sleeve panelled denim jacket in 'extreme aged' blue. Fading and distressing throughout. Spread collar. Button closure at front. Flap pockets at chest. Welt pockets at waist. Adjustable buttoned tabs at back hem. Single-button barrel cuffs. Logo-engraved silver-tone hardware. Contrast stitching in tan.

5. **Metriche raggiunte**

				BertScore		
Modello	BLEU	Meteor	RougeL	Precision	Recall	F1
Entropy	64.17	67.88	68.18	91.30	91.95	91.63
Entropy+Triplet	64.73	66.57	60.87	92.18	92.69	92.43

Esempio 2

1. **Immagine di input**



2. **Caption di riferimento**
   Skinny-fit stretch denim jeans in blue. Mid-rise. Panelled construction. Fading throughout. Mock pockets at front. Zippered vent at cuffs. Zip closure at back waistband. Gold-tone hardware featuring signature engraved detailing. Tonal stitching. Approx. 5" leg opening.

3. **Caption del modello 1 (Cross-Entropy)**
   Slim-fit stretch denim jeans in blue. High-rise. Fading throughout. Seven-pocket styling. Ribbed panel at knees. Zippered vent at cuffs. Zip-fly. Gold-tone hardware featuring signature engraved detailing. Tonal stitching. Approx. 4.5" leg opening.

4. **Caption del modello 2 (Cross-Entropy + Triplet)**
   Skinny-fit stretch denim jeans in washed blue. Mid-rise. Fading throughout. Panelled construction. Six-pocket styling. Ribbed panel at knees.

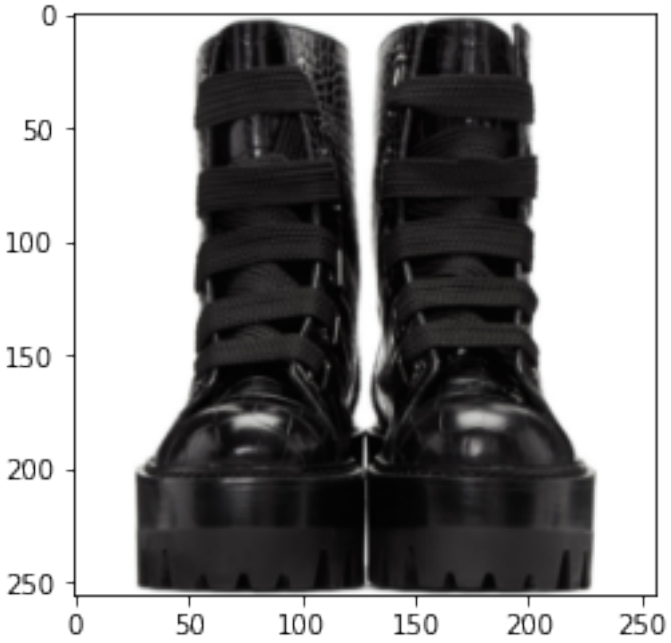
Zippered vent at cuffs. Zip-fly. Gold-tone hardware featuring signature engraved detailing. Tonal stitching. Approx. 4.5" leg opening.

5. **Metriche raggiunte**

				BertScore		
Modello	BLEU	Meteor	RougeL	Precision	Recall	F1
Entropy	66.26	73.76	70.00	92.81	94.48	93.64
Entropy+Triplet	75.15	79.75	74.99	95.95	96.78	96.36

Esempio 3

1. **Immagine di input**



2. **Caption di riferimento**
Ankle-high croc-embossed leather boots in black. Round toe. Tonal lace-up closure. Zip closure at heel. Tonal stacked leather midsole. Treaded foam rubber platform sole. Gunmetal-tone hardware. Tonal stitching. Approx. 2.5" platform.

3. **Caption del modello 1 (Cross-Entropy)**
Ankle-high buffed leather boots in black. Perforated and serrated detailing throughout. Round toe. Tonal lace-up closure. Logo embossed at padded tongue. Padded collar. Zip closure at inner side. Pull-loop at heel collar. Tonal treaded rubber sole. Silver-tone hardware. Tonal stitching.

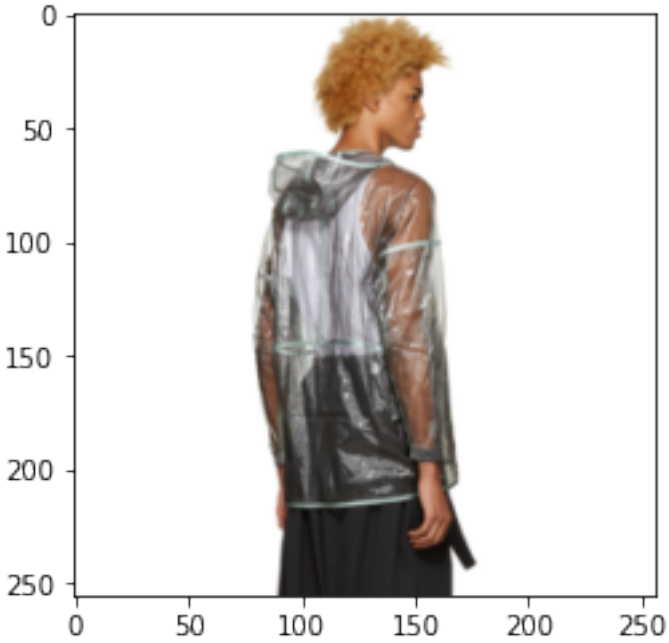
4. **Caption del modello 2 (Cross-Entropy + Triplet)**
   Ankle-high buffed leather boots in black. Topstitched detailing at pointed toe. Adjustable straps with pin-buckle fastening at vamp. Engraved detailing at pin-buckles. Covered elasticized gusset at heel collar. Stacked cuban heel. Tonal leather sole. Silver-tone hardware.

5. **Metriche raggiunte**

				BertScore		
Modello	BLEU	Meteor	RougeL	Precision	Recall	F1
Entropy	50.05	63.87	60.00	91.00	92.30	91.64
Entropy+Triplet	33.97	33.48	35.90	87.19	89.28	88.22

Esempio 4

1. **Immagine di input**



2. **Caption di riferimento**
   Long sleeve semi-sheer organza hooded jacket in 'fume' grey. Satin trim in green throughout. Half-zip closure at front. Kangaroo pocket at waist. Dropped shoulders. Elasticized cuffs. Tonal stitching.

3. **Caption del modello 1 (Cross-Entropy)**
   Long sleeve semi-sheer organza hooded jacket in metallic silver-tone. Embroidered text and graphic appliqués throughout. Drawstring at hood.

Concealed zip closure with press-stud placket at front. Flap pockets at waist. Elasticized cuffs. Bungee-style drawstring at hem. Unlined.

4. **Caption del modello 2 (Cross-Entropy + Triplet)**
   Long sleeve semi-sheer organza jacket in 'fume' grey. Satin trim in green and bouclé wool-blend panelling in beige throughout. Stand collar. Zip closure at front. Zippered pockets at waist. Raglan sleeves. Elasticized cuffs. Bungee-style drawstring at hem. Fully lined.

5. **Metriche raggiunte**

				BertScore		
Modello	BLEU	Meteor	RougeL	Precision	Recall	F1
Entropy	44.35	55.26	49.32	89.46	92.55	90.98
Entropy+Triplet	54.26	68.64	61.11	91.08	94.44	92.73

5.3.4 Metriche sull'intero validation set

Calcolando le metriche BLEU, Meteor, RougeL e BertScore su tutti i 32524 prodotti del set di validazione, otteniamo che anche se il secondo modello raggiunge una cross-entropy loss minore, entrambi i modelli raggiungono pressoché le stesse metriche. Come mostrato in tabella, il primo modello, che utilizza solo cross-entropy loss, raggiunge una precision leggermente maggiore e una recall leggermente minore nel BertScore rispetto al secondo modello, che usa triplet e cross-entropy loss.

				BertScore		
Modello	BLEU	Meteor	RougeL	Precision	Recall	F1
Entropy	38.00	50.10	43.30	88.72	90.89	88.72
Entropy+Triplet	38.90	47.80	43.10	89.19	90.38	89.19

La prima riga, marcata "Entropy", si riferisce al primo modello, addestrato solo con cross-entropy loss. La seconda riga, marcata "Triplet", si riferisce al secondo modello, addestrato con cross-entropy e triplet loss. Vediamo che la qualità delle caption come misurata dalle metriche è pressoché equivalente tra i due modelli. Il BertScore si divide in tre voci: precision, recall, e f1. F1 indica il classico f1-score, ovvero la media armonica tra precision e recall

Capitolo 6

Il codice prodotto

Mostriamo alcune parti rilevanti del codice del progetto, che costituiscono l'implementazione dei passaggi descritti nella fase di sviluppo, inclusi iperparametri del modello, generazione delle caption, e calcolo delle metriche.

6.1 Componenti del modello, batch size (4.1.2)

```
class modelComponents():
  def __init__(self, encoder, encoder_checkpoint, decoder,
      decoder_checkpoint, img_processor, tokenizer):
    self.encoder_checkpoint = encoder_checkpoint
    self.decoder_checkpoint = decoder_checkpoint
    self.img_processor = img_processor
    self.tokenizer = tokenizer
    self.encoder = encoder
    self.decoder = decoder

# component configurations
vit_gpt2 = modelComponents(encoder=ViTModel,
    encoder_checkpoint='google/vit-base-patch16-224-in21k',
                          decoder=GPT2Model, decoder_checkpoint='gpt2',
                          img_processor=ViTFeatureExtractor,
                             tokenizer=GPT2TokenizerFast)
# ## Device & Batch size
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 4

# ## Dataset class
```

6.2 Dataset class (4.1.3)

```
class FashionGenTorchDataset(torch.utils.data.Dataset):
    def __init__(self, file_name, caption_encodings, img_processor,
        n_samples:int, subcategory:bool=False):
        self.n_samples = n_samples
        self.file_path = file_name
        self.dataset = h5py.File(file_name, mode='r')["input_image"]
        self.caption_encodings = caption_encodings
        self.img_processor = img_processor
        self.subcategory = subcategory
        if(self.n_samples == -1): self.n_samples = len(self.dataset)
        else: assert n_samples <= len(self.dataset), 'n_samples must
            be <=' + str(len(self.dataset))

    def __getitem__(self, idx):
        return
            {"pixel_values":self.preprocess_image(self.dataset[idx]),
                "labels":self.caption_encodings[idx],
                "negative":self.get_product_same_category(idx)}

    def set_subcategory(self, subcategory:bool):
        self.subcategory = subcategory

    def __len__(self):
        return self.n_samples

    def preprocess_image(self, image):
        return self.img_processor(image,
            return_tensors="pt")['pixel_values'][0]

    def get_product_same_category(self, index:int):
        product = fashiongen_train.get_product(index)
        if self.subcategory:
          product.subcategory =
              product.subcategory.encode("ISO-8859-9")
          similiar =
              fashiongen_train.get_same_subcategory_of(product)[0].image
        else :
          product.category = product.category.encode("ISO-8859-9")
          similiar =
              fashiongen_train.get_same_category_of(product)[0].image
```

```
        return self.preprocess_image(similiar).to(device)

# ## Model and Dataset init
```

6.3 Caricamento dati e pre-processamento delle caption (4.2.2)

```
def load_data(tokenizer, img_processor, n_train, n_val,
    subcategory:bool=False):
  cap_train = list()
  for p in tqdm(FashionGenDataset(data_path +
      "fashiongen_train.h5").raw_h5()["input_description"],
      position=0, leave=True):
      cap_train.append(p[0].decode("ISO-8859-9"))
          #DEFUALT_STRINGS_ENCODING = "ISO-8859-9")

  cap_val = list()
  for p in tqdm(FashionGenDataset(data_path +
      "fashiongen_validation.h5").raw_h5()["input_description"],
      position=0, leave=True):
      cap_val.append(p[0].decode("ISO-8859-9"))
          #DEFUALT_STRINGS_ENCODING = "ISO-8859-9")

  # append all captions in a single shallow list, tokenize everything
  tokenizer.padding_side = "left"
  tokenizer.pad_token = tokenizer.eos_token
  cap = list(map(lambda caption:
      tokenizer.encode(caption.replace('<br>', ' '),
      return_tensors="pt", max_length=64, pad_to_max_length=True,
      truncation=True)[0], cap_train + cap_val))

  # split into train and validation again
  cap_train = cap[0:len(cap_train)]
  cap_val = cap[len(cap_train):]

  # create datasets
  data_train = FashionGenTorchDataset(data_path +
      "fashiongen_train.h5", cap_train, img_processor,
      n_samples=n_train, subcategory=subcategory)
  data_val = FashionGenTorchDataset(data_path +
      "fashiongen_validation.h5", cap_val, img_processor,
```

```
    n_samples=n_val, subcategory=subcategory)

  return data_train, data_val
```

6.4 Configurazione del modello (5.1)

```
def init_model_and_data(component_config:modelComponents,
    n_train:int=-1, n_val:int=-1, checkpoint:str=None,
    init_data:bool=True, subcategory:bool=False):
  # load models and their configs from pretrained checkpoints
  if(checkpoint is None):
    model = VisionEncoderDecoderModel.from_encoder_decoder_pretrained(
      component_config.encoder_checkpoint,
      component_config.decoder_checkpoint
      )
  else:
    model = VisionEncoderDecoderModel.from_pretrained(checkpoint)

  # set decoder config to causal lm
  model.config.decoder_is_decoder = True
  model.config.decoder_add_cross_attention = True

  # set img_processor & tokenizer
  img_processor = component_config.img_processor.from_pretrained(
    component_config.encoder_checkpoint
    )
  tokenizer = component_config.tokenizer.from_pretrained(
    component_config.decoder_checkpoint
    )

  # decoder-specific config
  if(component_config.decoder == BertModel):
    model.config.decoder_start_token_id = tokenizer.cls_token_id
    model.config.decoder_bos_token_id = tokenizer.cls_token_id
    model.config.decoder_eos_token_id = tokenizer.sep_token_id
  else:
    model.config.decoder_start_token_id = tokenizer.bos_token_id
    model.config.decoder_bos_token_id = tokenizer.bos_token_id
    model.config.decoder_eos_token_id = tokenizer.eos_token_id
    tokenizer.pad_token = tokenizer.eos_token
```

```
  model.config.pad_token_id = tokenizer.pad_token_id
  model.decoder.config.pad_token_id = tokenizer.pad_token_id
  model.config.encoder.pad_token_id = tokenizer.pad_token_id

  # generation arguments
  model.config.decoder.repetition_penalty = 10.0
  model.config.decoder.no_repeat_ngram_size = None
  model.config.decoder_start_token_id = tokenizer.bos_token_id
  model.config.decoder.eos_token_id = tokenizer.eos_token_id
  model.config.decoder.do_sample = False
  model.config.decoder.max_length = 64

  # load and prepare data
  if(init_data):
    data_train, data_val = load_data(tokenizer, img_processor,
        n_train, n_val, subcategory)
    return model, tokenizer, data_train, data_val
  else:
    return model, tokenizer
```

6.5 Inizializzazione del modello e di Tensorboard (5.1)

```
# ### Model and Data setup
loss_type = 'entropy'
step = 48000
checkpoint = checkpoints_path + loss_type + '-checkpoint-' + str(step)
model, tokenizer, data_train, data_val =
    init_model_and_data(vit_gpt2, checkpoint=checkpoint, n_train=-1,
    n_val=-1, subcategory=True)
bert = BertModel.from_pretrained('bert-base-uncased')
bert_tokenizer =
    BertTokenizerFast.from_pretrained('bert-base-uncased')
model = model.to(device)
bert = bert.to(device)

# ### Tensorboard Monitoring
tensorboard_path = drive_path+'tensorboard/'
#log_path =
    tensorboard_path+"swap_from_scratch_subcat_altnorm_lowmargin"
log_path = tensorboard_path+"entropy_subcat"
```

```
writer = SummaryWriter(log_dir=log_path)
```

6.6 Custom Trainer (5.1)

```
# ### CustomLossTrainer
class CustomLossTrainer(CustomTrainer):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.step = step

    def compute_loss(self, model, inputs, return_outputs=False):
        """
        How the loss is computed by Trainer. By default, all models
            return the loss in the first element.
        Subclass and override for custom behavior.
        """
        negative = inputs.pop("negative")
        if self.label_smoother is not None and "labels" in inputs:
            labels = inputs.pop("labels")
        else:
            labels = None
        outputs = model(**inputs)
        # Save past state if it exists
        if self.args.past_index >= 0:
            self._past = outputs[self.args.past_index]

        if labels is not None:
            entropy_loss = self.label_smoother(outputs, labels)
        else:
            # We don't use .loss here since the model may return
                tuples instead of ModelOutput.
            entropy_loss = outputs["loss"] if isinstance(outputs,
                dict) else outputs[0]

        triplet_loss = triplet_margin_loss(inputs["pixel_values"],
            negative)
        final_loss = entropy_loss + triplet_loss

        writer.add_scalar("Loss/train/triplet",
            torch.mean(triplet_loss), self.step)
```

```
        writer.add_scalar("Loss/train/entropy",
            torch.mean(entropy_loss), self.step)
        writer.add_scalar("Loss/train/final", torch.mean(final_loss),
            self.step)
        self.step += 1

        return (final_loss, outputs) if return_outputs else final_loss
```

6.7 Iper-parametri per il training e lancio dell'addestramento (5.1)

```
training_args = Seq2SeqTrainingArguments(
    dataloader_pin_memory = not device.type == 'cuda',
    per_device_train_batch_size = batch_size, # batch size per device
        during training
    per_device_eval_batch_size = batch_size, # batch size for
        evaluation
    output_dir = checkpoints_path, # output directory
    overwrite_output_dir = False,
    load_best_model_at_end = False,
    predict_with_generate = True,
    generation_num_beams = 3,
    eval_accumulation_steps = 4000, # send logits and labels to cpu
        for evaluation step by step, rather than all together
    evaluation_strategy = 'steps',
    save_strategy = 'epoch',
    save_total_limit = 3, # Only last [save_total_limit] models are
        saved. Older ones are deleted.
    # save_steps = 1000,
    eval_steps = 16281,  # Evaluation and Save happens every
        [eval_steps] steps
    learning_rate = 1e-5,
    num_train_epochs = 1,  # total number of training epochs
    warmup_steps = 500, # number of warmup steps for learning rate
        scheduler
    weight_decay = 0.01  # strength of weight decay
)

trainer = CustomLossTrainer(
    compute_metrics = compute_metrics,
    generation_function = generate_caption,
```

```
    tokenizer = None,
    data_collator = None,
    model = model, # the instantiated  Transformers model to be
        trained
    args = training_args, # training arguments, defined above
    train_dataset = data_train, # training dataset
    eval_dataset = data_val # evaluation dataset
)

#trainer.train(checkpoint)
trainer.train()
writer.flush()
```

6.8 Calcolo della Triplet Loss (5.1.2)

```
# ### Loss and Trainer config
def normalize_0_to_1(x:torch.Tensor):
  # x -= x.min(1, keepdim=True)[0]
  # x /= x.max(1, keepdim=True)[0]
  # writer.add_text('tensor_shape', str(list(x.shape)))
  return torch.nn.functional.normalize(x, dim=1)

def get_bert_embedding(text, normalize:bool=True, decode:bool=True):
  if(decode):
    text = tokenizer.batch_decode(text, skip_special_tokens=True)
  input_ids = bert_tokenizer(text, truncation=True, max_length=64,
      padding='max_length',
      return_tensors='pt')['input_ids'].to(device)
  with torch.no_grad():
    embedding = bert(input_ids)['pooler_output']
  embedding.requires_grad
  return normalize_0_to_1(embedding) if normalize else embedding

def get_encoder_embedding(pixel_values, normalize:bool=True):
  embedding = model.encoder(pixel_values.to(device))['pooler_output']
  return normalize_0_to_1(embedding) if normalize else embedding

def triplet_margin_loss(pixel_values, negatives, swap:bool=True):
    negative_embeddings = get_encoder_embedding(negatives)
    positive_embeddings = get_encoder_embedding(pixel_values)
    captions = generate_caption(model, pixel_values)
```

```
    caption_embeddings = get_bert_embedding(captions)
    return
        torch.nn.functional.triplet_margin_loss(anchor=caption_embeddings,
        positive=positive_embeddings, negative=negative_embeddings,
        margin=0.1, swap=swap)
```

6.9 Generazione caption (5.2)

```
# ## Generate captions
def generate_caption(model, pixel_values, num_beams:int=3,
    do_sample:bool=False, top_p:float=1.0, top_k:int=50,
    repetition_penalty:float=10.0, max_length:int=64,
    temperature:int=1.0):
    return model.generate(pixel_values,
                          num_beams=num_beams,
                          repetition_penalty=repetition_penalty,
                          #no_repeat_ngram_size=3,
                          decoder_start_token_id=tokenizer.bos_token_id,
                          pad_token_id=tokenizer.pad_token_id,
                          eos_token_id=tokenizer.eos_token_id,
                          do_sample=do_sample,
                          temperature=temperature,
                          top_k=top_k,
                          top_p=top_p,
                          max_length=max_length,
                          #bad_words_ids=[tokenizer.eos_token_id]
                          )
```

6.10 Calcolo delle metriche (5.3.4)

```
# ## Evaluation metrics
bleu_metric = load_metric('sacrebleu')
meteor_metric = load_metric('meteor')
rouge_metric = load_metric('rouge')

def compute_metrics(eval_preds, decode:bool=True):
    preds, labels = eval_preds
    if isinstance(preds, tuple):
        preds = preds[0]
```

```
    if(decode):
      preds = tokenizer.batch_decode(preds, skip_special_tokens=True)
      # Replace -100 in the labels as we can't decode them.
      labels = np.where(labels != -100, labels,
          tokenizer.pad_token_id)
      labels = tokenizer.batch_decode(labels,
          skip_special_tokens=True)

    # meteor
    meteor = meteor_metric.compute(predictions=preds,
        references=labels)

    # rougeL
    rouge = rouge_metric.compute(predictions=preds, references=labels)

    #split into list of tokens and remove spaces
    preds = [pred.split(' ') for pred in preds]
    labels = [[label.split(' ')] for label in labels]

    # bleu
    bleu = bleu_metric.compute(predictions=preds, references=labels)

    result = {"bleu": bleu["score"], "meteor": meteor['meteor']*100,
        "rougeL": rouge['rougeL'][1][2]*100}

    # prediction_lens = [np.count_nonzero(pred !=
        tokenizer.pad_token_id) for pred in preds]
    # result["gen_len"] = np.mean(prediction_lens)
    result = {k: round(v, 4) for k, v in result.items()}
    return result
```

Conclusioni

All'inizio di questa tesi, ci siamo posti l'obiettivo di sviluppare un sistema in grado di generare descrizioni dettagliate per immagini di articoli di moda. Abbiamo quindi deciso di affrontare il problema applicando tecniche di deep learning, utilizzando modelli transformer, anzichè reti convoluzionali o LSTM.

Durante lo sviluppo del progetto, abbiamo scelto un'architettura encoder-decoder, basata sui modelli Vision Transformer e GPT-2. Abbiamo poi addestrato due modelli basati su questa architettura, il primo dei quali utilizza la cross-entropy loss, e il secondo la somma di cross-entropy e triplet loss.

Terminato l'addestramento, durato 12 epoche, abbiamo potuto notare che mentre nel primo modello la curva della loss raggiunge un plateau dopo alcune epoche, smettendo di decrescere, la loss del secondo modello, che sfrutta il metric learning grazie all'aggiunta della triplet loss, continua a decrescere senza subire forti rallentamenti.

Una volta valutati i due modelli con le metriche BLEU, Meteor, RougeL e BertScore, ci accorgiamo che i due modelli ottengono punteggi molto simili. Anche effettuando una valutazione a campione delle caption generate, in modo da poter dare un giudizio umano, otteniamo che i due modelli raggiungono performance pressoché equivalenti.

Osserviamo però che il secondo modello, che utilizza la triplet loss, mostra ancora un significativo potenziale di miglioramento, in quanto al termine dell'addestramento la loss del modello sta ancora decrescendo rapidamente. E' quindi lecito pensare che continuando l'addestramento per un numero di epoche, il gap tra le loss dei due modelli continuerebbe ad aumentare, ed eventualmente il secondo modello, che sfrutta il metric learning tramite l'aggiunta della triplet loss, supererebbe il primo per qualità delle caption generate.

In ogni caso, gli esperimenti condotti in questa tesi confermano che è possibile ottenere buoni risultati nei task di image captioning utilizzando modelli Transformer, grazie soprattutto all'introduzione di Vision Transformer, studiato appositamente per l'applicazione nella computer vision. A seguito dell'introduzione di ViT, altri ricercatori hanno proposto varianti più accurate ed efficienti del modello, come BeiT [30] e DeiT [31]. Resta da vedere se nel tempo, l'evoluzione di questi modelli porterà l'architettura Transformer

a diventare il nuovo standard de-facto per la visione artificiale, così come è successo per il natural language processing grazie a BERT.

Ringraziamenti

Ringrazio il Prof. Gianluca Moro per avermi proposto questa interessante opportunità di tesi, e in particolare il Dott. Stefano Salvatori per avermi seguito passo a passo nella sua realizzazione.

Ringrazio di cuore la mia famiglia per avermi dato il supporto di cui avevo bisogno per arrivare fino a qui. Non ce l'avrei fatta da solo.

Bibliografia

[1] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. Image captioning with semantic attention. *CoRR*, abs/1603.03925, 2016.

[2] Jyoti Aneja, Aditya Deshpande, and Alexander G. Schwing. Convolutional image captioning. *CoRR*, abs/1711.09151, 2017.

[3] Xuewen Yang, Heming Zhang, Di Jin, Yingru Liu, Chi-Hao Wu, Jianchao Tan, Dongliang Xie, Jue Wang, and Xin Wang. Fashion captioning: Towards generating accurate descriptions with semantic rewards. *CoRR*, abs/2008.02693, 2020.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020.

[5] Negar Rostamzadeh, Seyedarian Hosseini, Thomas Boquet, Wojciech Stokowiec, Ying Zhang, Christian Jauvin, and Chris Pal. Fashion-gen: The generative fashion dataset and challenge. *CoRR*, abs/1806.08317, 2018.

[6] Tom Michael Mitchell. *The discipline of machine learning*, volume 9. Carnegie Mellon University, School of Computer Science, Machine Learning . . . , 2006.

[7] Diksha Sharma and Neeraj Kumar. A review on machine learning algorithms, tasks and applications. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 6(10):2278–1323, 2017.

[8] Jeff Heaton. Ian goodfellow, yoshua bengio, and aaron courville: Deep learning - the MIT press, 2016, 800 pp, ISBN: 0262035618. *Genet. Program. Evolvable Mach.*, 19(1-2):305–307, 2018.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nat.*, 521(7553):436–444, 2015.

[10] Robert Hecht-Nielsen. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[11] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[12] Jyoti Aneja, Aditya Deshpande, and Alexander G. Schwing. Convolutional image captioning. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 5561–5570. Computer Vision Foundation / IEEE Computer Society, 2018.

[13] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(4):652–663, 2017.

[14] Liwei Wang, Alexander G. Schwing, and Svetlana Lazebnik. Diverse and accurate image description using a variational auto-encoder with an additive gaussian encoding space. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5756–5766, 2017.

[15] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 2048–2057. JMLR.org, 2015.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[19] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[20] Philip Gage. A new algorithm for data compression. *C Users J.*, 12(2):23–38, feb 1994.

[21] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909, 2015.

[22] Claude E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27(4):623–656, 1948.

[23] Matthew Schultz and Thorsten Joachims. Learning a distance metric from relative comparisons. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pages 41–48. MIT Press, 2003.

[24] Vassileios Balntas, Edgar Riba, Daniel Ponsa, and Krystian Mikolajczyk. Learning local feature descriptors with triplets and shallow convolutional neural networks. In Richard C. Wilson, Edwin R. Hancock, and William A. P. Smith, editors, *Proceedings of the British Machine Vision Conference 2016, BMVC 2016, York, UK, September 19-22, 2016*. BMVA Press, 2016.

[25] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[26] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL, 2002.

[27] Satanjeev Banerjee and Alon Lavie. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare R. Voss, editors, *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for*

Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005, pages 65–72. Association for Computational Linguistics, 2005.

[28] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[29] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with BERT. *CoRR*, abs/1904.09675, 2019.

[30] Hangbo Bao, Li Dong, and Furu Wei. Beit: BERT pre-training of image transformers. *CoRR*, abs/2106.08254, 2021.

[31] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. *CoRR*, abs/2012.12877, 2020.