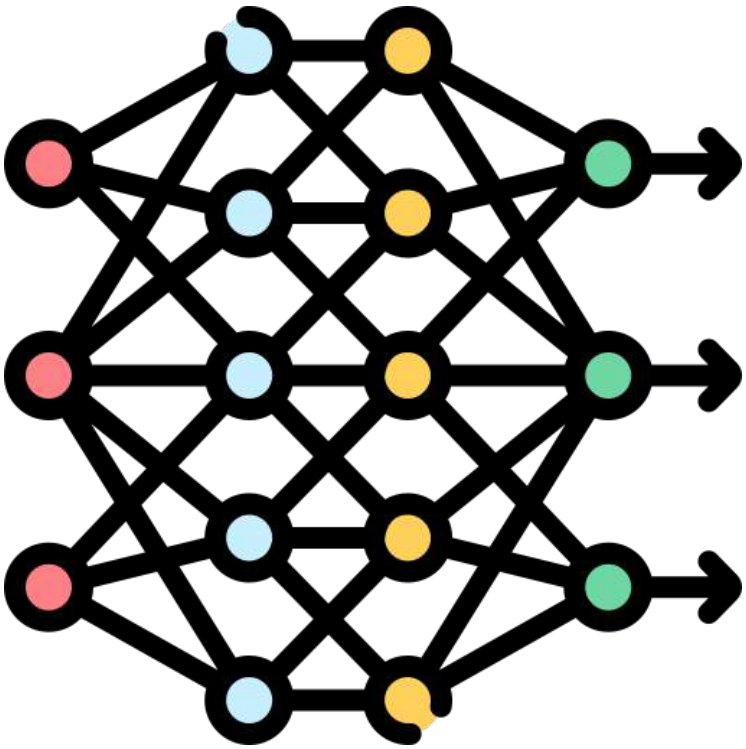
Appunti di Deep Learning

Appunti del corso tenuto dal Professor
Vito Walter Anelli



Domenico Calò

A.A. 2024/2025

Domenico Calò
Appunti di Deep Learning

Colophon
Questo lavoro è stato realizzato con LaTeX su Mac utilizzando il pacchetto ArsClassica, uno stile ispirato a *Gli elementi dello stile tipografico* di Robert Bringhurst.

In caso di necessità dell'effetuare segnalazioni per eventuali, refusi aggiunte o correzioni, non esitate a contattarmi.

Contatti
✉ domenico.calo44@gmail.com · Contatta Domenico Calò

To ask the right question is harder than to answer it.
— Georg Cantor

INTRODUZIONE

Questa dispensa nasce come supporto organico e approfondito al corso di **Deep Learning** tenuto dal **Prof. Vito Walter Anelli** nel secondo semestre del primo anno della **Laurea Magistrale** in **Ingegneria Informatica** presso il **Politecnico di Bari**. Il materiale raccolto e rielaborato in queste pagine ha l'obiettivo di fornire una guida completa e strutturata a tutti gli argomenti trattati durante il corso, seguendo passo dopo passo l'evoluzione delle tematiche affrontate a lezione, e integrando dove necessario approfondimenti teorici, esempi pratici e riferimenti alla letteratura scientifica, non cercando di elevare in maniera eccessiva il linguaggio di trattazione, in modo tale da risultare quanto più accessibile a tutti. Il Deep Learning è ad oggi una delle branche più attive e rivoluzionarie dell'Intelligenza Artificiale, con applicazioni che spaziano dalla visione artificiale alla comprensione del linguaggio naturale, dalla generazione di contenuti alla modellazione di fenomeni complessi. Per questo motivo, la presente dispensa non si limita a una trattazione superficiale, ma propone un percorso progressivo e dettagliato che accompagna il lettore dalla comprensione delle fondamenta teoriche fino alle architetture più avanzate e recenti della ricerca contemporanea.

Nel dettaglio, i capitoli che seguono coprono i seguenti macro-temi:

- **Introduzione al Deep Learning**: origini, motivazioni, contesto storico e differenze rispetto al Machine Learning tradizionale;
- **Architetture di base**: reti neurali feedforward, backpropagation e capacità di rappresentazione;
- **Learning Representation**: apprendimento di rappresentazioni distribuite e significative;
- **Funzioni di attivazione e di costo**: ruolo, forme comuni e impatto sull'ottimizzazione;
- **Ottimizzatori**: algoritmi per la discesa del gradiente, tecniche avanzate e analisi della convergenza;
- **Normalization Layers**: normalizzazione dei dati all'interno delle reti per migliorare la stabilità dell'apprendimento;

- **Convolutional Neural Networks (CNN)**: architetture per il trattamento dei dati strutturati spazialmente (immagini);
- **Recurrent Neural Networks (RNN)**: modelli sequenziali e gestione della memoria nel tempo;
- **Generalizzazione e Overfitting**: tecniche e strategie per migliorare la capacità predittiva del modello su dati non visti;
- **Autoencoder**: modelli non supervisionati per la compressione e la generazione dei dati;
- **Generative Adversarial Networks (GAN)**: modelli generativi basati su apprendimento competitivo;
- **Transformer**: architettura basata sull'attenzione che ha rivoluzionato l'elaborazione sequenziale;
- **Modelli di Diffusione**: tecniche generative recenti, basate su processi stocastici inversi;
- **Graph Neural Networks (GNN) e Graph Convolutional Networks (GCN)**: reti neurali estese a dati strutturati come grafi;
- **Energy-Based Models (EBM)**: architetture basate sull'energia con il loro training e l'inferenza;
- **Flow Matching & Stable Diffusion 3**: tecniche di flow matching e architetture generative avanzate per la sintesi di contenuti multimodali;;
- **Advanced Multi Head Attention**: tecniche avanzate di ottimizzazione dell'attenzione, inclusi KV Cache, Multi Query Attention, Grouped Query Attention, Rotary Positional Embeddings e Multi Head Latent Attention;

Ogni capitolo è pensato per essere il più auto-contenuto possibile, pur mantenendo connessioni logiche con gli altri argomenti, in modo da offrire al lettore sia una visione modulare che sistemica della disciplina. Gli argomenti più complessi sono accompagnati da esempi, schemi esplicativi e, ove necessario, approfondimenti matematici volti a chiarire i meccanismi interni dei modelli. Questa dispensa vuole dunque porsi come un ponte tra teoria e pratica, utile non solo per la preparazione all'esame ma anche utile per stimolare la curiosità altrui, ad approfondire alcune tematiche trattate, come accade nel Capitolo 14 e nel Capitolo 15.

RINGRAZIAMENTI

Ringrazio in primis Lorenzo Pantieri, il quale inconsapevolmente, ha arricchito le mie personali conoscenze di LaTeX, facendomi innamorare di questo mondo, semplicemente grazie ad alcuni suoi commenti presenti all'interno del forum del Gruppo Utilizzatori Italiani di TeX e LaTeX, ancora più in generale mi permetto di ringraziare inoltre tutti gli utenti e tutti i membri dello staff, che rendono quel forum una risorsa stupenda e meravigliosa di approfondimento, questo è l'internet che ci meritiamo e che personalmente gradisco e spero che non sparisca mai. Mi permetto di ringraziare il Prof. Vito Walter Anelli, il quale è stato in grado di raggiungermi con la sua passione per la materia del Deep Learning portandomi a scrivere questi appunti, senza di lui tutto ciò non sarebbe nemmeno esistito. Un ringraziamento speciale va ai miei colleghi che mi hanno accompagnato nel mio percorso universitario e che mi hanno arricchito personalmente, in particolare: Luca Crispino, Ivan Belvito, Antonio Favuzzi, Vincenzo Gentile, Monica Avella, Marzia Capuano, Elena De Michele, Fabrizio Ficarella e Luigi Racamato. Infine ci tengo a ringraziare tutti i futuri lettori di questi appunti, augurandoli un buono studio.

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

1 DI COSA STIAMO PARLANDO

In questo primo capitolo ci soffermeremo su l'introduzione generale al mondo del Deep Learning. Ci concentreremo sul suo apetto sociale, effettuando anche alcuni riferimenti storici. Tutto ciò che viene espresso in questo capitolo, viene trattato con un livello di dettaglio, notevolmente ridotto. Sorvoleremo su alcuni aspetti, poiché l'intento è semplicemente quello di passare a una trattazione di argomenti più specifici. Tuttavia questa introduzione può risultare un ottimo punto di partenza, per poter approfondire diversi aspetti che si discostano dalla parte più rigorosa e applicativa, permettendo di sviluppare la curiosità del lettore. Attualmente i modelli di Deep Learning sono integrati in numerosi campi: ospedali per la diagnosi automatica, agricoltura per il monitoraggio delle colture, domotica per la gestione intelligente degli ambienti, nei sistemi di rilevamento delle frodi, nella manutenzione predittiva degli impianti industriali, assistenti virtuali e molto altro.

1.1 APPLICAZIONI

Le applicazioni del Deep Learning risultano essere variegate e in continua espansione, sarebbe pressocché impossibile elencarle tutti, proprio per questo ci limitiamo ad elencarne solo alcuni qui di seguito:

- Traduzioni automatiche (es. Google Translate, Google Lens, DeepL, Wordvice AI, ecc. . .);
- Sistemi di guida autonoma (es. Tesla Model S, EQS di Mercedes, BMW iX, e-tron GT di Audi, ecc. . .);
- Sistemi di raccomandazione (es. Netflix, Spotify, E-commerce, Outbrains, ecc. . .);
- Generazione automatica di testi, poesie e immagini (es. DALL-E, Chat-GPT; Gemini, ecc. . .);
- Creazione di NPC intelligenti nei videogiochi;

- Agenti intelligenti in grado di competere o superare gli esseri umani in diversi giochi.

1.2 EVOLUZIONE DEI MODELLI DI AI

Prima di passare all'analisi dei diversi modelli che hanno portato a delle rivoluzioni nel mondo dell'intelligenza artificiale, bisogna effettuare una precisazione: la storia dell'intelligenza artificiale, del Machine Learning e successivamente del Deep Learning, ha radici profonde, fin dagli anni '50 grazie ad Alan Turing, il quale effettuo una teorizzazione dell'argomento. Dopo di lui, tutti questi argomenti hanno vissuto momenti di interessamento notevole da parte della comunità scientifica, e altri momenti di completo disinteresse. Negli ultimi anni, in particolare, a partire dagli anni '90, vi è stato un notevole incremento d'interesse nei confronti di queste tematiche. Essendo il Deep Learning una materia abbastanza giovane, ci saranno degli eventi molto vicini nel tempo i quali hanno portato dei cambiamenti e delle rivoluzioni nel campo della ricerca.

Turing, A. M. (1950). Computing machinery and intelligence. Mind, LIX(236), 433–460. [88]

1997 - Deep Blue

Nel 1997, IBM sviluppa **Deep Blue**, il primo sistema di intelligenza artificiale in grado di sconfiggere il campione mondiale di scacchi in carica, Garry Kasparov. Si trattava di un sistema basato su un'enorme potenza computazionale e un algoritmo di ricerca ottimizzato, allenato su numerose partite di scacchi, facendo sentire per la prima volta un umano il quale aveva investito la sua intera vita negli scacchi, impotente davanti a una macchina.

2011 - Watson

Nel 2011, sempre IBM, presenta **Watson**, un sistema in grado di rispondere a domande in linguaggio naturale. Watson vince il quiz televisivo *Jeopardy!* contro i migliori concorrenti umani, dimostrando una capacità impressionante di comprensione del linguaggio e accesso alla conoscenza.

2016 - AlphaGo

Nel 2016, DeepMind (azienda di Google) sviluppa **AlphaGo**, il primo programma in grado di battere il campione mondiale di Go, Lee Sedol. A differenza di Deep Blue, AlphaGo utilizza delle tecniche più avanzate di Deep Learning e Reinforcement Learning, il gioco del Go infatti aveva bisogno di tecniche più avanzate essendo sempre stato valutato come un gioco più complesso rispetto agli scacchi.

(a) La celebre partita tra Kasparov e Deep Blue.

(b) La storica sfida tra Lee Sedol e AlphaGo.

Figura 1: Sfide emblematiche tra esseri umani e intelligenze artificiali.

2017 - AlphaZero

Nel 2017, DeepMind sviluppa **AlphaZero**, un sistema generalista, capace di imparare a giocare a Go, scacchi e Shogi. Tutto questo, esclusivamente tramite autoapprendimento, senza l'ausilio di partite umane pregresse. Questo rappresenta un punto di svolta nell'addestramento tramite self-play.

2017 - Agenti per StarCraft

Blizzard, in collaborazione con DeepMind, inizia a sviluppare agenti intelligenti capaci di giocare a *StarCraft II*, un gioco particolarmente complicato per via dell'alto numero di azioni, segnando questo agente come uno dei primi in assoluto a essere in grado di interagire con un videogioco.

2016–2019 - OpenAI Five

Tra il 2016 e il 2019, OpenAI sviluppa **OpenAI Five**, un agente capace di giocare a *Dota 2*, vincendo contro team professionisti in una modalità 5 contro 5. Si tratta di una delle dimostrazioni più sofisticate di intelligenza artificiale collaborativa.

2019 - AlphaStar

Nel 2019, DeepMind introduce **AlphaStar**, che raggiunge prestazioni al livello di un campione mondiale di *StarCraft II*, grazie a una combinazione di Reinforcement Learning, imitazione e tecniche avanzate di training multi-agente.

2018–2020 - AlphaFold 2

Tra il 2018 e il 2020, DeepMind sviluppa **AlphaFold 2**, un sistema rivoluzionario in grado di prevedere la struttura tridimensionale delle proteine con un'accuratezza senza precedenti, partendo dalla sola sequenza amminoacidica.

2018–2020 - BERT

Google introduce **BERT** (Bidirectional Encoder Representations from Transformers), un modello per la comprensione contestuale del linguaggio, che consente una rappresentazione semantica più profonda, rispetto ai modelli precedenti.

2021 - DALL-E

Nel 2021, OpenAI presenta **DALL-E**, un modello generativo capace di produrre immagini realistiche a partire da semplici descrizioni testuali.

2022 - ChatGPT

Nel 2022, OpenAI introduce **ChatGPT**, un **LLM** (Large Language Model) capace di generare risposte coerenti e plausibili, con un'interfaccia conversazionale estremamente naturale, pur con limiti di accuratezza.

2023 - Bard

Nel 2023, Google rilascia **Bard**, un LLM concorrente di ChatGPT. Tuttavia, a causa di alcune imprecisioni e problemi di affidabilità nelle risposte, viene rapidamente ritirato.

2023 - Gemini

Sempre nel 2023, Google lancia **Gemini**, un modello multimodale progettato per gestire contemporaneamente testo, immagini e audio, aprendo la strada a interazioni AI più complesse e versatili.

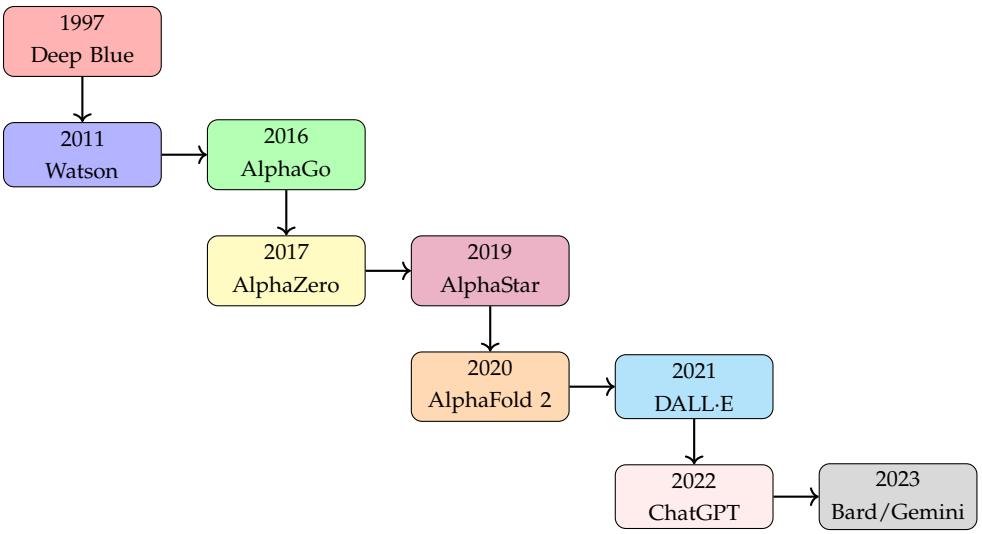


Figura 1: Evoluzione cronologica dei principali modelli di Deep Learning a partire dal 1997 fino al 2023.

Conclusione

Quelli analizzati finora, rappresentano solo una piccolissima parte dell'enorme ecosistema di modelli di Deep Learning sviluppati negli ultimi decenni. Sono stati scelti infatti solo alcuni di essi, legati alla loro rilevanza nel corso del Prof. Anelli. L'esplosione di questo ambito di ricerca, e la diffusione di questi modelli sono dovute a ingenti investimenti da parte di aziende private e da enti governativi, rendendo l'Intelligenza Artificiale un settore strategico globale. Tutto ciò ci permette di comprendere a grandi linee qual è stata la progressione naturale e i maggiori focus su degli specifici task, in cui ci si è soffermati negli ultimi anni da parte della ricerca, in modo da permetterci di capire soprattutto come siamo giunti agli attuali sviluppi.

1.3 ASPETTI ETICI E SICUREZZA

L'espansione dell'intelligenza artificiale ha portato a sollevare degli interrogativi di natura etica, sociale e legale. Tra le principali preoccupazioni troviamo: la sostituzione dell'essere umano in diversi settori lavorativi, la perdita di privacy, i rischi legati alla sicurezza e l'insufficiente trasparenza di molti sistemi. Il ruolo delle istituzioni pertanto, diventa centrale nell'effettuare una giusta regolamentazione. Questo compito tuttavia non è relegato solo ad esse, anche i ricercatori, hanno la responsabilità di considerare le implicazioni etiche venutesi a creare

attraverso le tecnologie che sviluppano. Purtroppo effettuare controlli sulla sicurezza porta un dispendio di costi ulteriore e dei rallentamenti dal punto di vista commerciale, proprio per questo motivo, molte aziende sottovalutano questi aspetti, concentrandosi principalmente su fini prettamente commerciali.

1.3.1 Banche dati esposte

In diversi casi, database contenenti dei dati biometrici, come il riconoscimento facciale o tattile, sono stati archiviati senza aver adottato in precedenza delle adeguate misure di sicurezza, esponendo dati sensibili degli utenti a rischi di uso improprio, questa problematica si è verificata spesso in contesti autoritari o militari, dunque contesti nei quali la perdita di informazioni di una tale portata risulta essere molto pericolosa. Questo potrebbe essere un problema che si correla con l'IA, poiché viene allenata su un vasto numero di dati, i quali il più delle volte non vengono revisionati in maniera accurata, portando a favorire la diffusione di dati in maniera scorretta.

1.3.2 Bias

Una delle più grandi sfide che si affronta nello sviluppo di sistemi di Intelligenza Artificiale, risulta essere, la presenza di **Bias**, il bias si riferisce a un errore sistematico che si presenta nel processo decisionale, portando a degli esiti non corretti o graditi. I Bias possono verificarsi per svariate cause come:

- Collezione dei dati;
- Design dell'algoritmo;
- Interpretazione umana.

Non esiste una vera e propria definizione di bias, di natura fondazionale, ma al più esso si coniuga nei vari contesti applicativi. In questa serie di appunti ho provato a darne una definizione non molto rigorosa, abbastanza intuitiva, ma che potesse comunque centrare l'obbiettivo:

Def: 1.3.1 *Un Bias è un errore sistematico nel processo decisionale, che risulta portare a degli esiti non desiderati, per lo più faziosi mancando di oggettività.*

Essendo il bias un preconcetto, insito nella natura umana, ed essendo che i modelli che conosciamo oggi, allenati su documenti scritti da umani, è naturale che anche i modelli stessi ne siano contagiati. Proprio per questo esiste un campo di ricerca che si occupa di studiare questi comportamenti nei modelli sul mercato, i quali mostrano la presenza di tendenze politiche, religiose, sociali, culturali, ecc. . . Ci sono stati eventi i quali hanno fatto emergere in maniera eclatante come l'intelligenza artificiale sia fortemente condizionata, a causa dei Bias presenti nei dataset sui quali viene allenata, portando a discriminazioni razziali nei modelli di giustizia predittiva o disparità di genere in modelli di selezione automatica del personale.

1.3.3 Filter Bubble

La *Filter Bubble* (in italiano: bolla di filtraggio) è un concetto sviluppato da Eli Pariser nel suo libro *The Filter Bubble: What the Internet Is Hiding from You (2011)*. Indica un ambiente informativo personalizzato, in cui un individuo risulta essere esposto a contenuti, che confermano le sue convinzioni preesistenti, mentre informazioni opposte vengono filtrate o nascoste da algoritmi online. Una filter bubble si crea quando algoritmi (di Google, Facebook, YouTube, TikTok, ecc. . .) selezionano e mostrano contenuti in base ai tuoi interessi, click, like, cronologia di navigazione, e comportamenti passati, limitando così l'esposizione a idee contrastanti.

1.3.4 Polarization Effect

Una conseguenza della filter bubble, è sicuramente la polarizzazione, essa scatena il polarization effect: gruppi omogenei finiscono per rafforzare le proprie opinioni in modo estremo, con conseguente aumento della divisione sociale, come visibile nei dibattiti online su temi sensibili. Questa caratteristica non restituisce un'immagine corretta della realtà, poiché si vede tutto in maniera polarizzata, dando pieno valore ai pareri estremi e non valutando tutte le possibili idee intermedie, presenti all'interno di un discorso.

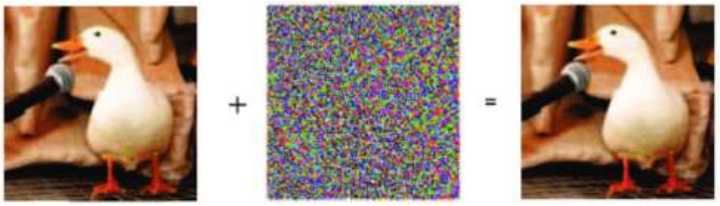


Figura 2: Esempio di Adversarial Attack, in cui una papera viene classificata come cavallo a seguito dell'aggiunta di un rumore esterno.

1.3.5 Adversarial Attacks

Un *Adversarial Attack* è una tecnica utilizzata volta ad ingannare un modello di machine learning, in particolare i modelli di deep learning, introducendo piccole perturbazioni appositamente progettate nei dati di input, tali da causare errori di classificazione o comportamenti indesiderati, senza che l'errore risulti essere evidente a un osservatore umano. Un classico esempio è immaginarsi un modello che classifica correttamente un'immagine di una papera etichettandola come "papera", ma nel momento in cui aggiungiamo una perturbazione invisibile all'occhio umano, il modello classifica questa papera come "cavallo" con altissima confidenza e sicurezza, pur essendo all'occhio umano una papera (Figura 2).

1.3.6 Data Poisoning

Il *Data Poisoning* è una forma di attacco contro i sistemi di machine learning in cui un attaccante manipola i dati d'addestramento allo scopo di compromettere il comportamento del modello durante l'inferenza.

Def: 1.3.2 *L'inferenza è il processo in cui un modello addestrato utilizza le sue conoscenze per fare previsioni, prendere decisioni o generare risultati da dati nuovi e mai visti prima.*

Il data poisoning è come "avvelenare il cibo del cervello dell'IA": se il modello impara da dati falsificati o manipolati, finirà per imparare cose sbagliate. Questa tipologia di attacco risulta essere una vera e propria minaccia per la sicurezza e affidabilità dei modelli, soprattutto in contesti critici (sanità, difesa, veicoli autonomi). Questo è anche un ruolo molto critico, difficile da rilevare, specialmente nei modelli addestrati su dati pubblici o crowdsourced (es. GitHub, Kaggle, internet). Intuitivamente si può pensare come aumentando il numero di

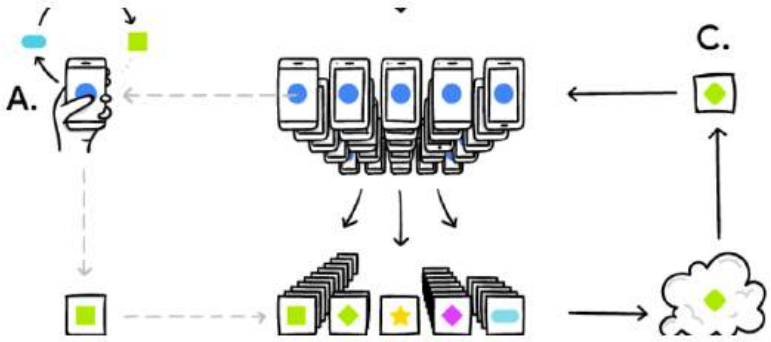


Figura 3: Architettura rappresentativa del Federated Learning.

parametri su cui si allena un modello, la necessità di file "avvelenati" da immettere vada di pari passo. Im realtà grazie a delle ricerche sviluppate dal team di Anthropic [79], si è potuto constatare come basti un numero notevolmente ridotto rispetto ai parametri di allenamento, per causare delle gravi problematiche. Pertanto, concentrarsi su questa problematica risulterà essere una grande sfida futura per la sicurezza di questi modelli.

1.3.7 Federated Learning

Il Federated Learning è un approccio di machine learning decentralizzato che consente a diversi dispositivi o entità di addestrare congiuntamente un modello AI senza condividere i dati grezzi. Invece di centralizzare i dati, l'addestramento avviene localmente sui dispositivi, e solo gli aggiornamenti del modello (come i parametri o i gradienti) vengono inviati a un server centrale per essere aggregati. Questo metodo preserva la privacy e la sicurezza dei dati, poiché le informazioni sensibili rimangono sui dispositivi locali (Figura 3).

1.3.8 Sostenibilità e AI

L'addestramento dei modelli di AI comporta un notevole consumo energetico. Basti pensare che GPT-3, ha prodotto livelli notevoli di CO2 a causa delle risorse richieste per l'allenamento e l'utilizzo, contribuendo significativamente all'impatto ambientale. Risulta dunque fondamentale sviluppare tecnologie più sostenibili, attraverso hardware efficiente e tecniche di training più leggere.

1.3.9 AI Act

L'**AI Act** dell'Unione Europea è una proposta legislativa che mira a classificare i sistemi di intelligenza artificiale secondo il livello di rischio. I sistemi ad alto rischio saranno soggetti a requisiti più stringenti, mentre quelli a basso rischio avranno meno restrizioni, promuovendo al contempo trasparenza e responsabilità.

2 INTRODUZIONE AL DEEP LEARNING

Ciò che bisogna consolidare prima di immergerci nello studio del Deep Learning è la differenza fra Machine Learning e Deep Learning. Per far questo bisogna analizzare quello che si fa con i programmi di Machine Learning, qui ci riferiremo a dei casi d'uso speicfici, partendo dal riconoscimento delle immagini. Nel Machine Learning si parte dal creare un programma, il quale deve essere in grado, data un'immagine, di estrarre alcuni attributi per poi utilizzarli, all'interno di un modello di ML e riconoscere immagini della stessa tipologia. Il passo ulteriore che viene effettuato nel Deep Learning è a monte, ossia durante l'estrazione degli attributi, dove costruiamo una gerarchia degli stessi, avendo la possibilità di stabilire attributi di basso, medio e alto livello. Questa selezione avviene con la combinazione di modelli di Machine Learning stratificati, creando nella sua complessità un meccanismo automatico più raffinato e profondo, da cui nasce il Deep Learning.

2.1 DEFINIZIONE

Il **Deep Learning** dunque è una sotto-disciplina del Machine Learning la quale utilizza *Reti Neurali Profonde*, ossia delle reti neurali con numerosi hidden-layer, per poter estrarre delle rappresentazioni gerarchiche dai dati. A differenza dei metodi tradizionali di Machine Learning, i quali richiedono l'ingegnerizzazione manuale delle caratteristiche, il Deep

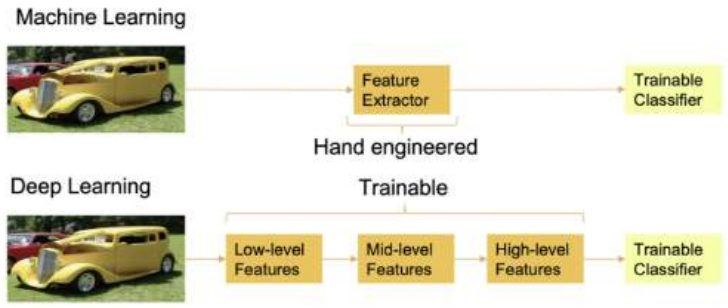


Figura 4: Schema rappresentativo della principale differenza fra un modello di Machine Learning e un modello di Deep Learning

Learning è in grado di apprendere automaticamente rappresentazioni multi-livello, direttamente dai dati grezzi. Questo approccio, si è dimostrato particolarmente efficace in campi come la visione artificiale, il riconoscimento vocale e l'elaborazione del linguaggio naturale.

2.2 ESEMPIO: IL DATASET MNIST

MNIST è un dataset ampiamente utilizzato per il riconoscimento di cifre scritte a mano. Composto da 60.000 immagini di addestramento e 10.000 immagini di test, ognuna delle quali è una griglia 28×28 di pixel in scala di grigi. L'obiettivo di un modello di apprendimento automatico addestrato su MNIST, risulta essere quello di classificare correttamente ogni immagine in una delle 10 categorie (da 0 a 9). Questo dataset viene spesso utilizzato come Benchmark, per poter valutare le prestazioni di nuovi algoritmi di Deep Learning.

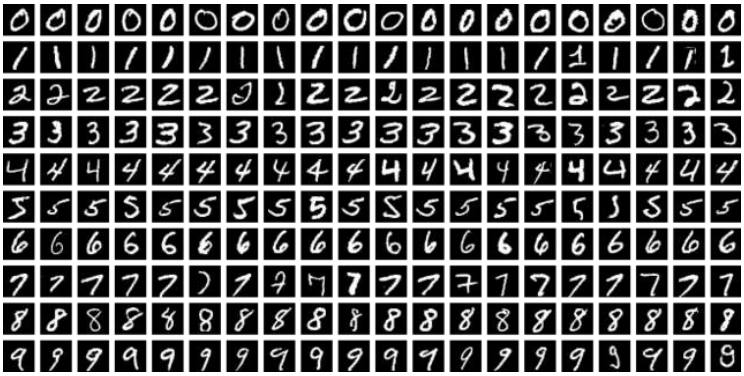


Figura 5: Un estratto delle immagini presenti all'interno del dataset del MNIST.

2.3 GRAFI COMPUTAZIONALI

Un **Grafo computazionale** rappresenta il flusso di operazioni effettuate tra variabili, esso viene utilizzato per modellare i calcoli svolti da una Rete Neurale. Ogni nodo del grafo rappresenta un'operazione matematica, mentre i bordi o archi, rappresentano i flussi di dati. Utilizzando questa struttura, possiamo visualizzare il flusso delle informazioni e il processo di apprendimento della rete, semplificando visivamente i

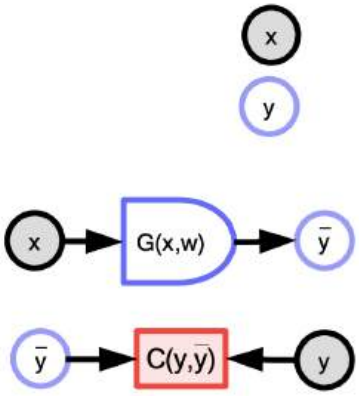


Figura 6: Notazione dei nostri modelli, dall'alto verso il basso ci sono: Variabili, Funzione deterministica e Funzione scalare

nostri modelli. Ad esempio, in una rete neurale profonda, ogni livello applica una trasformazione, tramite l'utilizzo di una funzione di attivazione generando un output, il quale verrà poi propagato fino al livello finale, per ottenerne una previsione. Il modello utilizzato da noi, distingue nettamente i vari elementi tramite l'utilizzo di forme e colori, avendo una differenza fra input, funzioni deterministiche e funzioni scalari (Figura 6).

2.4 FUNZIONE DI COSTO

Una **Funzione di Costo** (Loss Function) è un'operazione riportabile tramite l'utilizzo dei nostri grafi computazionali (Figura 7). Questa funzione misura l'errore pesente tra l'output fornito dal modello (predizione) e l'output desiderato (valore atteso), essa, è fondamentale per il processo di apprendimento. Il modello, cerca di minimizzare questa funzione di costo, aggiustando i pesi della rete di volta in volta attraverso un algoritmo di ottimizzazione. Una delle funzioni di costo più comuni è la Mean Squared Error (MSE), definita come:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Dove y_i rappresenta il valore reale e $\hat{y}_i$ l'output predetto dal modello per il campione i-esimo. Minore è il valore della funzione di costo, migliore sarà la capacità del modello di effettuare previsioni accurate.

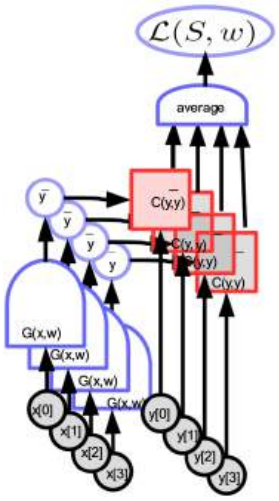


Figura 7: Rappresentazione tramite l'utilizzo di un grafico computazionale della funzione di costo, utilizzando la MSE

2.5 RETI NEURALI

Una Rete Neurale, tramite i nostri grafi computazionali, risulta essere una pila di blocchi funzionali lineari e non lineari. Le loro entità fondamentali prendono il nome di **Neuroni** (Figura 8). Ogni *Neurone* calcola una somma pesata degli input ricevuti e applica una funzione di attivazione per introdurre non linearità all'interno del modello. Questa architettura, permette alla rete di apprendere rappresentazioni complesse dei dati. Un singolo neurone in una rete neurale può essere rappresentato come segue:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

Dove w_i sono i pesi associati agli input x_i, e vengono sommati al bias b (un valore costante). Successivamente, viene applicata una funzione di attivazione $\sigma(z)$ per poter aggiungere non linearità, come detto in precedenza e poter ottenere l'output finale:

$$\hat{y} = \sigma(z)$$

Le reti neurali, sono strutture complesse composte da tanti neuroni, i quali possono essere completamente collegati fra loro, generando le *Fully-Connected Neural Network*, oppure parzialemente, i neuroni vengono concatenati fra loro, formando diversi strati, dando la possibilità di formare delle architettetture profonde.

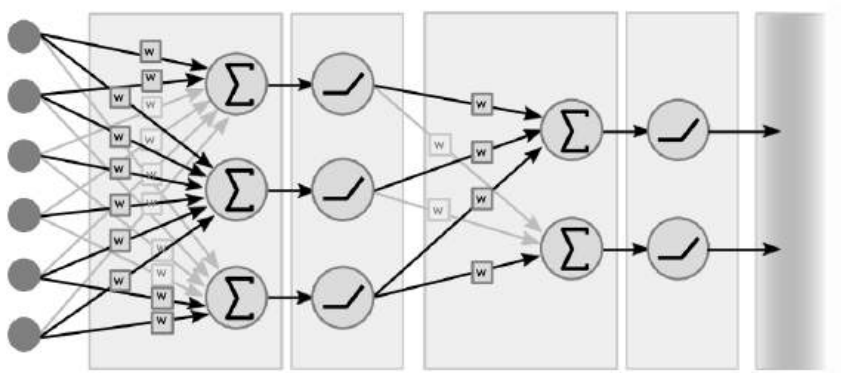


Figura 8: Figura rappresentativa di un esempio di rete neurale con diversi strati nella sua architettura.

2.6 BACKPROPAGATION

La **Backpropagation** (retropropagazione), è uno degli algoritmi principali quando si parla di *Reti Neurali*, poiché permette di allenare la rete, basandosi sulla **Discesa del Gradiente**.

Def: 2.6.1 *La Discesa del Gradiente è un algoritmo di ottimizzazione che trova il valore minimo di una funzione, spostandosi iterativamente nella direzione opposta al suo gradiente.*

L'idea della Backpropagation, è quella di propagare all'indietro l'errore generato dalla predizione e dal valore atteso, per aggiornare i pesi della rete e ridurre l'errore complessivo (la Loss Function), ottenendo dei risultati sempre più accurati di volta in volta fintantoché non si raggiunge una convergenza sui valori dei pesi.

2.6.1 Calcolo dei Gradienti

Il cuore della Backpropagation risiede nel calcolo dei gradienti della loss function rispetto ai pesi, utilizzando la **regola della catena**.

Def: 2.6.2 *La regola della catena afferma che se $f(x)$ e $g(x)$ sono funzioni derivabili, allora la derivata della funzione composta $h(x) = f(g(x))$ è:*

$$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$$

Approfondimento sulla Backpropagation

La formula generale per la derivata della funzione di costo rispetto ai pesi è la seguente:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial W}$$

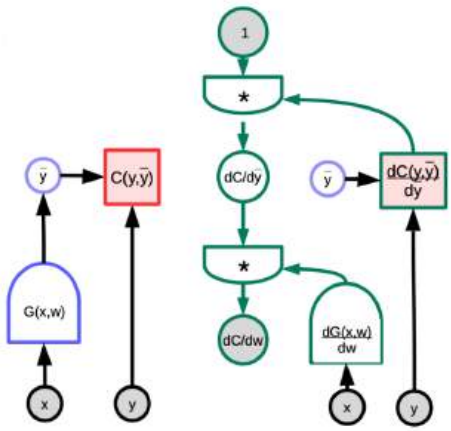


Figura 9: Grafo computazionale della Backpropagation, che si integra con quello già precedentemente visto della funzione di costo in modo tale da raffinare gli esiti finali correggendosi dinamicamente

Dove $\mathcal{L}$ è la funzione di costo, y l'output del modello e W il peso da aggiornare. Questo processo viene ripetuto per ogni livello della rete, fino all'ottimizzazione del valore dei pesi. Questa procedura può essere facilmente sintetizzata in un grafo computazionale (Figura 9). Nella backpropagation è possibile applicare qualunque grafo che sia diretto e aciclico. Nel caso in cui il grafo presentasse dei loop, sarà necessario scioglierli.

2.7 PROBLEMI DELLA BACKPROPAGATION

La Backpropagation soffre di due principali problemi legati al gradiente, ossia il *gradiente che scompare* e la sua *esplosione*. Questi due problemi si verificano quando il gradiente della funzione di costo, rispetto ai pesi, risulta essere molto piccolo o molto grande nei livelli più profondi della rete.

- **Gradiente che scompare:** nelle reti molto profonde, i gradienti calcolati per i livelli iniziali diventano sempre più piccoli man mano che vengono propagati all'indietro. Questo provoca degli aggiornamenti minimi dei pesi nei primi livelli, rallentando drasticamente l'addestramento o impedendo alla rete di apprendere;
- **Gradiente che esplode:** al contrario, se i gradienti aumentano esponenzialmente durante la propagazione all'indietro, i

pesi della rete possono diventare estremamente grandi, portando a instabilità nell'addestramento e rendendone difficile la convergenza.

Con il passare degli anni il Machine Learning evolvendosi e specializzandosi nel Deep Learning, portando allo sviluppo di diverse soluzioni per queste problematiche:

1. **Funzioni di attivazione avanzate:** L'utilizzo di altre funzioni come **ReLU** (Rectified Linear Unit) la quale ha ridotto il problema del gradiente che scompare rispetto alla funzione Sigmoide e tangente iperbolica. Varianti come Leaky ReLU e Parametric ReLU permettono di migliorare l'apprendimento;

2. **Inizializzazione dei pesi:** Metodi di inizializzazione dei pesi come **Xavier** per l'assegnamento dei pesi stabilizzano i gradienti all'inizio dell'addestramento, al posto di un'inizializzazione randomica o nulla;

3. **Batch Normalization:** Tecnica la quale permette di normalizzare le attivazioni intermedie riducendo la varianza dei gradienti durante la propagazione.

L'introduzione infatti di queste e altre tecniche, hanno permesso al Deep Learning di diventare un campo di studio sempre più interessante, divenendo anche più efficace come tecnica rispetto all'utilizzo di metodi tradizionali postulati dal Machine Learning.

2.8 IMPLEMENTAZIONE CON PYTORCH

Per effettuare una parallelizzazione della teoria, con l'aspetto implementativo del Deep Learning, quì mostriamo come implementare una rete neurale utilizzando **PyTorch** (una libreria open-source di Python), per far ciò utilizziamo la classe `nn.Module` così da definire la nostra architettura. Di seguito un esempio di una semplice rete con due livelli nascosti:

```
import torch
from torch import nn

class MyNet(nn.Module):
```

```
        def __init__(self, input_dim, hidden_dim, output_dim):
            super().__init__()
            self.fc1 = nn.Linear(input_dim, hidden_dim)
            self.fc2 = nn.Linear(hidden_dim, output_dim)

        def forward(self, x):
            x = torch.relu(self.fc1(x))
            x = self.fc2(x)
            return x
    model = MyNet(784, 128, 10)
```

La rete in questo codice prende in input delle immagini di 28x28 pixel (convertite in vettori di 784 elementi), le elabora utilizzando un livello nascosto con 128 neuroni e produce infine un output di 10 neuroni corrispondenti alle 10 classi del dataset MNIST.

3 ARCHITETTURE DI BASE

Quando si parla di Deep Learning, si usano spesso delle strutture grafiche, aggiungendo dei livelli di astrazione, per poter semplificare la comprensione delle architetture complesse, queste strutture prendono il nome di **Grafi Computazionali**. La rappresentazione grafica di alcuni dei modelli più complessi, permette di comprendere al meglio come vengano implementate alcune nuove tecniche. Adesso vedremo, semplici implementazioni, per dare un'idea di utilizzo, e lasciare al lettore, la possibilità di adottarle nei propri progetti.

3.1 MODULI MOLTIPLICATIVI

Iniziamo da delle architetture semplici: i **Moduli Moltiplicativi**. Creiamo immediatamente un collegamento con questi moduli, sfruttando un esempio pratico. Nel processo di Backpropagation essi vengono utilizzati, rispetto agli input, o rispetto ai pesi. La differenza fra operare rispetto agli uni o gli altri si trova nell'impossibilità di imparare direttamente i pesi, come avviene per gli input. In molti hanno pensato cosa succederebbe nel caso in cui i pesi diventassero degli output forniti da un'altra Rete Neurale, questa possibilità viene sfruttata tramite i moduli moltiplicativi.

$$s_i = \sum_j w_{ij} x_j \ : \ w_{ij} = \sum_k u_{ijk} z_k \Rightarrow s_i = \sum_{jk} u_{ijk} z_k x_j$$

Come vediamo nella formula di cui sopra, essa è una semplice sostituzione, ottenendo il risultato desiderato, senza modificare il senso dell'operazione. Questa semplice sostituzione, è stata adottata in molti contesti di Deep Learning.

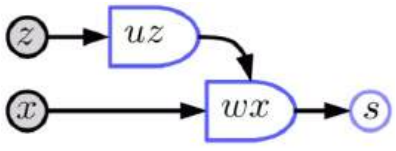


Figura 10: Grafo computazionale corrispondente all'equazione 3.1

Se avessimo una Rete Neurale che si occupa del controllo dei pesi, saremmo in grado di crearne una nuova, con una semplice variazione di essi, potendo persino ampliarne la portata, riuscendo a controllare molte reti con solo una a supervisionare. Questo non risulta essere un lavoro semplice, dovremmo effettuare delle limitazioni per non incorrere in problematiche di esplosione della rete stessa. Dunque la rete neurale al di sopra delle altre, funziona come un "controllore", pesando di più una sottorete sulle altre (Figura 3.1). Questo meccanismo si implementa adoperando la funzione **SoftMax**, che tratteremo più avanti, permettendoci di calcolare nuovi pesi ed effettuare un'attivazione. Questa architettura prende il nome di **Modulo dell'attenzione**, dando più o meno importanza in base all'input ricevuto alle singole reti neurali.

$$s_i = \sum_j w_j x_{ij} \; : \; w_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

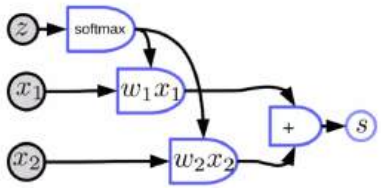


Figura 11: Grafo computazionale rappresentante il Modulo dell'attenzione, in cui ci sono più reti, controllate da una singola, la quale da più o meno importanza alle sottoreti in base all'input ricevuto.

Mixture of Experts

L'idea che stiamo analizzando, inserendo più reti neurali in un'unica architettura, prende il nome di **Mixture of Experts**, quindi partendo da una semplice rete neurale con x_1 e x_2 come ingressi, la estendiamo rendendola un'*esperta*, creando una rete con all'interno milioni di parametri. Il modello Mixture of Experts è una rete neurale, la quale controlla diversi task, spezzati nelle singole sottoreti, selezionando su quale fare affidamento a seconda della necessità (Figura 12). I modelli di linguaggio, utilizzano questa strategia, prendono in considerazione una combinazione di reti specializzate in diverse attività (e.g Traduzione, riassunto, calcoli, ecc. . .).

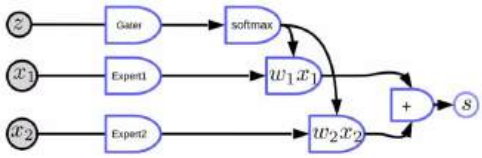


Figura 12: Grafo computazionale, del modello Mixture of Experts, vari modelli specializzati in compiti specifici, attivati o disattivati da un'altra rete neurale tramite l'utilizzo della funzione softmax.

Parameter Transoformation

Un modo generale per concettualizzare architetture come la Mixture of Experts è attraverso l'idea di **Parameter Transformation**. Invece di considerare i parametri di un modello, come i pesi W, dei valori statici appresi una sola volta, li concepiamo come l'output dinamico di un altro modulo. In pratica, la rete impara a generare o selezionare i parametri più adatti in base al contesto fornito dall'input. L'architettura Mixture of Experts (MoE) può essere vista come un caso specifico di questo principio. In una MoE, la rete generale non crea nuovi pesi, ma seleziona quale set di parametri (quello di un "esperto") utilizzare. Differentemente la Parameter Transformation è un'operazione di selezione dinamica. Una delle applicazioni più diffuse di questo concetto è la *Weight Sharing*. Questa tecnica consiste nell'utilizzare lo stesso modulo parametrico, definito da un set di pesi w, applicandolo ripetutamente in diverse parti di un modello, ma anche per elaborare porzioni differenti dell'input (Figura 13).

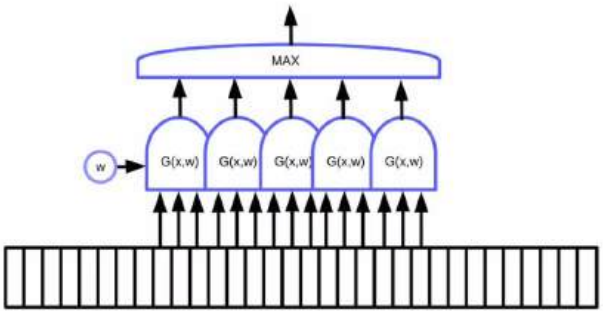


Figura 13: Esempio di grafo computazionale che implementa la Weight Sharing. Un unico set di parametri, il kernel w, riutilizzato ripetutamente dalla funzione G su diverse sezioni dell'input.

Avendo un modulo generatore di pesi, iniettato in diversi punti della rete principale otterremo due vantaggi principali:

- **Efficienza:** Invece di dover apprendere e memorizzare un set di pesi indipendente per ogni operazione, si apprende un unico e più compatto modulo che sa come generarli. Riduce drasticamente il numero totale di parametri del modello, rendendolo più leggero ed efficiente;

- **Generalizzazione:** Durante l'addestramento, il processo di Back-propagation diventa particolarmente efficace. Se lo stesso modulo di pesi condivisi viene utilizzato in N contesti diversi, esso riceverà i gradienti di errore da tutte e N le funzioni di costo parziali. Questo "costringe" il modulo ad apprendere parametri che non siano specifici per un singolo compito, ma che siano abbastanza robusti da funzionare bene in tutte le situazioni. Porta a un risultato con una migliore capacità di generalizzazione.

4 LEARNING RAPPRESENTATION

Uno degli obiettivi del Deep Learning è imparare rappresentazioni significative a partire da dei dati grezzi (logits). Un buon modello di Deep Learning, non si limita a classificare gli input, ma è in grado di estrarre delle caratteristiche gerarchiche le quali semplificano notevolmente i compiti di analisi e previsione.

4.1 CLASSIFICATORI LINEARI E I LORO LIMITI

I Classificatori Lineari, visti durante il corso di Machine Learning, suddividono lo spazio degli input in due regioni separate tramite un iperpiano. Questa soluzione però risulta essere molto limitata per due motivi:

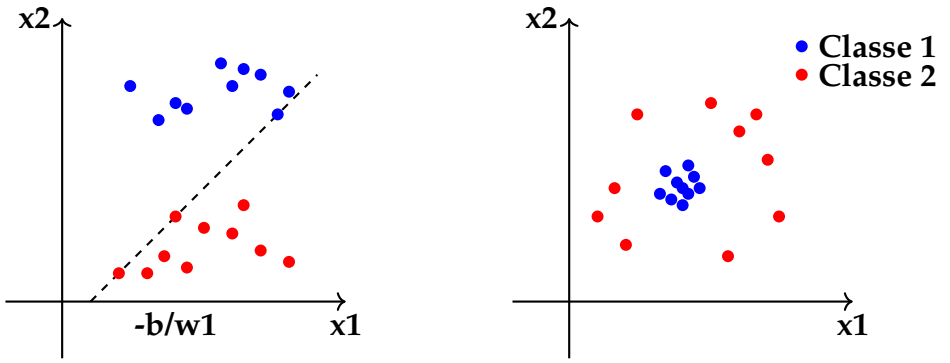
1. Non è in grado di gestire problemi con dati non linearmente separabili;

2. La probabilità che una distribuzione casuale di punti P sia separabile linearmente diminuisce all'aumentare della dimensionalità N nel momento in cui $P \geq N$ (Teorema di Cover, 1955 [25].

Se provassimo ad aumentare la dimensione di N per cercare di arginare il problema, a causa del teorema di Cover non possiamo ignorare la problematica della *Curse of Dimensionality* [11], la quale è anch'essa campo di studio.

4.1.1 La funzione XOR

Per superare i limiti dei classificatori lineari, si adottano diverse metodologie, una di queste è approfondita attraverso la funzione **XOR** [54], vista nel corso di Machine Learning, dove per definizione la funzione XOR genera dei dati che non sono linearmente separabili, ma attraverso la combinazione di modelli più semplici, riusciamo a ottenere la funzione complessiva attraverso una rete neurale con esiti linearmente separabili.



(a) Dati linearmente separabili (b) Dati non linearmente separabili

Figura 14: Confronto tra un dataset linearmente separabile (a sinistra) e uno non linearmente separabile (a destra).

Di seguito alcune metodologie per far fronte ai limiti dei classificatori lineari:

- Estrarre caratteristiche rilevanti dall'input grezzo;
- Trasformare i dati in una rappresentazione in cui la separabilità lineare sia più semplice;
- Usare estrattori di caratteristiche non lineari, come reti neurali profonde.

4.2 METODI DI ESTRAZIONE

Alcuni metodi per estrarre caratteristiche utili includono:

- **Tiling dello spazio:** suddivisione dello spazio in regioni più piccole;
- **Proiezioni casuali:** trasformazioni casuali per aumentare la dimensione dello spazio;
- **Classificatori polinomiali:** utilizzo di combinazioni di variabili di input per migliorare la separabilità;
- **Macchine a kernel**: mappatura non lineare dei dati in uno spazio a più alta dimensionalità.

L'idea alla base di tutte queste possibilità è quella di poter espandere le dimensioni della nostra rappresentazione, portando i nostri dati ad essere lineramente separabili con facilità.

Oss: 1 *Utilizzare le features in maniera lineare, risulta essere più semplice a livello di costo computazionale.*

4.3 RETI NEURALI POCO PROFONDE

Per far fronte a questa problematica, si possono utilizzare le strategie quì di seguito elencate:

- Le macchine a vettori di supporto (SVM) e i metodi a kernel utilizzano uno strato con funzioni di base non lineari, seguito da uno strato lineare;
- Utilizzare delle reti neurali con due strati diventando degli approssimatori universali 4.3.1, tuttavia richiedono un numero elevato di neuroni per rappresentare funzioni complesse.

Teorema 4.3.1 *Una rete con 2 strati (con 1 strato nascosto), è un approssimatore universale, può approssimare in modo arbitrariamente preciso ogni funzione continua* $g(u_1, u_2, \ldots, u_d)$.

A fronte di questo teorema potremmo chiederci del perché con il Deep Learning, utilizziamo delle reti neurali profonde, visto che ogni funzione continua è rappresentabile con un singolo layer nascosto, non sembra avere molto senso, anzi sembra quasi uno spreco.

Perché le Architetture Profonde?

Una rete neurale profonda è in grado di rappresentare funzioni complesse in modo più efficiente rispetto a una rete non profonda. Nel momento in cui passiamo a una rete neurale profonda otterremo un risultato in meno tempo, dunque si rinuncia a dello spazio per risparmiare del tempo. Inoltre esse hanno:

- Maggiore capacità di modellare strutture gerarchiche nei dati (e.g riconoscimento visivo, riconosco una figura a seconda di sue diverse caratteristiche, ognuna analizzata in un layer);
- Possibilità di apprendere trasformazioni sempre più astratte dei nostri dati.

4.4 IPOTESI DEL MANIFOLD

L'argomento della Data Manifold, viene trattato meglio in: Hadsell et al. CVPR 2006 [36]

Un concetto molto importante del Deep Learning è l'ipotesi del **Manifold** [84, 9]. Il nostro obbiettivo vuole essere quello di riconoscere la faccia della persona presente nella Figura 15, notiamo come in ognuna delle sottofigure vi è la stessa persona, noi riusciamo a caprilo facilmente, ma come questo accade ci permette di comprendere come potrebbero farlo le macchine. Ci sono delle caratteristiche che siamo in grado di riconoscere le quali appartengono alla stessa persona. Il nostro cervello, e più in generale, la nostra realtà si trova in uno spazio dimensionale che non è della stessa grandezza di una figura che possiamo creare.

Figura 15: Esempio dell'ipotesi del manifold.

È stato dimostrato come noi umani, siamo in grado di riconoscere una faccia di una persona rappresentando meno di 56 variabili, mentre un calcolatore usando un'immagine 1000x1000 pixel avrò bisogno di 1 000 000 di variabili. Questa è la così detta ipotesi del Manifold: la realtà vive in una dimensione notevolmente minore rispetto alla dimensione che utiliziamo per rappresentarla, differentemente da ciò che avviene per le macchine.

Correlazione con cio che stiamo facendo

Se avessimo un estrattore ideale, potremmo essere in grado di prendere questa immagine da un milione di dimensioni, ed estrarre le 56 features, in modo tale da essere in grado di riconoscere l'immagine. Rappresentando i cambiamenti su un piano cartesiano, e spostandoci lungo una singola dimensione, vedremo modificarsi esclusivamente una sola features (Figura 16).

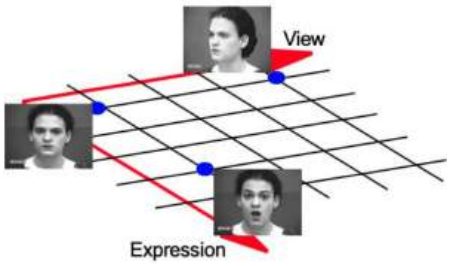


Figura 16: Rappresentazione di un estrattore ideale di features su un piano cartesiano.

4.5 STRUTTURA GERARCHICA DEI DATI

Le architetture multilivello riflettono la natura composizionale dei dati, analizzando parti dei dati ogni volta, queste possono diventare delle ottime possibilità per spostarci verso un estrattore ideale, permettendo un'efficiente rappresentazione delle informazioni:

- **Visione:** pixel ⇒ bordi ⇒ texture ⇒ oggetti;
- **Testo:** caratteri ⇒ parole ⇒ frasi ⇒ discorso;
- **Audio:** campioni ⇒ bande spettrali ⇒ fonemi ⇒ parole.

4.6 CONCLUSIONE

Le reti neurali profonde permettono di bilanciare tempo e spazio, garantendo che i più livelli implichino più operazioni sequenziali, riducono la necessità di risorse computazionali parallele e consentono una rappresentazione più compatta delle funzioni complesse, ma soprattuto una visione gerarchica delle nostre features.

5 FUNZIONI DI ATTIVAZIONE

Le funzioni di attivazione sono molto importanti nelle reti neurali, in quanto introducono la non-linearità nel modello, permettendo di apprendere relazioni complesse nei dati. Senza funzioni di attivazione, una rete neurale profonda risulterebbe equivalente a una semplice trasformazione lineare.

5.1 VANISHING GRADIENT PROBLEM

Uno dei problemi nell'addestramento delle reti neurali profonde è il **Vanishing Gradient Problem** [40, 12], una problematica che viene a verificarsi, quando il gradiente venendo propagato all'indietro diventa molto piccolo, a causa di ciò, smetterà di imparare. È un problema frequente nell'utilizzare funzioni di attivazione come la **Sigmoide** e la **Tangente Iperbolica**, poiché visionando i loro grafici, possiamo notare come ci siano grandi tratti dedicati al valore zero e al valore unitario. A causa di questa polarizzazione, le loro derivate avranno lunghi tratti vicini allo zero, portando a una non distinzione in maniera netta dei valori dei gradienti, pertanto essendo molti, vicino allo zero, tenderanno a scomparire, saturandone l'output.

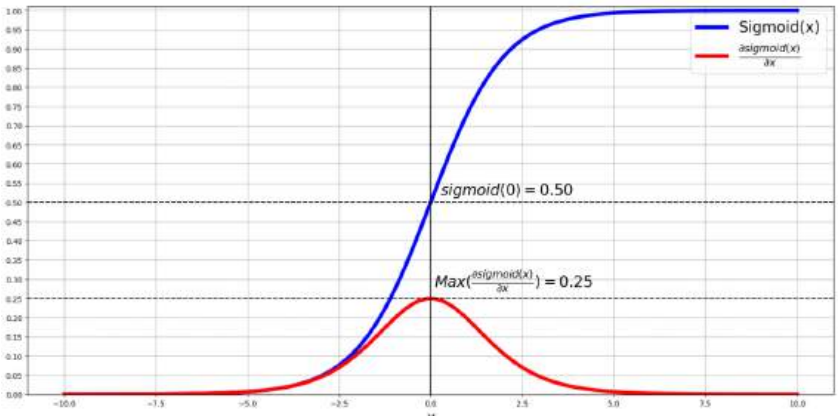


Figura 17: Grafico descrittivo del Vanishing Gradient Problem, come possiamo vedere la derivata (in rosso) risulta essere molto ridotta per la gran parte dei valori.

5.2 FUNZIONI NON SATURANTI

Per risolvere questo problema, vengono utilizzate funzioni di attivazione dette **Non Saturanti**, come:

- **ReLU (Rectified Linear Unit):** elimina i valori negativi rendendo i calcoli più efficienti, crea però dei problemi legati al gradiente dei valori negativi, rendendoli tutti uguali, inoltre soffre di un problema legato alla mancata derivabilità nell'origine, non favorevole alla computazione della discesa del gradiente;

$$\text{ReLU}(x) = \max(0, x)$$

- **Leaky ReLU:** introduce una piccola pendenza per i valori negativi, riuscendo a ridurre il problema creatosi con la *ReLU*;

$$\text{LeakyReLU}(x) = \max(\alpha x, x)$$

- **PReLU (Parametric ReLU):** simile a Leaky ReLU, con l'unica differenza della pendenza nei valori negativi, la quale viene appresa durante l'addestramento, permettendo di avere una pendenza più incisiva;

- **RReLU (Randomized ReLU)**: la pendenza anche qui viene appresa durante l'addestramento, ma si basa su una variabile casuale spostandosi all'interno di un range prefissato.

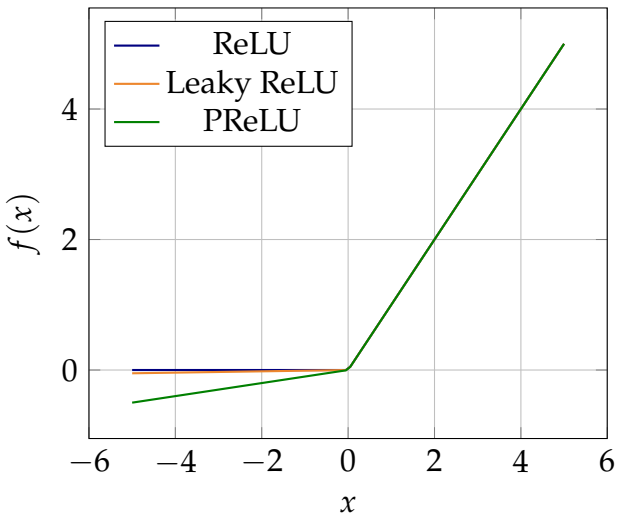
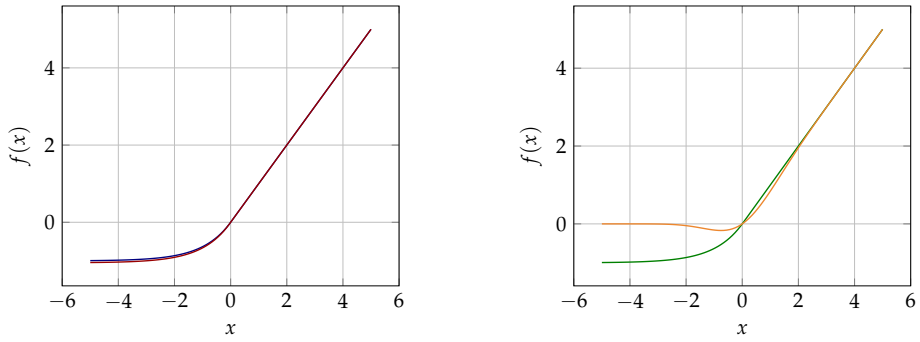


Figura 18: Grafico di ReLU, Leaky ReLU e PReLU, tutti e tre i grafici dopo l'origine si sovrappongono, e seguono lo stesso andamento.

5.3 FUNZIONI DI ATTIVAZIONE AVANZATE

Oltre alle funzioni non saturanti, esistono altre funzioni di attivazione avanzate, le quali puntano a gestire diversamente il problema dell'appiattimento dei valori negativi e della non differenziabilità nel punto di coordinata zero:

- **ELU (Exponential Linear Unit):** permette l'utilizzo di valori negativi tramite una decrescita esponenziale, migliorando la stabilità;
- **SELU (Scaled ELU):** nei valori positivi lavora esattamente come una funzione lineare scalata, mentre per quelli negativi si ricurva gradualmente, come succede nella ELU, tuttavia utilizzando un fattore di scala, permettendo di prevenire che i neuroni diventino inattivi;
- **GELU (Gaussian Error Linear Unit):** utilizza una distribuzione gaussiana, moltiplicandola per i valori ottenuti, avendo così delle transizioni più fluide fra i valori positivi e quelli negativi;
- **Softplus:** è un'approssimazione allisciata della funzione rampa vista precedentemente grazie alla funzione non saturante ReLU, la quale permette di avere delle transizioni più dolci.



(a) ELU e SELU **(b)** CELU e GELU

Figura 19: Nel grafico sono rappresentate le varie funzioni di attivazione avanzate, a sinistra, vi è il confronto fra ELU (in blu) e SELU (in rosso), a destra invece il confronto fra CELU (in arancione) e GELU (in verde).

Tutte queste funzioni di attivazione, permettono di avere un taglio non netto, fra valori negativi e positivi, garantendo una similarità fra i valori presenti prima e dopo l'origine.

5.4 FUNZIONI NORMALIZZANTI

Oltre alle moderne funzioni non saturanti, diventate lo standard nella maggior parte delle reti profonde, esiste un'altra categoria di funzioni storicamente importanti, utilizzate in contesti specifici. A differenza delle funzioni come ReLU, il cui scopo principale è introdurre non-linearità mitigando il problema dei gradienti che svaniscono, le funzioni che seguono, sono progettate per limitare i valori di output in modi specifici. Sebbene alcune di esse, come la funzione soglia (Treshold), siano oggi computazionalmente superate per l'addestramento di reti profonde, la loro comprensione è utile sia dal punto di vista storico sia perché i loro concetti di base riemergono in contesti più avanzati, come la *regolarizzazione* e l'*elaborazione dei segnali*.

- **Hardtanh:** la tanh ha una curva a forma di "S" che si avvicina asintoticamente a -1 e $+1$, la Hardtanh sostituisce questa curva con una funzione lineare a tratti. È una retta con pendenza 1 nell'intervallo $[-1, 1]$ e assume valori costanti al di fuori di questo intervallo. Evita il calcolo di funzioni esponenziali, e sebbene saturi in valori al di fuori dell'intervallo la transizione è netta e non graduale, il che in alcuni contesti può aiutare a gestire meglio il problema dei gradienti che svaniscono;

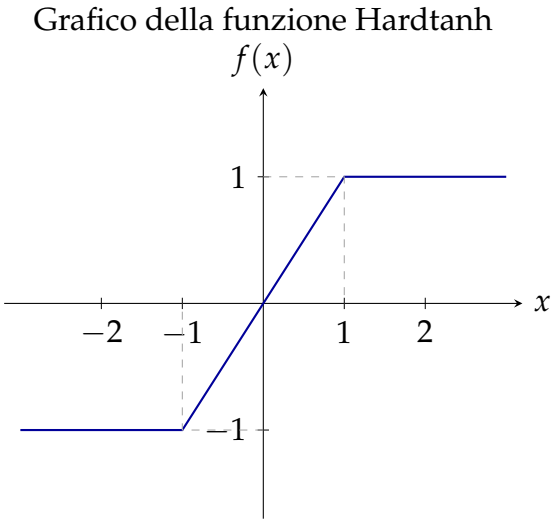


Figura 20: La funzione Hardtanh(x), usata come funzione di attivazione. È un'approssimazione lineare della funzione tangente iperbolica (tanh).

- **Threshold:** Questa è la funzione di attivazione più semplice e storicamente, la prima ad essere stata utilizzata, fra gli anni '60

e '70, ispirata al modello "tutto o niente" del neurone biologico. Essendo la sua derivata zero, praticamente ovunque, è stata immediatamente rigettata per gli studi sulle rete neurali profonde;

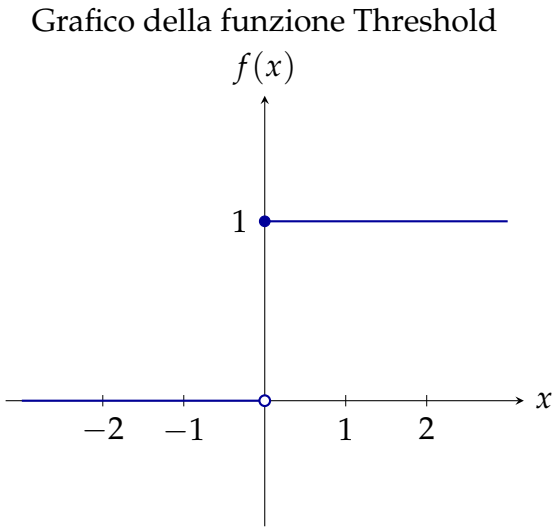


Figura 21: La funzione Threshold(x), nota anche come funzione a gradino di Heaviside.

- **Shrink Functions (Tanhshrink, Softshrink, Hardshrink):** Queste funzioni non sono tipicamente usate come attivazioni negli strati nascosti, ma in contesti di regolarizzazione, sparse coding (codifica sparsa) o elaborazione dei segnali. Il loro scopo è quello di "contrarre" (shrink) i valori di input verso lo zero, eliminando o riducendo il "rumore" di bassa ampiezza.

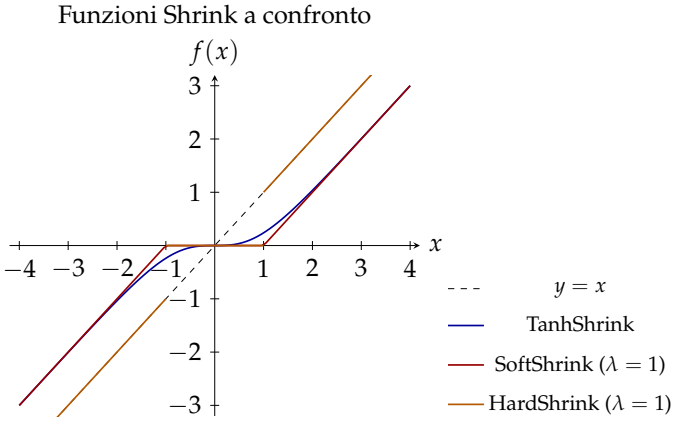


Figura 22: Confronto tra le funzioni di "shrinkage" TanhShrink, SoftShrink e HardShrink (queste ultime con $\lambda = 1$). La linea tratteggiata $y = x$ è mostrata come riferimento.

5.5 FUNZIONI PROBABILISTICHE

Fino ad ora, abbiamo analizzato funzioni di attivazione che operano in modo scalare, trasformando l'output di ogni neurone indipendentemente dagli altri. Esiste, tuttavia, una classe di funzioni che opera a livello vettoriale, trasformando un intero vettore di output in una distribuzione di probabilità. Queste funzioni sono fondamentali nello strato di output dei modelli di classificazione multiclasse, dove l'obiettivo non è semplicemente ottenere un punteggio per ogni classe, ma interpretare questi punteggi come la probabilità che l'input appartenga a ciascuna di esse. Convertendo i valori grezzi del modello (logits) in probabilità, possiamo non solo selezionare la classe più probabile, ma anche quantificarne la fiducia del modello nei confronti della sua previsione.

- **Softmax:** questa funzione di attivazione trasforma un vettore in una distribuzione di probabilità all'interno di un range, essa inoltre enfatizza le differenze assegnando a valori più grandi una probabilità più alta, mentre quelli piccoli vengono schiacciati vicino allo zero, la somma di tutti i valori è pari a 1 amplificando la differenza fra i valori scalandoli a elementi dopo la virgola;

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- **Softmin:** la differenza rispetto alla Softmax è l'utilizzo del segno meno all'esponente, dunque una semplice inversione. Questo fa sì che valori più piccoli abbiano una probabilità maggiore, mentre i valori più grandi abbiano probabilità più piccole, essa inoltre è simmetrica rispetto al softmax;

$$\text{Softmin}(x_i) = \frac{e^{-x_i}}{\sum_j e^{-x_j}}$$

- **LogSoftmax:** quest'altra tipologia di funzione di attivazione invece è una variante della Softmax, in cui si applica il logaritmo alla funzione Softmax per migliorare la stabilità numerica e semplificare il calcolo della perdita nelle reti neurali.

$$\text{LogSoftmax}(x_i) = \log(\frac{e^{x_i}}{\sum_j e^{x_j}})$$

Oss: 2 *La funzione Softmax può essere vista come la generalizzazione della funzione Sigmoide dal caso binario al caso multiclasse. In altre parole, la Sigmoide è un caso speciale di Softmax quando le classi da predire sono solo due ($K = 2$), e il valore del secondo elemento è pari a zero.*

$$z_i = \frac{\exp(x_i)}{\sum_j x_j} \Rightarrow z_1 = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} \Rightarrow x_2 = 0 \Rightarrow z_1 = \frac{x_1}{e^{x_1} + 1} \Rightarrow \frac{1}{1 + e^{-x_1}}$$

6 LOSS FUNCTION

Una **funzione di costo** (o loss function) è un elemento fondamentale nel Deep Learning, perché misura *quanto sia sbagliata la predizione di un modello rispetto ai valori reali*. L'obiettivo dell'apprendimento automatico è quello di minimizzare questa funzione, cosicché faccia previsioni più accurate. La misurazione delle funzioni di costo, risulta essere fondamentale per un criterio valutativo, inoltre permette di effetuarvi la Backpropagation nelle reti neurali.

6.1 MSE LOSS

La funzione `nn.MSELoss()` calcola la Mean Squared Error (MSE) tra le predizioni e i valori target, questa tipologia di funzione di costo utilizza la L2 Norm:

$$L = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2$$

Essa risulta essere molto sensibile ai valori marginali rispetto al fulcro dei dati, detti outliers, questo accade poiché gli errori vengono elevati al quadrato.

6.2 L1 LOSS

La `nn.L1Loss()` invece misura il Mean Absolute Error (MAE), sempre tra le predizioni e i valori target, questa tipologia di funzione di costo utilizza la L1 Norm:

$$L = \frac{1}{N}\sum_{i=1}^{N}|x_i - y_i|$$

Come possiamo immaginare rispetto alla precedente risulta essere più robusta alla presenza di outlier.

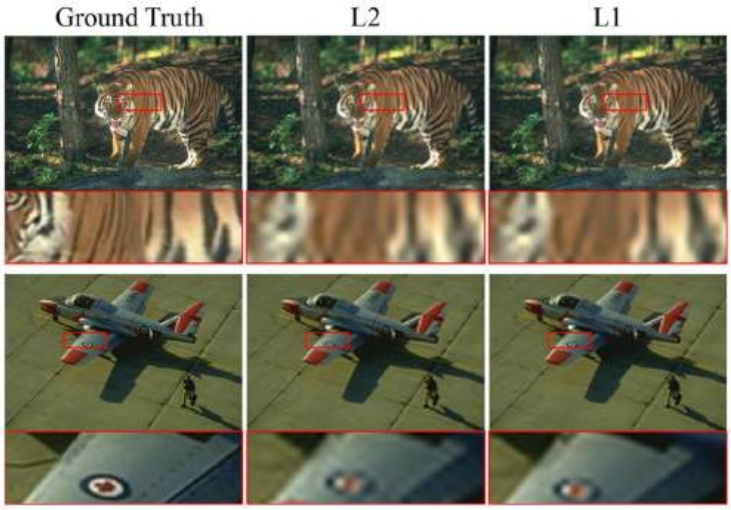


Figura 23: Utilizzando un modello di computer vision possiamo notare come è più accurata la norma 2 per figure meno spigolose, come quella della tigre, differentemente per l'immagine in cui vi è l'aereo la norma 1 è migliore poiché l'immagine risulta più spigolosa.

6.3 L1 VS L2

Ci si è chiesti pertanto, quale fra le due risulti essere migliore, alcune considerazioni si possono effettuare tramite delle visioni sperimentali. Si è mostrato come con L2,i valori vengono distribuiti uniformemente, mentre con L1 i risultati diventino netti, portando a delle eliminazioni di alcune features. Pertanto riportando questa considerazione nei modelli di Computer Vision, si verifica sperimentalmente come L1 sia migliore quando prendiamo in analisi delle immagini più "spigolose", diversamente otteniamo un risultato migliore con L2 in immagini che risultano essere più morbide nelle loro forme e nelle loro transizioni di colore (Figura 23).

6.4 SMOOTH L1 LOSS

Notando come in alcune situazioni è meglio utilizzare una norma 1 rispetto alla norma 2, si è strutturato un ulteriore modello di Loss Function il quale combina le due, il quale prende il nome di Smooth L1 Loss ed è costruito come segue:

$$L(x,y) = \begin{cases} \frac{1}{2}(x-y)^2, & \text{se } |x-y| < 1 \\ |x-y| - \frac{1}{2}, & \text{altrimenti} \end{cases}$$

6.5 NEGATIVE LOG LIKELIHOOD LOSS

La Negative Log Likelihood Loss (NLL Loss) è una funzione di costo usata nei problemi di classificazione multi-classe. Questa Loss Function, prende i dati di output di un modello, solamente se espressi in probabilità logaritmica, quindi un vettore ottenuto tramite la funzione di attivazione LogSoftMax, essa classificherà i vari dati, penalizzando in maniera ingente le predizioni sbagliate.

Es: 1 *Supponiamo di avere un modello il quale ci fornisca 3 dati grezzi (logits) come di seguito:*

$$z = [2.0\,, 1.5\,, -1.0]$$

Applicando la funzione LogSoftMax *ottengo il seguente risultato:*

$$\text{LogSoftMax(z)} = [-0.42\,, -1.92\,, -3.42]$$

*Ora la NLL Loss determina la classe corretta e ne nega il segno, nel caso in cui la classe corretta risulti essere la prima (*0.42*) essa avra un costo, molto piccolo, pertanto è stata scelta la classe corretta, ma se accidentalmente il modello dovesse considerare la terza come classe corretta, avremo un costo elevato (*3.42*), questo ci fa comprendere come questa funzione di costo penalizzi fortemente le classificazioni effettuate con alta confidenza, in maniera errata, mentre favorisca quelle corrette.*

La formula della NLL Loss è la seguente:

$$L = -\frac{1}{N}\sum_{i=1}^{N} \log P(y_i|x_i)$$

6.5.1 Problema delle classi sbilanciate

Nel caso in cui avessimo un dataset sbilanciato, il modello tenderà a minimizzare la loss, trovando come soluzione, quella di predirre sempre la classe più frequente, producendo un esito falsato. La possibilità di aggiungere dei pesi per ogni classe, potrebbe essere un'idea:

$$L = -\frac{1}{N}\sum_{i=1}^{N} w_{y_i} \log P(y_i|x_i)$$

In questo modo, valorizzeremo la classe minoritaria, che altrimenti verrebbe schiacciata dall'altra allenando il nostro modello nella maniera scorretta. Questa soluzione però non è molto efficiente, come quella di avere un dataset bilanciato in partenza.

6.6 CROSS ENTROPY LOSS

La funzione `nn.CrossEntropyLoss()` combina `LogSoftmax` e `NLLLoss` in un'unica funzione ed è la più utilizzata nei problemi di classificazione multi-classe, essa viene definita come segue:

$$L = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j}\log(\hat{y}_{i,j})$$

dove:

- C è il numero di classi,
- $y_{i,j}$ è 1 se l'osservazione i appartiene alla classe j, altrimenti è 0,
- $\hat{y}_{i,j}$ è la probabilità predetta per la classe j.

La Cross Entropy Loss, si comporta alla stessa maniera della Loss Function precedente, l'unica cosa per cui si diversifica, è la dimensione implementativa, poiché la Cross Entropy accetta a differenza dell'altra direttamente i logits, per poi effettuarci la funzione di LogSoftMax di sua inziativa.

6.7 ADAPTIVE LOG SOFTMAX WITH LOSS

Quando si hanno problemi con molte classi, l'uso di una SoftMax standard può essere inefficiente, pertanto si è introdotta l'Adaptive Log Softmax With Loss (`nn.AdaptiveLogSoftmaxWithLoss()`). Questa funzione di perdita suddivide le classi in cluster, basati sulla loro frequenza. Essa adopera la funzione SoftMax per le classi più frequenti, mentre per quelle meno frequenti calcola in primis la probabilità del cluster e poi la probabilità del singolo elemento. In questo modo si suddividono le classi in maniera gerarchica, permettendo anche di effettuare delle riduzioni notevoli sul costo computazionale complessivo.

6.8 BINARY CROSS ENTROPY LOSS

La Binary Cross Entropy Loss, è la versione binaria della Cross Entropy Loss, quindi nel momento in cui abbiamo solamente due classi nel nostro

modello di classificazione, useremo la BCE Loss, la quale segue gli stessi principi della CE Loss, si introduce quindi la funzione `nn.BCELoss()`:

$$L = -\frac{1}{N}\sum_{i=1}^{N} -w_i \left[y_i \log(x_i) + (1 - y_i) \log(1 - x_i)\right]$$

6.9 KULLBACK-LEIBLER DIVERGENCE LOSS

Nel momento in cui noi vogliamo misurare quanto la distribuzione target si discosti da quella predetta, possiamo utilizzare la Kullback Leibler Divergence Loss, basata sulla KL Divergence, una divergenza che misura quanto si discostano fra loro due distribuzioni di probabilità:

$$L = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

6.10 BCE LOSS WITH LOGITS

Un problema della BCE standard è che spesso per ottenere la sua probabilità usiamo la funzione sigmoide $\sigma(z)$, questo fattore porta a instabilità numerica. La `nn.BCEWithLogitsLoss()` incorpora direttamente la sigmoide nella perdita risolvendo il problema e rendendo tutto più stabile:

$$L = -\frac{1}{N}\sum_{i=1}^{N} - \left[y_i \log(\sigma(x_i)) + (1 - y_i) \log(1 - \sigma(x_i)\right]$$

6.11 HINGE EMBEDDING LOSS

Quando si lavora con problemi di similarità (es. metric learning), vogliamo una funzione di costo che enfatizzi la distanza tra esempi simili e dissimili. `nn.HingeEmbeddingLoss()` è utile in questi casi, lei insegna alla rete a tirare insieme le embedding di coppie simili e spingere lontano quelle diverse, finché la distanza supera una certa margin.

6.12 MARGIN RANKING LOSS

Simile alla Hinge Loss, ma si usa in compiti di ranking dove dobbiamo garantire che un elemento più rilevante abbia punteggio più alto rispetto a un elemento meno rilevante, questo permette al sistema di ordinare al meglio ciò che desideriamo.

$$L = \max(-y\,(x_1 - x_2) + \text{margin},\, 0)$$

Quì x_1 e x_2 sono gli output del modello per due esempi, mentre y può assumere il valore $+1$ o -1, il primo se $x_1 > x_2$ e il secondo nel caso in cui $x_1 < x_2$ il margin invece è quanto vogliamo che i punteggi differiscano fra loro.

6.13 TRIPLET MARGIN LOSS

Per migliorare la distinzione tra classi, `nn.TripletMarginLoss()` è ciò che viene utilizzato, questa confronta i valori con un'ancora (un valore fissato), con un esempio positivo e uno negativo, dando la possibilità di espanderlo anche a più di due categorie:

$$L = \max(0, d(a,p) - d(a,n) + \text{margin})$$

6.14 SOFT MARGIN LOSS

La Soft Margin Loss è una variante della Hinge Loss che introduce una penalità logistica per evitare margini rigidi:

$$L = \sum_{i=1}^{N} \log(1 + e^{-y_i x_i})$$

Questa funzione di costo integra la funzione SoftMax, alla logistica, aggiungendo un decadimento esponenziale alla funzione di costo.

6.15 COSINE EMBEDDING LOSS

Giungiamo ora all'ultima funzione di costo considerata in questa analisi. Immaginiamo di avere un insieme di frasi e di voler costruire un modello

capace di riconoscere quando due frasi esprimono lo stesso significato, anche se formulate in modo diverso. In altre parole, vogliamo che frasi semanticamente simili vengano rappresentate da vettori vicini nello spazio latente, mentre frasi con significati differenti risultino più distanti.

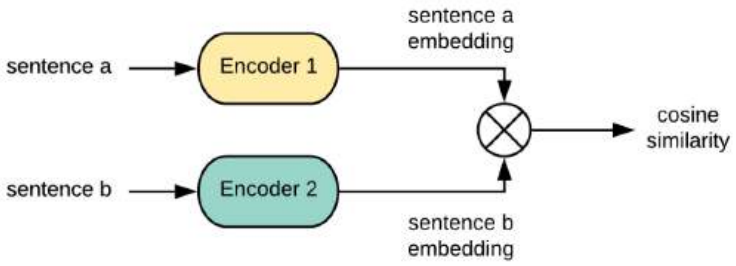


Figura 24: Rappresentazione di come vorrei siano trattate due frasi per determinarne la loro similarità.

Naturalmente, due frasi diverse, anche se trattano lo stesso argomento, tenderanno ad occupare posizioni diverse nello spazio delle embedding. Tuttavia, se il nostro obiettivo è misurare la somiglianza semantica, desideriamo che tali rappresentazioni siano quanto più possibile allineate, senza però perdere la ricchezza del linguaggio dovuta a tono, stile, gergo o espressività. Per quantificare la similarità fra due elementi come in questo caso, tra due vettori di embedding, si utilizza la **Cosine Similarity**, funzione di costo che misura il coseno dell'angolo compreso tra i due vettori:

$$\cos(x_1, x_2) = \frac{x_1 \cdot x_2}{\|x_1\| \|x_2\|}$$

Quando i due vettori sono perfettamente allineati, il coseno vale 1; quando sono ortogonali, vale 0; e quando puntano in direzioni opposte, vale -1.

Oss: 3 *Due vettori completamente diversi tendono ad essere ortogonali, indicando l'assenza di correlazione tra le loro direzioni nello spazio delle embedding.*

La `nn.CosineEmbeddingLoss()` di *PyTorch* sfrutta questa misura per addestrare modelli che devono apprendere relazioni di similarità o dissimilarità tra coppie di vettori. La sua formulazione è la seguente:

$$L = \begin{cases} 1 - \cos(x_1, x_2), & \text{se } y = 1 \text{ (coppie simili)} \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{se } y = -1 \text{ (coppie diverse)} \end{cases}$$

Questa funzione di costo incoraggia i vettori appartenenti a coppie simili a essere orientati nella stessa direzione (coseno vicino a 1), e quelli di coppie dissimili ad avere un coseno più basso, ovvero a essere separati da un angolo ampio.

7 OTTIMIZZAZIONE

Nel corso di Machine Learning abbiamo conosciuto lo **Stochastic Gradient Descent** e alcune delle sue varianti (Batch e MiniBatch), un meccanismo attraverso il quale si cercano delle condizioni di ottimalità. In questo capitolo affronteremo proprio i problemi legati all'**Ottimizzazione** e le tecniche che hanno portato all'evoluzione e alla modifica della classica discesa del gradiente. Noi sappiamo che ogni modello è caratterizzato da parametri (pesi e bias), ed essi determinano la capacità del modello nel trasformare gli input in output. L'obbiettivo del processo di training risulta essere, trovare valori dei pesi che minimizzino l'errore. La superficie d'errore (Figura 25), è spesso utilizzata per rappresentare graficamente come la funzione di costo varia al modificarsi dei pesi.

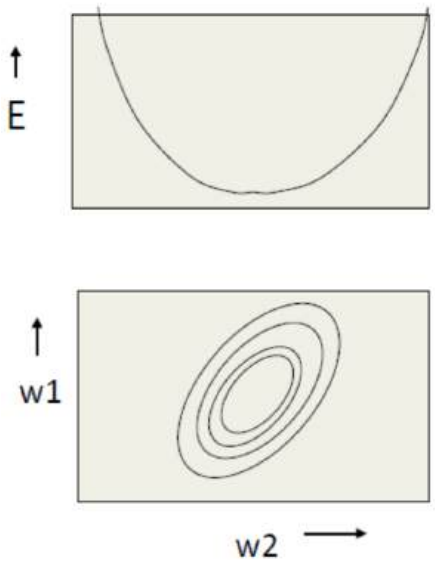


Figura 25: L'immagine illustra la "mappa" della funzione d'errore. Con l'errore rispetto a un peso di cui si mostra l'andamento parabolico, e il suo valore ottimizzato nel vertice (sopra). Inoltre una rappresentazione a due pesi, dove il valore ottimizzato si trova al centro delle ellissi, ognuna delle quali è una curva di livello (sotto).

7.1 DISCESA DEL GRADIENTE

La **Discesa del Gradiente** (Gradient Descent) è l'algoritmo che usiamo per muoverci sulla superficie d'errore per raggiungere il punto dove l'errore è minimo. Immaginando una pallina rotolare sulla superficie, all'inizio si trova in un punto alto (errore grande). Ad ogni passo, si muoverà nella direzione in cui la pendenza scende più rapidamente (cioè la direzione del gradiente negativo), essa continuerà a scendere fino ad arrivare in fondo alla "valle" della superficie, dove la pendenza è pari a zero (minimo locale o globale). Questa discesa si concretizza con l'aggiornamento continuo dei pesi nel seguente modo:

$$w_i^{(t+1)} = w_i^{(t)} - \eta \frac{\partial L}{\partial w_i}$$

Una valutazione da effettuare di volta in volta, è con quanta velocità giungiamo alla "valle" durante la nostra discesa, questa carattiristica può variare a seconda del valore assegnato al Learning Rate (η), proprio per questo motivo, scegliere il Learning Rate diventa un fattore cruciale durante l'implementazione dell'algoritmo di discesa del gradiente.

7.1.1 Scegliere il Learning Rate

Per poter decidere il valore ottimale del Learning Rate, si inizia impostandolo ad un valore elevato, per evitare di collassare all'interno di un minimo locale troppo presto. Successivamente, attraverso tecniche di decadimento (decay), si passa a ridurlo progressivamente per aumentare la probabilità di convergenza verso un minimo globale. Tra le strategie più comuni di decadimento possiamo considerare:

- **Step Decay:** il Learning Rate viene ridotto di un fattore fisso dopo un certo numero di epoche (iterazioni);
- **Exponential Decay:** il Learning Rate diminuisce esponenzialmente nel tempo;
- **Adaptive Learning Rate:** tecniche come Adam e RMSprop regolano dinamicamente, adattando il learning rate per ogni parametro della rete.

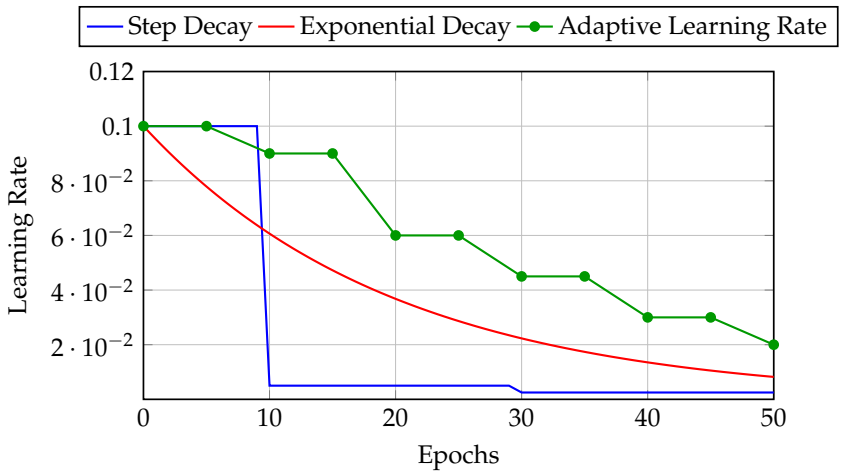


Figura 26: Confronto tra diverse strategie di decadimento del learning rate.

7.1.2 Inizializzazione dei pesi

Oltre all'ottimizzazione legata al valore del Learning Rate, risulta essere cruciale inizializzare i pesi delle reti neurali in maniera opportuna. A differenza di ciò che accade in una regressione lineare in cui l'ottimizzazione è diretta, nelle reti neurali l'algoritmo di backpropagation può essere compromesso nel caso in cui i pesi della nostra rete non siano scelti accuratamente.

Oss: 4 *Se due pesi all'interno di uno stesso layer hanno lo stesso valore iniziale, l'errore propagato all'indietro attraverso la rete, genererà gli stessi aggiornamenti, impedendo alla rete di apprendere in maniera corretta.*

Oss: 5 *Qual'ora i pesi vengano inizializzati con valori troppo grandi, riscontreremo una possibilità di esplosione della rete neurale.*

Oss: 6 *Se i pesi vengono inizializzati con valori troppo piccoli, potremmo incorrere in una possibile scomparsa del valore del gradiente.*

Oltre a queste osservazioni, si potrebbe incorrere, banalmente, in una non convergenze, generando il fenomeno dell'Overshoot, in cui il gradiente oscilla in continuazione da una parte all'altra del valore minimo a cui si dovrebbe idealmente convergere.

7.1.3 Xavier Glorot Initialization

Uno dei principali metodi di Inizializzazione è quello studiato da Xavier [34], ma prima di addentrarci in esso, vediamo due definizioni fondamentali per procedere:

Def: 7.1.1 (Fan-in) *Il Fan-in è il numero di neuroni che alimentano un neurone nel layer successivo;*

Def: 7.1.2 (Fan-out) *Il Fan-out è il numero di neuroni che un dato neurone alimenta nel layer successivo;*

Es: 2 *Consideriamo un layer completamente connesso (Fully Connected, FC) con: 256 neuroni in ingresso e 512 neuroni in uscita, il suo fan-in sarà 256 e il suo fan-out sarà pari a 512.*

Tecniche come quella studiata da Xavier Glorot prendono in considerazione proprio questi aspetti, utilizzando distribuzioni gaussiane o uniformi adeguate da cui pescare i valori dei pesi, invece di pensare a un'inizializzazione casuale. L'idea chiave di questa inizializzazione conosciuta come **Xavier** è quella di scegliere i pesi in modo tale che vengano soddisfatte le due seguenti proprietà:

1. La varianza dell'output di un neurone sia simile alla varianza del suo input;
2. La varianza dei gradienti rimanga costante tra gli strati;

In parole povere, si cerca di mantenere l'energia del segnale costante mentre attraversa la rete, sia in avanti che all'indietro. Per ottenere questo risultato, i pesi di ogni strato vengono estratti casualmente da una distribuzione (solitamente uniforme o normale) con una media nulla e una varianza specifica, nel caso della distribuzione normale, calcolata in base al numero di Fan-in e di Fan-out. Il valore della varianza solitamente è uno dei due:

$$\mathrm{Var}(W) = \frac{1}{n_{in}} \quad \text{o} \quad \mathrm{Var}(W) = \frac{2}{n_{in} + n_{out}}$$

Oss: 7 *In alcuni paper scientifici la seconda formula compare con un 1 al numeratore, questa non risulta essere sbagliata in senso assoluto, ma sono due varianti della stessa idea, quella che è riportata qui si basa sul paper ufficiale [34], in cui si adotta una media armonica tra i due requisiti prendendo il nome di He Initialization, differentemente dall'altra alle volte presente, la quale adotta una media aritmetica.*

Utilizzando una distribuzione uniforme la formula sarà la seguente:

$$\mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$$

La Xavier Initialization risulta essere un miglioramento rispetto all'inizializzazione a zero o quella casuale arbitraria, perché bilancia la propagazione del segnale nei vari strati.

7.1.4 Normalizzazione

Per normalizzare i dati, possiamo utilizzare tecniche come la *Min-Max Normalization* e la *Z-Score Normalization*, già viste nel corso di Machine Learning, esse rendono il dataset più coerente e scalabile. Il problema di queste tecniche, però è che effettuano un'assunzione quasi mai vera, ossia che le feature risultino scorrelate fra loro. Pertanto la sfida passa ad effettuare una decorrelazione delle feature in modo efficace, risultato ottenibile in diversi modi:

1. **Principal Component Analysis (PCA):** tecnica che trasforma le feature originali in nuove variabili non correlate [60, 42];
2. **Whitening:** trasformazione che rende le feature scorrelate con varianza unitaria [10, 43];
3. **Batch Normalization:** tecnica che normalizza le attivazioni durante l'addestramento per migliorare la stabilità e accelerarne la convergenza [44].

Tabella 1: Confronto tra PCA, Whitening e Batch Normalization

Caratteristica	PCA	Whitening	Batch Normalization
Tipo	Preprocessing	Preprocessing	In-training normalizzazione
Obiettivo	Riduzione dimensionalità	Decorrelazione delle feature	Accelerare il training
Agisce su	Intero dataset	Intero dataset	Mini-batch durante il training
Rende varianza uniforme?	No	Sì (varianza unitaria)	No (mantiene varianza originale)
Rende feature ortogonali?	Sì	Sì	No
Mantiene struttura dati?	Parzialmente	Meno rispetto a PCA	Sì
Uso tipico	Compressione dati, pretraining	Preprocessing per ICA, SVM	Normalizzazione nei layer di Reti Neurali
Effetto su reti neurali	Riduce dimensionalità in input	Raramente usato direttamente	Migliora stabilità e velocità di training

7.2 MOMENTUM

La Discesa del Gradiente è la base di gran parte dei metodi di ottimizzazione, nonostante la sua semplicità ed efficacia teorica, presenta alcune limitazioni pratiche. Uno dei principali problemi è la **Velocità di Convergenza** nel momento in cui la funzione di costo ha una forma fortemente non uniforme, caratterizzata da *valli strette e lunghe*. In tali casi, la pendenza della funzione può risultare molto ripida in una direzione e quasi piatta in altre, di conseguenza, il vettore gradiente, non punta direttamente verso il minimo, ma tende ad alternarsi tra le direzioni di massima e minima pendenza. Il risultato è una convergenza rallentata, come se ci trovassimo fra due pareti rocciose molto ripide e man mano

che scendiamo lungo un canale che è la direzione desiderata, andassimo a sbattere alle due pareti. Per mitigare tali effetti, sono stati introdotti ottimizzatori più avanzati estendendo la discesa del gradiente classica. Tra questi, primo fra tutti, il metodo del **Momentum** [63] in grado di ridurre le oscillazioni, accumulando una componente inerziale nella direzione di discesa, basandosi sulla storia dei gradienti precedenti, smorzando le oscillazioni.

$$\begin{cases} v_t = \beta v_{t-1} - \eta \nabla L(w_t) \\ w_{t+1} = w_t + v_t \end{cases}$$

Con v il *Momentum Parameter* il quale prende in analisi la velocità di discesa fino a un determinato momento, mentre β è il *Damping Factor*, un valore che controlla quanto la velocità passata influenza quella attuale, compreso fra zero e uno. Se fosse zero, staremmo facendo semplicemente la discesa del gradiente, nel caso unitario invece avrei un'esplosione del gradiente nella nostra rete neurale. Il valore del damping factor, ci permette di capire la velocità del cambio di direzione, più sarà piccolo, più la direzione cambierà velocemente.

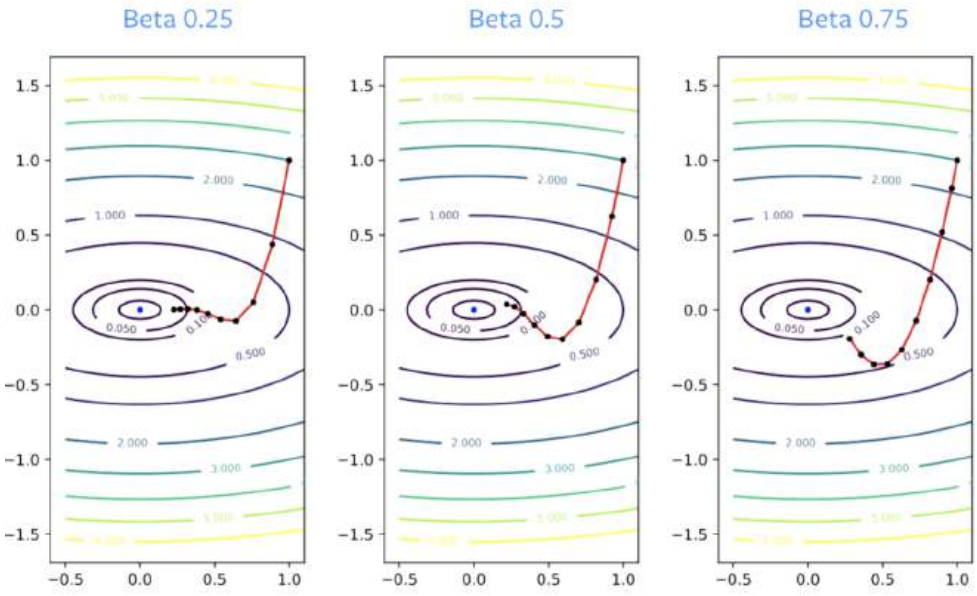


Figura 27: Confronto delle traiettorie per diversi valori di β nel Momentum.

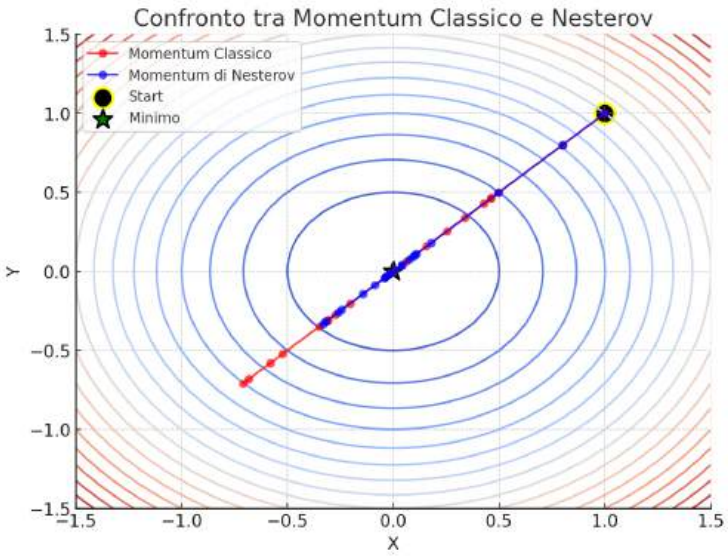


Figura 28: Nel grafico è possibile vedere la velocità di convergenza confrontando i due Momentum entrambi partendo da uno stesso punto d'inizio, il Momentum classico come potevamo aspettarci avrà delle oscillazioni molto più grandi rispetto a quelle del Momentum di Nesterov.

7.3 NESTEROV ACCELERATED GRADIENT

Il Momentum classico aiuta la discesa del gradiente accelerando il processo e smorzando le oscillazioni, ma ha un problema: aggiorna i pesi basandosi sulla direzione del gradiente calcolato nella posizione attuale, senza considerare dove il peso si troverà dopo l'aggiornamento. Il matematico Yurii Nesterov nel 1983 propose una modifica al Momentum classico, chiamata **Nesterov Accelerated Gradient** (NAG) [57, 82], migliorando la velocità di convergenza e la stabilità dell'ottimizzazione. Per comprenderne al meglio la differenza, ci basta immaginare una persona in bicicletta: nel momentum classico, la persona spinge sui pedali senza guardare avanti, e se la strada curva improvvisamente, potrei andare troppo veloce nella direzione sbagliata e dover correggere bruscamente; con il momentum di Nesterov, si guarda avanti prima di spingere sui pedali, valutando quanta forza imprimere sui pedali per correggere il movimento in base alla direzione da seguire. Il Momentum di Nesterov risulta essere più efficiente poiché calcola il gradiente in una posizione successiva vedendo in anticipo dove ci porterà la velocità calcolata, mantenendo così un'alta velocità senza instabilità.

$$\begin{cases} v_t = \beta v_{t-1} - \eta \frac{\partial L}{\partial (w_t + \beta v_{t-1})} \\ w_{t+1} = w_t + v_t \end{cases}$$

Nesterov è un piccolo ma potente miglioramento rispetto al Momentum classico. Questo metodo viene spesso usato negli ottimizzatori moderni come Adam, RMSprop con Momentum, e molte varianti di SGD, perché migliora stabilità e velocità di convergenza.

7.3.1 Perché funziona il Momentum ?

Il Momentum funziona principalmente per due motivi:

1. Accelerazione dell'ottimizzazione;

2. Attenuazione del rumore;

Accelerazione dell'ottimizzazione

L'idea del momentum è una spinta di un carrello in discesa, se dessi un piccolo colpo, si muoverà piano, se continuo a spingere il carrello guadagnerà velocità, e qual'ora smettessi di spingere, continuerà a muoversi per inerzia. Matematicamente questo viene considerato grazie all'aggiornamento dei pesi tramite l'utilizzo di un valore che tiene conto dei valori precedenti nel calcolo del gradiente. Pertanto se i gradienti puntano nella stessa direzione per più step, la velocità aumenterà e il modello sarà portato a convergere più velocemente.

Attenuazione del rumore

Con il Momentum, accumuliamo un valore mediato nel tempo, riducendo il rumore casuale come succedeva a ogni piccola variazione. Se il rumore nei gradienti punta in direzioni casuali, questi si annullano nel tempo, il movimento risulta più fluido e meno oscillante e infine permette di superare piccoli ostacoli senza rimanere bloccati in minimi locali poco profondi. Il Momentum di Nesterov, migliora ulteriormente entrambi gli aspetti anticipando il passo successivo prima di aggiornare la velocità, evitando così di andare troppo oltre e migliorando la precisione.

7.4 LEARNING RATE ADATTIVO

I parametri presenti in un modello, possono avere delle tipologie di scale di aggioranmento differenti, proprio per questo nasce l'idea di creare un **Learning Rate adattivo**. Prendendo una direzione di discesa al posto di un'altra, potremmo ritrovarci con richieste di un learning rate più grande o più piccolo, se ne usassimo uno fisso per tutti i parametri porterebbero a delle oscillazioni, o convergenze dilatate nel tempo. Quindi viene deciso di adattare il learning rate in modo indipendente per ogni parametro, basandosi sull'andamento del gradiente.

7.4.1 Additive increase multiplicative decrease

Adattare il learning rate individualmente per ogni peso con un piccolo incremento e un decremento moltiplicativo nasce per poter bilanciare due aspetti:

- Gradiente coerente in una direzione $\rightarrow$ Aumentiamo il Learning Rate accelerando la convergenza;
- Gradiente cambia direzione frequentemente $\rightarrow$ Riduciamo il Learning Rate evitando oscillazioni.

Questa tecnica è utilizzata in metodi come **Adadelta**, **Rprop** (Resilient Propagation) e alcune varianti di **Adam**, tutte tecniche che vedremo a breve. Il funzionamento è semplice: nel momento in cui valuto i singoli Learning Rate, se il gradiente mantiene la stessa direzione, aumento il valore del Learning Rate per fare passi più grandi, diversamente se cambia di segno, significa che stiamo oscillando troppo, perciò lo ridurremo per stabilizzarlo.

7.5 RESILIENT PROPAGATION (RPROP)

La **Resilient Propagation** (Rprop), introdotta da Riedmiller et al. 1993 [70], è un metodo di ottimizzazione che affronta una delle principali debolezze della discesa del gradiente: la dipendenza dalla magnitudine. L'idea consiste nel considerare unicamente il *segno* del gradiente per determinare la direzione di aggiornamento, mentre la grandezza del passo di apprendimento viene adattata dinamicamente per ciascun peso

in modo indipendente. Per ogni parametro w_i si definisce un valore di aggiornamento $\Delta_i(t)$, regolato in base alla coerenza del segno del gradiente nel tempo:

$$\Delta_i(t) = \begin{cases} \eta^+ \cdot \Delta_i(t-1), & \text{se } \frac{\partial L(t-1)}{\partial w_i} \cdot \frac{\partial L(t)}{\partial w_i} > 0 \\ \eta^- \cdot \Delta_i(t-1), & \text{se } \frac{\partial L(t-1)}{\partial w_i} \cdot \frac{\partial L(t)}{\partial w_i} < 0 \\ \Delta_i(t-1), & \text{altrimenti} \end{cases}$$

Solitamente η^+ e η^- hanno un valore rispettivamente di 1.2 e 0.5, l'aggiornamento del peso avviene quindi come:

$$w_i(t+1) = w_i(t) - \text{sign}\left(\frac{\partial L(t)}{\partial w_i}\right) \cdot \Delta_i(t)$$

In questo modo, ciascun peso dispone di un proprio Learning Rate, che cresce quando la direzione del gradiente è coerente, e diminuisce quando essa si inverte. Tale meccanismo permette di accelerare la convergenza nelle regioni piatte della funzione di costo e di ridurre le oscillazioni nelle zone di forte curvatura.

7.6 RMSPROP

Dopo gli approcci basati su aggiornamenti indipendenti dei pesi, come la **Resilient Propagation**, la ricerca sull'ottimizzazione dei modelli neurali si è orientata verso metodi in grado di adattare dinamicamente il Learning Rate in base al comportamento locale del gradiente. Uno dei risultati più efficaci in questa direzione è l'algoritmo **RMSprop** (*Root Mean Square Propagation*), introdotto da Geoffrey Hinton nel 2012 [85]. Stabilizzando la discesa del gradiente tramite il passo di aggiornamento di ciascun parametro, in funzione della sua storia recente di variazioni. Per evitare questo problema, RMSprop introduce una *media mobile esponenziale* dei quadrati dei gradienti, per mantenere un bilanciamento costante tra memoria storica e reattività alle variazioni recenti. Formalmente, il valore medio viene aggiornato secondo:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)(\nabla L(w_t))^2$$

dove γ rappresenta il fattore di decadimento (tipicamente pari a 0.9) e $\nabla L(w_t)$ è il gradiente calcolato al passo t. L'aggiornamento dei pesi diventa quindi:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla L(w_t)$$

dove η è il learning rate ed ϵ una piccola costante per evitare divisioni per zero. In questo modo, ciascun parametro dispone di un proprio passo di apprendimento adattivo:

- Se un peso riceve gradienti di grande ampiezza, il termine $E[g^2]_t$ cresce e il passo verrà ridotto, prevenendo oscillazioni e instabilità;
- Se i gradienti risultano piccoli o irregolari, il passo resterà più ampio, accelerando la convergenza.

RMSprop si distingue quindi come una strategia di ottimizzazione *localmente adattiva*, capace di bilanciare velocità e stabilità. Essa rappresenta il punto di transizione naturale tra i metodi a passo variabile per singolo parametro, come Rprop, e le tecniche completamente adattive di generazione successiva, come **AdaGrad**, che approfondiremo nella sezione seguente.

7.7 ADAGRAD

L'algoritmo **AdaGrad** (*Adaptive Gradient*) rappresenta uno dei primi approcci di ottimizzazione realmente *adattivi*, introdotto da Duchi et al. 2011 [30]. La sua idea di fondo è quella di modificare dinamicamente il tasso di aggiornamento per ciascun parametro del modello, in modo da rendere l'aggiornamento più sensibile alla frequenza e all'intensità dei gradienti associati a ogni peso. Nella discesa del gradiente tradizionale, un singolo valore di *Learning Rate* η viene applicato uniformemente a tutti i pesi. Ma i diversi parametri di una rete neurale possono contribuire in misura molto diversa alla funzione di costo: alcuni ricevono gradienti ampi e frequenti, altri piccoli o sporadici. AdaGrad affronta questo problema regolando automaticamente l'ampiezza del passo per ciascun peso in modo inversamente proporzionale alla somma cumulativa dei suoi gradienti al quadrato. Formalmente, si definisce per ogni parametro w_i un accumulatore $G_i(t)$:

$$G_i(t) = \sum_{k=1}^{t} \left(\frac{\partial L(w_i^{(k)})}{\partial w_i} \right)^2$$

e l'aggiornamento dei pesi avviene come:

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L(w_i^{(t)})}{\partial w_i}$$

dove ϵ è una piccola costante di stabilità numerica. Parametri che ricevono gradienti grandi in modo persistente avranno un denominatore elevato, quindi un passo di aggiornamento più piccolo; viceversa, quelli con gradienti rari o deboli verranno aggiornati con passi più ampi. Il risultato è un'**auto-regolazione locale del learning rate** che porta al miglioramento della convergenza. La caratteristica di AdaGrad di accumulare indefinitamente il quadrato dei gradienti comporta un effetto collaterale significativo: con il progredire dell'addestramento, $G_i(t)$ cresce costantemente, facendo diminuire il Learning Rate fino a valori prossimi allo zero. Portando a un **prematuro rallentamento della convergenza**, specialmente nelle fasi finali dell'ottimizzazione, quando il modello necessiterebbe ancora di piccoli aggiustamenti. Per ovviare a questo problema, sono state introdotte varianti più flessibili, come **Adadelta**, in grado di mantenere l'adattività di AdaGrad ma sostituendo la somma cumulativa, con una media mobile esponenziale dei gradienti al quadrato, preservando così un equilibrio più stabile nel tempo.

7.8 ADADELTA

L'algoritmo **Adadelta**, proposto da Zeiler nel 2012 [95], nasce come evoluzione di **AdaGrad**, con l'obiettivo di risolvere il problema del decadimento eccessivo del Learning Rate dovuto all'accumulo dei gradienti al quadrato. Come visto nella sezione precedente, AdaGrad ostacola la capacità del modello di continuare ad apprendere nelle fasi avanzate dell'addestramento. L'idea di Adadelta è sostituire la somma cumulativa con una **media mobile esponenziale** dei gradienti al quadrato, mantenendo una memoria a breve termine della loro variazione. In questo modo, l'algoritmo conserva la natura adattiva di AdaGrad, ma elimina il rallentamento asintotico. Formalmente, per ciascun parametro w_i, la media mobile viene aggiornata secondo:

$$E[g_i^2]_t = \rho E[g_i^2]_{t-1} + (1-\rho)\left(\frac{\partial L(w_i^{(t)})}{\partial w_i}\right)^2$$

dove ρ è il fattore di decadimento esponenziale (tipicamente compreso tra 0.9 e 0.95). L'ampiezza del passo viene poi calcolata come:

$$\Delta w_i^{(t)} = -\frac{\sqrt{E[\Delta w_i^2]_{t-1} + \epsilon}}{\sqrt{E[g_i^2]_t + \epsilon}} \frac{\partial L(w_i^{(t)})}{\partial w_i}$$

e infine i pesi vengono aggiornati come segue:

$$w_i^{(t+1)} = w_i^{(t)} + \Delta w_i^{(t)}$$

Un aspetto distintivo di Adadelta è l'introduzione di una seconda media mobile $E[\Delta w_i^2]_t$, tenente traccia dell'ampiezza media degli aggiornamenti precedenti. Questo termine funge da **meccanismo di autoregolazione del passo**, consentendo all'algoritmo di adattare dinamicamente non solo la direzione, ma anche la scala delle variazioni dei pesi. Il risultato è un comportamento più stabile e coerente, che non richiede l'impostazione manuale di un learning rate globale. Grazie a queste caratteristiche, Adadelta è particolarmente efficace in scenari in cui i gradienti variano in modo non uniforme o decrescono rapidamente nel tempo. Il metodo può essere interpretato come un ponte concettuale tra AdaGrad e gli ottimizzatori successivi, i quali ereditano il principio di adattività locale combinandolo con strategie di momentum e media mobile dei gradienti. In sintesi, Adadelta introduce una forma di apprendimento adattivo completamente autonoma, in grado di bilanciare efficacemente velocità di convergenza e stabilità numerica senza dipendere da un Learning Rate fisso.

7.9 ADAM

L'algoritmo **Adam** (*Adaptive Moment Estimation*), proposto da Kingma et al. 2015 [46], rappresenta una delle più efficaci e diffuse tecniche di ottimizzazione nel Deep Learning moderno. Adam combina i principi del **momentum** e dei metodi **adattivi** come **RMSprop**, fornendo un aggiornamento dei pesi: stabile, veloce e robusto rispetto alle variazioni della scala dei gradienti. L'idea consiste nel mantenere due medie mobili esponenziali:

- Una per la **stima del primo momento** (la media dei gradienti), analoga al momentum classico;
- Una per la **stima del secondo momento** (la media dei quadrati dei gradienti), come in RMSprop.

Formalmente, per ciascun parametro w_i, al passo t si definiscono:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla L(w_t)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla L(w_t))^2$$

dove:

- m_t rappresenta la stima del primo momento (velocità media),
- v_t rappresenta la stima del secondo momento (varianza),
- β_1 e β_2 sono coefficienti di decadimento esponenziale, tipicamente $\beta_1 = 0.9$ e $\beta_2 = 0.999$.

Entrambe le stime vengono inizializzate a zero al primo passo, proprio per questo Adam applica una correzione al bias introdotto dalle prime iterazioni trasformando le stime dei due momenti come segue:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Una volta calcolate le stime dei due momenti, si passa all'aggiornamento dei pesi:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

dove η rappresenta il Learning Rate, mentre ϵ una costante di stabilità numerica (solitamente 10^{-8}). Grazie a questa formulazione, Adam unisce:

- **Stabilità** dei metodi basati su media mobile (come RMSprop), che riducono il passo nelle direzioni ad alta curvatura;
- **Velocità di convergenza** del momentum, che mantiene la direzione media di discesa evitando oscillazioni.

L'algoritmo si adatta automaticamente alla scala dei gradienti di ciascun parametro, eliminando la necessità di un fine tuning del Learning Rate e rendendolo particolarmente efficace in reti profonde o con dati rumorosi. Per questo motivo, Adam è oggi uno degli ottimizzatori di default in numerosi framework di Deep Learning e architetture moderne (Transformer, GAN, reti convoluzionali, ecc.).

7.9.1 Limitazioni di Adam

Adam non risulta essere perfetto, poiché incorre in alcune limitazioni:

- **Sovra-adattamento ai dati rumorosi:** Adam si adatta velocemente, ma può adattarsi troppo a gradienti rumorosi, riducendo la capacità del modello di generalizzazione;

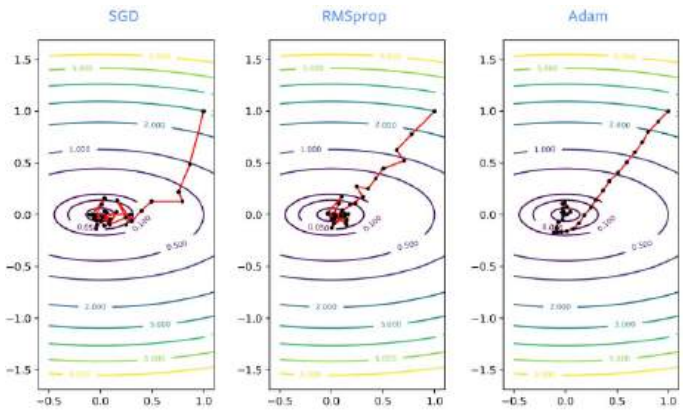


Figura 29: Grafico che mette in luce le differenze nella convergenza fra lo SGD, la RMSprop e Adam

- **Decay inefficace del peso:** Adam non fa un vero weight decay come la discesa del gradiente classica. Nel SGD viene incluso il parametro di regolarizzazione, in Adam viene fatta tramite il learning rate adattivo, dunque risulta essere meno efficace;
- **Rallentata convergenza finale:** Adam spesso converge rapidamente all'inizio, ma poi rallenta molto nel trovare il minimo ottimale. Questo poiché i Learning Rate adattivi riducono troppo la velocità di aggiornamento nelle fasi finali.

Proprio per cercare di mitigare questi comportamenti, è stata introdotta una variante evoluta di Adam, più stabile, la quale prende il nome di **AdamW**.

7.9.2 AdamW

L'ottimizzatore **AdamW** (*Adam with Weight Decay*), proposto da Loshchilov et al. 2017 [51], modifica la regolarizzazione di Adam separando esplicitamente il termine di *weight decay* dall'aggiornamento del gradiente. Nel metodo Adam classico, la penalizzazione L_2 sui pesi viene incorporata nel gradiente stesso, alterando le medie mobili e introducendo interazioni indesiderate con il meccanismo adattivo. AdamW rimuove questa interferenza applicando il decadimento dei pesi come un'operazione separata:

$$w_{t+1} = w_t - \eta \lambda w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

dove λ è il coefficiente di *weight decay*. Questa semplice modifica migliora la **generalizzazione** e la **stabilità numerica**, specialmente

nei contesti di addestramento di grandi modelli come BERT e GPT. AdamW è oggi la variante di riferimento dell'ottimizzatore Adam ed è implementato come impostazione predefinita nei principali framework di Deep Learning.

7.10 LION

Adam è l'ottimizzatore più utilizzato nel campo della ricerca, ma nonostante ciò, non risulta essere il migliore, nel Febbraio del 2023, è stato proposto un ottimizzatore completamente diverso da quelli precedenti, perché proposto da un Agente di Intelligenza Artificiale. Crearlo tramite un calcolatore, ha portato a raffinare le formule trasponendole direttamente in linee di codice, dando la possibilità di considerarne l'impatto sul costo computazionale. Per rendere più compatto il codice, l'agente ha introdotto una funzione, che è esattamente quella presente in tutti gli ottimizzatori che abbiamo visto fin'ora (funzione di interpolazione). L'algoritmo utilizzato per trovare Lion permetteva di analizzare vari ottimizzatori, migliorati nei vari cicli di aggiornamento, effettuando di volta in volta un torneo fra loro, in modo da combinarne i migliori e generarne dei figli adottando i pregi dei genitori, permettendo di raffinarsi sempre più, esattamente come accade con le manipolazioni genetiche. Di seguito il codice implementativo di Lion:

```
def train(weight, gradient, momentum, lr):
    update = interp(gradient, momentum, β1)
    update = sign(update)
    momentum = interp(gradient, momentum, β2)
    weight_decay = weight * λ
    update = update + weight_decay
    update = update * lr
return update, momentum
```

7.11 GRADIENTE E CURVATURA

Scegliendo la direzione in cui convergere, è corretto chiedersi di quanto diminuisca l'errore prima di ricomonciare a salire, solitamente, si assume la curvatura costante, cioè una superficie di errore quadratica, assumiamo nella nostra analisi che l'entità del gradiente diminuisca man mano che ci spostiamo verso il basso. La riduzione massima

dell'errore dipende dal rapporto fra il gradiente e la curvatura, e una buona direzione nella quale è efficiente muoversi è quella con un elevato rapporto fra gradiente e curvatura, dunque escludere la derivata seconda in ogni direzione, è un'approssimazione, considerarla, ci permette di aumentare la generalizzazione dei nostri algoritmi di convergenza, pertanto il giusto interrogativo da porsi risulta essere come trovare il giusto valore del rapporto.

7.12 METODO DI NEWTON

Il **Metodo di Newton** [58] è la soluziona a questo interrogativo, integrando la curvatura e il gradiente, moltiplicando la matrice Hessiana con il gradiente stesso. Prima di entrare nel fulcro del metodo di Newton, ricordiamo un paio di concetti fondamentali.

Gradiente

Sia $f : \mathbb{R}^n \to \mathbb{R}$ un campo scalare. Il gradiente $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ è un vettore tale che $(\nabla f)_i = \frac{\partial f}{\partial x_i}$. Poiché ogni punto nel dominio di f è mappato in un vettore, allora ∇f è uno spazio vettoriale.

Jacobian

Sia $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$ un campo vettoriale. Allora il Jacobiano può essere considerato come il campo vettoriale delle derivate. Considerando ogni componente di $\mathbf{F}$ come una singola funzione, il Jacobiano pertanto è una matrice la quale i-esima riga è il gradiente dell'i-esima componente di $\mathbf{F}$, dunque se il Jacobiano è J allora:

$$J_{i,j} = \frac{\partial F_i}{\partial x_j}$$

Hessiano

L'Hessiano è la matrice delle derivate seconde parziali di ogni combinazione di elementi:

$$H_{ij} = \frac{\partial^2 J}{\partial x_i \partial x_j}$$

- Le **componenti diagonali** indicano la curvatura lungo un singolo asse, coincidendo con la derivata seconda;

- Gli **elementi fuori diagonale** descrivono le interazioni tra i pesi.

Ogni elemento della matrice Hessiana, specifica come il gradiente si modifica in una direzione, nel momento in cui ci muoviamo in un'altra direzione diversa da quella del gradiente. Il metodo di Newton introduce tramite l'utilizzo dell'inversione della matrice Hessiana la seguente relazione:

$$\Delta w = -\epsilon H(w)^{-1} \frac{dL}{dw}$$

Aggiungendo un'informazione non solo sul primo ordine, come avviene con i gradienti, ma anche sul secondo ordine, avendo una: precisione maggiore e una velocità di convergenza migliore. Tuttavia questo risulta essere problematico, poiché calcolare una matrice inversa, oltre ad essere computazionalmente elevato, non sempre risulta essere possibile.

Oss: 8 *L'intuizione di utilizzare l'Hessiana risiede dal fatto che il gradiente ci fornisce l'informazione della direzione in cui muoverci, mentre l'Hessiana dice di quanto curvare il passo. La discesa del gradiente segue la pendenza più ripida, Newton invece "raddrizza" la traiettoria tenendo conto della curvatura.*

7.12.1 Metodi alternativi per gestire l'Hessiano

Calcolare l'inverso della matrice Hessiana H è computazionalmente complesso ($O(n^3)$) e spesso impraticabile per modelli di grandi dimensioni. Per questo motivo sono stati sviluppati diversi per ovviare a questa difficoltà, a partire dai Metodi quasi-Newton, i quali effettuano una stima di H basandosi sulle informazioni dei gradienti precedenti, o **Metodi Hessian-Free** come quello del *metodo del gradiente coniugato* per minimizzare l'errore di ottimizzazione. Recentemente, questa famiglia di tecniche è stata estesa anche a contesti più complessi, come l'*Online Certified Unlearning*, in cui si richiede di rimuovere selettivamente l'influenza di specifici dati di addestramento senza dover riaddestrare completamente il modello. Un esempio significativo è il lavoro di Qiao et al. [64], che propone un approccio **Hessian-Free Online Certified Unlearning**, nel quale l'informazione di secondo ordine viene stimata in modo efficiente per garantire una rimozione certificata e stabile dei dati. Tale metodologia dimostra come i principi dei metodi Hessian-Free possano essere estesi anche a problemi di sicurezza e robustezza dei modelli di apprendimento automatico.

7.13 GRADIENTE CONIUGATO

Il **Metodo del Gradiente Coniugato** rappresenta un'alternativa efficiente al metodo di Newton per la ricerca del minimo di una funzione di costo, in quanto consente di evitare l'inversione esplicita della matrice Hessiana. Il metodo del Gradiente Coniugato, evolve quello del Gradiente Classico, effettuando una serie di step affinché si trovi il minimo in una direzione, una volta ottenuto, procedo lungo un'altra direzione lasciando intatto il minimo lungo la direzione precedente (gradiente), trovando quello della direzione attuale. Se soddisfano questa caratteristica, le direzioni sono dette **coniugate** fra loro, da cui deriva il nome del metodo, il quale si articola nel seguente modo:

1. Si sceglie una direzione iniziale proporzionale al gradiente:

$$d_0 = -\nabla L(w_0);$$

2. Si determina il passo ottimale lungo tale direzione:

$$\alpha_0 = \arg\min_{\alpha} L(w_0 + \alpha\, d_0);$$

3. Si effettua l'aggiornamento del peso:

$$w_1 = w_0 + \alpha_0\, d_0;$$

4. Si calcola una nuova direzione che sia **coniugata** rispetto all'Hessiana, ossia ortogonale rispetto alla metrica indotta da H:

$$d_1 = -\nabla L(w_1) + \beta_1 d_1$$

dove il coefficiente β può essere calcolato come:

$$\beta_1 = \frac{\nabla L(w_1)^T \nabla L(w_1)}{\nabla L(w_0)^T \nabla L(w_0)}.$$

Ovviamente questa procedura è iterativa, pertanto basta sostituire i pedici, con il passo corretto a cui sto iterando, la grandezza di questo metodo risiede nella posibilità di al più N iterazioni in uno spazio di dimensione N, il gradiente coniugato raggiunge il minimo esatto per una funzione quadratica, evitando oscillazioni e migliorando sensibilmente la velocità di convergenza rispetto alla discesa del gradiente standard.

8 NORMALIZATION LAYERS

La normalizzazione rappresenta una delle tecniche più influenti e determinanti per l'addestramento stabile e rapido delle reti neurali profonde. L'idea alla base è quella di mantenere la distribuzione delle attivazioni intermedie entro range controllati, evitando che il flusso del gradiente diventi instabile. Tra i metodi di normalizzazione, la **Batch Normalization (BN)** ha avuto un impatto particolarmente significativo, poiché affronta efficacemente il fenomeno del cosiddetto *internal covariate shift*.

8.1 COVARIATE SHIFT

Il **Covariate Shift** [77] descrive una condizione in cui la distribuzione degli input $P(X)$ varia tra fase di addestramento e fase di test, mentre la relazione condizionale $P(Y|X)$ rimane invariata. Formalmente:

$$P_{\text{train}}(X) \neq P_{\text{test}}(X), \quad \text{ma} \quad P_{\text{train}}(Y|X) = P_{\text{test}}(Y|X)$$

Questo comporta che il modello, pur avendo appreso correttamente la funzione $P(Y|X)$, si trovi ad operare su una distribuzione di input differente da quella su cui è stato addestrato, con conseguente degrado delle prestazioni. Un esempio classico è quello di un modello di visione artificiale addestrato su immagini acquisite di giorno e testato su immagini notturne: le caratteristiche visive (input) cambiano, ma il concetto di "automobile" rimane invariato.

8.1.1 Covariate Shift Interno

Nelle reti profonde, un fenomeno analogo si verifica *all'interno* del modello. Durante l'addestramento, gli aggiornamenti dei pesi nei layer inferiori modificano costantemente la distribuzione delle attivazioni fornite ai layer successivi. In questo modo, ogni strato si trova a dover continuamente riadattarsi a un input che cambia distribuzione. Questo effetto, viene chiamato **Internal Covariate Shift (ICS)**, rallenta

la convergenza e rende l'ottimizzazione meno stabile. La **Batch Normalization** è stata introdotta proprio per mitigare questo problema, introducendo un layer di normalizzazione fra i vari layer (Figura 30), normalizzando così le attivazioni intermedie in modo che ogni layer riceva un input dalla distribuzione più stabile possibile, a differenza di ciò che accadeva normalmente effettuando una normalizzazione post-hoc. Nella normalizzazione effettuata senza *Batch Normalization* il flusso del gradiente portava a bias espolsivi, e la disconnessione fra ottimizzazione e normalizzazione era una causa dei problemi, non tenendo conto proprio della normalizzazione in alcuni casi.

8.2 BATCH NORMALIZATION

Adesso entriamo un po' più nell'aspetto matematico della **Batch Normalization**, la quale applica una normalizzazione per mini-batch, standardizzando ciascuna feature affinché abbia media zero e varianza unitaria:

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{2(k)} + \epsilon}}$$

dove $\mu_B^{(k)}$ e $\sigma_B^{2(k)}$ sono rispettivamente la media e la varianza calcolate sul batch per la k-esima feature. Successivamente, si introducono due parametri appresi $\gamma^{(k)}$ e $\beta^{(k)}$ che ripristinano la capacità rappresentativa dello strato:

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

In questo modo, la rete può annullare la normalizzazione se necessario, garantendo flessibilità e capacità espressiva.

8.2.1 Posizionamento nella rete

La BN è tipicamente inserita *dopo* la trasformazione lineare o convoluzionale e *prima* della funzione di attivazione, secondo lo schema:

$$\text{Linear/Conv} \rightarrow \text{BatchNorm} \rightarrow \text{ReLU}.$$

Questo approccio viene ripetuto in molteplici punti della rete, poiché ogni layer può introdurre una variazione statistica nelle proprie attivazioni. Nelle architetture moderne come le *ResNet*, gli strati di normalizzazione sono presenti in quasi ogni blocco convoluzionale.

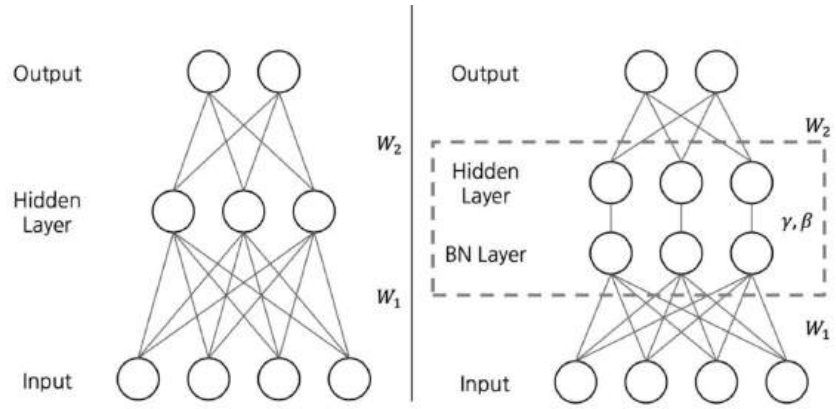


Figura 30: Inserimento di un layer di Batch Normalization fra un input layer e un hidden layer, i layer di Batch Normalization possono essere molteplici all'interno di un'architettura profonda, a volte sono presenti prima di ogni hidden layer.

8.2.2 Effetti e benefici

L'inserimento sistematico di BatchNorm comporta numerosi vantaggi pratici:

- Stabilizza le distribuzioni di attivazione e riduce l'*internal covariate shift*;
- Permette di utilizzare learning rate più elevati;
- Riduce gradienti esplosivi o vanescenti;
- Diminuisce la sensibilità all'inizializzazione dei pesi;
- Introduce una forma implicita di regolarizzazione dovuta alla variabilità stocastica del batch;
- Migliora la generalizzazione e la rapidità di convergenza.

8.3 VARIANTI DELLA NORMALIZZAZIONE

Sebbene la Batch Normalization sia estremamente efficace, la sua dipendenza dalle statistiche di batch può diventare un limite, specialmente in presenza di batch molto piccoli o in contesti sequenziali. Sono quindi nate diverse varianti, che differiscono per le dimensioni sulle quali vengono calcolate media e varianza.

8.3.1 Layer Normalization

La **Layer Normalization (LN)** [6] normalizza le attivazioni considerando tutte le feature di un singolo campione:

$$\mu_L = \frac{1}{C \cdot H \cdot W} \sum_{c,h,w} x_{c,h,w} \qquad \sigma_L^2 = \frac{1}{C \cdot H \cdot W} \sum_{c,h,w} (x_{c,h,w} - \mu_L)^2$$

Ogni campione è quindi normalizzato indipendentemente dagli altri. È particolarmente efficace in architetture **RNN** e **Transformer**, dove i batch possono essere di dimensione variabile e l'indipendenza temporale è fondamentale.

8.3.2 Instance Normalization

La **Instance Normalization (IN)** [89] normalizza ciascun canale e ciascun campione separatamente:

$$\mu_{n,c} = \frac{1}{H \cdot W} \sum_{h,w} x_{n,c,h,w} \qquad \sigma_{n,c}^2 = \frac{1}{H \cdot W} \sum_{h,w} (x_{n,c,h,w} - \mu_{n,c})^2$$

Viene utilizzata prevalentemente in applicazioni di *style transfer* e *image-to-image translation*, poiché rimuove le informazioni di contrasto e illuminazione specifiche di ciascuna immagine.

8.3.3 Group Normalization

La **Group Normalization (GN)** [93] costituisce un compromesso tra BN e LN. I canali vengono suddivisi in G gruppi, e la normalizzazione viene applicata all'interno di ciascun gruppo:

$$\mu_g = \frac{1}{(C/G) \cdot H \cdot W} \sum_{c \in g,h,w} x_{c,h,w}$$

La GN è indipendente dalla dimensione del batch e risulta particolarmente efficace nelle CNN quando si utilizzano batch piccoli, dove la BN tende a produrre statistiche instabili.

8.4 CONCLUSIONI

La normalizzazione ha avuto un impatto determinante nello sviluppo di reti sempre più profonde e stabili. Sebbene l'ipotesi iniziale dell'*internal*

Metodo	Media/Varianza su	Dip. dal batch	Vantaggi	Limiti
BatchNorm	Batch × Spaziale (per canale)	Sì	Stabilità e velocità	Instabile per batch piccoli
LayerNorm	Tutte le feature (per esempio)	No	Indipendente dal batch	Meno efficace in CNN
InstanceNorm	Spaziale (per canale e campione)	No	Rimuove info di stile	Perde dettagli discriminativi
GroupNorm	Gruppi di canali (per esempio)	No	Flessibile e stabile	Parametro G da scegliere

Tabella 2: Confronto tra diversi tipi di normalizzazione.

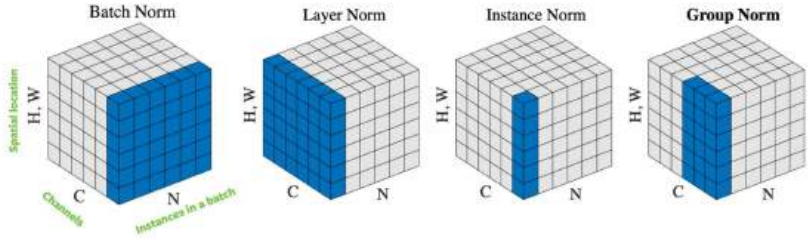


Figura 31: Rappresentazione schematica delle principali tecniche di normalizzazione impiegate nei modelli di Deep Learning.

covariate shift abbia fornito l'intuizione originale [45], studi successivi [74, 14, 13] hanno evidenziato che il successo di tali tecniche deriva principalmente dal miglioramento della geometria della funzione di costo e dalla stabilizzazione del flusso del gradiente. Oggi, l'uso di meccanismi di normalizzazione — sia in forma di *Batch*, *Layer*, *Instance* o *Group* — rappresenta uno standard de facto nella progettazione di architetture neurali moderne, migliorando la generalizzazione e la robustezza dei modelli in quasi ogni dominio applicativo.

9 CONVOLUTIONAL NEURAL NETWORKS

Le **Convulutional Neural Network** (CNN) [48] prendono ispirazione dal funzionamento dell'organizzazione della corteccia visiva animale, così come il nostro cervello essa riconosce: volti, forme e oggetti a partire da stimoli visivi grezzi. Le CNN sono in grado d'imparare a rilevare caratteristiche visive come: bordi, texture e forme; essendo in grado di combinarle per riconoscere oggetti più complessi. Le reti neurali trattate fin'ora, sono state quelle completamente connesse, non adatte per elaborare delle immagini grandi, infatti se considerassimo il dataset **CIFAR-10**, le immagini al suo interno sono composte da 32x32x3 pixel, e se dovessimo considerare una rete neurale completamente connessa, avrebbe nel suo primo layer ben 3072 pesi per un singolo neurone, un numero elevato, ma potrebbe essere considerato gestibile. Immaginiamo ora di spostarci a immagini con dimensioni sempre più grandi (e.g 200x200x3), avrò molti più pesi per neurone (e.g 120.000). Questo viene considerato uno spreco, portando all'overfitting e inefficienza computazionale. Proprio quì subentrano le reti convoluzionali, in grado di prendere degli input e trattarli come un volume tridimensionale (Larghezza, Altezza e Profondità), permettendo di produrre un nuovo volume tridimensionale chiamato **Activation Volume** tramite l'applicazione di **Filtri**. Ogni neurone, a differenza della *Fully Connected* è connesso solo a una porzione locale dello spazio, chiamata **Receptive Field**, il valore dell'estensione di questo campo di connettività è un iperparametro.

9.1 ANALISI DELL'ARCHITETTURA

Una Rete Convoluzionale è suddivisa in livelli, i quali si occupano di manipolazioni dei dati in maniere differenti in ognuno di essi: uno strato convoluzionale, uno strato di attivazione, uno strato di polling posto a conclusione di più coppie dei livelli precedenti e infine a conclusione uno strato completamente connesso. Le **CNN**, applicano dei filtri, immaginabili come dei piccoli parallelepipedi, aventi il compito di con-

Figura 32: Rappresentazione di una piccola parte del Cifar Dataset, contenente una delle collezioni di immagini più utilizzate nel campo del deep learning e della computer vision, per l'addestramento e la valutazione di reti neurali.

centrarsi su una specifica parte dell'immagine di volta in volta, il filtro scorre attraverso l'altezza e la larghezza della nostra immagine, concentrandosi però su tutta la profondità, apportando una convoluzione generando una mappa di attivazione (Figura 33).

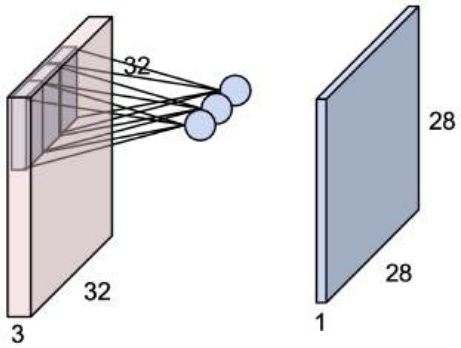


Figura 33: A sinistra la rappresentazione intera del nostro input al quale viene applicato un filtro ripetutamente, i piccolo parallelepipedi blu, in varie zone dell'input, generando degli output di volta in volta, portano alla creazione dell'activation layer, sulla destra.

Se dovessimo limitarci esclusivamente a una singola caratteristica la rappresentazione presente in Figura 33, sarebbe sufficiente. Per avere un'efficiente riconoscitore d'immagine, si utilizzano però, più filtri compattati, come se fossero un unico filtro, in modo tale da ottenere un'analisi più approfondita su vari aspetti dell'immagine stessa, ottenendo così quello che è visibile nella Figura 34. Ed è proprio qui che lo strato di profondità logicamente si inspessisce, differentemente da ciò che accadeva con il singolo filtro che lo lasciava immutato. A seguito dello strato convolutivo, vi è uno strato di attivazione non lineare, solitamente viene utilizzata la funzione **ReLU**, ma è possibile utilizzare

una qualunque delle funzioni di attivazione presenti nel Capitolo 5, ed è proprio quì che la nostra rete inizia ad apprendere vari elementi dell'immagine di partenza.

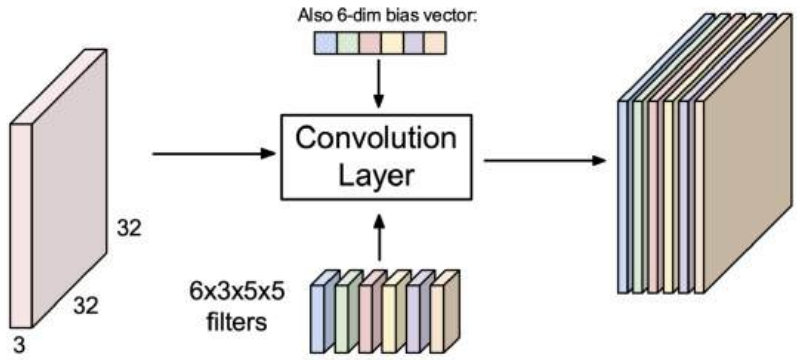


Figura 34: Viene applicata la convoluzione con un multi filtro che permette di rilevare più feature in un'unica volta, il quale ci genera uno stack di attivazione più profondo, di quello di partenza, ma con dimensioni ridotte nelle altre dimensioni.

9.2 IL FILTRO

Avendo analizzato lo strato di Convoluzione, soffermiamoci un momento su una parte costituente di questo strato: **il filtro**. Esso viene essere utilizzato con diversi obbiettivi, come effettuare *la decisione della dimensionalità* opportunatamente scelta in base alla dimensione dell'input layer. Il passo o stride del filtro, è una caratteristia molto importante, poiché determina di quanto si discosta il filtro una volta applicato la prima volta, per essere riapplicato iterativamente al di sopra del nostro input layer (Figura 35). Per poter calcolare dunque la dimensione dell'output, si utilizza una semplice formula, che tiene conto della dimensione dell'input (N), la dimensione del filtro (F) e la dimensione dello stride utilizzato (S).

$$Output = \frac{N - F}{S} + 1$$

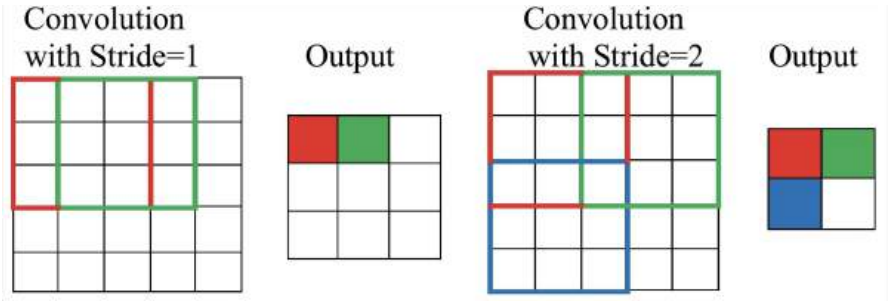


Figura 35: Applicazione di un filtro a uno stesso input layer, con a sinistra uno passo unitario, mentre a destra un passo di due, generando a un output layer di dimensionalità differente.

Nella pratica questa formula viene arricchita, poiché vi è la necessità di aggiungere un **Padding** al nostro input, applicato prima ancora che si applichi il filtro. Questo avviene poiché in primis, effettuando un rimpicciolimento dell'immagine (come desiderato), dopo diversi strati, l'immagine giunge a dimensioni molto piccole, non risultando una riduzione efficiente, in secondo luogo spesso nelle convoluzioni gli elementi posti negli angoli, non vengono efficaciemente rappresentati rispetto agli altri più centrali. Grazie al padding queste problematiche vengono arginate, permettendo una riduzione oculata e una partecipazione paritaria dei singoli pixel. L'equazione della dimensione dell'otuput pertanto si modifica nel seguente modo:

$$Output = \frac{N + 2P - F}{S} + 1$$

9.2.1 Filtri 1x1

I filtri 1x1 nelle reti neurali convoluzionali (CNN) possono sembrare controintuitivi, ma svolgono ruoli fondamentali nell'ottimizzazione e nella potenza espressiva delle reti. Un filtro 1x1 infatti applica una convoluzione su ogni pixel, considerando però tutti i canali contemporaneamente (e.g R, G, B). Non modifica la dimensione spaziale (altezza e larghezza), ma agisce sulla profondità (numero di canali). Esso può servire per:

- **Riduzione della dimensionalità:** Un filtro 1×1 può ridurre il numero di canali, diminuendo così i parametri e i costi computazionali. Ad esempio, da 256 canali a 64;

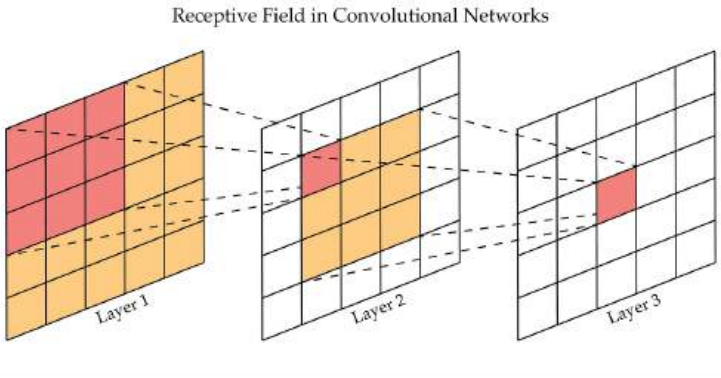


Figura 36: Rappresentazione del receptive field, su tre layer.

- **Aumento della dimensionalità:** Al contrario, può aumentare i canali per arricchire la rappresentazione dei dati;

- **Aggiunta di non linearità:** Combinato con funzioni di attivazione (eg. ReLU), introduce un'ulteriore non-linearità, migliorando la capacità di apprendimento della rete;

- **Interazione tra canali:** Permette di combinare informazioni tra diversi canali, facilitando l'apprendimento di caratteristiche complesse.

9.3 RECEPTIVE FIELDS ESTESI

Il **Receptive Field** (campo recettivo) è la porzione dell'immagine originale che influenza l'attivazione di un singolo neurone in uno specifico strato. Più si va in profondità nella rete, più il campo visivo del neurone si espande: cominciando a vedere regioni più grandi dell'immagine (Figura 36). Questo perché, se mi muovo in un layer più profondo, oltre a visualizzare i neuroni da cui il singolo elmento viene generato, vedrò anche a loro volta i neuroni da cui vengono generati i neuroni su cui mi sto soffermando, e per questo espanderò di volta in volta in ogni layer il mio campo recettivo, avendo una visione di insieme, sempre più espansa. Considerando L come il layer in cui ci troviamo e K come il numero di dimensione del filtro, possiamo calcolare il Receptive Field (R), nel seguente modo:

$$R = 1 + L \cdot (K - 1)$$

Dunque per ottenere una visione globale dell'immagine che stiamo analizzando, avremo bisogno di numerosi layer che mi diminuiscono la dimensionalità.

9.4 POOLING LAYER

I livelli di **Pooling** sono tipicamente interposti tra i vari strati convoluzionali e sono quelli che adoperano realmente la riduzione della dimensionalità, estendendo così il *Receptive Field*. La loro funzione è quella di ridurre progressivamente la dimensione spaziale delle mappe di attivazione, attraverso un processo di *Downsampling*. Questo consente di diminuire il numero di parametri e il carico computazionale della rete, mitigare il rischio di overfitting e favorire l'invarianza a piccole traslazioni dell'input. Le operazioni di Pooling più comuni includono:

- **Max Pooling:** seleziona il valore massimo all'interno di una finestra locale (kernel) sull'immagine di input;
- **Average Pooling:** calcola la media dei valori nella regione coperta dal kernel;
- **Sum Pooling:** somma tutti i valori nella regione considerata;

Attraverso il Pooling si ottiene una rappresentazione più compatta delle mappe di caratteristiche, permettendo alla rete di concentrarsi sulle caratteristiche più significative, riducendo al contempo la sensibilità a piccole variazioni locali. Combinando in modo efficace, strati convoluzionali, di attivazione e di pooling, una CNN è in grado di apprendere progressivamente, rappresentazioni gerarchiche dell'input, rilevando strutture sempre più complesse come: bordi, texture, oggetti o intere scene. Infine, l'output dello strato convoluzionale finale viene *appiattito* in un vettore monodimensionale, che viene successivamente fornito a uno o più strati completamente connessi (*fully connected*) per eseguire il compito di classificazione.

9.5 CONCLUSIONI

Le reti neurali convoluzionali (CNN) sfruttano in modo efficiente la struttura spaziale delle immagini attraverso tre principi fondamentali:

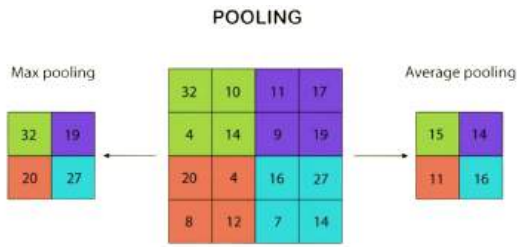


Figura 37: Effetto del pooling applicato con kernel 2×2 e stride 2: a sinistra, il valore massimo selezionato per ciascuna finestra (Max Pooling); a destra, la media dei valori nella stessa finestra (Average Pooling).

connettività locale, condivisione dei pesi filtrati e l'organizzazione gerarchica dei livelli. Questi elementi architetturali consentono una significativa riduzione del numero di parametri, migliorano la capacità di generalizzazione del modello e rendono la rete scalabile su immagini di grandi dimensioni. L'uso della connettività locale permette di concentrarsi su porzioni limitate dell'input, la condivisione dei pesi garantisce l'invarianza traslazionale e riduce il costo computazionale. L'approccio modulare e gerarchico, rende le CNN adatte a compiti complessi di riconoscimento visivo.

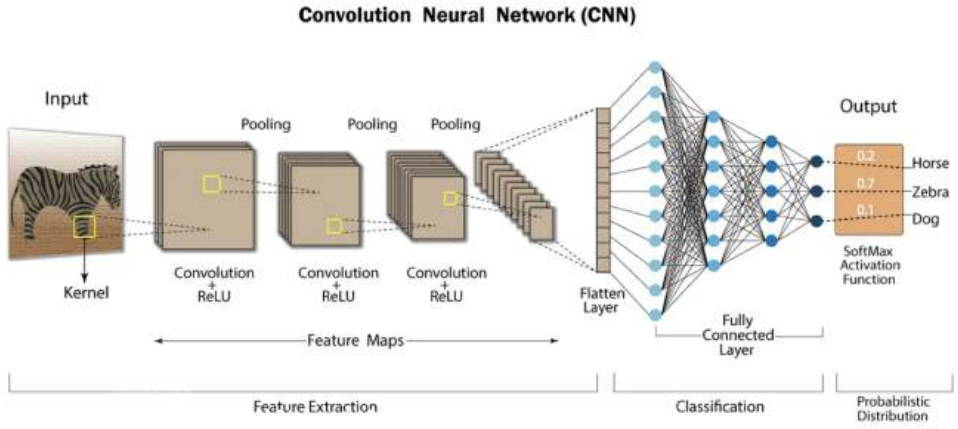


Figura 38: Rappresentazione schematica di una rete neurale convoluzionale. Ogni livello svolge un'operazione distinta, contribuendo all'estrazione progressiva delle caratteristiche rilevanti dall'immagine.

Tabella 3: Confronto tra reti convoluzionali (CNN) e reti completamente connesse (Dense)

Caratteristica	**Reti Dense**	**Reti Convoluzionali**
Architettura	Ogni neurone connesso a tutti gli altri	Connettività locale tra neuroni
Numero di parametri	Elevato, cresce rapidamente con la dimensione dell'input	Ridotto grazie alla condivisione dei pesi
Efficienza computazionale	Bassa per immagini ad alta risoluzione	Alta, grazie all'uso di kernel
Invarianza traslazionale	Non intrinseca	Intrinseca grazie ai filtri convoluzionali
Adattabilità alle immagini	Limitata, richiede preprocessing intenso	Alta, adatte all'elaborazione visiva diretta
Capacità di generalizzazione	Tende a overfittare su dataset piccoli	Migliore grazie al minor numero di parametri

10 RECURRENT NEURAL NETWORKS

Molti dei dati che incontriamo nel mondo reale sono di natura sequenziale: il testo è una sequenza di parole, la musica è una sequenza di note, i dati finanziari sono una serie temporale di valori, ecc. . . Lavorare con questo tipo di dati presenta una sfida, si note come il significato di un elemento, spesso, dipenda da quelli precedenti. Le tradizionali reti neurali feed-forward, che elaborano ogni input in maniera indipendente, non sono adatte a questo compito, poiché mancano di una "memoria" per conservare informazioni sugli eventi passati. In questo capitolo affronteremo le **Reti Neurali Ricorrenti** (RNN) le quali sono state progettate proprio per far fronte a questo limite [72, 31]. La capacità di possedere una memoria, risulta fondamentale in applicazioni dove il contesto è tutto, l'introduzione dello stato ricorrente permette di superare i limiti di approcci classici, come le catene di Markov [52, 65], che faticavano a catturare relazioni complesse tra eventi consecutivi a lungo termine.

10.1 ARCHITETTURA DELLE RNN

Le reti neurali **FeedForward** elaborano gli input in un'unica direzione, dall'input verso l'output, senza tenere conto della sequenzialità o della dipendenza temporale dei dati. In tal modo, presuppongono che ogni input e output siano indipendenti dagli altri. Questa assunzione le rende inadatte a compiti in cui il contesto o l'ordine temporale sono fondamentali (e.g traduzione, modellazione linguistica, riconoscimento vocale). Le **Reti Neurali Ricorrenti** (RNN), invece, sono progettate per lavorare su sequenze, riconoscendo pattern temporali, grazie alla capacità di mantenere una *memoria* implicita dello stato precedente della rete, grazie a una catena di retroazione. Considerando l'espressione idiomatica inglese "Feeling under the weather", comunemente usata per indicare che qualcuno è malato. L'espressione ha senso solo se le parole vengono pronunciate in quell'ordine specifico. Una RNN è in grado di interpretarla correttamente perché, elaborando ogni parola nel contesto delle precedenti, riesce a mantenere il significato dell'intera sequenza.

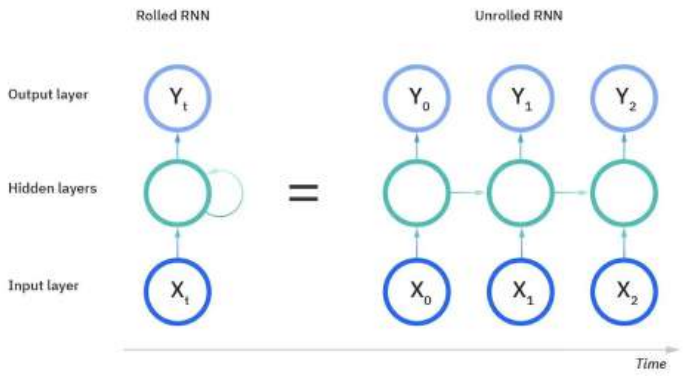


Figura 39: Rappresentazione schematica di una Recurrent Neural Network, nella versione "arrotolata" (a sinistra) e "srotolata" (a destra).

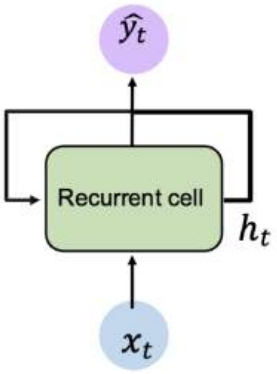


Figura 40: Rappresentazione della cella di ricorrenza, che integra input corrente e stato precedente attraverso la catena di retroazione.

La vista "arrotolata" della Figura 39 rappresenta l'intera rete come una singola entità che incorpora la memoria. La vista "srotolata" invece mostra i diversi *timestep*, ognuno dei quali corrisponde a un'istanza temporale della rete (e.g "Feeling", "under", "the", "weather"). Ogni nodo tiene conto dello stato nascosto accumulato nel tempo per poter predire correttamente il token successivo.

10.1.1 La cella di ricorrenza

La caratteristica distintiva delle RNN è la presenza di una struttura ricorsiva che consente di mantenere uno *stato nascosto* (*hidden state*) lungo la sequenza (Figura 40). A ogni istante temporale, la rete riceve l'input corrente (x_t) e lo stato nascosto del passo precedente (h_{t-1}), aggiornando lo stato attuale (h_t) tramite una funzione parametrizzata. Questo

comportamento si può formalizzare tramite la seguente equazione:

$$h_t = f_W(x_t, h_{t-1})$$

Dove f_W è una funzione non lineare parametrizzata dai pesi W. Mentre l'output della rete a un dato tempo t, indicato con $\hat{y}_t$, viene poi calcolato a partire dallo stato nascosto corrente h_t. Matematicamente, l'aggiornamento dello stato nascosto e il calcolo dell'output possono essere espressi come segue:

$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t) \qquad \hat{y}_t = W_{hy}^T h_t$$

Un modo intuitivo per visualizzare una RNN è attraverso la sua rappresentazione "srotolata" nel tempo (Figura 39). Sebbene la rete abbia un solo insieme di pesi (W_{hh},W_{xh},W_{hy}), possiamo immaginarla come una sequenza di copie della stessa cella, ognuna delle quali passa il proprio stato a quella successiva. Riducendo il carico computazionale rispetto a una rete profonda convenzionale e consente di modellare sequenze con efficienza. Un esempio semplificato del flusso di dati in una RNN può essere descritto dal seguente pseudocodice Python:

```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

sentence = ["I", "love", "recurrent", "neural"]
for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)

nextWordPrediction = prediction
# >>> "networks!"
```

10.1.2 Modelli di sequenza

La modellazione temporale può variare in base al tipo di task e dal modo in cui si desidera produrre l'output. In Figura 41 sono illustrate diverse strategie:

- **Many-to-One:** intera sequenza come input e un solo output finale (e.g sentiment analysis);
- **One-to-Many:** un input iniziale e genera una sequenza di output (e.g image captioning);
- **Many-to-Many:** input e output sequenziali (e.g machine translation).

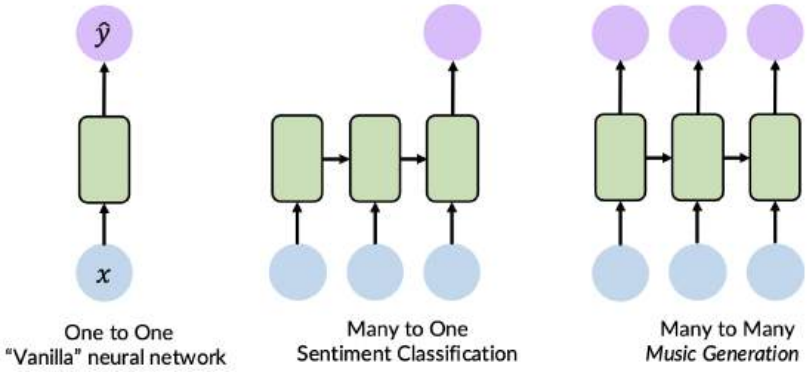


Figura 41: Esempi di modellazione della sequenza in RNN, adattabili al tipo di problema.

10.2 RAPPRESENTARE LE SEQUENZE

Prima di poter fornire una sequenza testuale a una RNN, dobbiamo trasformare le parole in un formato numerico che la rete possa elaborare. Esistono diversi approcci, con diversi livelli di complessità e capacità rappresentativa.

10.2.1 Rappresentazioni Vettoriali

- **One-hot encoding:** Una rappresentazione semplice dove ogni parola è codificata come un vettore binario lungo quanto il vocabolario, con un solo elemento pari a 1 (in corrispondenza dell'indice della parola) e tutti gli altri a 0. Sebbene facile da implementare, questa rappresentazione è inefficiente e non cattura alcuna relazione semantica tra parole simili;

- **Word Embeddings:** Sono la soluzione moderna. Si tratta di rappresentazioni dense e a bassa dimensionalità dove le parole sono mappate in uno spazio vettoriale continuo. In questo spazio, parole semanticamente simili (e.g "gatto" e "cane") risultano geometricamente vicine. Questi vettori possono essere pre-addestrati su grandi corpus di testo o appresi direttamente durante l'addestramento del modello.

Oss: 9 *La rappresentazione densa delle parole, tramite word embeddings, è stata resa popolare dai modelli Word2Vec [53] e successivamente migliorata da GloVe [61].*

10.2.2 Approcci Non-Ricorrenti

- **Finestra Fissa:** Una strategia che consiste nell'utilizzare solo un numero fisso n di parole precedenti per predire la successiva. Se la finestra è troppo piccola, il modello perde informazioni contestuali cruciali; se è troppo grande, diventa computazionalmente inefficiente e complesso da gestire;

- **Bag of Words:** In questo metodo, una frase viene rappresentata come un vettore che indica la presenza o la frequenza di ogni parola del vocabolario, ignorando completamente il loro ordine. Questo approccio perde tutta l'informazione sintattica, che è invece fondamentale per la comprensione del linguaggio. Un cambiamento nell'ordine delle parole può alterare radicalmente il significato di una frase, un aspetto che le RNN gestiscono nativamente.

La tabella di seguito riassume le caratteristiche di questi approcci:

Tabella 4: Confronto tra approcci per la rappresentazione del testo e la predizione sequenziale.

Metodo	Caratteristiche principali	Vantaggi	Svantaggi
Finestra fissa	Considera solo n parole precedenti	Computazionalmente semplice	Non gestisce dipendenze a lungo termine
One-hot encoding	Rappresentazione binaria con un solo 1 attivo	Facile da implementare	Non cattura relazioni semantiche
Bag of Words (BoW)	Rappresenta la presenza o frequenza delle parole ignorando l'ordine	Semplice e interpretabile	Perde completamente l'informazione sull'ordine
Word Embeddings	Vettori densi che catturano somiglianze semantiche	Efficienza e rappresentazione semantica	Richiede training e risorse computazionali

10.3 BACKPROPAGATION THROUGH TIME

L'addestramento di una RNN avviene tramite una variante dell'algoritmo di backpropagation chiamata **Backpropagation Through Time** (BPTT) [92]. Poiché la rete è "srotolata" nel tempo, l'errore calcolato all'ultimo passo temporale viene propagato all'indietro attraverso tutti i passi precedenti per aggiornare i pesi condivisi. Durante questo processo, il gradiente della funzione di costo rispetto ai pesi viene calcolato

applicando ripetutamente la regola della catena. Lo stato nascosto h_t dipende da h_{t-1}, che a sua volta dipende da h_{t-2}, e così via. Questo crea una lunga catena di moltiplicazioni di matrici Jacobiane:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W}$$

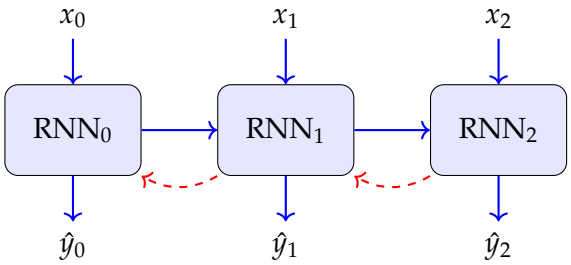


Figura 42: Illustrazione della Backpropagation Through Time (BPTT), in cui gli errori si propagano all'indietro lungo la sequenza temporale.

Questa struttura porta inevitabilmente a due problemi già noti:

- **Vanishing Gradient:** i gradienti si riducono esponenzialmente man mano che si propaga l'errore, impedendo l'apprendimento delle dipendenze a lungo termine;
- **Exploding Gradient:** i gradienti crescono esponenzialmente, destabilizzando l'addestramento.

10.4 GATED CELL

In entrambi i casi, il modello fatica a gestire sequenze molto lunghe o con relazioni temporali distanti. Per risolvere queste problematiche, vengono introdotte le **Gated Cells**, architetture più sofisticate, le quali utilizzano sempre le celle ma con l'aggiunta di meccanismi di gate. L'idea è quella di dotare la cella di meccanismi interni i quali controllino il flusso di informazioni, decidendo cosa dimentica, cosa memorizzare e decidere cosa inoltrare al passo successivo.

10.4.1 Long Short Term Memory Networks

Introdotte nel 1997 da Hochreiter e Schmidhuber [41], le **Long Short-Term Memory** (LSTM) sono la variante di RNN più celebre e diffusa.

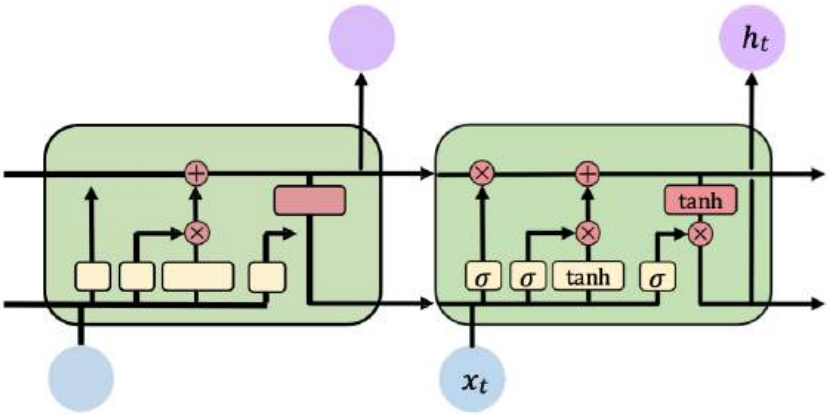


Figura 43: Rappresentazione della cella LSTM, che interagisce con un timestep precedente

Una cella LSTM (Figura 43), è molto più complessa di una cella RNN semplice. Oltre allo stato nascosto (h_t), gestisce uno stato della cella (C_t), il quale agisce come un nastro trasportatore su cui le informazioni possono viaggiare quasi inalterate lungo la sequenza. In `TensorFlow`, una cella LSTM può essere facilmente implementata con il comando `tf.keras.layers.LSTM(numUnits)`. Il cuore delle LSTM sono i quattro *gate*, che regolano l'informazione tramite funzioni sigmoidi e moltiplicazioni punto a punto, adesso li analizzeremo uno alla volta nel dettaglio.

Forget Gate

Il *Forget Gate* decide quali informazioni dello stato precedente devono essere mantenute. Per farlo, combina l'input corrente con lo stato nascosto precedente, e li elabora tramite una funzione sigmoide. I valori prodotti, compresi tra 0 e 1, indicano il grado con cui ciascun elemento della memoria deve essere conservato: più vicino a 0 indica dimenticare, più vicino a 1 conservare.

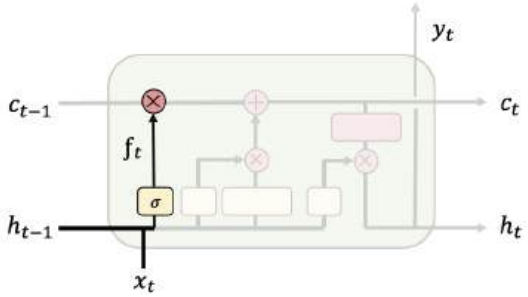


Figura 44: Rappresentazione del gate Forget della cella LSTM

Store (o Input Gate)

Lo *Store Gate*, o *Input Gate*, determina quali nuove informazioni devono essere aggiunte alla memoria della cella. Anche qui, l'input e lo stato nascosto precedente sono elaborati da due percorsi:

- Una funzione sigmoide, che seleziona le componenti informative da aggiornare;
- Una funzione tanh, che normalizza i nuovi valori da memorizzare in un intervallo tra -1 e 1.

Il prodotto tra questi due risultati indica quali parti della nuova informazione saranno effettivamente aggiunte allo stato interno.

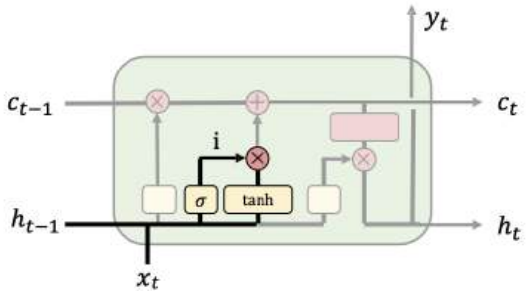


Figura 45: Rappresentazione del gate Store della cella LSTM

Update Gate

L'*Update Gate* rappresenta il cuore del funzionamento della LSTM. Qui lo stato della cella viene aggiornato combinando:

- Lo stato precedente, pesato dal Forget Gate;
- Le nuove informazioni, selezionate dallo Store Gate.

La somma dei due produce il nuovo stato della cella, considerando tutti i valori che adesso la rete neurale ritiene rilevanti.

Output Gate

Infine, l'*Output Gate* decide quale informazione deve essere trasmessa al prossimo timestep, ovvero, il nuovo stato nascosto. L'input e lo stato nascosto precedenti vengono processati da una funzione sigmoide, mentre lo stato interno aggiornato, attraversa una funzione tanh. Il prodotto punto a punto di questi due risultati rappresenta l'output finale della cella.

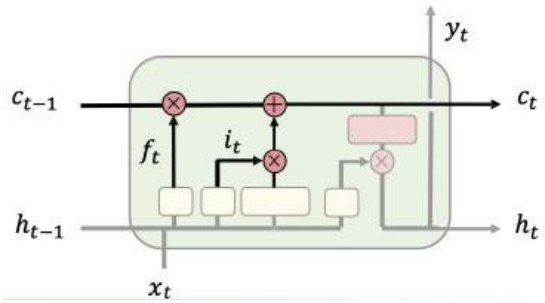


Figura 46: Rappresentazione del gate Update della cella LSTM

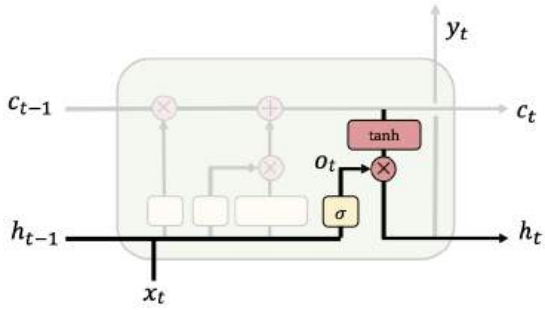


Figura 47: Rappresentazione del gate Output della cella LSTM

Backpropagation solved

Grazie a queste operazioni additive e moltiplicative, le LSTM creano dei "percorsi" attraverso i quali il gradiente può fluire senza svanire o esplodere, permettendo alla rete di apprendere dipendenze a lunghissimo termine.

10.4.2 Gated Recurrent Unit

A seguito delle LSTM, vi è stata l'implementazione di un'architettura più semplice chiamata **Gated Recurrent Unit** (GRU) proposte successivamente da Cho et al. [21], la quale rappresenta una versione semplificata con efficienza maggiore. Questa architettura infatti fonde lo stato della cella e lo stato nascosto in un unico vettore, utilizzando solo due gate invece che tre (l'update gate non viene considerato propriamente un gate), giungendo a delle performance in molti task praticamente equiparabili con le LSTM, di seguito i gate e i loro compiti:

- **Reset Gate:** Decide quanto dello stato precedente deve essere dimenticato;
- **Update Gate:** Controlla quanto dello stato precedente deve essere mantenuto e quanto della nuova informazione deve essere

aggiunta.

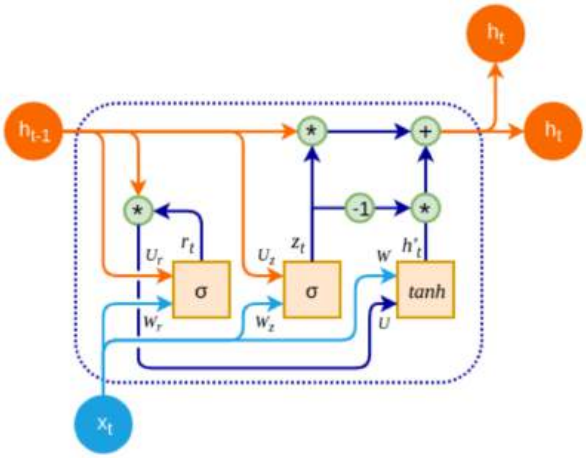


Figura 48: Rappresentazione della cella GNU, con tutti i suoi gate

10.4.3 Bi-LSTM

In molti compiti, come la traduzione o l'analisi di un testo, per comprendere il significato di una parola è utile conoscere non solo il contesto che la precede, ma anche quello che la segue. Le **RNN Bidirezionali** (Bi-LSTM) [75], riescono a risolvere questo problema, permettendo di elaborare la sequenza in due direzioni contemporaneamente:

1. Una LSTM forward processa la sequenza dall'inizio alla fine;

2. Una LSTM backward processa la sequenza dalla fine all'inizio.

Gli stati nascosti delle due reti vengono poi concatenati a ogni passo temporale, fornendo una rappresentazione ricca che tiene conto sia del passato che del futuro. Questa potenza ha un costo: due reti LSTM devono essere addestrate simultaneamente, il che comporta un maggiore uso di memoria e tempi di addestramento più lunghi. Un esempio applicativo di questa architettura è il modello ELMo [62], un modello di word embedding (rappresentazione vettoriale delle parole) contestualizzato, il quale risolve la polisemia, l'ambiguità lessicale di una parola a seconda del contesto in cui si trova, in quanto una stessa parola può avere diverso signficato.

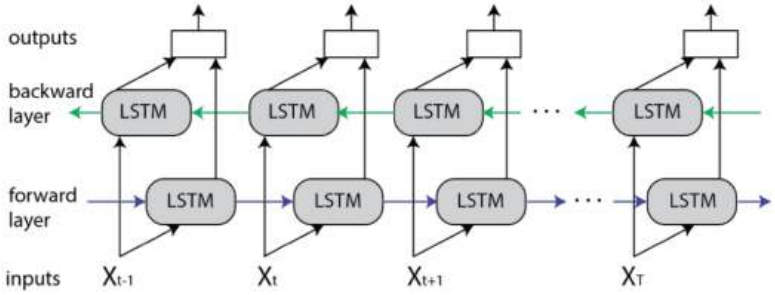


Figura 49: Architettura di una Bi-LSTM, che elabora la sequenza in entrambe le direzioni.

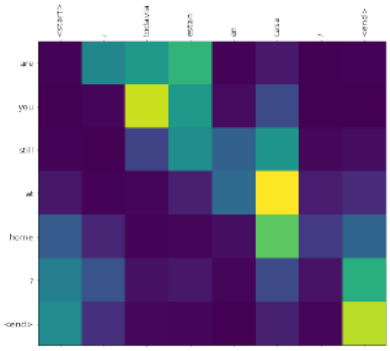


Figura 50: Rappresentazione del meccanismo dell'attenzione in un compito di traduzione: le zone più gialle indicano alta probabilità di corrispondenza tra parole.

10.4.4 Il Meccanismo dell'Attenzione

Un limite intrinseco delle architetture Encoder-Decoder basate su RNN è che devono comprimere l'intera sequenza di input in un unico vettore di contesto di dimensione fissa. Questo diventa un collo di bottiglia per sequenze molto lunghe, il **Meccanismo dell'attenzione** (attention mechanism) [7], che approfondiremo in seguito nel Capitolo 14, risolve questo problema in modo brillante. Invece di affidarsi a un singolo vettore di contesto, l'attenzione permette al modello di "guardare indietro" all'intera sequenza di input a ogni passo della generazione dell'output, e di decidere dinamicamente su quali parti concentrarsi. Assegna un "punteggio di attenzione" a ogni stato nascosto dell'input, indicando quanto è rilevante per predire l'output corrente. Questo non solo migliora drasticamente le performance, specialmente nella traduzione automatica, ma rende anche il modello più interpretabile: possiamo visualizzare su quali parole di input il modello si sta "concentrando" per produrre una certa parola di output.

10.4.5 Bi-LSTM con Attenzione

Un limite delle Bi-LSTM è che, nonostante processino la sequenza in entrambe le direzioni, alla fine restituiscono un singolo vettore che riassume l'intera sequenza. Questo può essere problematico se la sequenza è molto lunga o se alcune parti sono più rilevanti di altre. Proprio il **Meccanismo dell'Attenzione** risolve questo problema: assegna un peso α_t a ciascun passo temporale, indicando quanto ogni stato nascosto deve contribuire all'output finale.

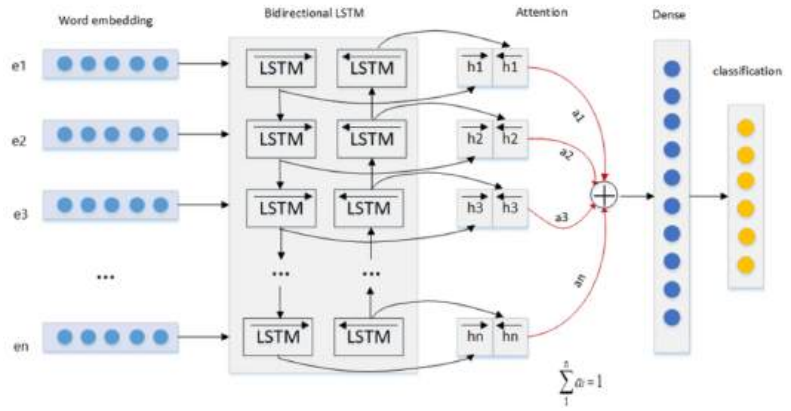


Figura 51: Architettura di una Bi-LSTM con meccanismo di attenzione

In sintesi, la Bi-LSTM cattura un contesto locale e posizionale ricco per ogni parola, mentre l'attenzione impara le relazioni a lungo termine e le dipendenze globali, permettendo al modello di concentrarsi selettivamente. Questa combinazione rappresenta una delle architetture ricorrenti più performanti per compiti complessi di NLP come la traduzione automatica, la classificazione del testo e il question answering.

10.4.6 Campi di Applicazione

Grazie alla loro capacità di modellare le dipendenze temporali, le RNN e le loro varianti, sono diventate uno strumento frequente in innumerevoli campi di applicazione dell'Intelligenza Artificiale, tra cui:

- **Elaborazione del Linguaggio Naturale (NLP):** Traduzione automatica, riassunto di testi, analisi del sentimento, chatbot e generazione di testo;
- **Riconoscimento Vocale:** Trascrizione dell'audio parlato in testo;

- **Generazione di Musica:** Composizione di melodie e armonie;
- **Analisi di Serie Temporali:** Previsioni finanziarie, meteorologiche e monitoraggio di segnali biomedici;
- **Computer Vision:** Generazione di descrizioni per immagini (image captioning) e analisi di video.

11 MIGLIORARE LA GENERALIZZAZIONE

La **Generalizzazione** rappresenta il cuore del Machine e del Deep Learning: un modello non deve semplicemente memorizzare i dati del training set, ma dev'essere in grado di effettuare previsioni accurate anche su esempi mai visti prima. Per questo motivo, i dataset sono solitamente suddivisi in training set e test set: quest'ultimo non viene mostrato al modello durante l'addestramento, ma serve a valutare la capacità di generalizzazione. Nel Deep Learning, la questione diventa ancora più delicata: i modelli hanno un'elevata capacità espressiva e possono facilmente adattarsi eccessivamente ai dati di addestramento, incorrendo nel fenomeno noto come **Overfitting**.

11.1 OVERFITTING

L'**Overfitting** si verifica nel momento in cui un modello, si adatta troppo ai dati di training, arrivando a modellare anche rumore ed irregolarità accidentali, spesso dovute a errori di campionamento. In questi casi, il modello perde la capacità di distinguere tra regolarità significative e dettagli irrilevanti. Questo accade in particolare con modelli molto flessibili, che hanno una capacità sufficiente per apprendere qualsiasi cosa, incluse le fluttuazioni casuali dei dati, limitandone la sua capacità di generalizzare.

> *"Se un modello è abbastanza potente da spiegare tutto, finirà per spiegare anche ciò che non è rilevante."*

Oss: 10 *Per visualizzare il fenomeno dell'overfitting, si può immaginare una persona vestita con taglie di vestiti differenti, nel caso in cui un vestito risultasse essere troppo aderente parleremo di overfitting, viceversa se troppo largo, parleremo di underfitting.*

11.1.1 Prevenire l'Overfitting

Per prevenire l'Overfitting, è possibile adottare tre strategie:

- **Espandere il dataset:** Risulta essere la soluzione più efficace, ma spesso difficile da realizzare nella pratica;
- **Utilizzare modelli con capacità adeguata:** si cerca di evitare reti troppo semplici (Underfitting) o troppo complesse (Overfitting);
- **Effettuare un ensamble:** effettuare una combinazione di più modelli utilizzando architetture diverse o addestramenti su sottoinsiemi.

L'espansione del dataset aiuta a ridurre il rumore e aumenta la significatività statistica. I modelli troppo complessi rischiano l'Overfitting, mentre quelli troppo semplici non riescono a catturare la struttura dei dati (Underfitting). Infine, l'ensembling permette di mediare tra i diversi comportamenti, bilanciando bias e varianza, questa possibilità viene effettuato utilizzando tecniche di *Bagging* e di *Boosting*, su cui ci soffermeremo più avanti.

11.2 LIMITARE LA CAPACITÀ DI UN MODELLO

La capacità di un modello può essere controllata in diversi modi:

- **Architettura:** definizione del numero di layer e neuroni;
- **Early Stopping:** interrompere l'addestramento prima che inizi l'Overfitting;
- **Weight Decay:** aggiungere una penalizzazione ai pesi grandi tramite regolarizzazione L1 o L2;
- **Noise Injection:** aggiungere rumore agli input o alle attivazioni per migliorare la robustezza;
- **Dropout:** È una delle forme di regolarizzazione più potenti e specifiche per il Deep Learning. Durante ogni passo di training, una frazione casuale di neuroni viene temporaneamente "spenta" (le loro uscite vengono impostate a zero).

Tutte queste tecniche mirano a ridurre la complessità effettiva del modello, favorendo una maggiore capacità di generalizzazione, vediamole ora meglio.

11.2.1 Early Stopping

L'**Early Stopping** è una tecnica che interrompe l'addestramento nel momento in cui il modello inizia a sovra-adattarsi ai dati. Nelle prime fasi, i pesi sono piccoli e i neuroni operano nella regione lineare; col progredire dell'allenamento, i pesi aumentano, i neuroni entrano nella regione non lineare e il modello diventa più potente, rischiando però l'Overfitting. Fermare l'allenamento nel momento opportuno consente di mantenere una buona capacità predittiva. È pratica comune utilizzare un **Validation Set** per monitorare l'andamento dell'errore e decidere quando arrestare l'addestramento.

11.2.2 Weight Decay

La regolarizzazione L2 introduce un termine additivo nella funzione di costo per penalizzare i pesi grandi:

$$L(\theta) = \mathcal{L}(\theta) + \lambda \sum_i \theta_i^2$$

Questo termine forza i pesi a rimanere piccoli, evitando soluzioni troppo complesse. Il parametro λ controlla l'intensità della penalizzazione:

- λ troppo grande $\Rightarrow$ Underfitting;
- λ troppo piccolo $\Rightarrow$ rischio di Overfitting.

La regolarizzazione L1 ($\sum_i |\theta_i|$) è un'alternativa che induce **sparsità**, cioè tende a portare molti pesi a zero, dunque adoperando una sorta di selezione delle features.

11.2.3 Noise Injection

Aggiungere rumore ai dati d'ingresso o ai pesi agisce come regolarizzatore. Ad esempio aggiungendo un **rumore Gaussiano**:

$$x_t^{(noisy)} = x_t + \mathcal{N}(0, \sigma^2)$$

Il rumore, amplificato dai pesi, penalizza indirettamente pesi grandi. L'effetto è simile al weight decay: il modello diventa più robusto e meno dipendente da feature specifiche.

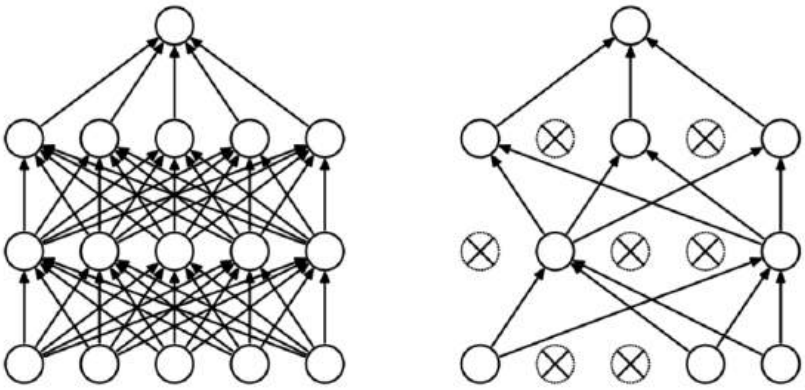


Figura 52: Una Rete Neurale completamente connessa (a sinistra). La stessa Rete Neurale dopo che è stato applicato il Dropout (a destra).

11.2.4 Dropout

Il **Dropout** è una tecnica di regolarizzazione che disattiva casualmente neuroni durante l'addestramento. Ogni forward pass attiva una sotto-rete diversa, riducendo la dipendenza da co-attivazioni specifiche. Con n neuroni, si campionano 2^n reti. I pesi sono condivisi, quindi ogni sottorete è fortemente regolarizzata, il dropout permette di ridurre gli adattamenti complessi tra neuroni e rafforza la robustezza della rete. Di conseguenza questa tecnica permette di avere due effetti principali:

1. **Forza la robustezza:** I neuroni non possono fare affidamento sulla presenza di altri neuroni specifici e sono costretti a imparare feature utili e robuste in modo indipendente;

2. **Simula un ensemble:** Ad ogni iterazione, si addestra una "sotto-rete" diversa. Il risultato finale è un'approssimazione efficiente dell'addestramento e della media di un numero esponenziale di reti diverse che condividono i pesi.

Pertanto il Dropout non è solo una forma di regolarizzazione numerica, ma impone una *robustezza strutturale*, in cui ogni neurone dev'essere **indipendentemente utile**. Questo argomento può essere approfondito in Srivastava et al. 2014 [80].

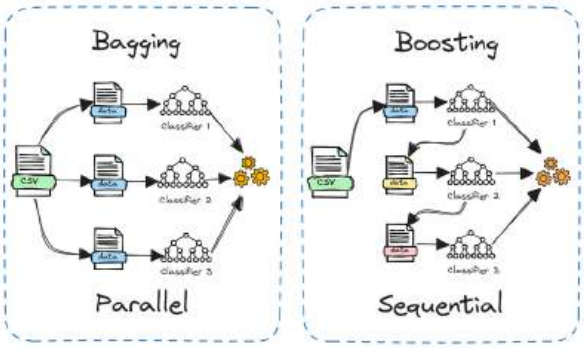


Figura 53: Rappresentazione dell'Ensemble tramite Bagging (modelli in parallelo) e Boosting (modelli in sequenza).

11.3 ENSEMBLING

Invece di cercare un singolo modello perfetto, le tecniche di ensemble combinano le previsioni di più modelli per ottenere un risultato più robusto e accurato. L'errore di un modello può essere scomposto in Bias (quanto le previsioni sono sistematicamente sbagliate) e Varianza (quanto le previsioni cambiano al variare del training set). L'ensemble è un modo potente per ridurre la varianza.

$$\text{Errore totale} = \text{Bias}^2 + \text{Varianza} + \text{Rumore}$$

Affinché un ensemble funzioni, i modelli che lo compongono devono essere diversi tra loro. La diversità può essere ottenuta addestrando modelli con architetture diverse, iperparametri diversi o su sottoinsiemi diversi dei dati. Le due strategie principali per creare ensemble sono:

- **Bagging:** Si addestrano più modelli indipendentemente su sottoinsiemi differenti del training set, creati tramite campionamento con rimpiazzo. Le previsioni vengono poi aggregate (e.g media o voto di maggioranza). È particolarmente efficace nel ridurre la varianza di modelli complessi (basso bias, alta varianza);

- **Boosting:** Si addestrano modelli in sequenza. Ogni nuovo modello si concentra sugli errori commessi dai modelli precedenti, dando più peso agli esempi classificati erroneamente. È ottimo per ridurre il bias di modelli semplici (alto bias, bassa varianza).

11.4 RIEPILOGO DELLE TECNICHE

Non esiste una singola tecnica valida per ogni problema. Nella pratica, le strategie per migliorare la generalizzazione vengono spesso combinate. Ad esempio, è comune usare Data Augmentation, Weight Decay e Dropout contemporaneamente all'interno di una rete neurale. La scelta e la calibrazione di queste tecniche sono una parte fondamentale del processo di sviluppo di un modello di successo.

Tabella 5: Strategie per migliorare la generalizzazione nei modelli di Deep Learning

Tecnica	Descrizione
Aumento dei dati	Espandere il dataset con esempi reali o sintetici
Architettura adeguata	Scegliere la capacità del modello con attenzione
Early Stopping	Interrompere l'addestramento in tempo utile
Weight Decay	Penalizzare pesi grandi nella funzione di loss
Noise Injection	Aggiungere rumore a input o attivazioni
Ensemble	Media di modelli diversi per ridurre la varianza
Bagging	Campionamento con rimpiazzo per ensemble
Boosting	Addestramento sequenziale di modelli deboli
Dropout	Spegnere neuroni casualmente evitando adattamenti

12 AUTOENCODER

Un **Autoencoder** [38], è una rete neurale progettata per apprendere una rappresentazione compressa dell'input, attraverso un processo di codifica e successiva decodifica, avente l'obiettivo di ricostruire fedelmente l'input ricevuto minimizzando la perdita di ricostruzione: $h_\theta(x) \approx x$. Sebbene questa formulazione possa risultare banale e apparentemente inutile, il valore dell'autoencoder risiede nella sua capacità di apprendere rappresentazioni utili dei dati, grazie a specifici vincoli architetturali. In particolare, l'apprendimento di una funzione identità approssimata, diventa un compito non banale, se si impongono vincoli strutturali quali:

1. **Compressione del livello nascosto:** La dimensionalità dello strato nascosto è inferiore rispetto a quella dello strato di input, costringendo la rete a catturare le caratteristiche più rilevanti dell'informazione;

2. **Sparsità:** Si introduce un vincolo per cui, durante la fase di training, solo un numero limitato di neuroni dello strato nascosto risulti attivo per ogni input. Ciò viene realizzato aggiungendo un termine di penalizzazione alla funzione di costo.

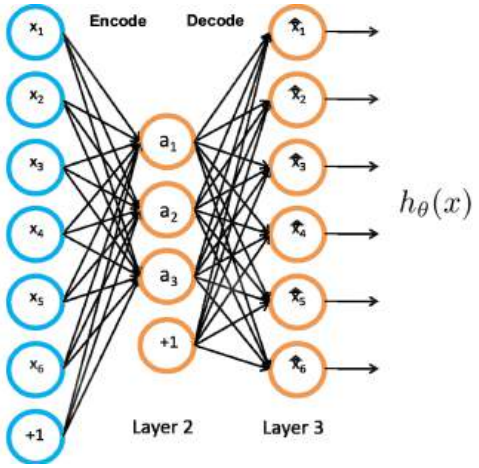


Figura 54: Architettura semplice composta da tre layer di un Autoencoder, evidene è la caratteristica della compressione del livello nascosto.

12.1 U-NET

Un'architettura che segue lo stesso principio di compressione e di successiva decompressione degli strati nascosti è quella delle **U-Net** [71], esse sono un'architettura di Deep Learning sviluppata per compiti di segmentazione semantica di immagini, con applicazioni rilevanti in ambito medico, come l'identificazione di lesioni o tumori. Esse sono state introdotte da Ronneberger et al., 2015, e nasce come variante convoluzionale dell'Autoencoder applicata alla segmentazione di immagini, generando una maschera al di sopra dell'input fornito.

Def: 12.1.1 *La segmentazione semantica è una tecnica di computer vision che assegna un'etichetta di classe (e.g "persona", "auto", "strada") a ogni singolo pixel di un'immagine, utilizzando algoritmi di Deep Learning.*

Pertanto l'U-Net risulta dedita a classificare ogni pixel dell'immagine in una determinata categoria, restituendone una mappa binaria o multiclasse. Un esempio è illustrato in Figura 55.

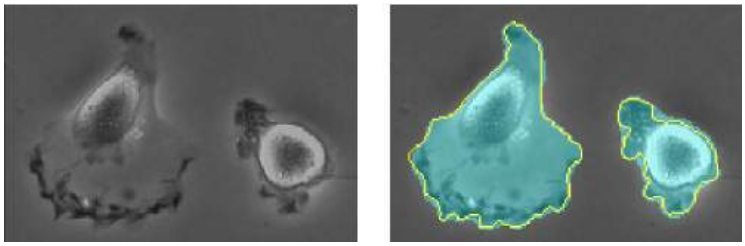


Figura 55: Esempio di segmentazione semantica: a sinistra l'immagine originale, a destra la mappa segmentata. I pixel evidenziati rappresentano la classe target.

L'architettura delle U-Net si discosta da quella degli Autoencoder per un componente in più, infatti essa è suddivisa in tre componenti principali: *Encoder*, *Decoder* e *Skip Connections*.

- **Encoder:** Composto da blocchi convoluzionali e operazioni di downsampling, estrae progressivamente feature sempre più astratte;
- **Decoder:** Costituito da operazioni di upsampling e convoluzioni, ricostruisce l'immagine nella sua risoluzione originale;
- **Skip Connections:** Collegano simmetricamente gli strati dell'encoder con quelli del decoder, trasmettendo direttamente informazioni spaziali dettagliate, utili alla ricostruzione dell'output.

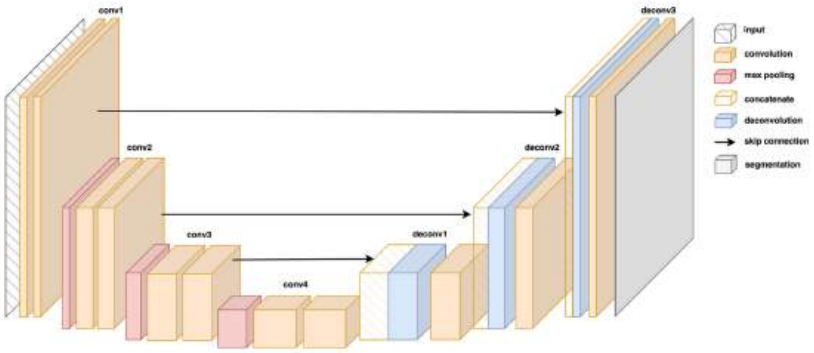


Figura 56: Architettura di una U-Net, la quale prende il nome dalla sua evidente forma ad U.

Durante la fase di downsampling, le informazioni spaziali precise tendono a degradarsi. Tuttavia, grazie alle Skip Connections, le caratteristiche di basso livello, conservate nei primi layer convoluzionali, vengono riutilizzate nel decoder per preservare la localizzazione spaziale dei dettagli. Questo migliora sensibilmente la qualità dell'output segmentato.

12.2 LIMITAZIONI DEGLI AUTOENCODER

Gli Autoencoder sono efficaci per compiti di compressione, riduzione del rumore e apprendimento non supervisionato. Tuttavia, presentano alcune limitazioni nel contesto generativo, in particolare, lo spazio latente appreso, non risulta essere strutturato per poter garantire la generazione di nuovi esempi coerenti.

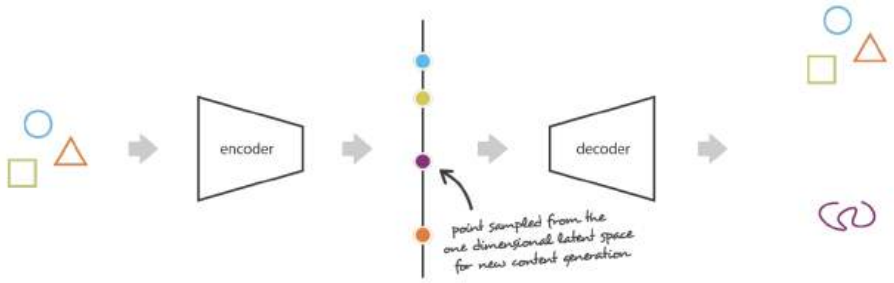


Figura 57: Distribuzione irregolare nello spazio latente: regioni inutilizzate o contenenti rumore

L'encoder può infatti mappare gli input in regioni isolate dello spazio latente, lasciando vaste aree vuote. Ciò implica che, campionando casualmente un punto in questo spazio, è probabile ottenere un output

privo di significato, non vi è in più garanzia che interpolazioni lineari tra due punti "validi" generino anch'esse esempi coerenti, dunque ricadiamo in due principali problematiche una legata alla **continuità** e una alla **completezza**.

12.3 VARIATIONAL AUTOENCODER (VAE)

Il **Variational Autoencoder** (VAE) è un'estensione probabilistica dell'autoencoder classico. Esso introduce una regolarizzazione nello spazio latente per consentire una generazione di dati coerente e continua. A differenza degli autoencoder tradizionali, i VAE non codificano l'input come un singolo punto nello spazio latente, ma come una distribuzione di probabilità, da cui poi successivamente campionare e ricostruire. Il processo di training, pertanto coinvolge i seguenti passaggi:

1. L'Encoder mappa l'input in una distribuzione probabilistica appresa dai dati, in uno spazio latente (tipicamente gaussiana);
2. Viene campionata una variabile latente da questa distribuzione:
$$z \sim p(z) = \mathcal{N}(\mu_x, \sigma_x)$$
3. Si calcola la Kullback Leibler Divergence per aggiungere un termine di regolarizzazione alla funzione di costo fra la distribuzione dello spazio latente e la distribuzione normale:
$$\mathrm{KL}[\mathcal{N}(\mu_x, \sigma_x), \mathcal{N}(0, I)]$$
4. Il Decoder genera un dato x a partire da z augurandosi che ottenga il valore iniziale:
$$x \sim p_\theta(x|z)$$
5. L'errore di ricostruzione successivamente viene propagato all'indietro.

La funzione di costo del VAE dunque combina due componenti, uno è il termine di ricostruzione che penalizza la differenza tra input e output, l'altro è un termine di regolarizzazione, costituito dalla divergenza di Kullback-Leibler tra la distribuzione appresa e una distribuzione normale standard.

$$L = \|x - \hat{x}\|^2 + \mathrm{KL}[\mathcal{N}(\mu_x, \sigma_x)\,, \mathcal{N}(0, \mathrm{I})] =$$
$$= \|x - \mathrm{D}(z)\|^2 + \mathrm{KL}[\mathcal{N}(\mu_x, \sigma_x)\,, \mathcal{N}(0, \mathrm{I})]$$

Dove il passaggio matematico da $\hat{x}$ a $\mathrm{D}(z)$, vuole rappresentare semplicemente che la stima è stata ottenuta sottoponendo z al Decoder della nostra architettura.

12.3.1 Proprietà desiderate dello spazio latente

Un buono spazio latente per definirsi tale deve soddisfare ben due proprietà fondamentali, in modo tale da garantire una corretta generazione dei dati:

- **Continuità:** Piccole variazioni nel latent space devono produrre output simili;
- **Completezza:** Qualsiasi punto dello spazio latente deve decodificarsi in un output plausibile.

Tali proprietà non sono garantite automaticamente poiché si incorre in diverse difficolta, le principali includono le seguenti:

- **Encoding inadeguato:** Distribuzioni malformate o disgiunte non assicurano continuità o completezza;
- **Mancata regolarizzazione:** Il VAE può comportarsi come un autoencoder classico;
- **Distribuzioni con varianza trascurabile:** Rendono l'encoding troppo deterministico;
- **Regolarizzazione completa:** È necessaria una regolarizzazione sia della media che della covarianza.

12.4 VISIONE PROBABILISTICA DEI VAE

Per poter comprendere al meglio perché si giunge all'utilizzo della Kullback Leibler Divergence, è opportuno analizzare la visione probabilistica del modello, consideriamo x un input e z una variabile latente.

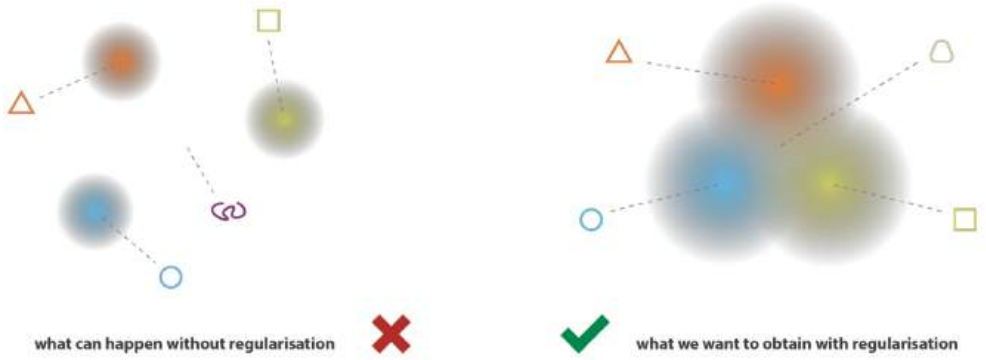


Figura 58: Effetti della regolarizzazione all'interno dello spazio latente, con la mancanza di essa (a sinistra) e la presenza della stessa (a destra).

L'obiettivo del VAE è apprendere un modello generativo che consenta di generare nuovi campioni x campionando da una distribuzione a priori su z, tipicamente $\mathcal{N}(0, I)$. Ci sono varie probabilità che entrano in gioco, ognuna rappresentativa di una caratteristica specifica:

- $p(x) \rightarrow$ Probabilità di osservare x rispetto a tutti i possibili parametri;
- $p(z) \rightarrow$ Distribuzione nello spazio latente;
- $p(x|z) \rightarrow$ Decoder probabilistico, da cui vengono campionati i valori di x;
- $p(z|x) \rightarrow$ Encoder probabilistico.

La distribuzione di probabilità $p(x)$ risulta essere la più complicata da calcolare, poiché cerca all'interno della distribuzione di dati in input, i quali possono rappresentare qualsiasi cosa. Tutte queste distribuzioni di probabilità possone essere facilmente essere interconnesse fra loro grazie al Teorema di Bayes:

$$p(z|x) = \frac{p(x|z)\, p(z)}{p(x)}$$

L'elemento più difficoltoso da calcolare è proprio l'elemento presente al denominatore, in quanto come già anticipato prima, porterebbe a calcolare un integrale molto complesso:

$$\log p(x) = \log \int p(x|u)\, p(u)\, du$$

12.4.1 Inferenza variazionale

Per affrontare questa difficoltà, si ricorre all'**Inferenza Variazionale**, effettuando delle assunzioni e poi creando una distribuzione $q_x(z)$ scelta all'interno di una famiglia parametrica (e.g gaussiana), portandola ad essere l'approssimazione della distribuzione di probabilità $p(z|x)$. Le due assunzioni sono le seguenti:

- $p(z) = \mathcal{N}(0, I)$
- $p(x|z) = \mathcal{N}(f(z), cI)$ con $f \in F$, una generica famiglia di funzioni e $c > 0$

La distribuzione $q_x(z)$ è costituita da due funzioni $g(x)$ e $h(x)$, rispettivamente per la media e per la varianza della distribuzione gaussiana. Il mio obbiettivo diventa minimizzare la KL Divergence fra questa approssimazione e la funzione $p(z|x)$, così da trovare la migliore approssimazione delle due funzioni g^*, h^*, che mi permettino di ottenere una buona distribuzione approssimata. Seguendo questa strategia seguiremo dunque i passaggi qui di seguito:

$$
\begin{aligned}
(g^*, h^*) &= \arg\min \left(\mathrm{KL}(q_x(z), p(z|x)\right) = \\
&= \arg\min \left(\mathbb{E}_{z\sim q_x}(\log q_x(z)) - \mathbb{E}_{z\sim q_x}\left(\log \frac{p(x|z)p(z)}{p_(x)}\right)\right) = \\
&= \arg\min \left(\mathbb{E}_{z\sim q_x}(\log q_x(z)) - \mathbb{E}_{z\sim q_x}(\log p(z)) \right. \\
&\quad \left. -\mathbb{E}_{z\sim q_x}(\log p(x|z)) + \mathbb{E}_{z\sim q_x}(\log p(x))\right) = \\
&= \arg\max \left(\mathbb{E}_{z\sim q_x}(\log p(x|z) - \mathrm{KL}(q_x(z), p(z))\right) = \\
&= \arg\max \left(\mathbb{E}_{z\sim q_x}\left[-\frac{\|x - f(z)\|^2}{2c}\right] - \mathrm{KL}(q_x(z), p(z))\right)
\end{aligned}
$$

Ciò che avviene nei vari passaggi, è un'esplicitazione e compattamento della KL Divergence, uno spezzettamento della funzione logaritmo, seguendo le sue proprietà, per poi invertire il segno e dunque passare da un problema di minimizzazione a uno di massimizzazione. Questo problema di ottimizzazione alla fine diventa un'ottimizzazione a tre parametri, poiché dovremmo trovare anche il valore ottimale di f^*.

12.4.2 Reparametrization Trick

Il campionamento da $q_x(z) = \mathcal{N}(\mu_x, \sigma_x)$ non risulta essere differenziabile poiché casuale (sebbene non completamente), impedendo così

l'utilizzo diretto della retropropagazione. Esiste per fortuna, un piccolo trucco chiamato **Reparametrization Trick**, il quale permette di arginare questa problematica: invece di campionare direttamente, effettuo un campionamento di un vettore $\zeta \sim \mathcal{N}(0, I)$ e imposterò $z = \mu_x + \sigma_x \odot \zeta$. Permettendo di far ricadere la casualità sul nostro parametro e non nelle funzioni parametrizzate in se, permettendomi di effettuare la Backpropagation senza alcun tipo di problema.

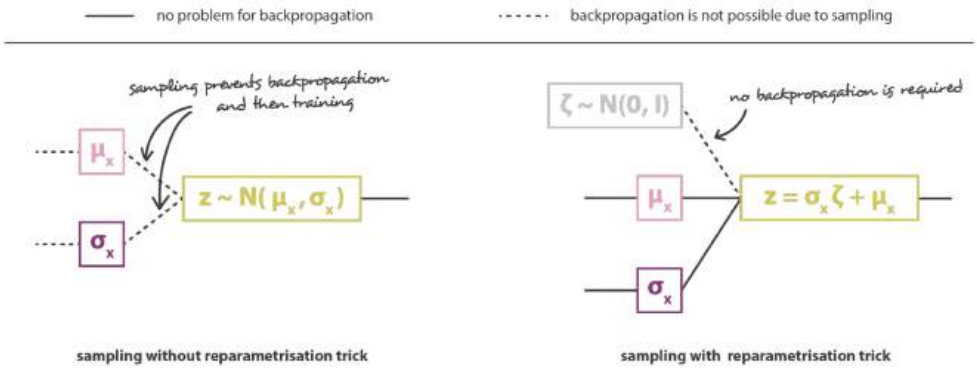


Figura 59: Effetto del reparametrization trick nel rendere il campionamento differenziabile.

13 | GAN

Le **Generative Adversarial Networks** (GAN), introdotte da *Ian Goodfellow* nel 2014 [35], rappresentano una tecnica nel campo dell'Intelligenza Artificiale diversa dalle altre. Si tratta di una classe di reti neurali progettate per generare dati nuovi e realistici, a partire da esempi osservati. Alla base della generazione di dati realistici c'è sempre un processo di trasformazione di semplici variabili casuali (spesso uniformi) in variabili complesse. Nonostante i calcolatori siano sistemi **deterministici** (dato un input, producono sempre lo stesso output), è comunque possibile costruire algoritmi capaci di generare sequenze numeriche che si comportano similmente a sequenze casuali. A partire da questa base, esistono diverse tecniche per ottenere variabili casuali con distribuzioni sempre più sofisticate. Tutti questi metodi sfruttano "trucchi" matematici diversi, ma condividono un'idea comune: ottenere variabili casuali complesse come risultato di una trasformazione applicata a variabili semplici.

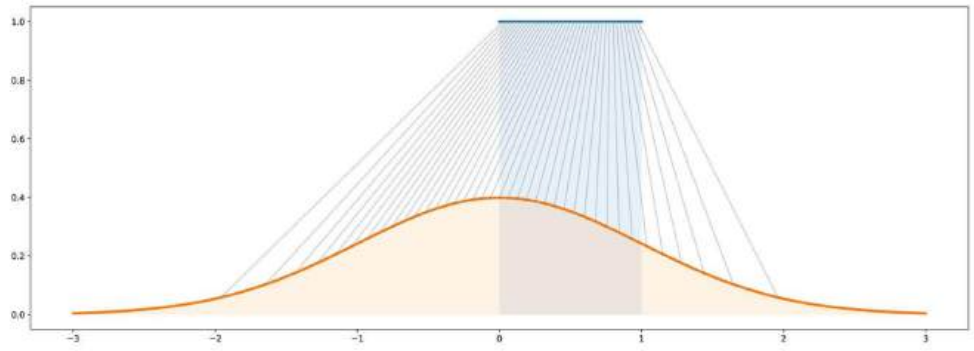


Figura 60: In blu, la distribuzione uniforme su $[0, 1]$ mentre in arancione, una distribuzione gaussiana standard, in grigio, le linee che mostrano la mappatura dalla distribuzione uniforme a quella gaussiana.

13.1 LA TRASFORMAZIONE INVERSA

Una tecnica semplice è il metodo della **Trasformazione Inversa**. Supponiamo di voler generare una variabile casuale X che segua una

distribuzione con funzione di distribuzione cumulativa $F_X(x)$. L'idea è partire da una variabile casuale u distribuita uniformemente in $\mathcal{U}(0,1)$, e ottenere x tramite:

$$x = F_X^{-1}(u)$$

In questo modo, x sarà distribuito secondo la legge desiderata. L'idea si può estendere anche a funzioni di trasformazione generali, che mappano variabili semplici (non necessariamente uniformi) in variabili con una distribuzione target.

13.2 MODELLI GENERATIVI

Se volessimo generare immagini in bianco e nero di cani, con dimensione $n \times n$ pixel. Ogni immagine può essere "linearizzata" in un vettore N di lunghezza n^2, mettendo le colonne una sopra l'altra. In questo modo, ogni immagine può essere rappresentata da un punto nello spazio vettoriale $\mathbb{R}^N$. Una cosa a cui bisogna stare attenti, è che non tutti i vettori in $\mathbb{R}^N$ rappresentano cani, infatti solo una piccola regione di questo spazio contiene vettori che corrispondono a immagini di cani. La distribuzione che vogliamo seguire la chiamiamo "distribuzione dei cani", ma come già detto è una distribuzione molto complessa in uno spazio notevolmente più grande, e non sappiamo ovviamente come esprimerla esplicitamente, pertanto il nostro obbiettivo di generare elementi che seguino questa distribuzione, diventa di una complessità notevole. Per rendere più semplice questo compito si è pensato di far riferimento ai campioni, delle immagini vere e di quelle generate, ottimizzando il nostro modello per riduerre la distanza dei campioni di volta in volta, facendo sì che le immagini si somiglino fra di loro. Seguendo pressocché gli stessi passaggi effettuati da un modello generativo:

1. Generare input casuali da una distribuzione semplice (es. uniforme);
2. Passare questi input attraverso una rete neurale generativa;
3. Confrontare i campioni generati con quelli reali;
4. Usare la retropropagazione per migliorare il modello e ridurre la distanza tra le due distribuzioni.

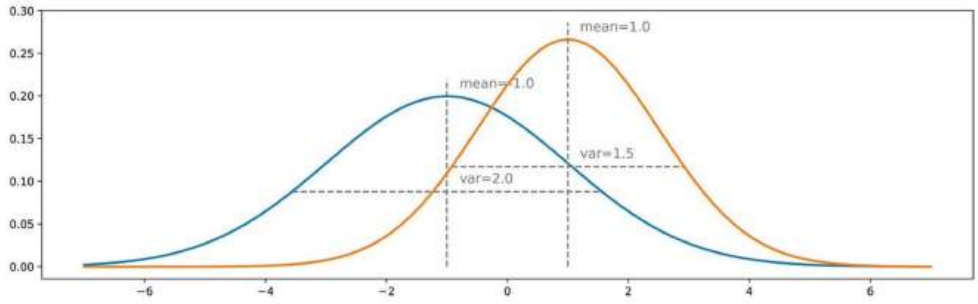


Figura 61: Grafico che mostra come la distribuzione reale (in blu) e la distribuzione generata (in arancione), in un momento intermedio dell'allenamento del generatore, le quali con il passare del tempo tenderanno a sovrapporsi.

13.3 GAN

Le **GAN** (Generative Adversarial Networks) [35], sono una potente architettura generativa che adotta una strategia diversa: invece di confrontare direttamente le distribuzioni o i singoli campioni, il modello impara attraverso un compito indiretto. L'allenamento forza la distribuzione generata ad avvicinarsi sempre più alla distribuzione reale dei dati, questo avviene utilizzando un task di discriminazione dei valori generati e dei valori reali, l'obbiettivo cardine è che questa discriminazione fallisca il più possibile, così che le due distribuzioni ottenute risultino praticamente uguali. Per effettuare la generazione e il task di discriminazione delle distribuzioni, avviene un addestramento di due reti neurali in competizione tra loro, dette una *Generatore* e *Discriminatore*.

13.3.1 Metodo diretto

Il metodo chiamato *diretto* si basa sull'aggiustare iterativamente il generatore, tramite le iterazioni della discesa del gradiente, in modo tale da correggere la differenza misurata degli errori fra la distribuzione generata e quella dei valori reali. Ovviamente il processo di ottimizzazione (Figura 61), si conclude con la distribuzione generata che aderisce perfettamente a quella dei valori reali.

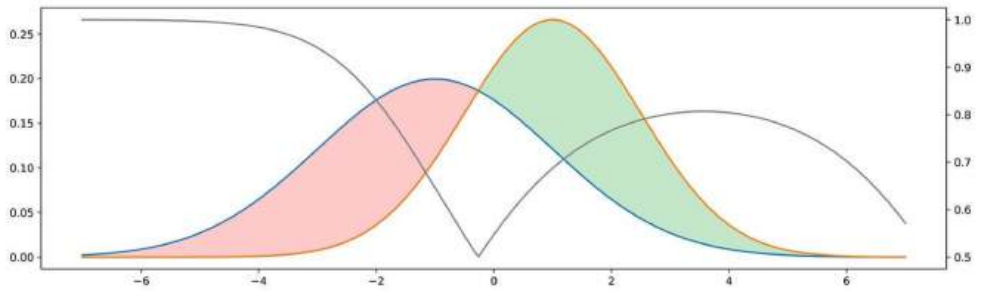


Figura 62: Grafico che mostra la distribuzione dei dati reali (in blu) e dei dati generati (in arancione), e l'andamento del discriminatore (in grigio), il quale segue la scala presente alla sinistra del grafico, in cui si vede che nei punti di sovrapposizione dei due grafici si giunge a un 50% di probabilità.

13.3.2 Metodo indiretto

Per utilizzare un approccio *indiretto*, invece si considera anche un discriminatore, il quale assumiamo per il momento come se sia un oracolo il quale conosce esattamente la distribuzione reale e a quella generata, e grazie a questa informazione sia in grado di determinare l'appartenenza a una data classe per ogni punto fornito in input. Se le due distribuzioni sono distanti fra loro, il discriminatore sarà in grado di determinare facilmente la differenza fra le due, con un livello di confidenza molto alto su la maggior parte dei punti che li forniamo. Se volessimo ingannare il discriminatore, dobbiamo fare in modo che la distribuzione generata risulti essere più vicina a quella reale, così che il discriminatore abbia più difficoltà nel predirre la classe di appartenenza nel momento in cui le due distribuzioni sono uguali in tutti i punti. Pertanto ci sarà una equa probabilità in ogni punto che appartengano alla distribuzione reale o quella dei dati generati, e dunque il discriminatore non potrà fare altro che centrare la previsione il 50% delle volte. A questo punto, può esser legittimo chiedersi se questo metodo indiretto sia davvero una buona idea visto che sembra essere più complicato e richiede un discriminatore che qui consideriamo come un oracolo dato, ma che, in realtà, non è né noto né perfetto. In realtà la difficoltà di confrontare direttamente due distribuzioni di probabilità basate sui campioni, controbilancia l'apparente maggiore complessità del metodo indiretto. Mentre il discriminatore sebbene non sia noto, può essere facilmente appreso.

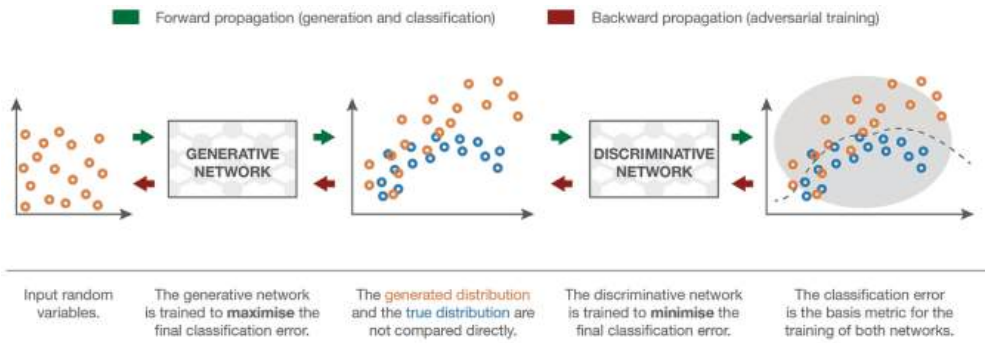


Figura 63: Flusso di lavoro di una GAN. L'addestramento si basa su un gioco a somma zero: il Generatore prende variabili casuali in input e produce dati falsi (arancioni), mirando a ingannare il Discriminatore. Il Discriminatore riceve sia dati reali (blu) che generati, e viene addestrato a minimizzare l'errore di classificazione, separando le due distribuzioni (linea tratteggiata).

13.3.3 I due modelli

Nella creazione di questi due modelli, il generatore prende una variabile casuale z a partire da una distribuzione uniforme $\mathcal{U}$ o normale $\mathcal{N}$, viene mandata come input al generatore e otterremo $x = G(z)$ dove il valore dell'output x sarà un elemento nella distribuzione di probabilità dei dati generati che seguirà la distribuzione dei dati desiderata $p_g(x) \approx p_{data}(x)$. Per quanto riguarda il discriminatore, ovviamente non sarà un oracolo in grado di distinguere a priori i dati d'ingresso, ma riceverà come input un dato o appartenente alla distribuzione generata, o alla distribuzione reale z e restituirà, un valore appartenente all'intervallo $[0, 1]$, dove se il valore si avvicina a 0 sarà un elemento appartenente alla distribuzione generata, mentre se si avvicina a 1 a quella dei dai reali. I due modelli a ogni iterazione portano ad aggiornare i pesi di volta in volta per seguire gli obbiettivi seguenti:

- **Generatore:** Massimizzare l'errore di classificazione, in modo tale da riuscire ingannare per bene il discriminatore;
- **Discriminatore:** Minimizzare l'errore di classificazione, così da distinguere per bene i dati reali, da quelli ottenuti dal generatore;

13.3.4 Visione probabilistica

Il Discriminatore è un classificatore binario il quale si occupa di determinare se l'input ricevuto sia reale, ricevuto dalla distribuzione reale,

o falso, ricevuto dalla distribuzione generata. Assumendo che la label y per i dati veri sia pari a 1 e 0 per i dati generati. Alleneremo il discriminatore a minimizzare la Binary Cross Entropy Loss seguente:

$$\min_D \{-y \log(D(x)) - (1-y) \log(1 - D(x))\}$$

Dove chiaramente se stiamo trattando con i dati reali, rimarrà soltanto la prima parte e nel caso in cui il Discriminatore determina che stiamo analizzando un dato reale $D(x) \approx 1$ e avrò una bassa perdità poiché il logaritmo di 1 è zero, differentemente avrò un'alta perdita. Nel caso in cui stiamo trattando i dati falsi (generati), rimarrà solo la seconda parte con ovviamente una bassa perdita se $D(x) \approx 0$, sennò avremo un'alta perdita. Per il generatore invece, si prende in input una variabile latente z da una sorgente di numeri pseudo-casuali, ad esempio, un campione da una distribuzione normale. Successivamente applichiamo la funzione per generare un'uscita $x' = G(z)$, che dovrebbe assomigliare a un dato reale, poiché il nostro obbiettivo è ingannare il Discriminatore per ottenere $D(G(z)) \approx 1$. In altre parole, dato un Discriminatore aggiorniamo i parametri del Generatore per massimizzare la Binary Cross Entropy Loss quando $y = 0$:

$$\max_G \{-(1-y) \log(1 - D(G(z)))\} = \max_G \{-\log(1 - D(G(z)))\}$$

Tuttavia vi è un problema non ignorabile, legato al fatto che nelle prime fasi il Discriminatore risulti essere molto bravo, portando a non permettere al Generatore di imparare poiché porta a saturazione, pertanto Goodfellow [35], fa notare come se optiamo per quest'altra ottimizzazione non incorreremo nello stesso problema:

$$\max_G \{-\log(D(G(z)))\}$$

Infine l'intero processo può essere descritto come un problema min-max, in cui il discriminatore e il generatore si sfidano a vicenda:

$$\min_D \max_G \{-\mathbb{E}_{x \sim Real}(\log D(x)) - \mathbb{E}_{z \sim Fake}(\log(1 - D(G(z))))\}$$

Questa formulazione rappresenta il cuore delle GAN: uno sport con due giocatori che si allenano fra loro, fino a raggiungere un equilibrio in cui il generatore produce dati indistinguibili da quelli reali, detto *Equilibrio di Nash* [56].

14 | TRANSFORMER

I **Transformer** sono un'architettura di rete neurale introdotta nel paper *"Attention Is All You Need", Vaswani et al., 2017* [90]. Essa è diventata lo standard per poter affrontare problemi relativi alle sequenze, come la traduzione automatica, o problemi di visione artificiale, tutto questo grazie alla sua efficienza e capcaità di modellare realzioni a lungo termine. I Transformer si basano quasi esclusivamente su *Meccanismi di Attenzione*, eliminando l'utilizzo tradizionale di CNN o RNN.

14.1 PROBLEMI CON RNN E CNN

Nei capitoli precedenti abbiamo analizzato le Reti Convoluzionali e le Reti Ricorrenti, notando come riscontrassero delle criticità in diversi aspetti:

- **RNN**: Processamento di dati in maniera sequenziale, paralellizzazione dei processi complicata, scomparsa o esplosione del gradiente su sequenze eccessivamente lunghe;
- **CNN**: Inefficienti nel momento in cui si vogliono catturare delle dipendenze molto lunghe, con la necessità di aumentarne la profondità.

Una delle soluzioni proposte è stata quella di utilizzare il **Meccanismo dell'Attenzione** per modellare direttamente tutte le dipendenze in una singola sequenza, a prescindere dalla distanza, proprio questa implementazione è possibile ritrovarla nei Transformer.

14.2 VISIONE AD ALTO LIVELLO

Architetturalmente, i Transformer sono composti da due grandi blocchi distinti:

- **Encoder:** una pila di **N strati** identici fra loro, nel paper ufficiale ne sono presenti 6, con dei sottolivelli;

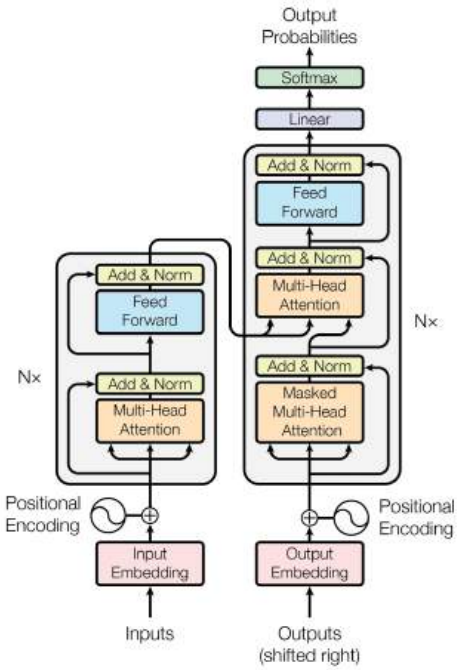


Figura 64: La struttura Encoder-Decoder dell'architettura di un Transformer.

- **Decoder:** una pila di **N strati** identici fra loro, e anche in questo caso nel paper ufficiale ne sono presenti 6, specificando come il loro numero è strettamente collegato a quello degli encoder, strutturati anch'essi in più sottolivelli.

Queste due macrocomponenti, non condividono i pesi fra loro, analiziamo ora partendo dagli Encoder i loro sottostrati e come si relazionano nel modello complessivo:

1. **Self-Attention Layer:** permette a ciascun token di "guardare" tutti gli altri token dell'input per poter raccogliere il contesto;

2. **Feed-Forward Neural Network:** una piccola rete Fully-Connected applicata in modo identico e indipendente a ogni posizione.

I Decoder, sono composti invece da entrambi gli stati presenti negli Encoder, ma con l'aggiunta di uno strato ulteriore, vediamoli:

1. **Masked Self-Attention:** come sopra, ma mascherata, in modo tale da non utilizzare informazioni future durante la generazione;

2. **Encoder-Decoder (Cross) Attention:** "aggancia" l'output dell'Encoder, così da potersi concentrare sulle parti più rilevanti dell'input;

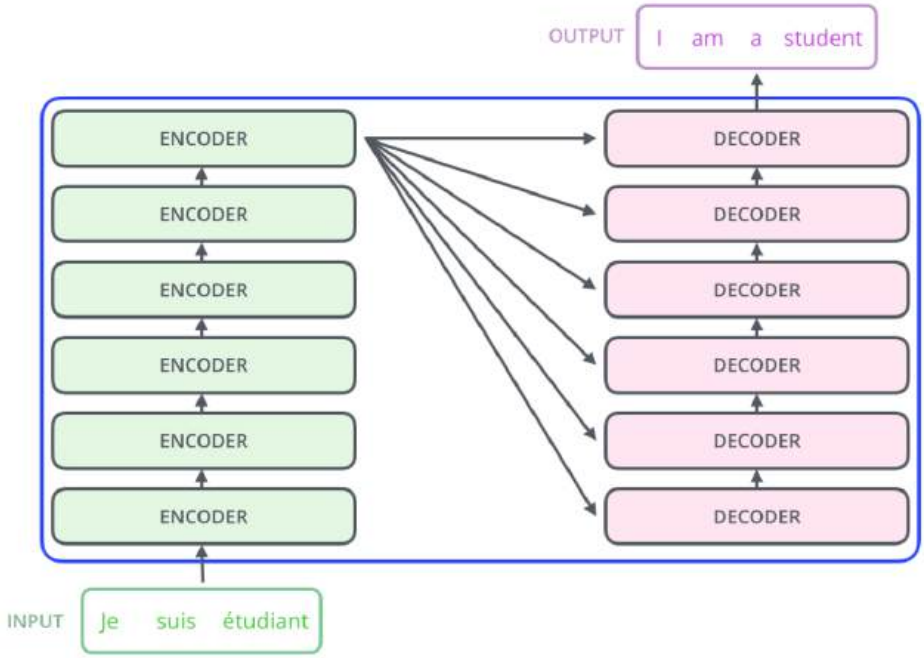


Figura 65: Rappresentazione ad alto livello di un'architettura di un Transformer, focalizzandosi principalmente sulla parte in cui vi sono gli Encoder e i Decoder.

3. **Feed-Forward:** identico per posizione, come nell'encoder.

I pesi, come già detto, non sono condivisi tra Encoder e Decoder. Ogni sottolivello è avvolto da *Connessioni Residue* e *Layer Normalization* (che approfondiremo a breve). Mentre per quanto riguarda le rappresentazioni in ingresso agli stack esse vengono arricchite con un *Positional Encoding* in modo da codificare l'ordine dei token.

14.3 TENSORI

Nell'ambito del Deep Learning un **Tensore** viene considerato come un array multidimensionale, in realtà il Tensore ha una spiegazione teorica molto più profonda, basata sull'algebra astratta, ma noi ci serviremo semplicemente di questa semplice astrazione, per cui ogni numero risulta essere un tensore, di diversa dimensione. Uno scalare è considerabile come un tensore a zero dimensioni, un vettore come un tensore a una dimensione, una matrice un tensore a due dimensioni e se dovessimo pensare a più dimensioni considereremo più matrici messe sopra l'altra, quindi maniere per inglobare quelli allo stato precedente

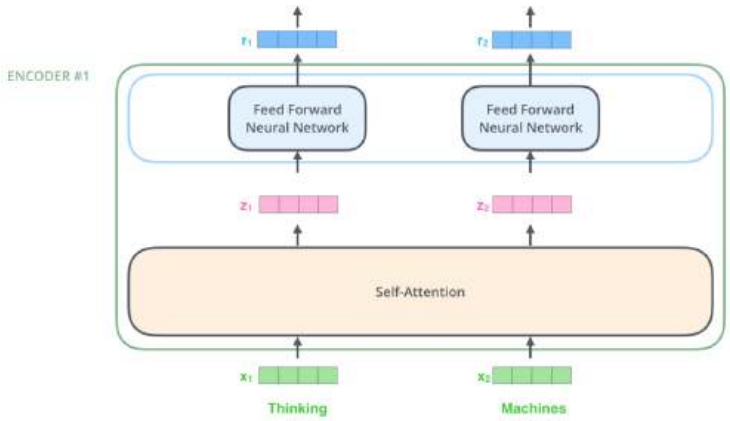


Figura 66: Rappresentazione interna di un Encoder, in cui nel primo strato di Self Attention, gli embedding vengono inseriti e creano delle relazioni fra loro, una volta generati gli output, questi andranno separatamente all'interno dello strato Feed-Forward, seguendo ognuno un loro percorso.

come un unico elemento. Nel contesto applicativo in cui ci troviamo, ogni parola fornita come input viene trasformata in un vettore (tensori monodimensionali) di dimensione 512 tramite un *Algoritmo di Embedding*. La trasformazione in Embedding avviene prima dell'ingresso nel primo Encoder, la lunghezza della frase, risulterà essere un *iperparametro*, impostabile durante la progettazione. Ogni parola seguirà un suo percorso indipendente all'interno dell'Encoder, e verranno create delle relazioni fra questi percorsi tramite il Meccanismo dell'Attenzione. Nel layer di Feed-Forward invece, non ci sono dipendenze quindi sarà permessa l'esecuzione parallela, aumentandone l'efficienza (Figura 66).

14.4 SELF-ATTENTION

Entriamo finalmente nel dettaglio per approfondire il *Meccanismo dell'Attenzione*, questo meccanismo è l'effettiva rivoluzione che viene proposta da questa architettura [90], permettendo a ogni parola di ponderare la rilevanza delle altre in una frase. Considerando una frase con un soggetto sottointesto, per noi umani risulta semplice determinarne chi sia il soggetto, diversamente per una macchina questo risulta un meccanismo complesso. Grazie al **Self-Attention Mechanism**, questo problema viene aggirato, potendo collegare i vari token presenti in una frase e determinando le loro relazioni (Figura 67).

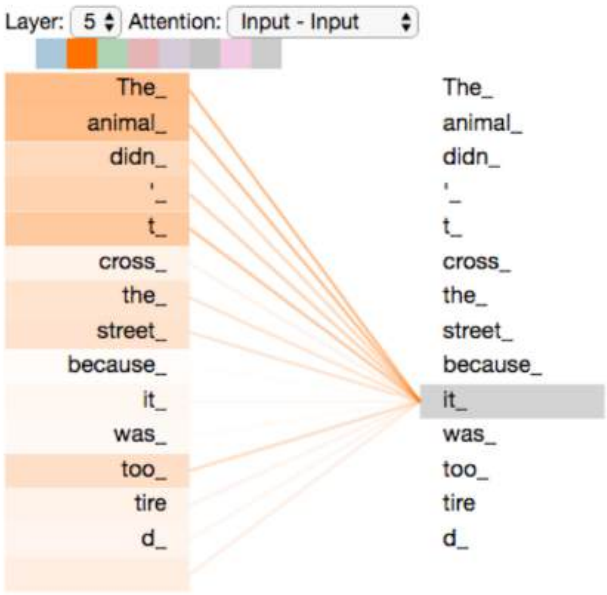


Figura 67: Nell'immagine possiamo vedere le singole relazioni, rappresentate dalle linee di connessione del token it_ con gli altri token della frase, più spessa è la linea, più solida risulta essere la relazione.

14.4.1 Self-Attention in dettaglio

La prima fase per il calcolo della **Self-Attention** comincia dalla creazione di tre vettori a partire da ogni vettore di Embedding mandato in input agli Encoder. Vengono generati quindi: un vettore Query (Q), un vettore Key (K) e infine un vettore Value (V). Ogni token ha un suo Embedding associato E, il quale viene moltiplicato per tre matici differenti apprese durante il training: W^Q, W^K, W^V. Queste matrici vengono inizializzate a valori piccoli, e ci permettono di ottenere rispettivamente il Query vector, il Key vector e il Value vector.

$$Q = E \times W^Q, \qquad K = E \times W^K, \qquad V = E \times W^V$$

Una volta ottenuti i tre vettori Q, K e V, si procede calcolando il grado di affinità tra ogni query Q e tutte le chiavi K, moltiplicando le singole Query per tutte le Key. Il risultato di questi confronti viene normalizzato dividendo per la radice quadrata della dimensione del vettore $\sqrt{d_k}$ (nel paper si divide per 8), e poi viene applicata la funzione SoftMax fornendoci il grado di compatibilità fra una Query e tutte le Key. Ovviamente la Query con la sua Key corrispondente ci darà lo score, maggiore, proprio per questo solitamente, si va a scalare verso la seconda. Dopo questi passaggi entra in gioco il vettore dei Value V il quale andrà a moltiplicarsi con l'esito di queste operazioni, e otterrò

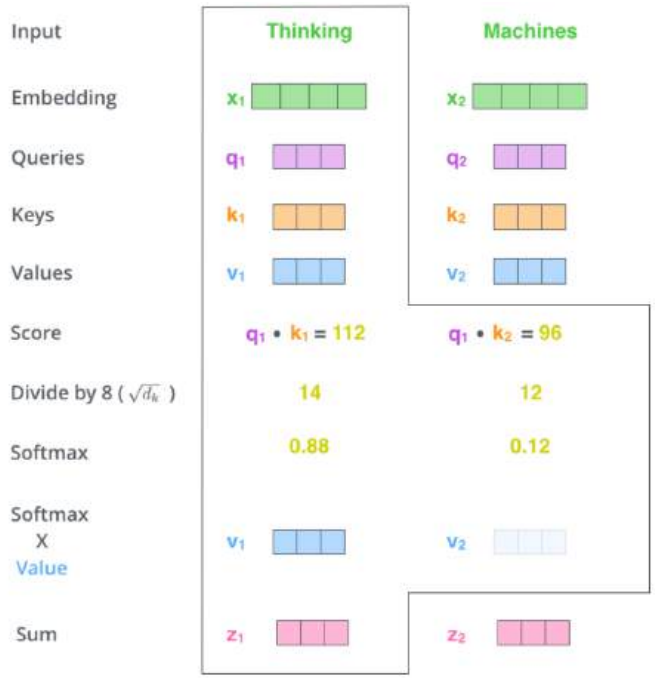


Figura 68: Rappresentazione visiva dei calcoli effettuati per ottenere lo score dell'attenzione a partire dagli Embedding iniziali.

i singoli valori pesati a seconda dello score ottenuto dalla funzione SoftMax, per poi sommarli fra loro, ottenendone il risultato da mandare in ingresso al layer di Feed-Forward (Figura 68), l'intero processo viene sintetizzato dalla seguente equazione:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q\,K^T}{\sqrt{d_k}}\right) V$$

L'analogia delle cartelle

Supponiamo di avere un armadietto, al quale interno ci sono numerose cartelle ognuna con un identificativo, il nostro obbiettivo è trovare la cartella più affine al post-it che abbiamo in mano (Figura 69). Ogni cartella viene presa e analizzata in base alla corrispondenza fra ciò che c'è scritto sul post-it e l'identificativo della cartella stessa, al suo interno ci sono dei fogli con delle informazioni, alla fine della mia ricerca prenderò in considerazione maggiore i fogli presenti nelle cartelle che erano più affini con quanto scritto sul post-it. Questa analogia ci permette di semplificare il ragionamento, i post-it sono le Queries, le cartelle le Keys, mentre i fogli all'interno delle cartelle sono i Values.

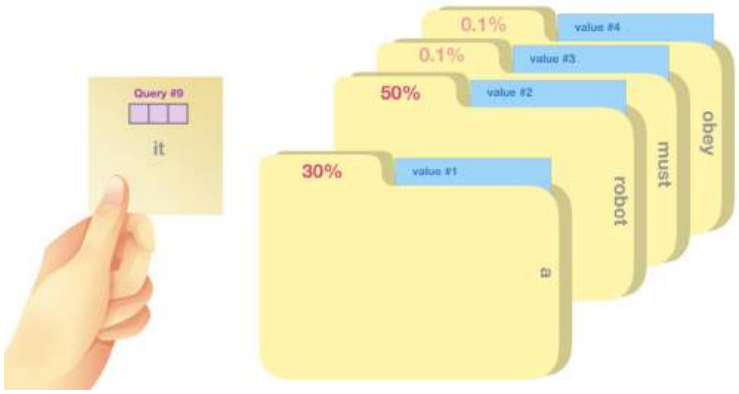


Figura 69: Rappresentazione dell'analogia delle cartelle, nel quale si rappresentano i tre vettori, query vector come il post-it, key vector come l'identificativo di ogni cartella, value vector come il valore all'interno di ogni cartella, ogni cartella ha un punteggio che mappa la percentuale di corrispondenza.

14.5 MULTI-HEAD ATTENTION

Il meccanismo di Self-Attention può essere raffinato ulteriormente introducendo la **Multi-Head Attention**, una tecnica in grado di migliorare l'efficacia dell'attenzione secondo due principali direttrici:

1. Ampliare la capacità del modello di focalizzarsi su posizioni differenti nella sequenza di input. Ad esempio, il vettore z_1 associato alla prima parola potrebbe rappresentare un misto di tutte le codifiche, ma al contempo essere fortemente influenzato dalla parola stessa;

2. Introduce molteplici *sottospazi di rappresentazione* all'interno dello stesso livello di attenzione. Invece di utilizzare un singolo insieme di matrici di peso per Q, K e V, si impiegano più insiemi distinti, ciascuno inizializzato in modo indipendente. Al termine dell'addestramento, ogni insieme proietta gli embedding d'ingresso in uno spazio latente diverso, catturando così vari aspetti delle relazioni tra parole.

Questo meccanismo introduce diverse **Teste di Attenzione** (*Heads*), ognuna delle quali applica il calcolo di attenzione in maniera indipendente, utilizzando parametri distinti. Ogni testa analizza l'input da una prospettiva differente, permettendo al modello di cogliere vari tipi di relazioni semantiche e sintattiche tra le parole. Alla fine, i vettori prodotti da ciascuna testa (z_i) vengono concatenati e successivamente

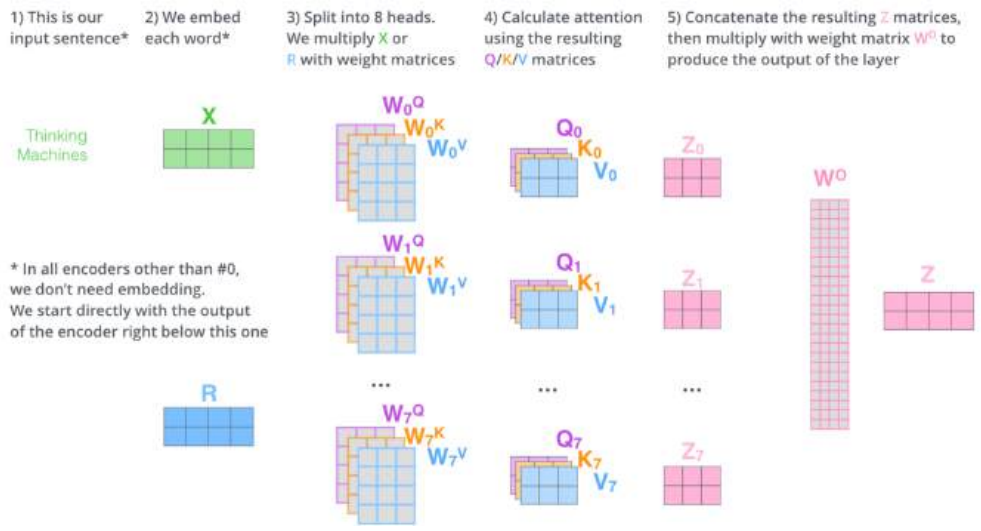


Figura 70: Rappresentazione grafica del meccanismo della Multi Head Attention, mettendo in luce tutta la procedura distinta nelle singole teste di attenzione, che culmina nella concatenazione dei singoli esiti per poi moltiplicarli per la matrice W_o

proiettati in uno spazio comune tramite una matrice di pesi addizionale, W_o, appresa durante l'addestramento. Questo passaggio ha lo scopo di restituire un singolo vettore di output da fornire al livello Feed-Forward della rete (Figura 70).

Esempio

Per poter semplificare ulteriormente il meccanismo della Multi Head Attention, andiamo a considerare un esempio, tramite la frase seguente:

The animal didn't cross the street because it was too tired.

In questa frase, notiamo come siano presenti diverse relazioni, focalizzandoci sulla parola *it*, troviamo relazioni di natura diversa con le altre parole, a partire dalla parola *animal*, di cui it è il suo sostituto, la parola *cross*, che descrive l'azione che è stata compiuta o ancora la parola *tired* che descrive lo stato in cui si trova l'animale. Insomma, possiamo notare come con ognuna di queste parole ci sia una sfumatura di relazioni, cosa che una singola testa di attenzione faticherebbe a modellare da sola, proprio per questo la Multi-Head Attention riesce a risolvere questa problematica, riuscendo a tessere delle relazioni con ogni singolo elemento presente nella frase (Figura 71).

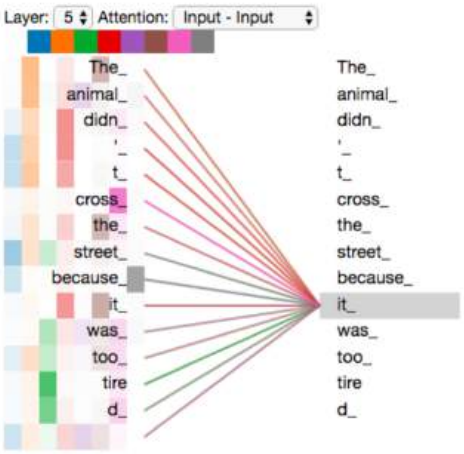


Figura 71: Esempio visivo della Multi-Head Attention: ogni testa stabilisce relazioni differenti tra la parola "it" e le altre nel contesto, evidenziate con colori distinti.

14.6 POSITIONAL ENCODING

Una limitazione intrinseca del meccanismo di *Self-Attention* risulta essere la mancanza di una nozione di ordine all'interno della sequenza. A differenza delle reti ricorrenti, i Transformer elaborano tutti i token in parallelo, senza alcuna informazione sulla loro posizione nella frase. Questo significa che, senza un intervento aggiuntivo, il modello non sarebbe in grado di distinguere tra frasi come "il cane morde l'uomo" e "l'uomo morde il cane". Per ovviare a questo problema, i Transformer introducono un meccanismo chiamato **Positional Encoding**, il quale ad ogni embedding di input aggiungono un *vettore di posizione*, codificando la posizione del token nella sequenza. A differenza degli altri parametri del modello, questi vettori non vengono appresi durante l'addestramento, ma sono definiti in modo deterministico secondo un pattern **Sinusoidale**. La loro costruzione è tale da garantire che le differenze tra posizioni risultino ancora percepibili dopo la proiezione nei vettori Q, K e V, influenzando così il calcolo dell'attenzione. I vettori di Positional Encoding sono definiti dalle seguenti funzioni periodiche:

$$PE_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \quad PE_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

in cui pos è la posizione del token nella sequenza, i l'indice della dimensione dell'embedding e d_{model} la dimensione totale dell'embedding del modello. Ogni embedding seguen un'onda sinusoidale con

frequenza diversa, un aspetto di questa scelta progettuale è che le posizioni relative possono essere descritte in modo lineare: dato un offset k, il Positional Encoding della posizione pos $+k$ può essere espresso come una funzione lineare di quello in pos, facilitando così il compito del modello nel riconoscere distanze e relazioni relative tra i token, indipendentemente dalla loro posizione assoluta.

Figura 72: Esempio di combinazione tra gli embedding delle parole e i vettori di Positional Encoding. Ogni token dell'input viene rappresentato dal proprio embedding lessicale, al quale viene sommato un vettore di Positional Encoding. Quest'ultimo fornisce al modello l'informazione sulla posizione relativa del token nella sequenza, consentendo al Transformer di distinguere l'ordine delle parole pur elaborandole in parallelo.

14.7 RESIDUAL CONNECTIONS

Un elemento dell'architettura Transformer importante sono le **Connessioni Residue** (*Residual Connections*). Queste connessioni sono nate con le reti *ResNet* e hanno rivoluzionato il modo in cui vengono addestrate le reti neurali profonde, permettendo di preservare il flusso dell'informazione e rendendo l'ottimizzazione più stabile. L'idea alla base è semplice ma estremamente efficace: l'input non passa attraverso una lunga catena di trasformazioni che rischiano di distorcere o attenuare il segnale, ma si consente all'informazione originale di "saltare" direttamente uno o più strati. Formalmente, se uno strato applica una trasformazione $F(x)$ al suo input x, l'uscita dello strato diventa:

$$\text{Output} = F(x) + x \tag{1}$$

Iil risultato finale quindi, non è solo la trasformazione $F(x)$, ma una combinazione dell'input originale e della sua versione modificata. Questo semplice schema consente al modello di apprendere più facilmente funzioni identitarie (cioè $F(x) \approx 0$), riducendo il rischio che

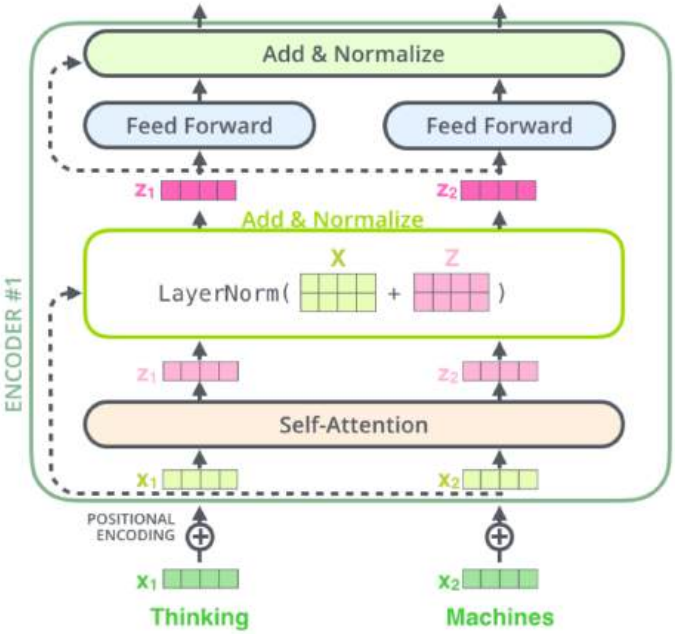


Figura 73: Struttura del blocco *Add & Norm*, che combina una connessione residua con la normalizzazione per mantenere la stabilità numerica e il flusso informativo.

l'informazione iniziale venga "dimenticata" nei livelli successivi. All'interno del Transformer, ogni blocco, sia nell'Encoder che nel Decoder, utilizza questo principio attraverso una struttura chiamata **Add & Norm** (Figura 73). Ogni sottoblocco viene avvolto da questa componente che effettua due azioni:

1. Somma l'input del blocco con la sua uscita trasformata (connessione residua);

2. Normalizza il risultato tramite una *Layer Normalization*.

Le connessioni residue riescono a creare diversi vantaggi come la mitigazione del *Vanishing Gradient Problem*, riesco a preservare l'informazione originale e riesco a facilitare l'apprendimento. Le connessioni residue possono essere viste come una sorta di **corsia preferenziale per l'informazione**, il modello può decidere, in ogni blocco, se utilizzare il segnale trasformato, mantenere quello originale o combinarli fra loro. Questo meccanismo diventa particolarmente prezioso in architetture molto profonde, dove l'accumulo di trasformazioni rischierebbe di attenuare o distorcere il contenuto semantico iniziale, compromettendo la coerenza dell'apprendimento.

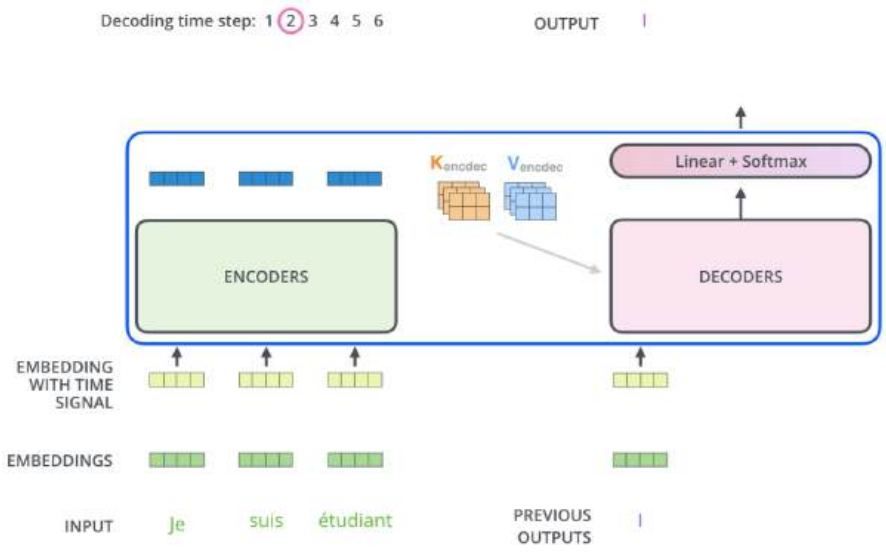


Figura 74: Esempio del processo di generazione nel decoder al secondo step temporale. Nel primo step è stata generata la parola *I* come traduzione di *je*.

14.8 DECODER SIDE

Il **Decoder**, ha il compito di generare la sequenza di output a partire dalla rappresentazione codificata dell'input, fornita dall'Encoder. L'output dell'ultimo strato dello stack degli Encoder viene trasformato in un insieme di vettori chiave (K) e valore (V), i quali saranno utilizzati in ogni strato del decoder all'interno del modulo *Encoder-Decoder Attention*. Meccanismo che permette al Decoder di concentrarsi su specifiche porzioni della sequenza di input (Figura 74). Il processo si ripete finché non viene generato un simbolo speciale di fine sequenza (`<eos>`), è importante sottolineare che anche nel Decoder, viene incorporata l'informazione posizionale. L'attenzione nel Decoder viene *Mascherata* per evitare che il modello acceda a posizioni future della sequenza di output. Precisamente, alle posizioni successive viene assegnato il valore $-\infty$ prima dell'applicazione della funzione SoftMax, così da annullarne il contributo. Il modulo *Encoder-Decoder Attention*, invece, opera in maniera analoga alla Self-Attention Multi-Head, con la differenza che le matrici K e V provengono direttamente dall'output dello stack degli Encoder, mentre le matrici Q (Query) vengono calcolate a partire dallo strato sottostante del Decoder stesso.

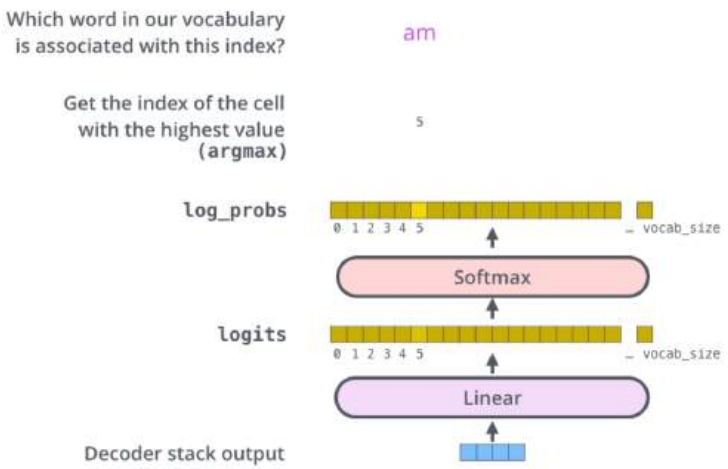


Figura 75: Struttura dei layer finali del decoder nel Transformer, che trasformano la rappresentazione vettoriale in una parola.

14.8.1 Final Layer e Softmax

Lo stack dei Decoder, al termine della generazione, produce un vettore continuo (un vettore di float), che deve essere convertito in una parola del vocabolario. A questo scopo intervengono due componenti finali: il **Final Linear Layer** e il **Softmax Layer**. Il primo è una rete completamente connessa che trasforma il vettore generato dal decoder in un **vettore di logit**, con una dimensione pari alla cardinalità del vocabolario. Il *Softmax Layer* converte questo vettore di logit in una distribuzione di probabilità: tutti i valori risultanti saranno positivi e la loro somma sarà pari a uno. L'indice con la probabilità più alta indicherà la parola da generare in quello specifico time step.

14.9 TRAINING

Durante la fase di **Training**, il Transformer esegue un *forward pass* sui dati forniti, poiché il dataset di training è etichettato, è possibile confrontare le uscite del modello con i target attesi, valutando così la qualità delle predizioni. In presenza di discrepanze tra output previsto e desiderato, si procede con un aggiornamento dei pesi tramite backpropagation, al fine di minimizzare l'errore. I vocabolari utilizzati nei Transformer contengono generalmente un numero elevato di parole, oltre a simboli speciali. Prima del training, vi è un *preprocessing* dei dati, in cui ogni parola viene mappata su un vettore numerico. Una

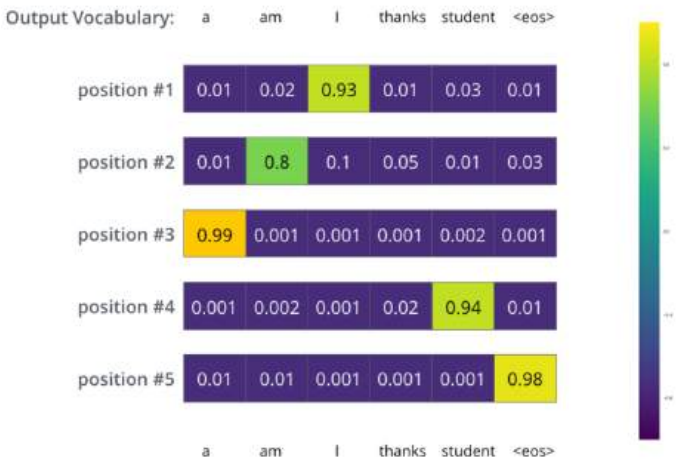


Figura 76: Distribuzioni di probabilità prodotte dal Decoder: ogni posizione temporale è associata a una distribuzione su tutto il vocabolario, la parola target dovrebbe essere quella con la probabilità massima.

rappresentazione comune è la **One-Hot Encoding**, in cui ogni parola, è rappresentata da un vettore con tutti zeri, tranne un valore pari a uno in corrispondenza della posizione associata alla parola nel vocabolario. L'obiettivo del training è guidare il modello a produrre, per ciascun time step, una distribuzione di probabilità in cui la parola attesa abbia la probabilità più alta, fino all'ultima distribuzione, in cui la probabilità più alta dovrebbe corrispondere al token `<eos>`, segnalando la fine della sequenza.

14.10 OLTRE IL TRANSFORMER

Con la comprensione completa dell'architettura originaria del Transformer, siamo ora pronti ad analizzare alcune delle estensioni che hanno reso questo modello uno standard de facto nel campo del Deep Learning. In particolare, approfondiremo come il Transformer venga adattato a contesti specifici mediante il **fine-tuning**, quali siano le sue principali **applicazioni** in scenari reali e accademici, e infine esamineremo alcune delle **varianti architetturali** che ne estendono le potenzialità, spesso superandone i limiti originali.

14.11 FINE-TUNING DEL TRANSFORMER

Il **Fine-Tuning** è una tecnica di apprendimento trasferito (*Transfer Learning*) ampiamente adottata nel contesto dei Transformer, che consente di adattare un modello pre-addestrato a un compito specifico.

Def: 14.11.1 *Il Transfer Learning è una tecnica di Machine Learning in cui si riutilizza un modello pre-addestrato su un compito per risolverne uno simile. Anziché addestrare un modello da zero, si parte da una conoscenza già acquisita, risparmiando tempo, risorse computazionali e quantità di dati necessari.*

Il modello viene inizialmente addestrato su un vasto corpus general-purpose (come Wikipedia o Common Crawl), apprendendo rappresentazioni linguistiche ricche e versatili. Successivamente, viene ottimizzato su un dataset mirato, spesso molto più piccolo, relativo a un task specifico (e.g Sentiment Analysis, Entity Recognition, Code Generation, ecc. . .). Il processo si articola come segue:

1. **Pretraining:** il modello viene addestrato in maniera auto supervisionata su un task generale;

2. **Fine-tuning supervisionato:** si sostituisce o si estende la testa del modello con uno o più layer adatti al task target, e si effettua un training supervisionato con gradient descent.

Uno degli aspetti cruciali del Fine-Tuning è la scelta del *Learning Rate*: un tasso troppo elevato potrebbe sovrascrivere le rappresentazioni apprese nel pretraining; al contrario, uno troppo basso potrebbe non fornire l'adattamento desiderato. Il Fine-Tuning può essere visto come l'aggiustamento delicato di migliaia (o miliardi) di piccole manopole, ovvero i parametri del modello, che sono già stati impostati in una configurazione utile durante la fase di pre-training. Durante il Fine-Tuning, queste manopole non vengono azzerate né completamente ricalibrate, ma solo ritoccate per adattare il modello a un nuovo compito specifico. In questo modo, si parte da una conoscenza generale già appresa e la si "rifinisce", ottenendo una configurazione dei parametri più adatta al contesto desiderato.

14.12 APPLICAZIONI DEL TRANSFORMER

L'architettura Transformer è stata adottata in una vasta gamma di applicazioni, sia in ambito linguistico che in domini non testuali. Tra le principali possiamo trovare:

- **Natural Language Processing (NLP):** è il dominio originario del Transformer, dove viene impiegato in:
  - Traduzione automatica (e.g. Google Translate);
  - Generazione di testo (e.g. ChatGPT, GPT-5);
  - Sentiment Analysis;
  - Named Entity Recognition (NER);
  - Riassunto automatico.
- **Visione artificiale (CV):** con modelli come Vision Transformer (ViT), l'architettura viene adattata al dominio visivo per compiti come classificazione, segmentazione e object detection;
- **Bioinformatica e Chimica Computazionale:** i Transformer vengono utilizzati per la modellazione di sequenze proteiche, il drug discovery, e la predizione di interazioni molecolari;
- **Codice e programmazione automatica:** modelli come Codex e CodeBERT sfruttano il Transformer per comprendere e generare codice sorgente;
- **Musica, immagini e altre modalità:** modelli come MuseNet o DALL-E impiegano architetture Transformer per generare musica e immagini, integrando capacità multi-modali.

14.13 VARIANTI ARCHITETTURALI

L'efficacia e la flessibilità del Transformer hanno portato allo sviluppo di numerose varianti architetturali, ciascuna con caratteristiche peculiari. Tra le più importanti troviamo: BERT, GPT, T5, ViT, Longformer, Performer e Linformer.

14.13.1 BERT (Bidirectional Encoder Representations from Transformers)

BERT è una variante del Transformer basata esclusivamente sulla pila di Encoder. Il pretraining viene effettuato tramite *Masked Language Modeling* (MLM) e *Next Sentence Prediction* (NSP), rendendolo particolarmente adatto per task di classificazione, QA e NER.

14.13.2 GPT (Generative Pretrained Transformer)

GPT, in particolare nelle sue versioni più recenti, utilizza esclusivamente la pila di Decoder con attenzione causale. È ottimizzato per la generazione di testo autoregressiva e mostra prestazioni notevoli in numerosi task senza necessità di fine-tuning esplicito (few-shot learer).

14.13.3 T5 (Text-To-Text Transfer Transformer)

T5 propone un approccio uniforme in cui ogni task NLP è riformulato come un problema di traduzione da testo a testo, permettendo al modello di utilizzare una singola architettura per una varietà di compiti diversi, come classificazione, traduzione e completamento.

14.13.4 Vision Transformer (ViT)

ViT adatta il Transformer all'elaborazione di immagini, dividendo un'immagine in patch (simili a token testuali) e trattandole come una sequenza da processare tramite attenzione. Questa strategia ha ottenuto risultati competitivi nei Benchmark di visione artificiale.

14.13.5 Longformer, Performer, Linformer

Queste varianti propongono meccanismi di attenzione ottimizzati per lunghe sequenze, affrontando il limite di complessità quadratica dell'attenzione standard, mediante sparsità, kernelizzazione o proiezioni lineari.

Tabella 6: Confronto tra principali varianti del modello Transformer.

Modello	Stack	Task Principali	Caratteristiche Peculiari
BERT	Encoder	Classificazione, NER, QA	Pretraining con Masked Language Modeling (MLM) e Next Sentence Prediction (NSP). Attenzione bidirezionale.
GPT	Decoder	Generazione di testo, completamento, traduzione	Addestramento autoregressivo (unidirezionale) tramite language modeling.
T5	Encoder-Decoder	Tutti i task NLP (in forma testo-testo)	Architettura unificata: ogni task è trattato come una traduzione. Buona generalizzazione.
ViT	Encoder	Classificazione immagini, segmentazione	Applica il Transformer alla visione computazionale: utilizza patch e positional embedding.
Longformer	Encoder	Elaborazione di lunghe sequenze testuali	Combina attenzione locale e globale per ridurre la complessità. Adatto a documenti estesi.
Performer	Encoder	Elaborazione efficiente di sequenze lunghe	Approssima l'attenzione softmax tramite metodi kernel, riducendo la complessità a $O(n)$.
Linformer	Encoder	Task sequenziali su lunghi input	Proietta K e V in spazi ridotti, ottenendo attenzione lineare.

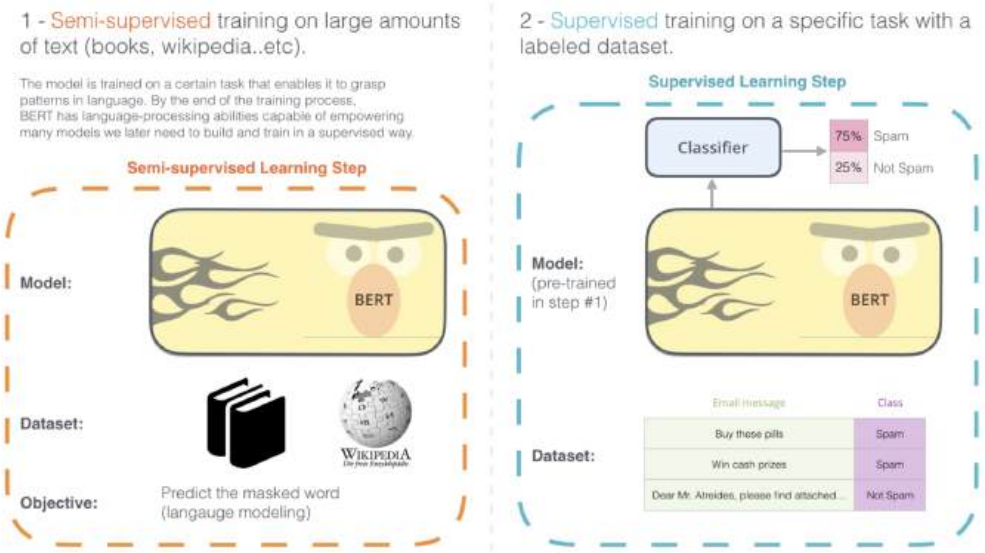


Figura 77: Rappresentazione dei due step su cui BERT è stato sviluppato, gli utenti possono scaricare il modello allenato nella prima fase, per poi occuparsi del Fine-Tuning e del secondo step da se.

14.14 BERT

BERT (Bidirectional Encoder Representations from Transformers), è un modello di linguaggio basato esclusivamente sulla componente Encoder dell'architettura Transformer, proposto da Google nel 2018 [28]. Gli utenti pensano, di non aver mai avuto a che fare con questo modello, ma in realtà lo abbiamo utilizzato trasversalmente più volte di quelle che crediamo, semplicemente effettuando una ricerca su Google. BERT infatti, è presente in varie modalità: nel momento in cui ci viene raccomandato un risultato di ricerca, sulla base di similirità testuali presenti nella nostra query di ricerca, o ancora, quando ci viene riassunta per sommi capi un'informazione presente in un sito il quale una volta cliccato la arricchirà, o in ultima analisi, quando in un sito vengono evidenziate alcune parti automaticamente che rispondono alla nostra richiesta. BERT è un modello che è stato pre-addestrato su un ampio corpus di testi (Wikipedia e BooksCorpus), acquisendo una solida conoscenza linguistica generale in maniera *Semi-Supervisionata*, poi addestrato in maniera *Supervisionata* su degli specifici task, adoperando dataset etichettati (Figura 77).

14.14.1 Il modello

La caratteristica di BERT è la sua natura *bidirezionale*: il modello è in grado di analizzare simultaneamente il contesto precedente e successivo di una parola. Questo differisce dai modelli precedenti, che si focalizzavano unicamente sul contesto passato o futuro. Inoltre, BERT è *contestuale*, ovvero assegna un significato a ogni parola in funzione del contesto in cui appare. Sono disponibili due versioni principali di BERT le quali differiscono per valori numerici:

- **BERT Base:** 12 strati di Encoder, 768 unità nascoste, 12 teste di attenzione;
- **BERT Large:** 24 strati di Encoder, 1024 unità nascoste, 16 teste di attenzione.

Oss: 11 *Il paper originale dei Transformers adottava 6 Encoder Layers, 512 Hidden Units e 8 Attention Heads, BERT Large è praticamente il doppio per tutti i parametri, eccetto gli Encoder Layers che sono addirittura il quadruplo.*

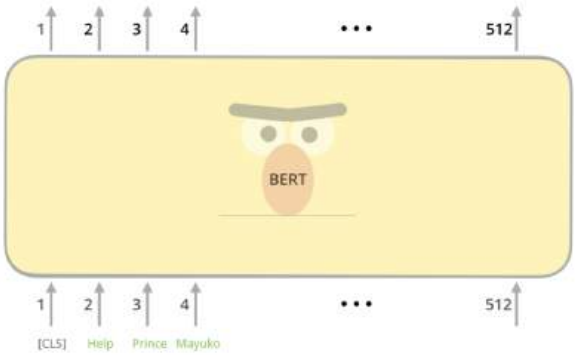


Figura 78: Visualizzazione di come gli input vengono inseriti all'interno del modello BERT.

14.14.2 Gestione dell'input

L'input testuale viene tokenizzato e successivamente trasformato in vettori di embedding, viene inserito un token speciale `[CLS]`, anteponendolo alla sequenza, utile per i compiti di classificazione, mentre tra due frasi distinte viene inserito un token `[SEP]`, il quale fungerà da delimitatore. A ogni token viene sommato un *Positional Encoding* in modo da preservare l'informazione relativa all'ordine dei token, e un

Segment Embedding il quale passerà l'informazione a ogni embedding di appartenenza a una delle frasi separate dal token separatore (Figura 79). A ogni encoder layer infine verrà applicato un meccanismo di self-attention seguito da una feed-forward network.

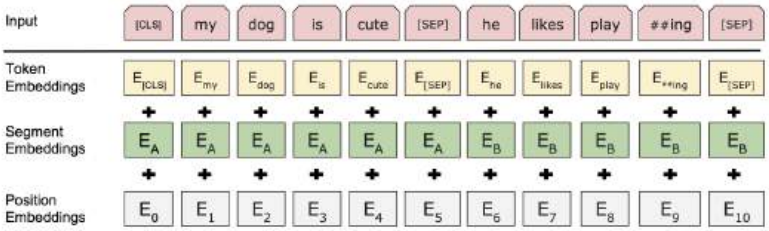


Figura 79: Rappresentazione degli input di BERT. Gli embedding di input sono la somma degli embedding dei singoli token, degli embedding di segmentazione e dei positional embedding.

14.14.3 Gestione dell'output

Ogni token genera in output, un vettore di dimensione `hidden_size` (768 nel caso di BERT base), per i compiti di classificazione verrà utilizzato il vettore associato al token `[CLS]`, questo vettore verrà poi passato a un classificatore seguito da una funzione softmax per poter ottenere una distribuzione di probabilità sulle singole classi (Figura 80). Nel caso in cui ci non si trattassero compiti di classificazione vengono prese in considerazione le diverse unità di output, a seconda del task preso in analisi, proprio per questo motivo BERT risulta essere un modello molto versatile. La rappresentazione della gestione delle posizioni di output per diversi task, è visibile anche nel paper di riferimento [28].

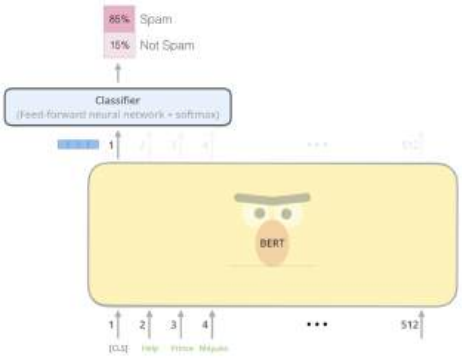


Figura 80: Visualizzazione di come gli output vengono gestiti all'interno del modello BERT in casi di task di classificazione.

14.14.4 Pre-Training

Il Pre-Training di BERT ha permesso a questo questo modello di divenire potente e generalizzabile, giungendo a ottenere nella raccolta dati circa 3.3B (miliardi) di parole, esso si basa su due task principali:

Masked Language Modeling (MLM)

Questo task è stato scelto poiché, nel momento in cui abbiamo un modello bidirezionale come BERT, che si discosta dai classici modelli con contesto passato o futuro, incorriamo in una problematica relativa alla visibilità di ciascuna parola con se stessa, portando magari il modello a predirre se stessa in un contesto multi-layer. Per arginare tutto ciò si opta per un **Mascheramento** di alcune parole in input, tramite l'utilizzo di un token speciale `[MASK]`, restituendo come output solo la parola errata, invece che l'intero input. Sebbene questo ci consenta di ottenere un modello pre-addestrato bidirezionale, uno svantaggio è che creiamo una discrepanza tra pre-training e Fine-Tuning, poiché il token `[MASK]` non compare durante il Fine-Tuning. Per mitigare questo problema, non sostituiamo sempre le parole "mascherate" con il token `[MASK]` effettivo (Figura 81). Infatti solo il 15% dei token viene selezionato casualmente e ognuno di questi avrà un diverso trattamento:

- 80% di essi verrà sostituito con il token speciale `[MASK]`;
- 10% verrà sostituito con un token casuale;
- 10% rimarrà invariato.

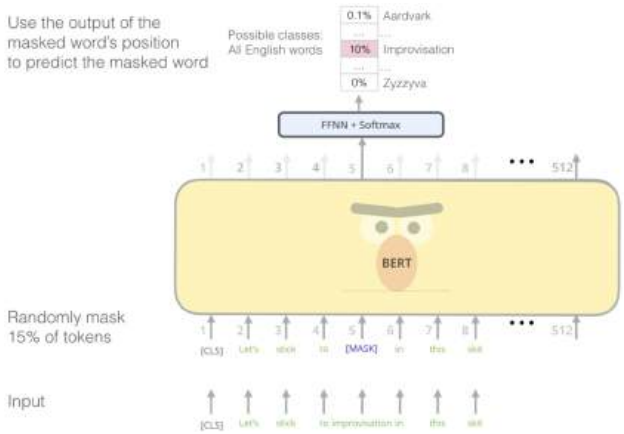


Figura 81: Il metodo intelligente di BERT per il task di Language Modeling, mascherando il 15% delle parole presenti nell'input e chiedendo al modello di predirre la parola mancante.

Next Sentence Prediction (NSP)

Molti importanti compiti a valle, come il Question Answering (QA) e l'Inferenza del Linguaggio Naturale (NLI), si basano sulla comprensione della relazione tra due frasi, che non viene catturata direttamente dalla modellazione linguistica. Per poter addestrare un modello che comprenda le relazioni tra frasi, è stato eseguito un pre-training per un compito di predizione binaria della frase successiva, che può essere generato in modo semplice da qualsiasi corpus monolinguistico. A BERT dunque verranno fornite coppie di frasi:

- Nel 50% dei casi, la seconda frase sarà effettivamente quella che segue la prima nel testo originale, etichettata con `IsNext`;
- Nel restante 50%, è una frase casuale, etichettata con `NotNext`.

Questo task è pensato per insegnare al modello le relazioni semantiche tra frasi consecutive.

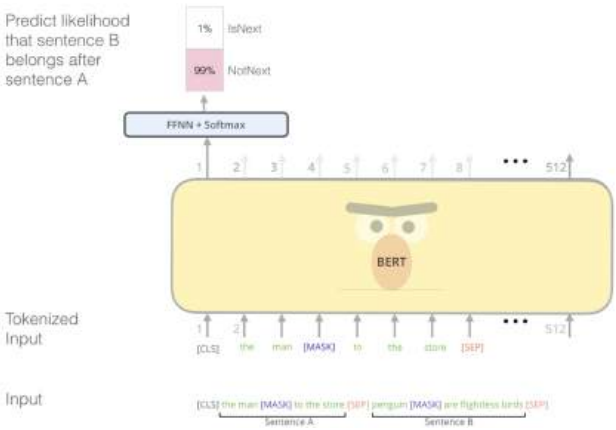


Figura 82: Il secondo task su cui BERT viene pre-addestrato: la classificazione fra due frasi successive.

14.14.5 Feature Extraction

BERT può essere impiegato non solo in tecniche di Fine-Tunin, in modo tale da poter addestrare il modello su uno specifico task, ma è possibile che venga utilizzato per una **Feature Extraction**, utilizzando gli embedding contestuali generati da BERT come input per modelli esterni, sfruttando la ricca rappresentazione appresa, proprio come già con modelli precedenti si erano ottenuti buoni risultati in questa tecnica (e.g ELMo [62]), addirittura con BERT si raffinano e migliorano le performance.

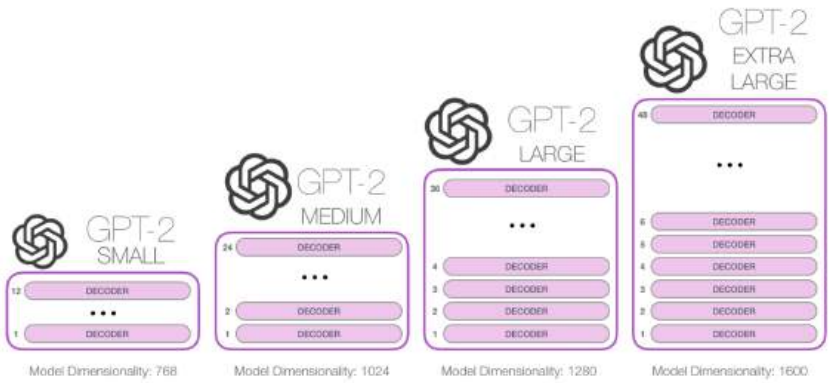


Figura 83: Esempio delle varie tipologie dei modelli di GPT-2, a seconda della dimensione del modello, si utilizzano più Decoder.

14.15 GPT NEL DETTAGLIO

GPT (Generative Pretrained Transformer) è una famiglia di modelli linguistici autoregressivi, introdotti da OpenAI. A differenza di BERT, il quale basa la sua architettura su una pila di Encoder, GPT utilizza esclusivamente la componente *Decoder* dell'architettura originale [90], l'idea di utilizzare una struttura *Decoder Only* [66], è stata dettata dall'osservazione dei ricercatori di OpenAI, di come essa fosse la scelta migliore, poiché questa architettura garantisse una gestione migliore per la scalabilità di sequenze molto lunghe, molto più lunghe della classica architettura Transformer [90]. La prima versione di GPT è stata rilasciata nel 2018, successivamente con il passare degli anni l'azienda ha rilasciato nuove versioni del modello, migliorandolo sempre più e creando funzioni e interfacce ad hoc per alcuni compiti specifici, la più famosa ChatGPT, l'evoluzione di questo modello è segnata da una scala crescente di parametri e capacità di generalizzazione.

14.15.1 Autoregressione

GPT è un modello Transformer unidirezionale, basato sulla *Next Token Prediction*, ovvero la predizione del token successivo all'interno di una frase, esso genera una parola alla volta, considerando solo il contesto precedente, definendosi dunque, come un modello Autoregressivo (Figura 84).

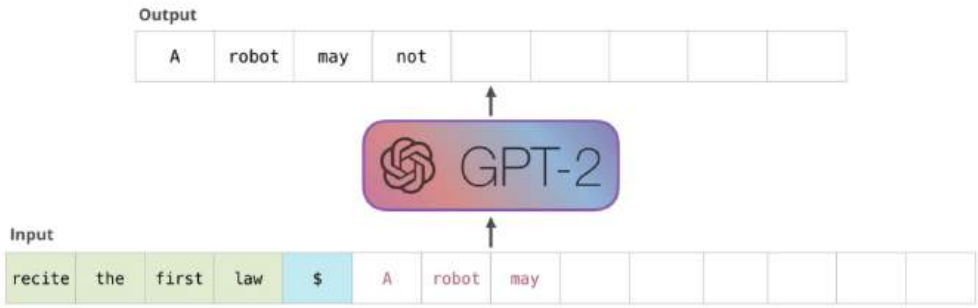


Figura 84: Rappresentazione dell'Autoregressione che avviene all'interno di GPT, ogni token generato come output viene reinserito come input, per ottenerne il successivo.

14.15.2 Tokenizzazione

Quando analizziamo gli input forniti a un modello di Natural Language Processing come BERT o GPT, spesso utilizziamo, in maniera interscambiabile il termine *parola* con il termine **token**, tuttavia vi è una grande differenza fra le due. Per comprenderla al meglio entriamo nello specifico di ciò che viene utilizzato da GPT per effettuare il processo di passaggio da parola a token detto **Tokenizzazione**, mediante l'uso di **Byte Pair Encoding** [76].

Byte Pair Encoding

La **Byte Pair Encoding** (BPE) è una tecnica di tokenizzazione usata per rappresentare il testo in modo più efficiente, molto comune nei modelli di linguaggio. L'idea su cui essa si basa è estremamente semplice: ridurre il numero di simboli unendo le coppie di byte o caratteri più frequenti in nuove unità (token). In altre parole, BPE trova sequenze ripetute di caratteri nel testo e le sostituisce con un nuovo simbolo unico, iterativamente. Alla fine, invece di trattare il testo come singoli caratteri o parole intere, si ottiene un insieme di sottoparole (subwords) che rappresentano meglio la lingua. Risulta essere molto utile per i seguenti motivi:

- **Gestione di parole sconosciute:** Se il modello non conosce ad esempio la parola "lowering", può comunque riconoscere singolarmente i token che la compongono cioè: "low" + "er" + "ing";
- **Bilanciamento fra caratteri e parole:** se si usano caratteri il vocabolario risulta molto piccolo, ma le sue sequenze sono lunghe,

se uso le parole invece il vocabolario tende ad esplodere, pertanto questa soluzione si trova nel mezzo;

- **Migliora efficienza e generalizzazione:** riducendo la dimensione del dizionario, facilita l'addestramento e consente di gestire testi in lingue diverse o con neologismi.

In realtà i modelli di GPT, utilizzano una tencnica con una leggera modifica apportata alla Byte Pair Coding, chiamata **Byte Level Byte Pair Encoding**, questa variante, non lavora sui caratteri, ma sui byte grezzi del testo. Quindi prima di iniziare a fondere le coppie frequenti, il testo viene convertito nella sua rappresentazione in byte (0–255), così che ogni carattere, simbolo o emoji diventi gestibile in modo uniforme, permettendo così di risolvere problemi legati a caratteri speciali e testi multilingua e codifiche diverse da UTF-8, permettendo inoltre alla codifica di essere reversibile. La trattazione dell'input dunque risulta essere composta dalla *tokenizzazione*, *conversione in embedding* e successivamente *somma del positional embedding*, non avendo in se token speciali come succedeva in BERT con i token `[CLS]` e `[SEP]`.

14.15.3 Decoder

GPT utilizzando unicamente blocchi Decoder del Transformer, viene meno uno strato che è presente nella formulazione classica [90], il quale si occupava dell'Encoder-Decoder Self Attention, a differenza da ciò che accadeva in BERT però il primo stato di Self-Attention, viene mascherato, così da non poter vedere i token successivi, e limitare l'osservazione solo ai token precedenti a quello attuale mediante una *casual mask*. Applicando una matrice triangolare superiore, alla matrice di calcolo dello score, prima dell'applicazione della funzione softmax, portando i valori presenti nella matrice nella parte triangolare superiore a un valore pari a $-\infty$ (Figura 85). Per quanto riguarda la il meccanismo di Self-Attention, non cambia nulla rispetto a quanto già visto, dunque vengono ricavati i vettori Query, Key e Value, viene effettuato uno splitting nelle singole teste di attenzione, prese le query e confrontate con tutte le key, ottenendo lo score di ogni testa per poi venire concatenato e moltiplicato per una matrice che ci genera l'output di questo strato passandolo al Feed Forward Layer.

Figura 85: Applicazione della maschera di attenzione sugli score ottenuti, prima dell'intervento della funzione SoftMax, mettendo in luce come ogni riga corrisponde alla presenza di una parola alla volta, nella prima abbiamo solo lo score di una sola parola, nella seconda di due, e così via.

Feed Forward Layer

Questo strato, come ci è ben noto è composto da una Fully Connected Neural Network la quale quadruplicherà nel suo primo layer la grandezza dell'input ricevuto, in analogia con quanto accade nell'architettura Transformer [90], poiché sperimentalmente portava a degli ottimi livelli di generalizzazione, e successivamente riportato alla dimensione di partenza.

14.15.4 Scelta del Token successivo

Una volta effettuato il processo di analisi attraversando tutti i Decoder, verrà generato a conclusione del processo in cui sono presenti diverse probabilità di preddizione relative alla parola successiva della nostra frase, per poter scegliere di quale parola si tratti possono essere adottate diverse modalità:

- **Argmax:** Scelto il token con probabilità massima;
- **Sampling:** Si estra un token a caso, seguendo la distribuzione di probabilità fornita;
- **Top-k sampling:** Si considerano solo i top-k token probabili, per poi selezionare il token fra questi;
- **Top-p nucleus sampling:** Vengono considerati solo i token la cui somma delle probabilità supera un valore p, per poi selezionare un token fra questi in maniera casuale;

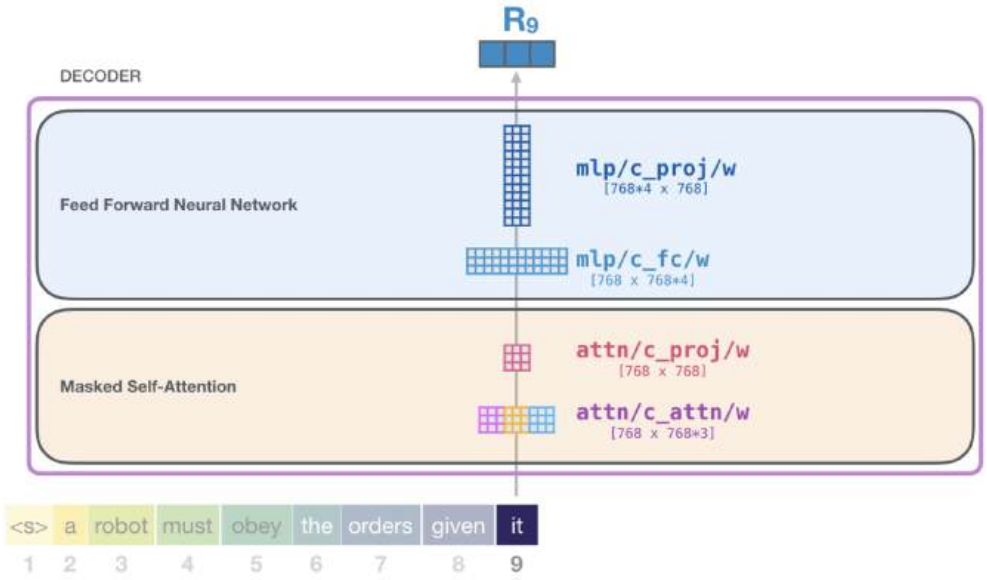


Figura 86: Analisi dell'intero processo, di predizione del token successivo, all'interno del Decoder di GPT, in cui è possibile vedere la matrice di attenzione composta da Query, Key e Value, la matrice di per ottenere l'output del layer di attenzione e i prodotti che avvengono nel layer di Feed Forward.

- **Beam Search:** Vengono mantenute più sequenze candidate contemporaneamente, il costo computazionale è elevato;

Oss: 12 *L'argmax, è una specializzazione del Top-k sampling, dove il valore di k è unitario.*

14.15.5 Pretraining

In GPT-1 [66] avviene un pretraining non supervisionato utilizzando il Dataset BooksCorpus [97], il quale comprende circa 7000 libri unici non pubblicati di vari generi, fra cui Avventura, Romantici e Fantasy, con circa 800 milioni di parole, permettendo di imparare dei pattern di strutture molto lunghe come avviene nei libri. Segue poi un Fine-Tuining supervisionato su vari task di NLP usando il GLUE Benchmark [91]. Differentemente in GPT-2 [67] non avviene alcun Fine-Tuning, ma avviene soltanto un training su un dataset più vasto e variegato costruito ad hoc da OpenAI, comprendente circa 8 milioni di documenti, composti da 40 GB di testo pulito, essendo in grado il modello di generalizzare facilmente, grazie alla grande scala di dati presenti a differenza di quello precedente. GPT-3 [17], risulta essere 100 volte più grande del suo predecessore e viene allo stesso modo solamente trainato su

Tabella 7: Confronto tra BERT e GPT.

Caratteristica	BERT	GPT
Architettura	Encoder Transformer	Decoder Transformer
Direzionalità	Bidirezionale	Unidirezionale (autoregressivo)
Obiettivo pretraining	MLM + NSP	Language Modeling
Token speciali	[CLS], [SEP], [MASK]	Nessuno
Uso principale	Comprensione del linguaggio	Generazione del linguaggio
Modalità di utilizzo	Fine-tuning, feature extraction	Prompting, fine-tuning

un dataset con una miscela di fonti diverse, raggiungendo circa i 570 GB di testo pulito. La forza di GPT-3 risiede nel fatto che, durante l'inferenza, si comporta come un "meta-modello" diventando in grado di capire il compito dai prompt, specificandoglielo e facendoli esempi, aumenterò l'accuratezza, dando l'impressione di creare un Fine-Tuning personalizzato all'utente, anche se ciò che accade nella conversazione con il modello, non porta alcuna modifica ai pesi dello stesso. GPT-4 ci è solo noto che esso sia multimodale, e utilizza il **Reinforcement Learning with Human Feedback** (RLHF), cioè apprende da feedback umano per migliorare la sicurezza e la correttezza delle risposte. GPT-5 invece adotta due modalità, una *Standard* e una *Thinking*, utilizzando un router per decidere quale modello utilizzare a seconda del task inserito nel prompt, questo router viene continuamente allenato grazie alle risposte degli utenti. Per quanto riguarda, queste due ultime versioni citate, le informazioni riguardanti tipologie di training e dataset utilizzate in maniera più approfondita, non sono note.

APPROFONDIMENTI

14.16 VARIANTI DEI MODELLI TRANSFORMER

Nel corso degli anni, numerose varianti dell'architettura Transformer sono state sviluppate per migliorarne l'efficienza, la generalizzazione e l'adattabilità a specifici compiti. Di seguito ne riportiamo alcuni:

14.16.1 RoBERTa (Robustly Optimized BERT Approach)

RoBERTa è una versione migliorata di BERT, introdotta da Facebook AI, che elimina il task di pre-training NSP, utilizza una maggiore quantità di dati e addestra il modello più a lungo. Le principali modifiche

includono:

- Rimozione del Next Sentence Prediction (NSP);
- Addestramento su batch più grandi e sequenze più lunghe;
- Dinamica del masking modificata ad ogni epoca.

RoBERTa ha mostrato prestazioni superiori a BERT in numerosi benchmark NLP.

14.16.2 ALBERT (A Lite BERT)

ALBERT propone un modello più leggero e scalabile mediante:

- Condivisione dei pesi tra i layer;
- Fattorizzazione della matrice di embedding;
- Nuovo obiettivo di pretraining: *Sentence Order Prediction (SOP)*.

Queste modifiche riducono drasticamente il numero di parametri, rendendo ALBERT più efficiente senza compromettere l'accuratezza.

14.16.3 ChatGPT e InstructGPT

Con l'espansione della famiglia GPT, l'attenzione si è spostata dal semplice addestramento generativo verso la **messa a punto comportamentale** dei modelli, ossia la capacità di rispondere in modo utile, sicuro e coerente con le intenzioni umane.

- **InstructGPT** (2022): è una versione di GPT-3 sottoposta a un processo di *fine-tuning* mediante **Reinforcement Learning from Human Feedback** (RLHF). In questo approccio, annotatori umani valutano diverse risposte generate dal modello e assegnano un punteggio di preferenza. Un secondo modello, chiamato *reward model*, apprende da queste valutazioni e guida il modello principale ad allinearsi meglio alle istruzioni fornite dall'utente. Il risultato è un comportamento più conforme alle aspettative umane, con risposte più pertinenti e un'evidente riduzione di bias e contenuti indesiderati;

- **ChatGPT** (fine 2022): nasce come applicazione conversazionale basata su InstructGPT e, successivamente, sulle versioni **GPT-3.5** e **GPT-4**. È progettato per generare risposte naturali, coerenti e contestualmente pertinenti, mantenendo la continuità del dialogo su più turni conversazionali. L'interfaccia di ChatGPT ha reso accessibile la potenza dei modelli linguistici a un vasto pubblico, promuovendo un'adozione senza precedenti dell'intelligenza artificiale generativa nel lavoro, nella ricerca e nella didattica.

14.16.4 GPT-4 Turbo e GPT-4o

Con questi modelli della famiglia GPT hanno ulteriormente migliorato la gestione del contesto, l'efficienza computazionale e la capacità multimodale del modello.

- **GPT-4 Turbo** (2023): è una versione ottimizzata di GPT-4, progettata per offrire prestazioni equivalenti con un costo computazionale ridotto e una latenza inferiore. Supporta un contesto esteso fino a **128k token**, consentendo al modello di mantenere coerenza anche in testi di grande lunghezza o in conversazioni molto prolungate. È diventato il motore predefinito di ChatGPT a partire da fine 2023.
- **GPT-4o** (2024): segna una tappa fondamentale nello sviluppo dei modelli di linguaggio, introducendo la prima architettura **multimodale unificata** in grado di elaborare testo, immagini e audio all'interno di un singolo modello end-to-end. La sigla "o" sta per *omni*, a indicare la capacità di operare su più modalità contemporaneamente. GPT-4o può percepire, comprendere e generare contenuti in tempo reale, con risposte vocali naturali e interpretazione visiva diretta, aprendo la strada a interfacce conversazionali sempre più immersive e interattive.

14.16.5 GPT-5

GPT-5 rappresenta l'evoluzione più recente della serie *Generative Pre-trained Transformer* al momento in cui scriviamo (Novembre 2025), proseguendo la direzione tracciata da GPT-4o verso un'intelligenza realmente multimodale, interattiva e contestualmente consapevole. A differenza delle versioni precedenti, GPT-5 non si focalizza unicamente

sull'aumento della scala, ma sulla **integrazione dinamica di memoria, ragionamento e apprendimento adattivo**. Le caratteristiche principali attese o già emerse includono:

- Un'architettura **multimodale unificata**, capace di gestire testo, immagini, audio e video in un unico spazio di rappresentazione condiviso;
- **Memoria contestuale persistente**, per mantenere coerenza tra interazioni nel tempo e supportare apprendimento continuo;
- Forme avanzate di **ragionamento composizionale**, basate su strategie di *chain-of-thought* e *graph-of-thought*;
- Un'ottimizzazione computazionale tramite **mixture-of-experts** e *sparse attention*, che consente al modello di attivare solo i sotto-moduli necessari al compito corrente.

GPT-5 si configura dunque come un passo verso sistemi cognitivi più flessibili e adattivi, in cui la distinzione tra pre-addestramento e apprendimento continuo tende gradualmente a sfumare. L'obiettivo non è più soltanto generare testo, ma costruire un'intelligenza in grado di **interagire, ragionare e apprendere dal proprio contesto**.

14.16.6 Verso la Prossima Generazione

Le evoluzioni future dei modelli GPT e dei *foundation models* in generale sembrano orientarsi verso tre direzioni principali:

1. **Efficienza e sostenibilità:** riduzione dei costi computazionali mediante compressione dei parametri, quantizzazione e tecniche di *sparse attention*;
2. **Memoria a lungo termine:** sviluppo di architetture dotate di memoria persistente, capaci di conservare conoscenze e aggiornarsi nel tempo;
3. **Ragionamento multimodale e adattivo:** integrazione sempre più stretta tra linguaggio, visione, audio e interazione dinamica con l'ambiente.

Questa traiettoria evidenzia come i modelli linguistici si stiano evolvendo da semplici predittori di testo a veri e propri **sistemi cognitivi generali**, capaci di apprendere, adattarsi e comprendere il mondo in maniera sempre più olistica e interattiva.

14.16.7 Altri modelli noti

- **DistilBERT:** versione compressa di BERT, con il 40% di parametri in meno e il 97% delle performance;
- **ELECTRA:** introduce un pretraining discriminativo invece del classico MLM;
- **T5 (Text-to-Text Transfer Transformer):** unifica i compiti NLP in un'unica formulazione testuale.

14.17 APPLICAZIONI PRATICHE DEI TRANSFORMER

L'adattabilità dei modelli Transformer consente la loro applicazione in una vasta gamma di task di NLP, sia in modalità supervisionata (fine-tuning) che non supervisionata (prompt engineering):

- **Classificazione testuale:** assegnazione di etichette a frasi, documenti, tweet, ecc. (es. analisi del sentiment);
- **Named Entity Recognition (NER):** identificazione di entità rilevanti nel testo (nomi, date, organizzazioni);
- **Risposta a domande (QA):** estrazione o generazione di risposte date una domanda e un contesto;
- **Traduzione automatica:** conversione da una lingua a un'altra (es. inglese $\rightarrow$ francese);
- **Generazione di testo:** produzione di frasi o paragrafi coerenti a partire da un prompt (storytelling, copywriting);
- **Riassunto automatico:** estrazione o generazione di versioni sintetiche di testi lunghi;
- **Conversational AI:** chatbot e assistenti virtuali, spesso costruiti su InstructGPT o ChatGPT;
- **Codice e programmazione:** generazione di codice (es. Codex), spiegazione di funzioni o completamento automatico.

14.18 EVOLUZIONE DEI TRANSFORMER

L'evoluzione dei modelli Transformer è stata caratterizzata da un rapido progresso sia in termini di architettura che di scala computazionale. Di seguito è riportata una timeline dei principali modelli:

- **2017 – Transformer [90]:** introduzione dell'architettura self attentiva in "Attention is All You Need";
- **2018 – BERT [28]:** encoder bidirezionale pre-addestrato con obiettivi MLM e NSP;
- **2019 – GPT-2 [67]:** modello autoregressivo di grandi dimensioni per generazione testuale;
- **2019 – RoBERTa [50]:** ottimizzazione di BERT tramite addestramento più esteso e rimozione NSP;
- **2019 – DistilBERT[73]:** distillazione di BERT per maggiore efficienza;
- **2020 – T5 [69]:**approccio "text-to-text" per unificare i compiti NLP;
- **2020 – GPT-3 [17]:** 175 miliardi di parametri, abilità few-shot learning;
- **2020 – ELECTRA [24]:**discriminatore al posto di un tradizionale generatore;
- **2021 – InstructGPT [59]:** RLHF per migliorare l'allineamento con le intenzioni umane;
- **2022 – ChatGPT:** versione ottimizzata per il dialogo basata su InstructGPT;
- **2023 – GPT-4:** modello multimodale con prestazioni avanzate in molti contesti;
- **2025 - GPT-5:** verso un'intelligenza realmente multimodale, interattiva e contestualmente consapevole.

15 DIFFUSION MODELS

In questo capitolo affrontiamo una tematica profondamente diversa da quanto visto finora: l'utilizzo dei modelli di Deep Learning per la generazione di immagini. In precedenza, abbiamo esplorato l'approccio delle **Generative Adversarial Networks** (GAN), in cui si apprende direttamente la distribuzione di probabilità da cui si generano nuovi dati. Concentriamoci su una nuova metodologia: i **Modelli di Diffusione**. I modelli di diffusione si fondano su un'intuizione tanto semplice quanto potente: corrompere progressivamente un dato, aggiungendo rumore, fino a trasformarlo in rumore puro, per poi addestrare un modello per compiere il processo inverso, rimuovere gradualmente il rumore, a partire da un rumore casuale l'immagine che mi aspetto di ottenere (Figura 87). L'ispirazione teorica proviene dalla termodinamica, in cui i processi stocastici descrivono il passaggio da uno stato ordinato a uno disordinato e viceversa.

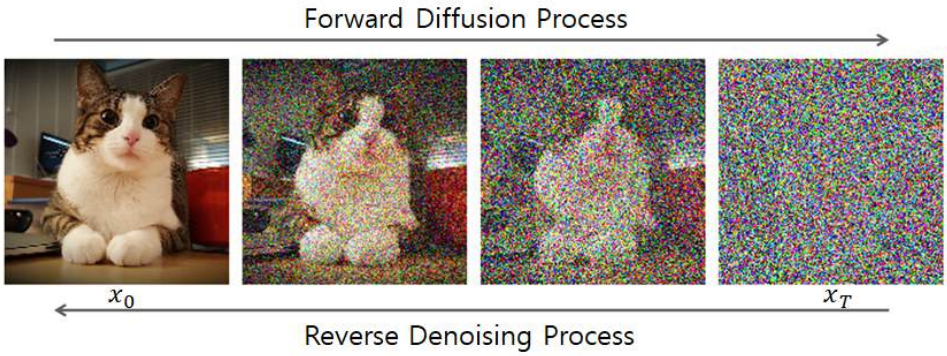


Figura 87: Schema di un modello di diffusione: nel processo forward si aggiunge rumore progressivamente a un'immagine, mentre nel processo inverso (Reverse Diffusion) si parte dal rumore per ricostruire l'immagine originale.

15.1 STABLE DIFFUSION

Una delle implementazioni più celebri ed efficienti di questa famiglia è **Stable Diffusion**. La sua innovazione principale consiste nel non

applicare la diffusione direttamente sui pixel, ma utilizzando uno **Spazio Latente Compresso**. Questo spazio, ottenuto tramite un Autoencoder, rappresenta le informazioni essenziali dell'immagine in una forma compatta e gestibile.

- Il modello lavora con vettori di dimensioni molto più piccole rispetto ai pixel originali;
- Il costo computazionale e la memoria richiesta si riducono drasticamente;
- Il processo diventa più efficiente, pur mantenendo un'elevata qualità visiva.

Questa strategia ricorda quanto visto nei **Variational Autoencoder (VAE)**, dove si opera in uno spazio latente che cattura la struttura semantica dei dati.

15.2 TASK DEI MODELLI DI DIFFUSIONE

Stable Diffusion consente di affrontare diversi compiti di generazione, chiamati *task*, tra cui:

- **`text2img`:** generazione di immagini a partire da un prompt testuale;
- **`text+image`:** generazione o modifica di immagini a partire da un'immagine e un prompt testuale.

In entrambi i casi, il modello è composto da tre moduli principali:

1. **Text Encoder:** converte il testo del prompt in un embedding numerico (tramite il modello CLIP);
2. **Image Information Creator:** una rete U-Net che, insieme a uno scheduler, effettua il processo di denoising nello spazio latente;
3. **Image Decoder:** un Autoencoder che ricostruisce l'immagine finale dallo spazio latente.

Figura 88: Esempio di task `text2img` (a sinistra), ed esempio di task `text+image` (a destra).

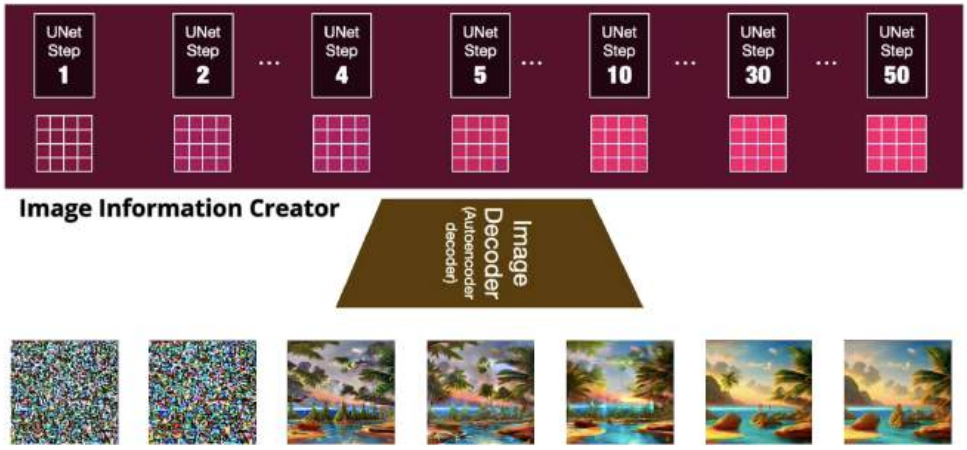


Figura 89: Output decodificato progressivamente a ogni step della diffusione inversa.

15.3 MECCANISMO DI DIFFUSIONE

Il processo di diffusione si sviluppa in una serie di step progressivi. A ogni passo, il modello prende in input un vettore latente e lo aggiorna, aggiungendo o rimuovendo rumore a seconda della direzione del processo (forward o inversa). Durante la fase di generazione, il vettore latente viene gradualmente raffinato fino ad assumere la forma desiderata. Se osservassimo i risultati decodificati dopo ciascun passo, vedremmo immagini sempre più nitide, che da un semplice rumore convergono progressivamente verso un'immagine coerente con quanto richiesto (Figura 89).

15.4 DIFFUSIONE INVERSA

Il processo inverso è il cuore dei modelli di diffusione. Per ricostruire un'immagine dal rumore, il modello deve sapere quanto rumore è

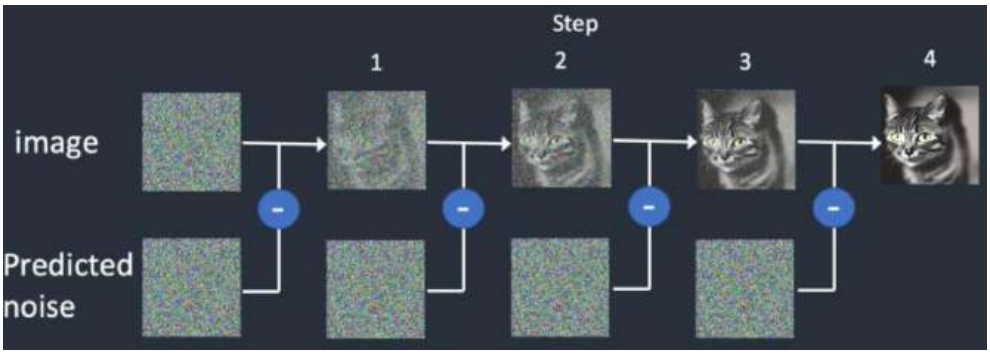


Figura 90: Esempio di reverse diffusion: il rumore viene rimosso progressivamente fino a ottenere un'immagine pulita.

stato aggiunto a ogni passo. Per questo, si addestra una rete neurale, chiamata **Noise Predictor**, per stimare la quantità di rumore presente. L'addestramento segue questi passaggi:

1. Si seleziona un'immagine dal dataset;
2. Si genera una quantità casuale di rumore;
3. Si corrompe l'immagine originale sommando il rumore;
4. Si addestra la rete a predirre il rumore aggiunto.

Oss: 13 *Per rendere il Training meno costoso, si possono utilizzare le stesse immagini con diverse quantità di rumore, così da poter avere un dataset più ampio, su cui allenare in Noise Predictor, a partire da poche immagini.*

Durante la generazione (*Denoising*), si parte da puro rumore e si sottrae, a ogni step, la stima del rumore fornita dal modello. Iterando il processo decine o centinaia di volte, il rumore scompare gradualmente, rivelando un'immagine coerente con quanto aspettato (Figura 90). Senza alcun vincolo esterno, questo processo è detto **incondizionato**, poiché il contenuto dell'immagine è determinato casualmente.

15.5 CONDITIONING

Per generare immagini coerenti con un prompt testuale, è necessario introdurre un meccanismo di **Condizionamento** (*conditioning*). In pratica, il modello non parte più solo dal rumore, ma riceve anche un'informazione aggiuntiva (il testo) che guida il processo di generazione. Il prompt viene elaborato da un **Text Encoder**, che produce un embedding

vettoriale. Questo embedding viene poi integrato nella U-Net attraverso un meccanismo di **Cross-Attention**, consentendo al modello di generare contenuti visivi coerenti con la descrizione testuale.

15.5.1 Cross-Attention

Il meccanismo di **Cross-Attention** permette a una sequenza di essere "guidata" da un'altra. A differenza della *Self-Attention*, dove una sequenza interagisce con sé stessa, qui l'interazione avviene tra due sequenze distinte (ad esempio, testo e immagine latente). Nel caso di Stable Diffusion:

- Il **Text Encoder CLIP** genera un embedding per il prompt;
- La **U-Net** usa Self-Attention per la coerenza interna e Cross-Attention per collegarsi semanticamente al testo;
- Il risultato è un'immagine generata che riflette fedelmente il contenuto descritto nel prompt.

CLIP

Il modello **CLIP** (Contrastive Language–Image Pretraining) è fondamentale per collegare linguaggio e visione. Durante il suo addestramento, CLIP riceve coppie immagine-testo e impara a rappresentarle in uno spazio comune: due embedding (uno testuale e uno visivo) sono tanto più vicini quanto più l'immagine e la descrizione corrispondono semanticamente. Questa similarità viene misurata tramite la **cosine similarity** e ottimizzata via backpropagation.

Cross-Attention nei Transformer

Nel contesto dei Transformer, la Cross-Attention combina due sequenze: una da cui si ottengono le *Key* e le *Value*, e un'altra da cui derivano le *Query*. L'output finale mantiene la dimensione della sequenza di Query.

15.6 ULTERIORI FORME DI CONDITIONING

Il testo non è l'unico modo per guidare la generazione. Stable Diffusion e i suoi derivati permettono anche di condizionare il processo su

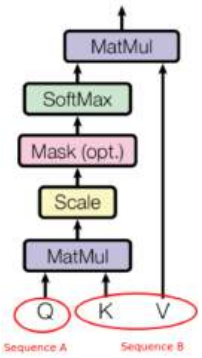


Figura 91: Schema del meccanismo di cross-attention all'interno di un Transformer.

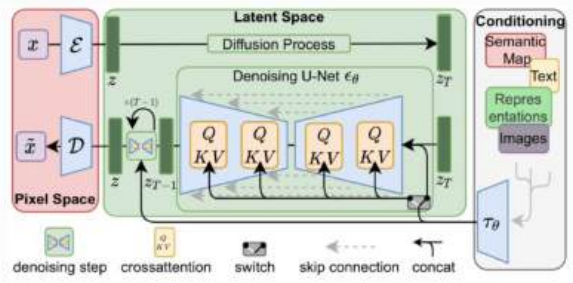


Figura 92: Architettura completa della Stable Diffusion.

altre informazioni, come bordi, profondità o pose. Queste varianti consentono un controllo più preciso e mirato sull'immagine generata.

15.6.1 Image-to-Image

Nel metodo **Image-to-Image**, un'immagine di partenza e un prompt testuale vengono utilizzati insieme per produrre una nuova immagine. Questo approccio, introdotto con *SDEdit*, segue questi passaggi:

1. L'immagine viene codificata nello spazio latente;
2. Si aggiunge una quantità controllata di rumore;
3. La U-Net predice il rumore in base al prompt;
4. Il rumore stimato viene sottratto e il processo ripetuto più volte;
5. L'immagine finale viene decodificata tramite il VAE.

15.6.2 Depth-to-Image

Il metodo **Depth-to-Image** estende il precedente introducendo una mappa di profondità, ottenuta tramite il modello MiDaS. Questa aggiunta fornisce informazioni strutturali sulla scena, migliorando la coerenza spaziale dell'immagine generata.

1. Si codifica l'immagine nello spazio latente;
2. Si estrae la mappa di profondità con MiDaS;
3. Si aggiunge rumore controllato;
4. La U-Net predice il rumore condizionata su prompt e profondità;
5. Si rimuove progressivamente il rumore e si decodifica l'immagine finale.

15.7 CLASSIFIER GUIDANCE E AGGIORNAMENTI DELLA STABLE DIFFUSION

Nei modelli di diffusione condizionati, un aspetto fondamentale è la capacità di guidare il processo generativo in modo preciso verso un risultato desiderato. A tal fine sono state introdotte due tecniche chiave: il **Classifier Guidance** e il successivo **Classifier-Free Guidance** (CFG). Entrambe mirano a controllare il contenuto dell'immagine generata, ma con approcci concettualmente diversi.

15.7.1 Classifier Guidance

Il **Classifier Guidance** rappresenta la prima estensione importante dei modelli di diffusione condizionati. L'idea alla base è intuitiva: aggiungere una "forza" esterna che spinga la generazione verso una determinata direzione semantica, come una classe, un oggetto o uno stile desiderato. In altre parole, si vuole che l'immagine non sia solo coerente, ma anche *più fedele* alla condizione imposta (e.g "gatto", "tramonto", "in stile Van Gogh").

Intuizione di base

Durante la generazione, il modello di denoising (ad esempio una U-Net) produce a ogni step un'immagine leggermente meno rumorosa. Il Classifier Guidance introduce un secondo modello, un **classificatore**, che valuta quanto l'immagine corrente corrisponde alla condizione desiderata. La sua derivata rispetto all'immagine latente indica in che direzione modificare il processo di denoising per aumentare la probabilità della condizione voluta. In pratica, questo gradiente funziona come una bussola: il classificatore indica la direzione nel "paesaggio probabilistico" che porta verso immagini più coerenti con la condizione desiderata.

Formalizzazione Matematica

Nel caso di una generazione condizionata su una variabile y, vogliamo campionare da $p(x_0|y)$, cioè generare immagini coerenti con la condizione y. Possiamo esprimere il gradiente di questa distribuzione come:

$$\nabla_x \log p(x_t|y) = \nabla_x \log p(x_t) + \nabla_x \log p(y|x_t) \tag{2}$$

dove:

- x_t è lo stato latente (rumoroso) al tempo t;
- $p(x_t)$ rappresenta la distribuzione incondizionata appresa dal modello di diffusione;
- $p(y|x_t)$ è la probabilità, stimata da un classificatore, che lo stato corrente corrisponda alla condizione desiderata y.

Il primo termine è ciò che il modello generativo apprende normalmente; il secondo termine, fornito dal classificatore, aggiunge la spinta verso la condizione y. Durante la diffusione inversa, questo gradiente viene usato per modificare la previsione del rumore della U-Net:

$$\epsilon_\theta^{\text{guided}} = \epsilon_\theta(x_t, t) - s \cdot \sigma_t \cdot \nabla_{x_t} \log p(y|x_t) \tag{3}$$

dove:

- $\epsilon_\theta(x_t, t)$ è il rumore stimato dal modello generativo;
- s è un coefficiente che controlla l'intensità della guida;
- σ_t è la deviazione standard del rumore al passo t.

Aumentando s, il modello viene "spinto" maggiormente verso immagini che il classificatore considera coerenti con la condizione desiderata, ma se s è troppo alto si rischia di perdere diversità o introdurre artefatti visivi.

15.7.2 Classifier-Free Guidance (CFG)

Sebbene efficace, il Classifier Guidance richiede un classificatore esterno, addestrato separatamente. Questo rende l'approccio più complesso e meno flessibile. Per semplificare, nasce il **Classifier-Free Guidance (CFG)**, oggi utilizzato in modelli come Stable Diffusion, Imagen e DALL-E 3.

Intuizione

Nel CFG non si usa un classificatore esplicito. Si addestra invece lo stesso modello generativo a lavorare in due modalità:

- **Condizionata:** quando riceve il prompt testuale y;
- **Incondizionata:** quando genera senza prompt (solo dal rumore).

Durante la generazione, si combinano le due previsioni per controllare quanto il risultato finale debba aderire al prompt testuale. Formalmente:

$$\epsilon_\theta^{\text{guided}} = (1 + w) \cdot \epsilon_\theta(x_t, t, y) - w \cdot \epsilon_\theta(x_t, t) \tag{4}$$

dove:

- $\epsilon_\theta(x_t, t, y)$ è la predizione condizionata (con prompt);
- $\epsilon_\theta(x_t, t)$ è quella incondizionata (senza prompt);
- w è un iperparametro che regola la forza della guida (*guidance scale*), tipicamente compreso tra 1.5 e 7.5.

Quando $w = 0$, il modello ignora completamente il prompt e genera in modo casuale; quando w aumenta, il modello segue più rigidamente le istruzioni testuali, generando immagini più coerenti ma anche meno varie. Questa strategia consente di ottenere un controllo fine sull'aderenza semantica, senza la complessità di un classificatore aggiuntivo. Inoltre, il CFG ha dimostrato empiricamente di fornire risultati visivamente superiori e più coerenti con il testo rispetto al Classifier Guidance tradizionale.

15.7.3 Stable Diffusion 2.0 e versioni successive

Con la diffusione mondiale di Stable Diffusion, sono state introdotte numerose versioni migliorate del modello, ciascuna con affinamenti architetturali e prestazionali. Le principali innovazioni introdotte a partire dalla versione 2.0 includono:

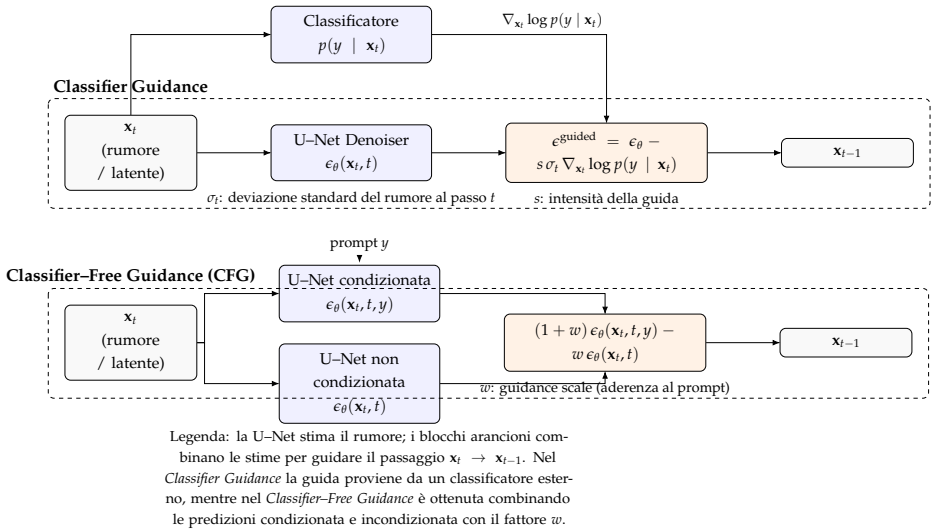
- **Addestramento su risoluzioni più elevate** (es. 768x768), per ottenere immagini nitide e dettagliate;
- **Nuovi text encoder** (e.g OpenCLIP), n grado di comprendere in modo più preciso la semantica dei prompt complessi;
- **Depth-conditioning** nativamente integrato, per generazioni controllate tramite mappe di profondità;
- **Miglioramenti al VAE decoder**, riducendo artefatti e distorsioni cromatiche;
- **Introduzione di ControlNet**, un'estensione che consente un controllo esplicito della struttura dell'immagine (pose, bordi, schizzi, profondità, ecc. . .).

Grazie a questi avanzamenti, Stable Diffusion 2.x è diventato un riferimento industriale e accademico per la generazione di immagini controllata. Il suo equilibrio tra efficienza, qualità visiva e flessibilità ha aperto la strada a un'enorme varietà di applicazioni: dall'arte digitale alla pubblicità, fino alla ricerca scientifica e al design industriale. In sintesi, il passaggio da *Classifier Guidance* a *Classifier-Free Guidance* segna una tappa fondamentale nell'evoluzione dei modelli di diffusione: da un approccio supervisionato e complesso, si è giunti a un metodo più snello, stabile e adattabile, oggi alla base dei modelli generativi più avanzati.

APPROFONDIMENTI

DDPM (Denoising Diffusion Probabilistic Models)

I **DDPM** rappresentano il framework classico dei modelli di diffusione, introdotti da Ho et al. 2020 [39]. Il processo è modellato come una catena Markoviana di lunghezza T, con una forward process che aggiunge



Legenda: la U–Net stima il rumore; i blocchi arancioni combinano le stime per guidare il passaggio $\mathbf{x}_t \rightarrow \mathbf{x}_{t-1}$. Nel *Classifier Guidance* la guida proviene da un classificatore esterno, mentre nel *Classifier–Free Guidance* è ottenuta combinando le predizioni condizionata e incondizionata con il fattore w.

Figura 93: Confronto tra **Classifier Guidance** (in alto) e **Classifier–Free Guidance** (in basso). Nel primo, la direzione di guida deriva da un classificatore esterno; nel secondo, la guida è interna al modello e regolata dal parametro w.

rumore gaussianamente e una reverse process approssimata da una rete neurale. Il modello apprende a predire il rumore ϵ iniettato in x_t:

$$\mathcal{L}_{\text{simple}} = \mathbb{E}_{t,x_0,\epsilon}[\|\epsilon - \epsilon_\theta(x_t, t)\|^2] \tag{5}$$

Pur offrendo alta qualità, il sampling è lento (richiede centinaia di passaggi).

DDIM (Denoising Diffusion Implicit Models)

Proposto da Song et al. (2021) [78], **DDIM** permette un processo di campionamento deterministico (non stocastico), più rapido ed efficiente. Invece di ricampionare il rumore a ogni step, l'algoritmo definisce una traiettoria implicita nell'input space:

$$x_{t-1} = \sqrt{\alpha_{t-1}} \cdot x_0 + \sqrt{1 - \alpha_{t-1}} \cdot \epsilon_t \tag{6}$$

DDIM consente di ridurre il numero di step di generazione (e.g da 1000 a 50) senza perdita significativa di qualità.

Score-based Generative Models

Questi modelli, sviluppati in parallelo ai DDPM, si fondano sull'apprendimento dello **score function** $\nabla_x \log p(x_t)$, sfruttando tecniche di Stochastic Differential Equations (SDEs). Le score functions vengono apprese a diversi livelli di rumore, e successivamente usate in una

reverse SDE per il sampling. La connessione tra questi modelli e i DDPM è stata formalizzata, rendendoli due facce della stessa medaglia: i DDPM sono una discretizzazione specifica di un processo continuo definito dalle SDE.

ControlNet: Controllo strutturato nella generazione

ControlNet è un'estensione di Stable Diffusion che consente di aggiungere forti vincoli spaziali o semantici durante la generazione di immagini, preservando fedelmente la struttura in input.

Motivazione

La generazione condizionata solo sul testo può risultare ambigua o imprecisa rispetto a vincoli geometrici desiderati (pose, silhouette, contorni, ecc.). ControlNet introduce un secondo input (es. mappa di pose, bordi Canny, depth map), utilizzato per controllare la generazione a livello strutturale.

Architettura

ControlNet replica la U-Net di Stable Diffusion, inizializzandola con i pesi pre-addestrati, ma aggiunge dei **moduli di controllo** (ad es. convolutioni residue) che elaborano l'input strutturale. Questi moduli sono inizialmente disattivati (zero-conv) e vengono addestrati separatamente. Durante il training, il modello apprende come combinare la guida testuale (prompt) e quella strutturale, garantendo coerenza visiva e semantica.

Tipologie di controllo supportate

ControlNet supporta diverse modalità di conditioning:

- **Pose estimation (OpenPose):** genera persone in pose specifiche;
- **Canny edges:** preserva bordi di oggetti o silhouette;
- **Scribble / Sketch:** permette agli utenti di disegnare forme rudimentali;
- **Depth maps:** rispetta la disposizione spaziale degli oggetti;
- **Normal maps:** per superfici e orientamenti 3D.

Questa flessibilità rende ControlNet uno strumento potente per applicazioni artistiche, design, fashion, architettura e prototipazione visiva.

Modelli a confronto

Oss: 14 ***Stable Diffusion*** *è la soluzione preferita per chi desidera pieno controllo, estensibilità e utilizzo locale. Supporta l'intero ecosistema open-source, inclusi modelli LoRA, ControlNet, DreamBooth, ecc.*

Oss: 15 ***DALL-E 2*** *è adatto a chi cerca semplicità d'uso e buone prestazioni, ma con meno flessibilità.*

Oss: 16 ***Midjourney*** *produce risultati artisticamente curati e stilizzati, ma non offre controllo diretto sulla struttura.*

Oss: 17 ***Imagen****, pur dimostrando risultati eccezionali nei paper, non è accessibile pubblicamente per motivi di sicurezza e bias.*

Prompt Engineering

Prompt ben progettati sono essenziali per ottenere immagini coerenti e di qualità. Un prompt efficace include:

- **Soggetto primario:** "a futuristic city skyline";
- **Modificatori visivi:** "at night, with neon lights";
- **Stile:** "cyberpunk, digital art, trending on ArtStation";
- **Fotografia:** "wide-angle, bokeh, depth of field";
- **Compositing:** "symmetrical composition, ultra-detailed".

Prompt negativo

Stable Diffusion supporta anche **prompt negativi**, usati per escludere elementi indesiderati:

[scale=0.9] `-negative_prompt="blurry, low resolution, bad anatomy, extra limbs, disfigured"`

Questi aiutano a ridurre artefatti e risultati indesiderati.

Tabella 8: Confronto tra principali modelli di generazione di immagini

Caratteristica	Stable Diffusion	DALL·E 2	Midjourney	Imagen
Architettura	Diffusione latente (VAE + U-Net + CLIP)	Diffusione con VQGAN e CLIP	Proprietaria (non pubblicata)	Diffusione pura + T5 encoder
Accessibilità	Open source, estendibile	Accesso tramite API OpenAI	Solo via Discord (chiuso)	Non disponibile al pubblico
Controllabilità	Alta: ControlNet, CFG, img2img	Limitata	Bassa, prompt tuning empirico	Alta, ma solo testata internamente
Qualità visiva	Molto alta (variabile con CFG)	Alta, dettagli precisi	Alta, stile molto curato	Eccellente, fotorealistico
Customizzazione	Altissima (plugin, estensioni, modelli LoRA)	Limitata	Nessuna	Nessuna
Prompt understanding	Buono (dipende dal modello CLIP)	Ottimo	Medio-alto	Eccellente (T5 encoder)
Uso offline	Sì (tutto in locale)	No	No	No

Prompt chaining e combinazioni

Tecniche avanzate includono:

- **Prompt chaining:** usare l'immagine generata da un prompt come base per il successivo (img2img);
- **Attention weighting:** parentesi per controllare l'importanza di certe parole: `(cat:1.3)`, `[tree:0.5]`;
- **Multiple subjects:** separati da "AND" o virgole per composizioni complesse.

Token embedding personalizzati

È possibile introdurre token personalizzati (es. `<sks style>`, `<johndoe face>`) associati a concetti specifici via fine-tuning o DreamBooth, utili in contesti professionali (moda, branding, VFX).

16 GNN & GCN

Molti dati del mondo reale non seguono delle strutture regolari, proprio in questi casi, i grafi offrono una rappresentazione più adatta: strutture composte da nodi (gli elementi) e archi (le connessioni tra essi), in grado di modellare relazioni complesse come quelle presenti nei social network, nelle molecole, nei testi, nelle immagini, nei sistemi biologici e in quelli di raccomandazione. Questo capitolo è dedicato al Graph Machine Learning, ovvero all'applicazione del Deep Learning ai dati rappresentati come grafi. Analizzeremo le Graph Neural Networks (GNN) e le principali famiglie di modelli, i problemi che possono risolvere e le tecniche per rappresentare e manipolare i dati strutturati all'interno di una rete neurale, e ci soffermeremo sulle Graph Convolutional Networks (GCN). Approfondiremo poi modelli più recenti e avanzati, come le Graph Attention Networks (GAT), GraphSAGE e le Graph Isomorphism Networks (GIN).

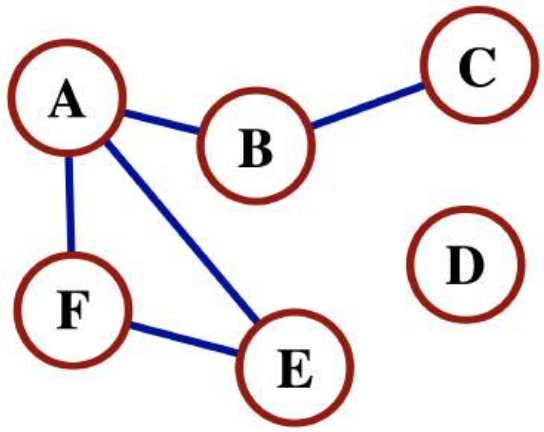


Figura 94: Esempio di grafo non orientato composto da 6 nodi e 5 archi. Se fosse orientato, gli archi avrebbero delle frecce che ne indicherebbero la direzione.

16.1 GRAFI

Un **grafo** ($G = (V, E)$) è una struttura flessibile in grado di rappresentare informazioni interconnesse. È composto da elementi chiamati *vertici* (V o nodi), connessi tra loro mediante *Archi* (E o *Spigoli*), a differenza di una lista o di una sequenza, un grafo non presenta un **ordine implicito**, ed è quindi ideale per descrivere dati complessi che non seguono un ordine lineare.

- Gli archi possono essere unidirezionali o bidirezionali, definendo i grafi orientati o non orientati;
- Ogni componente possiede degli attributi: i nodi sono caratterizzati da feature (vettori di attributi), dalla loro identità o dal numero di connessioni (grado); gli archi possono avere un peso che quantifica la forza della connessione;
- Talvolta, si introduce anche un vettore di Contesto Globale ($\mathcal{U}$ o Master Node), che sintetizza le informazioni dell'intero grafo.

16.1.1 Immagini come grafi

Le immagini possono essere descritte tramite grafi a struttura regolare (griglie). Si effettua una corrispondenza fra pixel e nodo, connettendoli fra loro in base all'adiacenza spaziale (spesso 4 o 8 vicini). L'informazione contenuta in ciascun nodo è il vettore RGB (o altri canali) del pixel. Per rappresentare le connessioni tra i nodi in modo computabile si utilizza la **Matrice di Adiacenza** (A, Figura 95) , una struttura che indica quali nodi sono connessi, facilitando le operazioni computazionali sul grafo.

16.1.2 Testi come grafi

Modellare testi come grafi non è la pratica più comune in **NLP** (Natural Language Processing) perché questi dati possiedono una struttura regolare e lineare (sequenza di token). Nei testi, ogni parola è canonicamente connessa solo a quella precedente e a quella successiva, producendo matrici di adiacenza fortemente strutturate, spesso con una diagonale che riflette questa linearità. L'utilizzo dei grafi in NLP diventa vantaggioso solo quando si vogliono modellare relazioni complesse e irregolari che

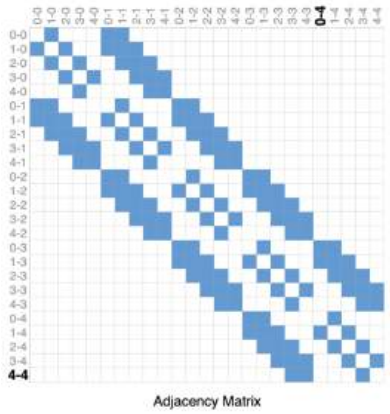


Figura 95: Matrice di adiacenza per un'immagine 5×5 raffigurante una faccina sorridente. I nodi (pixel) sono connessi se condividono un lato.

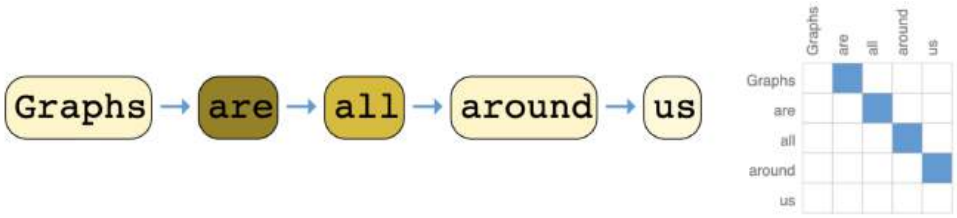


Figura 96: Matrice di adiacenza di un testo: ogni parola è connessa alla precedente e alla successiva, dando origine a una struttura diagonale.

esulano dalla semplice sequenza (e.g *Semantic Parsing*, *Dependency Trees*, *o Knowledge Graphs*).

16.1.3 Task sui grafici

Le GNN possono essere applicate su tre diverse tipologie di task:

- **Graph-level task:** Obiettivo di predire una proprietà che riguarda l'intero grafo (un singolo output);
  - *Esempio:* Prevedere la tossicità, l'odore, o la funzione di una molecola (modellata come grafo);
  - *Parallelo:* Classificazione di immagini; Sentiment Analysis di un intero documento;
- **Node-level task:** Obiettivo di classificare o predire una proprietà per ciascun nodo individualmente;
  - *Esempio:* Il dataset Zachary's Karate Club (prevedere l'appartenenza a una fazione, Figura 97);

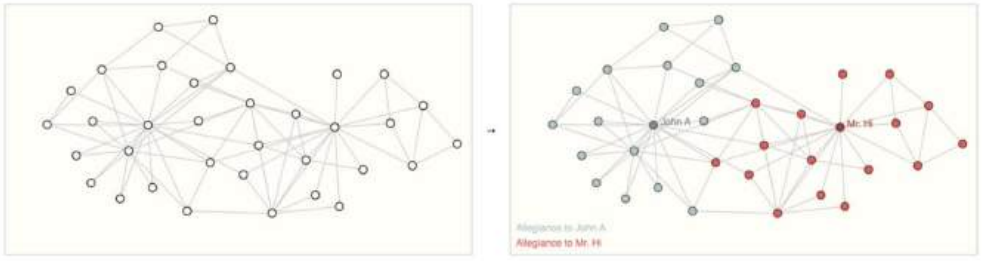


Figura 97: Zachary's Karate Club: a sinistra, la rete prima della divisione; a destra, i due gruppi dopo lo scisma.

  - *Parallelo:* Segmentazione semantica nelle immagini (etichettare ogni pixel); Part-of-Speech Tagging (assegnare una categoria grammaticale a ogni parola);

- **Edge-level task:** Valutare le relazioni (link prediction) o le proprietà degli archi fra i singoli nodi;

  - *Esempio:* Prevedere se due persone si collegheranno in un social network (Link Prediction); determinare le relazioni spaziali tra oggetti in una scena. In questi compiti si costruisce spesso un grafo completamente connesso o un bipartite graph per esplorare le interazioni.

16.2 COMPUTAZIONE SUI GRAFI

La flessibilità dei grafi implica l'**assenza di una struttura fissa**. Grafi diversi (es. due molecole) hanno dimensioni variabili. Anche lo stesso grafo può essere rappresentato da matrici di adiacenza completamente diverse a seconda dell'ordine in cui si elencano i nodi. Questo rende la rappresentazione sensibile alle **permutazioni** dei nodi. La sfida principale è che il modello Deep Learning deve essere progettato per essere **invariante alla permutazione** (o **equi-variante**), ovvero il risultato finale del task (e.g la tossicità di una molecola) non deve cambiare se si riordinano i nodi.

16.2.1 Ordine e sparsità

Convertire un grafo in un vettore (o in una matrice di adiacenza) richiede un ordinamento dei nodi. Questo ordinamento è arbitrario e,

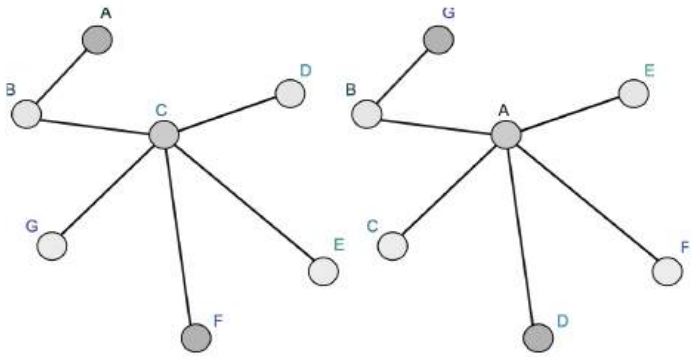


Figura 98: Due rappresentazioni diverse di un alfabeto attraverso un grafo. Sebbene l'ordine dei nodi sia differente, il significato sottostante è identico.

come detto, porta a diverse rappresentazioni equivalenti. Nel contesto delle GNN, non si cerca di ordinare i nodi in modo canonico (cosa difficile e costosa per grafi grandi), ma si cerca di utilizzare tecniche di aggregazione (Message Passing) che sono *intrinsecamente invarianti all'ordine*. La sparsità è una caratteristica strutturale (avere pochi archi rispetto al numero massimo possibile) che è cruciale per la scalabilità computazionale. Nei grandi grafi (e.g social network), A è sparsa e ciò permette di evitare di memorizzare e computare interazioni inesistenti.

16.2.2 La lista di adiacenza

I grafi possono essere rappresentati in diverse modalità. La **Lista di Adiacenza** (Figura 99), è una delle più efficienti, specialmente per grafi sparsi. La connettività viene descritta tramite delle tuple (coppie di nodi e attributi dell'arco). Questa rappresentazione è preferita in termini di memoria rispetto alla Matrice di Adiacenza quando il grafo è molto sparso, poiché si memorizzano solo le connessioni esistenti. È importante notare che, sebbene l'esempio della figura usi valori scalari, nella pratica i modelli GNN operano con **vettori di caratteristiche** (feature vectors) (embedding) per nodi, archi e contesto globale.

16.3 GRAPH NEURAL NETWORK

Le **Graph Neural Network** (GNN) modelli progettati per apprendere rappresentazioni (embedding) su strutture a grafo. Una GNN effettua

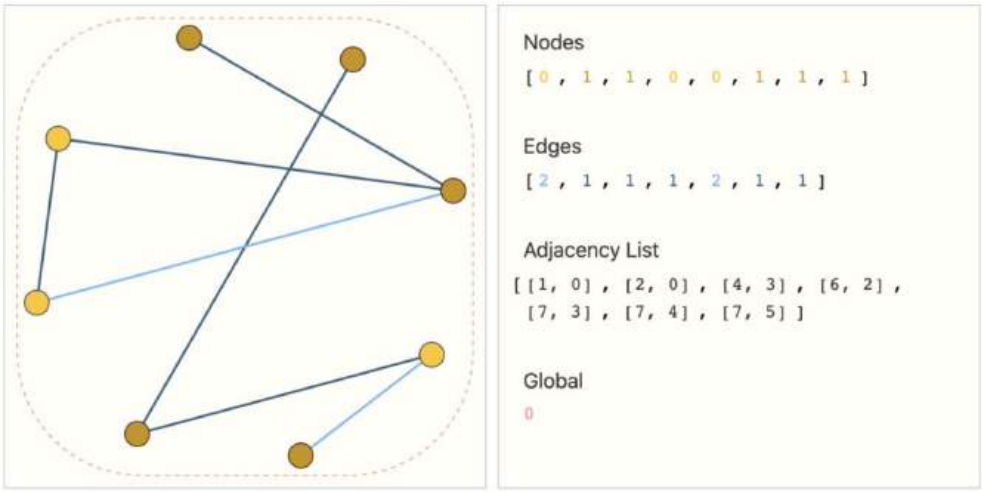


Figura 99: Rappresentazione di un grafo mediante lista di adiacenza. Ogni nodo è collegato ad altri nodi secondo le coppie definite nella Adjacency List. Gli archi rappresentano le connessioni tra i nodi, mentre la lista di adiacenza descrive in forma strutturata quali nodi sono connessi tra loro.

una trasformazione che agisce su tutti gli attributi del grafo, mantenendo invariata la sua struttura topologica. Questi modelli adottano una filosofia **Graph-in**, **Graph-out**: prendono in input un grafo con feature associate a nodi, archi e contesto globale, e restituiscono un grafo trasformato, con la stessa struttura ma con rappresentazioni aggiornate. Queste architetture seguono il framework delle **Message Passing Neural Networks** (MPNN) introdotto da Gilmer et al. (2017) [33], e implementato secondo lo schema delle Graph Nets proposte da Battaglia et al. (2018) [8].

16.3.1 La GNN più semplice

La GNN più elementare ignora la connettività del grafo e sfrutta solo le *feature* iniziali. Applica un **Multilayer Perceptron (MLP) separato** e **condiviso** a ciascun elemento:

- Un MLP per ogni nodo (v), trasformando x_v in h_v;
- Un MLP per ogni arco (e), trasformando x_e in h_e;
- Un MLP per il vettore del contesto globale (u).

Questo *layer* elementare apprende *embedding* aggiornati per ogni componente, ma **non sfrutta** la topologia per scambiare informazioni.

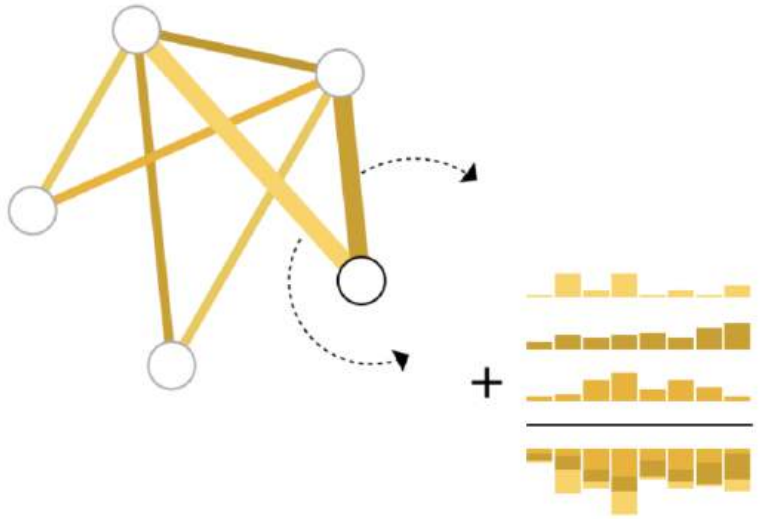


Figura 100: In assenza di informazioni nei nodi, è possibile inferirle aggregando quelle presenti negli archi adiacenti.

16.3.2 Pooling delle informazioni

Nelle GNN, il termine *pooling* (o **aggregazione invariante**) fa riferimento all'operazione di aggregazione delle informazioni provenienti da *insiemi disordinati* di elementi del grafo, solitamente, l'insieme dei vicini di un nodo, con lo scopo di costruire una rappresentazione che non dipenda dall'ordine degli elementi. L'aggregrazione è cruciale nei task a livello di grafo. Il processo si articola infatti seguendo due passaggi fondamentali:

1. **Raccolta:** Si raccolgono gli *embedding* degli elementi correlati (e.g tutti i vicini $\mathcal{N}(v)$;
2. **Aggregazione:** L'insieme raccolto viene aggregato tramite una funzione **invariante all'ordine** (tipicamente una somma, una media o il valore massimo), per ottenere un singolo embedding rappresentativo considerato un *messaggio*.

In un *node-level task* ad esempio, se il nodo non ha feature iniziali, è necessario inferirle aggregando quelle dagli archi adiacenti (Figura 100).

16.3.3 Message Passing

Il **Message Passing** è la filosofia che integra la topologia del grafo nel processo di apprendimento, superando il limite dell'aggregazione non strutturata. Gli elementi del grafo scambiano informazioni con i loro

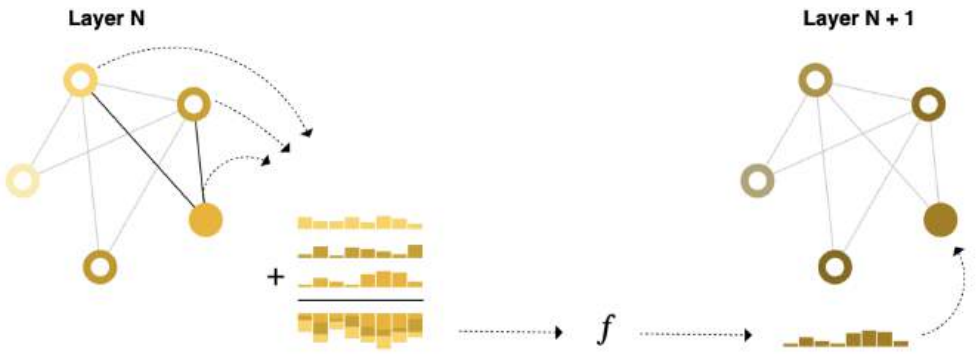


Figura 101: Fasi della Message Passing: aggregazione dei nodi adiacenti, trasformazione tramite MLP, aggiornamento degli embedding.

vicini in modo iterativo, aggiornando i rispettivi embedding in base alla struttura del grafo. Il meccanismo si articola in tre fasi per aggiornare lo stato di un nodo v:

1. **Messaggio (Raccolta):** Si raccolgono gli embedding dei nodi adiacenti $u \in \mathcal{N}(v)$, e se necessario, degli archi adiacenti $e_v u$;
2. **Aggregazione:** I messaggi raccolti vengono aggregati in un unico vettore, tramite somma o media, garantendone l'invarianza all'ordine;
3. **Aggiornamento:** Il risultato dell'aggregazione viene passato a una funzione di aggiornamento, solitamente un MLP o una funzione non lineare, per produrre un nuovo embedding $h_v^{(l+1)}$ per il nodo.

$$h_v^{(l+1)} = \text{UPDATE}^{(l)}\left(h_v^{(l)}, \text{AGGREGATE}^{(l)}(\{m_{vu} \forall u \in \mathcal{N}(v)\})\right)$$

Questo processo è locale: in uno strato (l), il nodo riceve informazioni solo dai vicini a distanza 1. Applicando L strati di Message Passing, il nodo integra informazioni da tutti i vicini a distanza L (il campo ricettivo del nodo).

16.3.4 Rappresentazione globale

Una delle limitazioni del Message Passing è la difficoltà di trasferire efficacemente informazioni tra nodi molto distanti, causando problemi

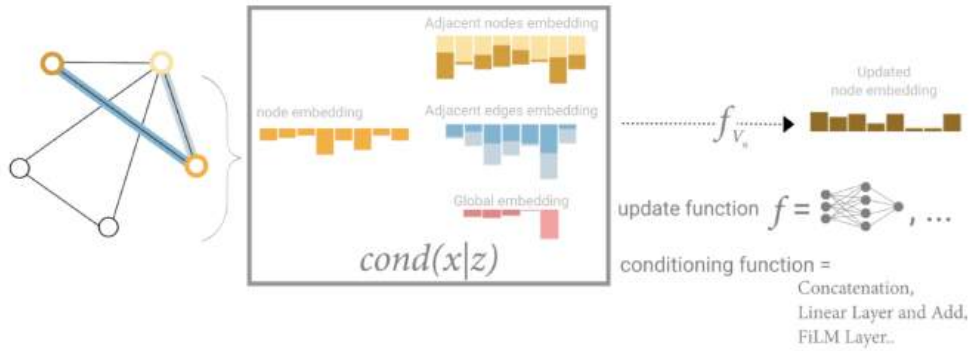


Figura 102: Rappresentazione del Conditioning applicato all'embedding di un nodo, arricchito mediante gli embedding dei nodi e degli archi adiacenti, e quello relativo al contesto globale.

di bottleneck o oversmoothing. Una soluzione è fornire a ciascun nodo una **Rappresentazione Globale** (U), nota anche come **Contesto Globale** o **Master Node**. Questo vettore U è connesso a tutti gli archi e i nodi, agendo come un canale condiviso per lo scambio di informazioni globali tra le componenti del grafo.

Conditioning

Il **Conditioning** è un metodo efficace per arricchire l'embedding di un nodo integrando le informazioni locali con il contesto globale. L'idea si basa sull'aggiornamento di un nodo che non dipenda soltanto dai suoi vicini, ma anche dal contesto globale U:

$$h_v^{(l+1)} = \text{MLP}\left(h_v^{(l)} \| \text{AGGREGATE}(\mathcal{N}(v)) \| u^{(l)}\right)$$

Questa tecnica aiuta a costruire rappresentazioni più informative e robuste, specialmente nei task a livello di grafo (Figura 102).

16.3.5 Filtri polinomiali sui grafi

I **Filtri Polinomiali** sono lo strumento matematico chiave per estendere il concetto di convoluzione dal dominio regolare delle immagini al dominio irregolare dei grafi (approccio spettrale). Nei grafi, non si può usare un filtro fisso. Si usa il **Laplaciano del grafo (L)**, un operatore che funge da analogo del Gradiente/Laplaciano continuo e descrive la connettività.

Il Laplaciano del grafo

Dato un grafo $G = (V, E)$ con matrice di adiacenza A, dove A_{ij} ha valore unitario, qualora i nodi i e j sono connessi, la **Matrice diagonale dei Gradi** D è definita come:

$$D_{ii} = \sum_j A_{ij}$$

Il **Laplaciano non normalizzato** del grafo si ricava tramite:

$$L = D - A$$

Questo operatore misura "quanto un nodo differisce dalla media dei suoi vicini", una versione alternativa è il **Laplaciano normalizzato** utilizzato per fattori di stabilità:

$$\mathbf{L} = I - D^{-1/2} A D^{-1/2}$$

Questo serve a rendere il contributo di ciascun nodo indipendente dal suo grado (cioè dal numero di connessioni). Tale operatore discreto è l'analogo del Laplaciano continuo impiegato in analisi matematica e fisica. Definito L, è possibile costruire polinomi in L:

$$p_w(L) = \sum_{k=0}^{d} w_k L^k$$

dove:

- w_k sono coefficienti, i parametri appresi dal modello;
- d è il grado del polinomio, che controlla la profondità del campo ricettivo.

Applicare l'operazione $p_w(L)x$ consente di propagare le feature $x \in \mathbb{R}^n$ tra nodi del grafo.

Interpretazione: Convoluzione Locale

Nel dominio dei grafi L^k ha l'effetto di collegare ogni nodo con i suoi vicini a distranza k. La convoluzione con un filtro polinomiale agisce come un kernel con raggio d:

- $L^0 x = x$: il nodo vede solo se stesso;
- Lx: il nodo riceve informazioni solo dai suoi vicini diretti;
- $L^2 x$: il nodo integra le informazioni dai nodi entro k-hop.

Esempio: Filtro di grado 1

Il caso più semplice è proprio il Filtro di grado 1:

$$p_w(L) = w_0 I + w_1 L$$

Applicandolo a un vettore x, otteniamo:

$$x' = w_0 x + w_1 L x$$

Dove il primo termine ($w_0 x$), manterrà le feature originali del nodo, mentre il secondo termine ($w_1 Lx$) aggiunge le informazioni dai nodi adiacenti, il Laplaciano agisce come un operatore di differenziazione/smoothing. Questo filtro di grado 1 è la base concettuale della **Graph Convolutional Network** (GCN) [47], dove ogni nodo aggiorna la propria rappresentazione combinando sé stesso e i suoi vicini.

16.3.6 Chebyshev Polynomials (ChebNet)

Nei filtri polinomiali, il calcolo diretto di L^k è computazionalmente costoso e instabile (a causa degli autovalori di L che possono essere grandi). Per risolvere questo problema, Michaël Defferrard et al. [27] introducono i **Polinomi di Chebyshev** come base ortogonale per approssimare il filtro:

$$x' = \sum_{k=0}^{K} w_k T_k(\tilde{L}) x$$

- w_k: Parametri appresi dal filtro (uno per ogni ordine del polinomio);
- $T_k(\cdot) \rightarrow$ Rappresenta la propagazione dell'informazione a distanza k, ed è il k-esimo Polinomio di Chebyshev;
- $\tilde{L} = \frac{2L}{\lambda_{\max}(L)} - I$: È il Laplaciano scalato e traslato per avere autovalori nell'intervallo $[-1, 1]$, garantendo stabilità.

I Polinomi di Chebyshev hanno una forma ricorsiva efficiente, permettendo di calcolare $T_k(\tilde{L})x$ senza calcolare esplicitamente gli autovalori, principale causa di instabilità che portava a un costo computazionale pari a $O(n^3)$.

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{k+1}(x) &= 2xT_k(x) - T_{k-1}(x) \end{aligned}$$

16.4 MODERN GRAPH NEURAL NETWORKS

Le GNN moderne si sono evolute dal framework spettrale (ChebNet, GCN di primo grado) verso l'approccio **spaziale** (Message Passing) che utilizza tecniche di aggregazione più raffinate, mantenendo l'invarianza all'ordine.

16.4.1 Graph Convolutional Network (GCN)

La **Graph Convolutional Network (GCN)** è la rete che ha reso l'approccio GNN pratico. Essa non utilizza i polinomi di Chebyshev per esteso, ma è una semplificazione del filtro di grado 1 (la GCN è un filtro polinomiale limitato a $K = 1$). La formulazione operativa (usando la matrice di adiacenza rinormalizzata $\tilde{A}$) è:

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W^{(l)})$$

La sua forma di *Message Passing* è la seguente:

$$h_v^{(k)} = \sigma\left(W^{(k)} \cdot \sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{\sqrt{\tilde{D}_{vv}\tilde{D}_{uu}}} h_u^{(k-1)}\right)$$

- La **somma pesata** (normalizzata dal grado $\tilde{D}$) sulle feature dei vicini è la fase di **aggregazione**;
- L'operatore **lineare** (W) seguito dalla non linearità (σ) è la fase di **aggiornamento**;
- Le matrici $W^{(k)}$ sono condivise tra i nodi, garantendo la scalabilità.

16.4.2 Graph Attention Network (GAT)

La **Graph Attention Network (GAT)** si distacca dalla normalizzazione statica della GCN e introduce un **meccanismo di attenzione** che assegna pesi dinamici all'aggregazione:

$$h_v^{(k)} = f^{(k)}\left(W^{(k)} \cdot \sum_{u \in \mathcal{N}(v) \cup \{v\}} \alpha_{vu}^{(k-1)} h_u^{(k-1)}\right)$$

- $\alpha_{vu}^{(k-1)}$: Coefficiente di attenzione appreso;

- **Calcolo dell'attenzione:** Il punteggio di attenzione α_{vu} è calcolato tramite una funzione di compatibilità (e.g una single-layer feedforward network) tra i nodi v e u e normalizzato con **softmax** sui vicini:

$$\alpha_{vu}^{(k)} = \frac{\exp(e_{vu}^{(k)})}{\sum_{w \in \mathcal{N}(v)} \exp(e_{vw}^{(k)})}$$

Questo permette alla GAT di dare maggiore importanza ai vicini più rilevanti per il task, aumentando l'espressività rispetto alla GCN.

16.4.3 Graph SAGE

Graph SAGE (Graph Sample and AggregatE) è un'architettura progettata per l'apprendimento induttivo e la scalabilità su grafi di grandi dimensioni. Invece di apprendere un embedding fisso per ogni nodo, GraphSAGE apprende una funzione di aggregazione generale che può essere applicata anche a nodi non visti durante il training (apprendimento induttivo). Il principio chiave è il campionamento locale: ogni nodo aggiorna la sua rappresentazione aggregando le feature di un sottoinsieme campionato dei suoi vicini:

$$h_v^{(k)} = \sigma\left(W^{(k)} \cdot \text{AGGREGATE}^{(k)}\left(\{h_v^{(k-1)}\} \cup \{\text{SAMPLE}(\, h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\}\right)\right)$$

Dove $\text{AGGREGATE}^{(k)}$ può essere una media, una somma o una rete neurale più complessa (e.g LSTM o max-pooling). L'approccio di Graph SAGE è particolarmente utile per scenari di *streaming* o *large-scale learning*, poiché permette di scalare a grafi di grandi dimensioni grazie al campionamento locale e al riutilizzo della stessa funzione aggregatrice.

16.4.4 GIN

La **Graph Isomorphism Network (GIN)** è stata proposta con l'obiettivo di massimizzare il potere discriminante delle GNN, equiparandolo a quello del Weisfeiler-Lehman Test (WL Test). GIN dimostra che le GNN più potenti non hanno bisogno di meccanismi complessi come l'attenzione o la normalizzazione statica, ma di una semplice ma espressiva funzione di aggregazione e aggiornamento:

$$h_v^{(k)} = \text{MLP}^{(k)}\left((1 + \epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}\right)$$

Dove:

- $\epsilon^{(k)}$: è un parametro che bilancia il contributo del nodo rispetto ai vicini;
- $\text{MLP}^{(k)}$: Applica una trasformazione non lineare sulla somma, fungendo da hashing delle etichette del WL Test;
- $\sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}$: Semplice somma dei vicini (un'operazione invariante all'ordine e massimamente espressiva per il WL Test).

GIN ottiene la massima capacità di distinguere strutture complesse pur mantenendo una formula di aggregazione apparentemente semplice.

Tabella 9: Confronto tra le principali architetture di Graph Neural Networks.

Caratteristica	**GCN**	**GAT**	**GraphSAGE**	**GIN**
Tipo Aggregazione	Media normalizzata	Attenzione pesata	Custom (media, LSTM, pooling)	Somma + MLP
Pesatura dei vicini	Statica (matrice normalizzata)	Dinamica (self-attention)	Parametrica o fissa	Fissa (somma)
Apprendimento Induttivo	No	Parziale	Sì	No
Espressività	Limitata	Superiore a GCN	Dipende dall'aggregatore	Massima (WL test)
Funzione di aggiornamento	Lineare + ReLU	Lineare + Attenzione	MLP + Aggregazione	MLP su somma
Parametri appresi	W	W, Attenzione	W, Aggregatore (opz.)	MLP, ϵ
Sensibile alla struttura del grafo	Sì	Sì	Sì	Sì (con maggiore discriminatività)
Scalabilità	Buona	Limitata (per grafi molto grandi)	Alta (con campionamento)	Media
Proprietà chiave	Semplicità	Flessibilità	Generalizzazione	Massima capacità discriminante

17 ENERGY-BASED MODELS (EBM)

Gli **Energy-Based Models** (EBM) costituiscono un paradigma di modellazione che si distacca dagli approcci discriminativi o generativi classici. Invece di apprendere direttamente una funzione esplicita di predizione normalizzata, un **EBM** definisce un'*energia* associata a ogni configurazione possibile di input x e output y, misurandone la compatibilità. Questa energia viene definita tramite una funzione scalare $E(x, y)$, il cui obbiettivo è minimizzarla, definito come *obbiettivo inferenziale*. Gli **Energy Based Model**, si articolano principalmente in due fasi: quella di **allenamento** (o training), solitamente molto costosa, e quella di **inferenza**.

17.1 TRAINING

Nella fase di allenamento, il modello impara la funzione d'energia tramite l'uso di un input e di un'etichetta di riferimento. L'idea chiave è che le configurazioni "Plausibili" abbiano bassa energia, mentre le configurazioni "implausibili" abbiano alta energia. Effettuare il training di un EBM richiede la minimizzazione di una funzione di costo che coinvolge questa energia, spesso confrontando l'energia di esempi reali (positivi) con quella di esempi generati (negativi).

17.2 INFERENZA

Una volta che il modello è stato allenato, possiamo usarlo per l'inferenza, ossia determinare quale output y (o quale input x) minimizza l'energia, in altre parole, trovare la configurazione più probabile secondo il modello. Ad esempio, in un EBM discriminativo $E(x, y)$, possiamo usare il modello per inferire l'etichetta $\hat{y}$ più probabile per un dato x:

$$\hat{y} = \arg\min_{y} E(x, y) \tag{7}$$

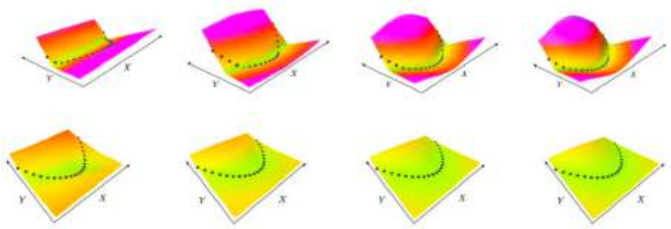


Figura 103: Applicazione dell'inferenza a seguito di un buon training (sopra), e l'esito a seguito di un pessimo training (sotto).

In molti casi pratici il modello EBM è già allenato da qualcun altro, e viene utilizzato solo per l'inferneza (non vengono modificati i pesi) per selezionare le configurazioni a bassa energia. Alcuni esempi includono:

- Trovare la label più probabile per un'immagine;
- Campionare un'immagine simile a un target;
- Scegliere tra alternative quella più compatibile con un certo contesto.

Il fenomeno dell'inferenza viene solitamente visualizzato tramite un'interpretazione grafica della superficie energetica (Figura 103). Se la superficie è piatta, l'allenamento non è stato corretto, poiché non identificherà delle differenze significative tra le varie zone. Se invece la superficie risulta essere ricurva (con minimi ben definiti), il modello sarà stato allenato bene, distinguendo le zone a bassa energia da quelle ad alta. Il passo successivo sarà quello di trovare i valori precisi che generano il minimo.

17.2.1 Quando usare un EBM

I casi d'uso degli EBM includono:

- Problemi che richiedono calcoli complessi per determinare l'output (e.g modelli con molteplici variabili accoppiate);
- Situazioni con uscite multiple plausibili (problemi multimodali o one-to-many);
- Inferenza come soddisfacimento di vincoli, tipico di traduzioni linguisticamente corrette o trascrizioni fonetiche coerenti.

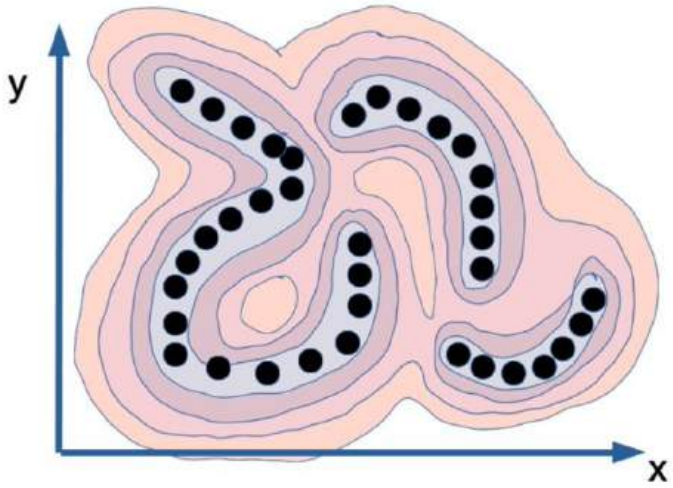


Figura 104: Visualizzazione dell'energia, vicino ai datapoint abbiamo un'energia più bassa, mentre man mano che ci si allontana l'energia aumenta.

17.3 MODELLI ESPLICITI VS IMPLICITI

Un modello *feed-forward* classico è una funzione **esplicita** $y = f(x)$, che calcola direttamente y a partire da x. Al contrario, un EBM definisce una relazione **implicita** tra x e y tramite la funzione energetica $E(x, y)$. L'inferenza consiste nel risolvere il problema di ottimizzazione ($\arg\min_y E(x, y)$). Questo approccio implicito consente l'esistenza di molteplici y compatibili con uno stesso x, rendendo l'approccio adatto per compiti **multimodali**. Se y è continuo, è essenziale che E sia *differenziabile* e *liscia*, per consentire l'uso di algoritmi di ottimizzazione basati sul gradiente.

17.4 EBM PER PREDIZIONE MULTIMODALE

Due degli approcci più utilizzati che estendono il paradigma classico degli EBM, sfruttati in ambiti come multimodalità e modellazione generativa profonda, sono:

- **Joint Embedding Architectures:** basate sulla distanza nello spazio delle feature;
- **Latent Variable Models:** introducono variabili latenti z per rappresentare fattori di variazione non osservati.

17.4.1 Joint Embedding

In un **Joint Embedding** EBM, la funzione di energia è definita sullo spazio latente condiviso tra input e output, tipicamente tramite embedding. È l'approccio alla base dei modelli contrastivi e multimodali. Algoritmi noti includono *Siamese Networks* e tecniche di *Metric Learning* [16, 22, 36]. Il vantaggio principale è l'assenza di necessità di ricostruzione a livello di pixel. Considerando x come un'immagine di input e y come output testuale e $f(x)$ e $g(y)$ due reti neurali, esse proiettano x e y in uno spazio comune l'energia può essere definita come:

$$E(x, y) = -\langle f(x)\,,\, g(y)\rangle \tag{8}$$

L'energia pertanto viene ottenuta come il valore negativo del prodotto scalare (o cosine similarity) dei due embedding. L'energia risulterà di basso valore se x e y sono compatibili, indicando una forte similarità nello spazio latente. Esempi applicativi includono:

- DeepFace [83];
- PIRL [55], MoCo [37];
- SimCLR [19].

L'obbiettivo è *minimizzare* l'energia (massimizzare la similarità) per le coppie vere (positive) e **massimizzare** l'energia (minimizzare la similarità) per le coppie false (negative).

17.4.2 Latent Variable EBMs

In questa tipoliga di modello vengono sfruttate le variabili latenti z, le quali parametrizzano lo spazio delle predizioni. Idealmente, z rappresenta fattori indipendenti non direttamente osservabili. Per evitare che la variabile latente z prenda il soppravvento sull'informazione di input e porti a una superficie energetica piatta, la sua capacità informativa deve essere *minimizzata* (regolarizzata), in modo da influenzare solo parzialmente la predizione.

Esempi applicativi si riscontrano nella lettura di parole scritte a mano (segmentazione dei caratteri) e nella comprensione del parlato (segmentazione in fonemi o parole).

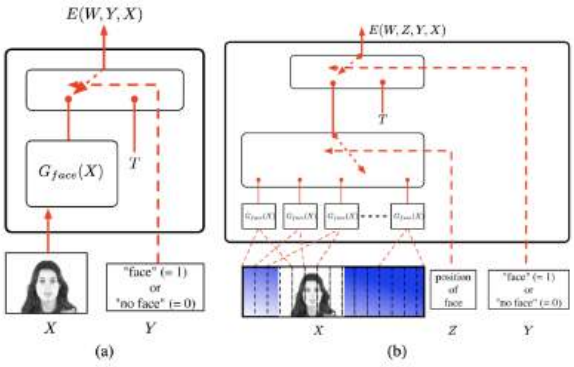


Figura 105: Confronto tra due architetture Energy-Based: (a) un EBM discriminativo semplice, e (b) un EBM con variabile latente. Nel primo caso, la funzione di energia $E(W, Y, X)$ valuta direttamente la compatibilità tra l'immagine X e l'etichetta binaria Y. Nel secondo caso, viene introdotta una variabile latente Z che rappresenta la posizione del volto, e il modello calcola l'energia congiunta $E(W, Z, Y, X)$. Questo consente al modello di apprendere rappresentazioni strutturate e di migliorare la localizzazione e classificazione del volto all'interno dell'immagine.

17.4.3 EBM con variabili latenti condizionali

Per limitare la capacità informativa di z e prevenire l'appiattimento della superficie energetica, è fondamentale introdurre un regolarizzatore $R(z)$. Altrimenti, ogni y potrebbe essere ricostruito perfettamente, rendendo $E(x, y)$ non informativo. Alcuni approcci di regolarizzazione includono:

- Quantizzazione/discretizzazione dello spazio di z;
- Penalizzazioni tipo L_0, L_1;
- Inibizione laterale;
- Aggiunta di rumore controllato alla variabile latente.

17.5 TRAINING DEGLI ENERGY-BASED MODELS

L'addestramento di un **Energy-Based Model** (EBM) consiste nel parametrizzare una funzione di energia $E(x, y; \theta)$ in modo che le coppie

corrette (quelle osservate nei dati) abbiano energia più bassa rispetto a tutte le altre combinazioni possibili:

$$E(x^{(i)}, y^{(i)}) < E(x^{(i)}, y), \quad \forall y \neq y^{(i)} \tag{9}$$

In questo modo, il modello impara una funzione energetica che distingue correttamente le coppie plausibili da quelle implausibili.

17.5.1 Due approcci principali

L'addestramento di un EBM può essere condotto secondo due famiglie principali di metodi:

1. **Metodi contrastivi:** minimizzano l'energia associata alle coppie corrette e la massimizzano per le coppie errate (e.g *Contrastive Divergence* e *Margin-Based Loss*);

2. **Metodi con Regolarizzazione o Architetturali:** impongono vincoli sulla forma di E o penalizzano regioni troppo ampie a bassa energia, evitando di modellare esplicitamente coppie negative e garantendo una funzione più liscia e generalizzabile.

17.5.2 Problemi con il massimo di verosimiglianza

L'approccio probabilistico tradizionale agli EBM mira a modellare la distribuzione dei dati tramite la massimizzazione della verosimiglianza. Tuttavia, questo equivale a creare un *"canyon"* infinitamente stretto e profondo attorno alla distribuzione dei dati osservati. Questo porta a due problematiche principali:

- La funzione energetica diventa troppo **non liscia**, rendendo difficile la discesa del gradiente;

- Il modello tende a **overfittare**, concentrandosi eccessivamente sui dati osservati.

Per questo motivo, spesso si preferiscono approcci **regolarizzati o contrastivi**, che mantengono una funzione energetica più stabile.

17.5.3 Distribuzione di Gibbs

In un'ottica probabilistica, la funzione di energia può essere interpretata come il negativo del logaritmo di una distribuzione non normalizzata:

$$P(y|x) \propto e^{-\beta E(x,y)}.$$

dove:

- β è un parametro di scala chiamato *Inverse Temperature*;
- $E(x, y)$ è l'energia associata alla coppia (x, y);

Il termine di normalizzazione (partizione di Gibbs) è dato da:

$$Z(x) = \int_{y'} e^{-\beta\, E(x,y')} dy' \tag{10}$$

Tuttavia, $Z(x)$ è notoriamente **molto costoso** da calcolare, poiché richiede di integrare su tutte le possibili y'. Per questo, nella pratica, si evita di stimare $P(y|x)$ in forma esplicita, e si preferiscono strategie di ottimizzazione basate su confronti di energia.

17.6 METODI CONTRASTIVI

I **metodi contrastivi** addestrano il modello riducendo l'energia per le coppie corrette (positive) e aumentandola per tutte le altre (negative). In termini probabilistici, si può scrivere la probabilità come:

$$P(y \mid x) = -\frac{e^{-\beta\, E(x,y;\theta)}}{\int_{y'} e^{-\beta\, E(x,y';\theta)} dy'} \tag{11}$$

e la funzione di perdita (log-verosimiglianza negativa) corrispondente è:

$$L(x, y; \theta) = E(x, y; \theta) + \frac{1}{\beta} \log \int_{y'} e^{-\beta\, E(x,y;\theta)} dy' \tag{12}$$

La prima parte riduce l'energia dei campioni corretti, mentre il secondo termine (che include il logaritmo della funzione di partizione $Z(x)$) penalizza il modello se assegna energia bassa anche ad altre configurazioni non corrette.

17.6.1 Esempi di contrastive training

- **Contrastive Divergence (CD):** si utilizza un set di campioni reali e un set di campioni "negativi" generati tramite catene di Markov (MCMC);
- **Loss a margine:** funzioni come la *Square-Square loss* o la *Negative Log-Likelihood* introducono un margine esplicito tra energia positiva e negativa;
- **Embedding contrastivo:** nello spazio delle *feature*, i campioni positivi vengono spinti vicini e quelli negativi allontanati.

17.7 EMBEDDING NON CONTRASTIVO

Gli **embedding non contrastivi** superano la necessità di generare "esempi negativi" o di effettuare *negative mining*, sfruttando due reti quasi identiche (*online* e *target*) con pesi aggiornati con strategie lente o mediate. Esempi noti:

- **SimSiam** [20];
- **BYOL** (*Bootstrap Your Own Latent*).

Questi metodi si basano sull'idea di *auto-supervisione*, imparando rappresentazioni coerenti da due diverse *viste* dello stesso campione, anche senza etichette o esempi negativi espliciti.

17.8 SELF-SUPERVISED LEARNING (SSL)

Il **Self-Supervised Learning** (SSL) nasce dall'osservazione del modo in cui gli esseri umani apprendono, esplorando l'ambiente e costruendo internamente rappresentazioni del mondo a partire da osservazioni parziali. Un modello SSL impara a:

- Predire parti mancanti o corrotte dell'input a partire dalle parti osservate (e.g inpainting);
- Ricostruire dati incompleti o rumorosi;
- Comprendere relazioni temporali o spaziali implicite.

L'obiettivo è imparare una rappresentazione utile del mondo senza etichette, sfruttando solo la struttura intrinseca dei dati stessi.

17.8.1 Applicazioni di SSL

Esempi moderni di apprendimento *self-supervised* includono:

- **Wav2Vec 2.0:** addestrato su 960 ore di parlato non etichettato, ha raggiunto prestazioni competitive con modelli che richiedono molta più supervisione esplicita;
- **BERT** e **GPT**: basati sulla predizione di token mascherati o successivi (task di auto-supervisione), hanno rivoluzionato il NLP generando rappresentazioni linguistiche estremamente potenti.

18 FLOW MATCHING & SD3

Un modello generativo ha come scopo: modellare la distribuzione di probabilità di un insieme di dati. Supponiamo di avere un dataset con i seguenti dati osservati $x = \{x^{(1)}, x^{(2)}, \dots, x^{(N)}\}$. L'obbiettivo da noi prefissato, è quello di: apprendere una distribuzione di probabilità tale che la distribuzione generata sia il più possibile simile a quella reale dei dati.

$$p_{\text{model}}(x) \approx p_{\text{data}}(x)$$

Dove $p_{\text{data}}(x)$ è la distribuzione reale da cui i dati sono stati campionati (a noi sconosciuta), mentre $p_{\text{model}}(x)$ è la distribuzione appresa dal modello generativo. Una volta che il modello apprende questa distribuzione, il suo compito principale è quello di generare nuovi dati realistici, coerenti con quelli del dataset originale. Inoltre, può essere in grado di completare parti mancanti o condizionare la generazione a partire da informazioni aggiuntive.

18.1 MAPPATURA DIRETTA

Un primo approccio è stato adottare una **Mappatura Diretta** tra lo spazio latente e lo spazio dei dati osservati. Costruendo un modello in grado di traformare un vettore latente z in un campione realistico x, deformando la distribuzione latente $p(z)$ fino a sovrapporla (o avvicinandola) alla distribuzione reale $p_{\text{data}}(x)$. Questo approccio risulta essere molto rigido per fenomeni complessi con alta dimensionalità. Servirebbe un processo graduale e continuo, che trasformi progressivamente una distribuzione in un'altra. Da questa esigenza nasce il concetto di **Flusso Continuo**, modellato come un processo di Markov continuo nel tempo, in cui lo stato evolve secondo una certa velocità $v_t(x)$.

18.2 FLUSSO E VELOCITÀ

Il **Flusso** descrive come una distribuzione evolve nel tempo. Ogni punto x "si muove" nello spazio dei dati seguendo un campo di velocità $v_t(x)$. La dinamica del processo può essere modellata tramite un'equazione differenziale ordinaria (ODE):

$$\frac{dx}{dt} = v_t(x), \quad x(0) \sim p_{\text{data}}(x) \tag{13}$$

Il modello grazie a questa caratteristica, imparera a spostare i punti nel tempo, per poter trasformare una distribuzione iniziale p_0 (come un rumore gaussiano) in una distribuzione finale p_1 (delle immagini reali).

18.3 PROCESSI DI MARKOV E TRANSIZIONE CONTINUA

Per passare da un modello a mappatura diretta a uno a flusso continuo, si introduce l'idea di un **Processo di Markov**. Una catena di Markov $(x_0, x_1, \ldots, x_T)$ è un processo stocastico in cui avviene la seguente evoluzione:

$$p(x_{t+1} \mid x_t, x_{t-1}, \ldots, x_0) = p(x_{t+1} \mid x_t) \tag{14}$$

cioè lo stato futuro dipende solo ed esclusivamente dallo stato corrente, invece che dipendere da tutti gli stati precedenti. Nel contesto generativo invece accade:

- Nel *processo in avanti* $q(x_t \mid x_{t-1})$ si aggiunge progressivamente rumore ai dati reali fino a ottenere rumore puro (x_T);
- Nel *processo inverso* $p_\theta(x_{t-1} \mid x_t)$ il modello apprende a invertire tale rumore per rigenerare dati realistici.

Questo schema è alla base dei **Modelli di Diffusione**, visti nel Capitolo 15 (dove il processo è stocastico), e dei **Flow Models**, dove l'evoluzione è continua e deterministica. Il **Flow Matching** nasce proprio per unificare queste due prospettive.

18.4 I MODELLI DA CUI CI DISCOSTIAMO

I modelli dai quali il Flow Matching trae ispirazione, e dai quali si discosta, sono principalmente:

- **Modelli di Diffusione**: basati su processi stocastici (catene di Markov);
- **Normalizing Flows:** e le loro versioni continue (**Continuous Normalizing Flows**), basati invece su trasformazioni deterministiche e invertibili.

18.4.1 Normalizing Flows

I **Normalizing Flows** (NF) sono modelli a **distribuzione esplicita**, cioè permettono di calcolare esattamente la densità di probabilità dei dati generati grazie al cambiamento di variabili. Seguono un approccio deterministico e invertibile: una distribuzione semplice (e.g una gaussiana) viene trasformata in una distribuzione complessa (e.g immagini) mediante trasformazioni invertibili f_θ.

$$\log p(x) = \log p_z(f_\theta^{-1}(x)) + \log \left| \det \left(\frac{\partial f_\theta^{-1}(x)}{\partial x} \right) \right|. \tag{15}$$

Il limite di questi modelli è la complessità computazionale del determinante jacobiano, che rende difficile l'applicazione a reti molto profonde o ad alta dimensionalità.

18.4.2 Continuous Normalizing Flows

Per superare il limite del determinante, i **Continuous Normalizing Flows** (CNF) [18] modellano la trasformazione come un flusso continuo nel tempo:

$$\frac{dx}{dt} = f_\theta(x, t). \tag{16}$$

Questi modelli utilizzano un'ODE per "spingere" la distribuzione nel tempo. Il log-determinante del Jacobiano viene calcolato come integrale nel tempo, sfruttando l'equazione di continuità, migliorando così l'efficienza rispetto ai flussi discreti.

18.5 FLOW MATCHING

Il **Flow Matching** (FM) [94] nasce come ponte concettuale tra i modelli di diffusione (stocastici) e i Continuous Normalizing Flow (deterministici). Invece di simulare interi flussi e calcolare likelihood complesse, il Flow Matching impara direttamente il campo di velocità $v_t(x)$ che collega due distribuzioni. Immaginiamo di voler insegnare a un'auto a guida autonoma a muoversi dal punto x_0 al punto x_1. Il modello traccia una linea retta tra i due punti e, in un punto intermedio γ_t, deve prevedere la direzione e la velocità corretta per restare sulla traiettoria. La velocità target v_t è nota, mentre il modello predice $\hat{v}_t$; l'allenamento consiste nel ridurre la differenza tra le due.

$$\mathcal{L}_{FM} = \mathbb{E}_{x_0,x_1,t}\left[\|\hat{v}_t(\gamma_t) - v_t(\gamma_t)\|^2\right]. \tag{17}$$

18.5.1 Marginalization Trick

Il **Marginalization Trick** serve a evitare la simulazione completa delle traiettorie. Si considera la distribuzione condizionata $\psi_t(x \mid x_0, x_1)$, che rappresenta la probabilità di trovarsi in un certo punto γ_t lungo il percorso. Si calcola quindi la velocità corretta in quel punto e si marginalizza il processo, rendendo l'addestramento efficiente:

$$\mathcal{L}_{FM} = \mathbb{E}_{x_0,x_1}\left[\mathbb{E}_t\left[\mathbb{E}_{\gamma_t\sim\psi_t(\cdot|x_0,x_1)}\|f_\theta(\gamma_t,t) - v_t(\gamma_t|x_0,x_1)\|^2\right]\right]. \tag{18}$$

18.5.2 Percorsi Affini e Gaussiani

Il parametro γ_t può essere calcolato anche tramite degli studi differenti, tramite l'utilizzo dei Percorsi Affini o quelli Gaussiani:

PERCORSO AFFINE

$$\gamma_t = (1-t)\,x_0 + t\,x_1 \tag{19}$$

È una traiettoria lineare a velocità costante: il modo più semplice per collegare due distribuzioni.

PERCORSO GAUSSIANO

$$\gamma_t = (1-t)\,x_0 + t\,x_1 + \sqrt{t\,(1-t)}\cdot\epsilon, \qquad \epsilon \sim \mathcal{N}(0,I) \tag{20}$$

È una linea retta perturbata da rumore gaussiano, con massima incertezza a metà percorso. Questo rende il modello più robusto e più simile

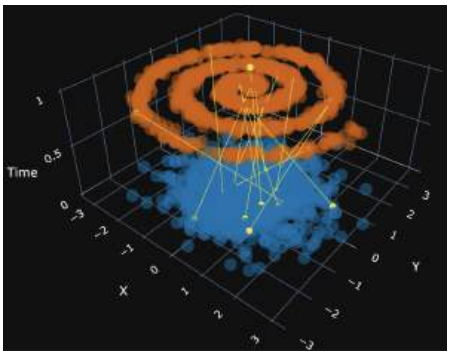


Figura 106: Visualizzazione 3D della procedura di Training, in blu la distribuzione iniziale dei datapoint, in arancione la distribuzione target.

al comportamento dei modelli di diffusione.Come se fosse un aereo che vola da una città all'altra. La rotta principale è una linea retta, ma piccole turbolenze lo fanno deviare leggermente.

18.5.3 Metodologia di Addestramento

Il training del Flow Matching può essere implementato in tre modalità:

- **Simulation-Free:** ottimizzazione diretta del campo vettoriale, evitando simulazioni costose;
- **Gradient-Based:** ottimizzazione tramite discesa del gradiente dei parametri del modello;
- **Conditional Flow Matching:** introduce un obiettivo condizionale che garantisce stime non distorte dei gradienti e un addestramento efficiente.

In generale, si campionano coppie di punti dalle distribuzioni di origine e di destinazione, si calcolano posizioni e velocità intermedie lungo le traiettorie, e si addestra la rete a predire le velocità corrette, approssimando il campo di flusso globale in prima analisi in maniera rettilinea Figura 106. Il **Problema della Confusione** è una problematica che si può riscontrare, nel momento in cui accoppiamo due punti fra rumore e target. I percorsi rettilinei quasi certamente si incroceranno e nel punto di incrocio, il modello riceve segnali contraddittori non sapendo quale traiettoria seguire. Il modello tuttavia non è progettato per imparare i percorsi individuali, ma per imparare un campo di velocità complessivo una "mappa del vento" che sposta l'intera distribuzione da una forma all'altra.

Oss: 18 *Questo fenomeno può essere associato al versare una caraffa d'acqua in un bicchiere, dove la caraffa è la nostra distribuzione iniziale, mentre il bicchiere quella target. Se ci concentrassimo su due gocce d'acqua, i loro percorsi saranno caotici e si incroceranno, mentre il flusso complessivo dell'acqua ha una sola direzione non incorciandosi con se stesso.*

Pertanto il modello neurale, vedendo milioni di esempi, ignora il caos individuale e impara il flusso aggregato. Il codice di seguito nella classe `VelocityNet` è il "cervello" che impara questa mappa del vento. Prende in input una posizione x e un tempo t e restituisce la velocità che un punto dovrebbe avere in quel punto dello spazio-tempo:

```
import torch.nn as nn
import torch.nn.functional as F

class VelocityNet(nn.Module):
    def __init__(self, input_dim, h_dim=64):
        super().__init__()
        self.fc_in  = nn.Linear(input_dim + 1, h_dim)
        self.fc2    = nn.Linear(h_dim, h_dim)
        self.fc3    = nn.Linear(h_dim, h_dim)
        self.fc_out = nn.Linear(h_dim, input_dim)

    def forward(self, x, t, act=F.gelu):
        t = t.expand(x.size(0), 1)
        # Ensure t has the correct dimensions
        x = torch.cat([x, t], dim=1)
        x = act(self.fc_in(x))
        x = act(self.fc2(x))
        x = act(self.fc3(x))
        return self.fc_out(x)

# Instantiate the model
input_dim = 2
model = VelocityNet(input_dim)
```

La parte più complessa del Flow Matching è come alleniamo questa rete, per ogni training step dobbiamo:

1. Campionare punti casuali dalle distribuzioni di origine e di destinazione e accoppiare i punti;
2. Campionare tempi casuali compresi tra 0 e 1;
3. Calcolare le posizioni in cui questi punti si troverebbero in quei momenti se si muovessero a velocità costante dalla sorgente al bersaglio;

4. Calcolare la velocità che avrebbero in quei punti se si muovessero a velocità costante, effettuata dalla rete neurale, per tanto dovrà scoprire da se la velocità costante;

5. Addestrare la rete a prevedere queste velocità, che finiranno per "cercare la media" quando la rete dovrà fare lo stesso per molti punti.

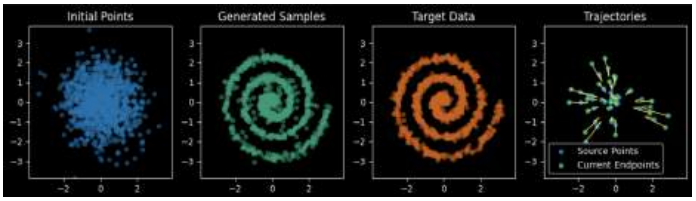


Figura 107: Rappresentazione di ciò che accade dopo il training alle traiettorie che seguiranno i punti fino a giungere alla distribuzione target.

Anche se le traiettorie risultino essere lisce e non incrociate, la loro curvatura significa che dobbiamo integrare lentamente e con attenzione per evitare di accumulare errori significativi. Ci sono stati dei paper i quali hanno approfondito questo dettaglio [94], offrendo un modo efficace per accelerare l'integrazione "raddrizzando" le traiettorie curve, attraverso un metodo che prende il nome di **Reflow**.

18.5.4 Reflow

Una volta addestrato, il modello di Flow Matching apprende un campo di velocità che sposta gradualmente la distribuzione di partenza verso quella dei dati reali. Tuttavia, le traiettorie che ne risultano, pur essendo lisce e coerenti, tendono spesso a essere **ricurve**. Per ottenere una trasformazione accurata lungo queste curve, durante l'inferenza è necessario integrare l'ODE con molti piccoli passi temporali, rendendo il processo di generazione lento e computazionalmente costoso. Il **Reflow** è una procedura iterativa che ha come scopo quello di "*raddrizzare*" queste traiettorie, ossia di ridurre la loro curvatura pur mantenendo la correttezza della trasformazione complessiva. In sostanza, il Reflow riutilizza il campo di velocità appreso in un primo addestramento per generare traiettorie più dirette e riaddestrare il modello su percorsi più lineari, migliorando così l'efficienza del flusso. Il processo si articola in tre fasi:

1. **Primo addestramento:** si addestra un modello di Flow Matching standard utilizzando percorsi rettilinei tra coppie di punti (x_0, x_1). Il modello impara un campo di velocità $v_t(x)$ che trasforma correttamente la distribuzione di partenza, ma le traiettorie risultano comunque curve a causa della natura complessa dei dati;

2. **Generazione del nuovo target (Reflow step):** si utilizza il modello addestrato per simulare le traiettorie complete, partendo da x_0 e seguendo il flusso fino a un nuovo punto terminale x_1'. Questa traiettoria, prodotta dal modello, rappresenta una trasformazione corretta ma non rettilinea.

3. **Riaddestramento:** si costruisce un nuovo dataset utilizzando le coppie (x_0, x_1') ottenute nel passo precedente, e si riaddestra il modello imponendo che il nuovo flusso segua la linea retta che collega x_0 a x_1'. In questo modo il modello impara a generare campi di velocità che approssimano direttamente la trasformazione desiderata con percorsi più brevi e quasi lineari.

L'effetto del Reflow è quello di ridurre la curvatura media delle traiettorie, producendo flussi più "dritti" e regolari. Questo ha due conseguenze fondamentali:

- **Efficienza:** poiché le traiettorie sono più lineari, è possibile integrare l'ODE con un numero di passi temporali minore, accelerando notevolmente l'inferenza;

- **Stabilità:** percorsi meno curvi riducono l'accumulo di errore numerico durante l'integrazione, migliorando la fedeltà dei campioni generati.

In altre parole, il Reflow "*distilla*" la conoscenza del modello originale in una versione più diretta e computazionalmente efficiente, senza alterare la qualità della generazione. Le traiettorie risultanti tendono a muoversi direttamente dalla configurazione iniziale a quella finale, evitando la fase di "aggregazione" intermedia tipica del Flow Matching standard (Figura 108).

Sono presenti online numerose animazioni, che mettono in luce come le traiettorie con l'utilizzo del Reflow, siano più lineari rispetto a quelle prive di esso, inoltre il passaggio dalle configurazioni iniziali a quelle finali nel Flow Matching, subiscono una sorta di aggregamento iniziale prima di giungere nella posizione finale, cosa che non succede

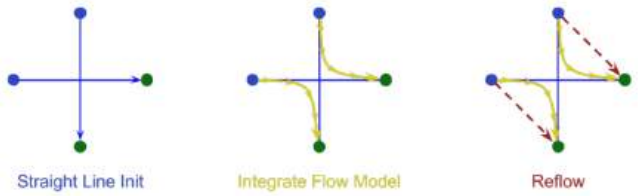


Figura 108: Idea alla base del Reflow: il nuovo flusso viene addestrato a seguire traiettorie più lineari tra i punti di origine e destinazione.

con il Reflow, le quali vanno direttamente nella posizione finale dei valori target.

18.6 T5

T5 sta per **Text-To-Text Transfer Transformer**, ed è un modello sviluppato da Google Research nel 2020 [68], il suo *key-concept* è semplice ma potente, ossia, tutti i compiti di *Natural Language Processing*, vengono formulati come trasformazioni di testo in testo. T5 usa un'architettura

Task	Input	Output
Traduzione	`translate English to German: How are you?`	`Wie geht es dir?`
Riassunto	`summarize: The book was about...`	`It was a book about...`
Classificazione	`classify sentiment: I love this movie!`	`positive`
Domanda-Risposta	`question: What is the capital of France? context: France is a country...`	`Paris`

Tabella 10: Esempi di task gestiti da T5 con input testuali specifici, quì tutto è testo in input, e tutto è testo in output.

Transformer Encoder-Decoder, simile a quella dei modelli di traduzione neurale [90], l'Encoder legge l'input testuale e lo rappresenta come sequenza di vettori, mentre il Decoder genera l'output testuale token per token, usando l'attenzione sui vettori prodotti dall'Encoder, dunque è diverso da BERT e GPT come abbiamo visto nel capitolo 14 poiché è come se combini i due modelli.

18.6.1 Masked Attention

T5 utilizza la **Masked Attention**, nella Self-Attention classica ogni token può guardare tutti gli altri token della sequenza. In questo caso i token,

non possono guardare in avanti nella sequenza, nel Decoder si genera un testo token per token e durante l'addestramento si ha già tutto l'output target a disposizione, ma bisogna impedire che il modello effettui delle scorrettezze guardando ai token futuri.

18.6.2 Obbiettivi di Training

T5 non è addestrato su una Next-Token Prediction o Masked Token Prediction Standard, invece è addestrato sulla **Span Corruption**, la quale rimuove casualmente degli span (frammenti di testo) dal testo, facendo diventare l'input il testo con span mascherati. L'output saranno gli span rimossi, permettendo al modello di imparare a ricostruire il testo, la forza di T5 è anche quella di essere allenato su uno dei dataset più puliti e grandi disponibili al momento, chiamato C4 Dataset (Colossal Clean Crawled Corpus), inoltre dopo il pre-training viene effettuato un fine-tuning su dei task specifici come la traduzione, il riassunto, ecc. . .

18.7 DIT

DiT è un **Vision Transformer** [29], questi trattano un'immagine come una sequenza di patch (piccoli blocchi dell'immagine) e la elaborano come se fosse una frase, usando un Transformer (Figura 109). DiT tuttavia è addestrato sui documenti come immagini [49], progettato per comprendere la struttura e il contenuto visivo di documenti senza testo esplicito. DiT utilizza la stessa struttura dei Vision Transformer, l'immagine del documento viene divisa in patch, e ognuno di questi patch diventa un embedding, gli embedding vengono processati da stack di Transformer encoder, e come nel ViT, c'è un [CLS] token il cui embedding finale può essere usato per classificazione o altre task globali.

18.8 MMDIT

MMDiT è un'estensione di DiT che integra testi e immagini in un unico framework Transformer [96], è pensato per task multimodali, come la comprensione congiunta del layout e del contenuto testuale.

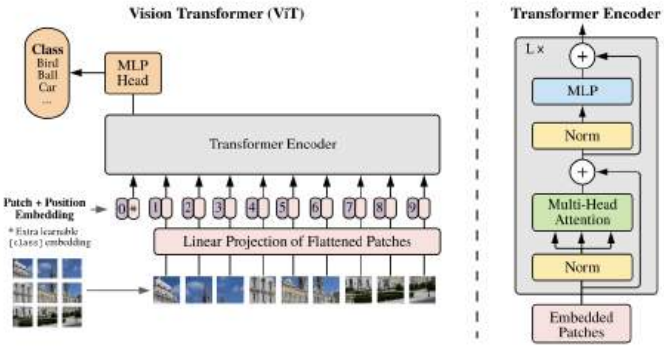


Figura 109: Panoramica del modello. Dividiamo un'immagine in patch di dimensioni fisse, incorporiamo linearmente ciascuna di esse, aggiungiamo le incorporazioni di posizione e diamo la sequenza di vettori risultante a un codificatore Transformer. Eseguendo la classificazione, utilizziamo l'aggiunta di un "token di classificazione" apprendibile dalla sequenza.

18.9 SD3

SD3 è un modello di generazione condizionata di immagini basato sull'architettura dei modelli di diffusione [5], rappresenta una delle più recenti evoluzioni di Stable Diffusion, ottimizzato per qualità, scalabilità e flessibilità d'uso in vari contesti multimodali, **SD3** (Stable Diffusion 3) è la terza generazione del modello *Stable Diffusion*, rilasciata da Stability AI. È stato progettato per superare i limiti delle versioni precedenti, con un focus particolare sulla coerenza testuale, l'alta risoluzione, multi-condizionalità e la scalabilità. A differenza dei precedenti modelli, SD3 non è un puro modello di diffusione tradizionale, ma è influenzato dal paradigma **Flow Matching** affrontato in questo capitolo, questo significa che è in grado di impara una mappa deterministica dello spazio latente e l'immagine target, ispirandosi alla teoria del flusso. Utilizza T5 come modello di testo, avendo una vera e propria archiettutra a blocchi.

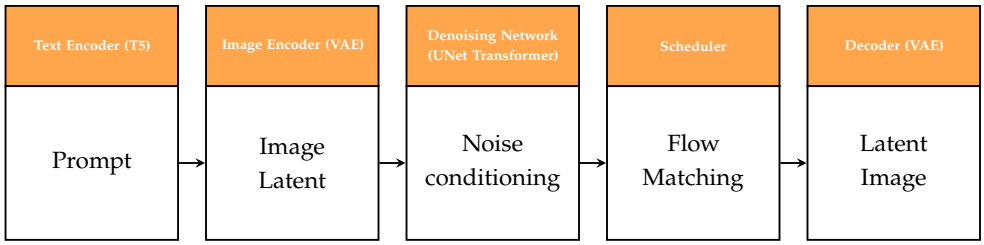


Figura 110: Schema della pipeline del modello di Stable Diffusion 3.

19 MULTI HEAD ATTENTION

Questo capitolo esplora le tecniche avanzate di attenzione, in particolare quelle Multi-Head. Analizzando le ottimizzazioni che hanno reso possibile l'implementazione efficiente dei modelli di linguaggio di grandi dimensioni. Tratteremo le tecniche di Caching KV, le varianti Multi Query Attention (MQA) e Grouped Query Attention (GQA), gli Embeddings Posizionali Rotazionali (RoPE) e la Multi Head Latent Attention (MLA).

19.1 INTRODUZIONE

L'architettura Transformer ha rivoluzionato il campo del Deep Learning nel campo del Natural Language Processing (NLP). Tuttavia, con l'aumento delle dimensioni dei modelli e delle sequenze di input, l'efficienza computazionale e di memoria del meccanismo di attenzione Multi-Head è diventata una sfida critica, esploriamo insieme le tecniche avanzate volte a ottimizzare l'attenzione, mantenendo o migliorando le prestazioni dei modelli.

19.2 KV CACHE

Durante la fase d'inferenza, nei modelli autoregressivi, il calcolo dell'attenzione per ogni nuovo token richiede l'accesso alle rappresentazioni di tutti i token precedenti. Senza ottimizzazioni, questo comporterebbe il ricalcolo completo delle matrici Key (K) e Value (V) per ogni posizione ad ogni step di generazione. La tecnica del **KV Cache** risolve questo problema memorizzando le matrici Key e Value calcolate nei passi precedenti all'interno della Cache:

$$\text{Attention}(Q_t, K_{1:t}, V_{1:t}) = \text{softmax}\left(\frac{Q_t K_{1:t}^T}{\sqrt{d_k}}\right) V_{1:t} \tag{21}$$

dove Q_t è la query per il token corrente alla posizione t, mentre

$K_{1:t}$ e $V_{1:t}$ rappresentano tutte le chiavi e valori a partire dalla prima posizione, fino a giungere alla posizione t.

19.2.1 Implementazione della Cache

Ad ogni step di generazione:

1. Si calcola la nuova coppia key-value per il token corrente;
2. Si concatena questa coppia alla Cache esistente;
3. Si utilizza la Cache aggiornata per calcolare l'attenzione.

Questo approccio riduce la complessità temporale da $o(n^2)$ a $o(n)$ per ogni nuovo token, dove n è la lunghezza della sequenza.

19.2.2 Requisiti di Memoria

La dimensione del KV Cache per un layer di attenzione Multi-Head standard è:

$$\text{Cache Size} = 2 \times \text{batch\_size} \times \text{num\_heads} \times \text{seq\_length} \times \text{head\_dim}$$

Il fattore 2 deriva dalla necessità di memorizzare sia la matrice Key che la matrice Value. Questa crescita lineare con la lunghezza della sequenza rappresenta una limitazione significativa per sequenze molto lunghe.

19.3 MULTI QUERY ATTENTION (MQA)

La **MQA** (Multi Query Attention), introdotta nei modelli PaLM (Google) e Falcon, affronta il problema della memoria del KV Cache attraverso una strategia di condivisione delle rappresentazioni. L'idea è quella di utilizzare un'unica coppia Key-Value condivisa tra tutte le teste di attenzione, mantenendo invece Query separate per ogni testa:

$$\begin{aligned} Q_i &= xW_i^Q \quad \text{per } i = 1, \ldots, H \\ K &= xW^K \\ V &= xW^V \end{aligned}$$

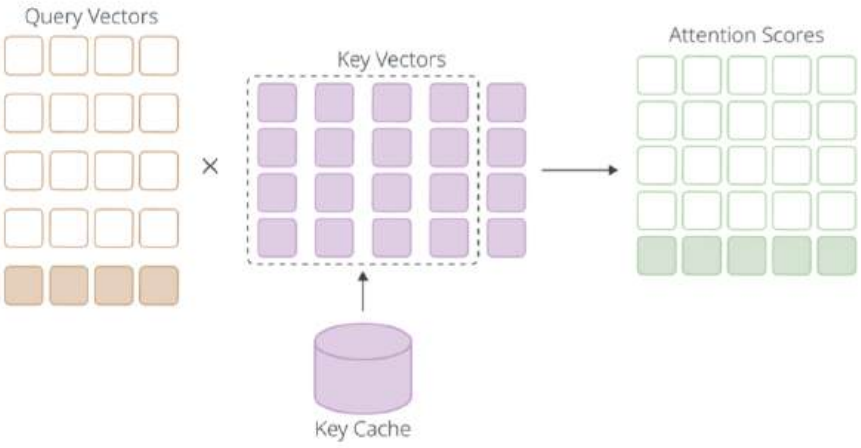


Figura 111: Illustrazione del meccanismo KV Cache: i vettori query correnti vengono moltiplicati con i vettori key memorizzati nel cache per calcolare gli attention scores, evitando il ricalcolo delle rappresentazioni dei token precedenti.

dove H è il numero di teste di attenzione, x è la rappresentazione di input e W_i^Q, W^K, W^V sono le matrici di proiezione.

19.3.1 Calcolo dell'Attenzione

L'attenzione per ogni testa viene calcolata come:

$$\text{head}_i = \text{Attention}(Q_i, K, V) = \text{softmax}\left(\frac{Q_i K^T}{\sqrt{d_k}}\right) V$$

L'output finale è ottenuto concatenando tutte le teste:

$$\text{MQA}(h) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H) W^O$$

19.3.2 Riduzione della Memoria

Con MQA, la dimensione del KV Cache diventa:

$$\text{MQA Cache Size} = 2 \times \text{batch\_size} \times 1 \times \text{seq\_length} \times \text{head\_dim}$$

Questo rappresenta una riduzione di un fattore H (numero di teste) rispetto all'attenzione Multi-Head standard, portando a significativi risparmi di memoria per modelli con molte teste di attenzione.

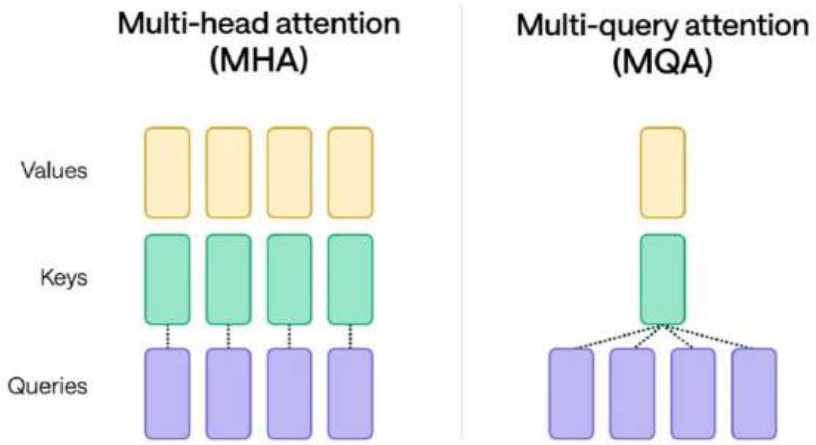


Figura 112: Architettura di Multi-Head Attention (MHA) vs Multi-Query Attention (MQA). MQA utilizza Key e Value condivisi tra tutte le teste di attenzione, mantenendo Query separate, riducendo così i requisiti di memoria del cache.

19.3.3 Prestazioni

Gli esperimenti mostrano che MQA mantiene prestazioni competitive rispetto all'attenzione Multi-Head classica, pur riducendo drasticamente i requisiti di memoria. Questo trade-off favorevole ha reso MQA una scelta popolare per modelli di grandi dimensioni.

19.4 GROUPED QUERY ATTENTION (GQA)

La **GQA** (Grouped Query Attention) invece, utilizzata in modelli come LLaMA 2 [87], LLaMA 3 [86] e Mistral 7B [4], rappresenta un compromesso tra Multi Head Attention classica e Multi Query Attention. Invece di utilizzare una singola coppia Key-Value per tutte le teste (MQA) o coppie separate per ogni testa (MHA), GQA organizza le teste in gruppi che condividono le stesse rappresentazioni Key-Value. Dato un numero totale di teste H e un numero di gruppi G, ogni gruppo contiene $\frac{H}{G}$ teste di query che condividono la stessa coppia Key-Value:

$$\begin{aligned} Q_{g,i} &= xW_{g,i}^{Q} \quad \text{per gruppo } g, \text{testa } i \\ K_g &= xW_g^{K} \quad \text{per gruppo } g \\ V_g &= xW_g^{V} \quad \text{per gruppo } g \end{aligned}$$

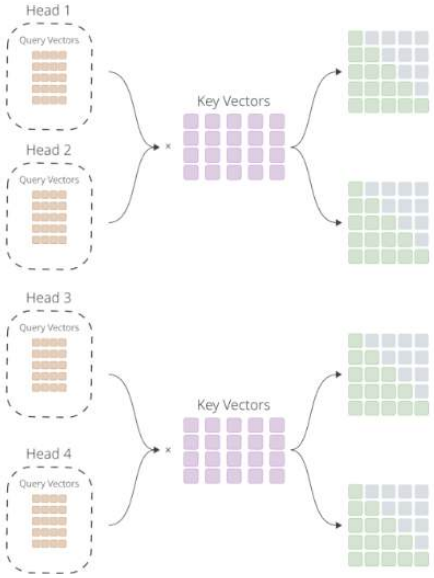


Figura 113: Architettura di un meccanismo di attenzione multi-head con quattro teste (Head 1-4). Ogni testa elabora un gruppo di Query Vectors in parallelo con Key Vectors, producendo matrici di attenzione indipendenti che possono essere successivamente combinate.

19.4.1 Flessibilità della Configurazione

GQA offre flessibilità nella scelta del numero di gruppi:

- $G = H$: equivale a Multi Head Attention standard;
- $G = 1$: equivale a Multi Query Attention;
- $1 < G < H$: compromesso ottimale tra prestazioni e efficienza.

19.4.2 Dimensione del Cache

La dimensione del KV Cache per GQA è:

$$\text{GQA Cache Size} = 2 \times \text{batch\_size} \times G \times \text{seq\_length} \times \text{head\_dim}$$

dove G è il numero di gruppi, fornendo una riduzione di memoria di un fattore $\frac{H}{G}$.

19.4.3 Prestazioni

GQA dimostra prestazioni superiori rispetto a MQA mantenendo significativi vantaggi in termini di memoria rispetto a MHA standard. Questa caratteristica l'ha resa una scelta popolare per molti modelli moderni di grandi dimensioni.

19.5 LIMITI DEGLI EMBEDDINGS

Gli embeddings posizionali assoluti presentano diverse limitazioni, due degli aspetti principali per cui si definiscono abbastanza limitati sono i seguenti:

- **Lunghezza di Sequenza Limitata:** I modelli possono rappresentare posizioni solo fino a un limite predefinito durante l'addestramento, non potendo rappresentare posizioni oltre quel limite;
- **Indipendenza delle Posizioni:** Ogni embedding posizionale è indipendente dagli altri, questo significa che per il modello, la differenza fra la posizione 1 e la posizione 2 è la stessa che vi è fra la posizione 2 e la posizione 500, impedendo al modello di comprendere le relazioni relative tra posizioni.

La mancanza di posizionamento relativo può ostacolare la capacità del modello sul comprendere la struttura linguistica, dove le relazioni tra parole dipendono, spesso dalla loro distanza relativa. Gli embeddings posizionali relativi, come quelli utilizzati nel modello T5, si concentrano sulle distanze tra coppie di token e non sulle posizioni assolute. Tuttavia, presentano sfide pratiche:

- **Problemi di Prestazioni:** Possono essere più lenti, specialmente per sequenze lunghe;
- **Complessità nel KV Cache:** Ogni token aggiuntivo altera l'embedding per tutti gli altri token, complicando l'uso efficiente della cache.

Proprio a causa di queste complessità ingegneristiche, gli Embedding Posizionali non sono stati adottati largamente, soprattutto nel contesto dei *Large Language Models* (LLM), differentemente dall'utilizzo di quelli

rotazionali che vedremo a breve, i quali si basano sulle *Matrici di Rotazione*.

19.5.1 Matrici di Rotazione

Le **Matrici di Rotazione** costituiscono la base matematica dei **RoPE** (Rotational Positional Embeddings). Una matrice di rotazione 2D è definita come:

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{22}$$

Quando questa matrice viene applicata a un vettore, tramite un prodotto, questa matrice cambia l'angolo del vettore mantenendone invariata la sua lunghezza.

19.5.2 Formulazione di RoPE

Proprio su questo principio si basano i Rotary Positional Embedding, differendo dagli Embedding Posizionali Assoluti e Relativi, ch assegnano uno a ogni posizione un vettore che verrà sommato alla parola, non addattandosi a frasi lunghe, l'altro sulla distanza fra parole, complicando però il calcolo dell'attenzione. **RoPE** propone un'idea molto elegante, ispirata alla geometria. Pensiamo a un orologio e immaginiamo che ogni parola sia rappresentata da una lancetta (un vettore) che parte dal centro del quadrante, la prima parola verrà ruotata di dieci gradi, la terza di trenta gradi e così via. Questa è considerata un'idea geniale poiché quando il modello calcola l'attenzione tra due parole (usando il prodotto scalare), il risultato dipende solo dalla differenza angolare tra le due lancette. Quindi, applicando una rotazione basata sulla posizione assoluta, il modello impara automaticamente le relazioni di posizione relativa "gratis". RoPE per esempio applica rotazioni position-dependent alle rappresentazioni Query e Key, per due token alle posizioni m e n:

$$q_m = R(\theta, m) \cdot W^Q h_m$$
$$k_n = R(\theta, n) \cdot W^K h_n$$

dove $R(\theta, \text{pos})$ è la matrice di rotazione dipendente dalla posizione, in questo caso l'ennesima e l'emmesima posizione. Le matrici di rotazione possiedono la proprietà fondamentale:

$$R(\theta, m)^T R(\theta, n) = R(\theta, n - m)$$

Questa proprietà, consente al prodotto scalare tra query e key di dipendere solo dalla loro distanza relativa $(n - m)$, ottenendo così un posizionamento relativo naturale. Torna utile nel momento in cui si vuole calcolare l'attention score come quanto segue, il quale grazie a questa proprietà subisce una trasformazione nel suo calcolo evidenziandone come lo score dipenda solo dalla differenza:

$$\text{score} = q_m \cdot k_n = q^T R_m^T R_n k = q^T R_{m-n} k$$

Per concludere analizziamo i vantaggi che offre RoPE, i quali sono diversi rispetto agli embedding tradizionali che abbiamo sempre considerato:

- **Posizionamento Relativo:** Intrinsecamente codifica le relazioni relative, senza effettuare dei calcoli complessi;
- **Lunghezza Flessibile:** Può gestire sequenze più lunghe di quelle viste durante l'addestramento, perché essendo una rotazione, può girare all'infinito;
- **Efficienza:** È una tecnica che non aggiunge parametri da apprendere e si integra facilmente con le tecniche di caching KV;
- **Adozione Diffusa:** Utilizzato in modelli come RoFormer [81], LLaMA 2 [87], LLaMA 3 [86], Gemma [26], PaLM [23], GPT-NeoX [15], Falcon [32], DeepSeek V2 [2], DeepSeek V3 [3], DeepSeek R1 [1].

19.6 MULTI HEAD LATENT ATTENTION (MLA)

La **MLA** (Multi Head Latent Attention), è stata introdotta nei modelli DeepSeek V2, V3 e R1, rappresentando un approccio più avanzato per ridurre i requisiti di memoria del KV Cache. L'idea centrale è comprimere l'input dell'attenzione in uno spazio latente a bassa dimensionalità come visibile nella parte più a destra della Figura 114.

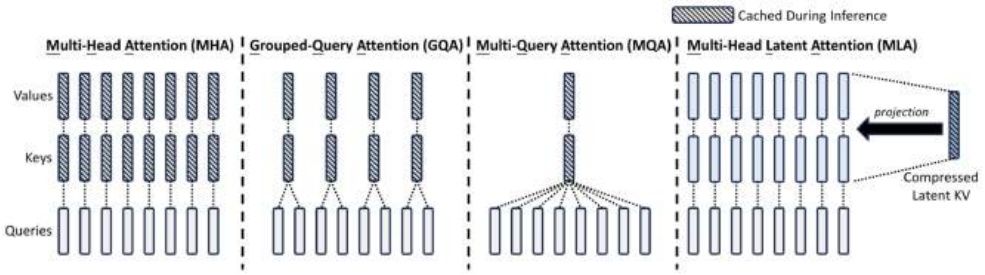


Figura 114: Rappresentazioni dei diversi meccanismi dell'attenzione presentati in questo capitolo da sinistra verso destra: Multi Head Attention, Grouped Query Attention, Multi Query Attention e Multi-Head Latent Attention.

19.6.1 Principio Fondamentale

MLA comprime l'input dell'attenzione h_t in un vettore latente a bassa dimensionalità c_t^{KV} con dimensione d_c, dove $d_c \ll h_n \cdot d_h$:

$$c_t^{KV} = W^{DKV} h_t$$

dove W^{DKV} è la matrice di compressione che riduce la dimensionalità da $(h_n \cdot d_h)$ a d_c.

19.6.2 Recupero delle Chiavi e Valori

Quando necessario per il calcolo dell'attenzione, il vettore latente viene mappato nuovamente nello spazio ad alta dimensionalità:

$$K_t = c_t^{KV} W^{UK}$$
$$V_t = c_t^{KV} W^{UV}$$

dove W^{UK} e W^{UV} sono matrici di up-projection che mappano dallo spazio latente allo spazio originale.

19.6.3 Trattamento delle Query

Analogamente, le query possono essere trattate attraverso lo spazio latente:

$$c_t^{Q} = h_t W^{DQ}$$
$$Q_t = c_t^{Q} W^{UQ}$$

19.6.4 RoPE Disaccoppiato

L'integrazione di RoPE con MLA presenta sfide tecniche. Nel caso standard, W^{UK} può essere "assorbito" in W^{Q}, riducendo ulteriormente l'uso di memoria. Tuttavia, con RoPE, la matrice di rotazione position-dependent si interpone tra $(W^{Q})^{T}$ e W^{UK}, impedendo questo assorbimento. Per risolvere questa problematica, MLA introduce il così chiamato "RoPE disaccoppiato":

- Vettori query aggiuntivi vengono introdotti insieme a un vettore key condiviso;
- Questi vettori aggiuntivi sono utilizzati esclusivamente per il processo RoPE;
- Le chiavi originali rimangono isolate dalla matrice di rotazione.

19.6.5 Calcolo dell'Attenzione MLA

Il processo completo di calcolo dell'attenzione MLA include:

1. Compressione dell'input nello spazio latente;
2. Up-projection per recuperare keys e values;
3. Applicazione del RoPE disaccoppiato;
4. Calcolo dell'attenzione standard;
5. Aggregazione dei risultati.

19.6.6 Riduzione della Memoria

La dimensione del cache per MLA è drasticamente ridotta:

$$\text{MLA Cache Size} = \text{batch\_size} \times \text{seq\_length} \times d_c$$

dove d_c è significativamente più piccolo della dimensione originale, risultando una grande riduzione di memoria di diversi ordini di grandezza.

19.6.7 Prestazioni

Gli esperimenti mostrano che MLA mantiene prestazioni competitive o superiori rispetto alle tecniche precedenti, pur ottenendo le maggiori riduzioni di memoria tra tutte le tecniche discusse. Questo ha reso MLA particolarmente attraente per modelli che devono gestire sequenze molto lunghe o operare con risorse computazionali limitate.

19.7 CONFRONTO DELLE TECNICHE

19.7.1 Analisi Comparativa

La tabella seguente riassume le caratteristiche principali delle diverse tecniche di attenzione:

Tecnica	Cache Size	Qualità	Complessità	Adozione
Multi Head Attention	$2 \times B \times H \times L \times D$	Alta	Bassa	Standard
Multi Query Attention	$2 \times B \times 1 \times L \times D$	Media	Bassa	Diffusa
Grouped Query Attention	$2 \times B \times G \times L \times D$	Alta	Bassa	Molto diffusa
Multi Head Latent Attention	$B \times L \times d_c$	Alta	Alta	Emergente

Tabella 11: Confronto delle tecniche di attenzione avanzate

dove B è la batch size, H il numero di teste, L la lunghezza della sequenza, D la dimensione delle teste, G il numero di gruppi, e d_c la dimensione compressa.

19.7.2 Scelta della Tecnica Ottimale

La scelta della tecnica di attenzione dipende da diversi fattori:

- **Risorse di Memoria:** MLA per limitazioni estreme, GQA per compromessi bilanciati;
- **Lunghezza delle Sequenze:** RoPE per sequenze lunghe, MLA per sequenze molto lunghe;
- **Prestazioni:** GQA per il miglior equilibrio prestazioni-efficienza;
- **Semplicità di Implementazione:** MQA per implementazioni rapide.

19.8 IMPLEMENTAZIONI PRATICHE

L'implementazione efficiente di queste tecniche richiede considerazioni specifiche:

- **Gestione della Memoria:** Allocazione dinamica del Cache per sequenze di lunghezza variabile;
- **Parallelizzazione:** Ottimizzazione per GPU e TPU;
- **Precision:** Uso di mixed precision per ridurre ulteriormente l'uso di memoria;
- **Batch Processing:** Gestione efficiente di batch con sequenze di lunghezza diversa.

19.8.1 Trade-off Prestazioni-Memoria

Ogni tecnica presenta specifici trade-off:

MQA Massima semplicità implementativa con buone riduzioni di memoria;

GQA Miglior equilibrio tra prestazioni e efficienza;

MLA Massima riduzione di memoria con complessità implementativa maggiore;

ROPE Miglior gestione delle sequenze lunghe con overhead computazionale minimo.

19.9 TENDENZE FUTURE E SVILUPPI

Le direzioni future includono la così detta **Sparse Attention**, volta a ridurre la complessità quadratica dell'attenzione, l'**Adaptive Attention**, introducendo meccanismi che adattano la complessità alla difficoltà del task, avanzate tecniche di compressione più sofisticate per MLA, e ovviamente delle ottimizzazioni specifiche per le architetture hardware emergenti. Una cosa da considerare è come queste ottimizzazioni delle tecniche abbiano permesso a modelli di diventare enormi quasi 1T di parametri, garantire un inferenza efficiente sull'hardware dei consumatori, dare la possibilità di processare dei documenti molto lunghi

e la possibilità infine di effettuare il deployment su dispositivi mobili, tutte non cose da poco. Le tecniche avanzate di attenzione Multi-Head rappresentano progressi fondamentali nell'ottimizzazione dei modelli Transformer. Dall'introduzione del KV Cache per accelerare l'inferenza, attraverso le varianti MQA e GQA per ridurre l'uso di memoria, fino agli Embeddings Posizionali Rotatori per una migliore gestione delle sequenze lunghe e alla rivoluzionaria MLA per compressioni estreme. Ognuna di queste innovazioni ha contribuito a rendere i modelli di linguaggio di grandi dimensioni più pratici e accessibili. La scelta della tecnica ottimale dipende dai requisiti specifici dell'applicazione, dalle risorse disponibili e dai trade-off accettabili tra prestazioni ed efficienza. Con il continuo sviluppo di nuove architetture e l'aumento delle dimensioni dei modelli, queste ottimizzazioni continueranno a evolversi, aprendo nuove possibilità per applicazioni sempre più sofisticate del Deep Learning nel processamento del linguaggio naturale. L'adozione diffusa di queste tecniche nei modelli moderni testimonia la loro efficacia e importanza pratica. Comprendere questi meccanismi è essenziale per chiunque lavori con modelli Transformer avanzati e rappresenta la base per future innovazioni nel campo.

CONCLUSIONI

Questa raccolta di appunti per poter essere al passo coi tempi dovrebbe essere aggiornata praticamente ogni mese, poiché la velocità alla quale questo settore di ricerca si muove, è impressionante e sarebbe a dir poco impossibile raccontare tutto, essendoci sviluppi in qualsiasi direzione. Proprio per questo motivo il mio invito è sempre quello di essere curiosi e voler approfondire le proprie conoscenze attraverso la lettura dei paper pubblicati ed addentrarsi sempre più in profondità, stimolando la propria curiosità. Ho cercato in questi appunti di ridurre al minimo l'eccessiva formalizzazione dei concetti, in modo tale da poter rendere di più semplice comprensione argomenti che sono tutto fuorché semplici, mi auguro di essere riuscito nel mio intento (ben consapevole che i formalismi siano necessari in alcuni momenti), cercando di distanziarmi da quel rigore formale, di cui la produzione accademica il più delle volte è piena, proprio poiché il target verso cui si è direzionati è tutt'altro. Auguro a tutti voi lettori di riuscire ad avere un bellissimo proseguimento nelle vostre vite accademiche e non, e spero dal profondo del cuore che questi appunti possano esservi tornati utili in una qualche maniera, sperando non diventino obsoleti solo dopo qualche mese dalla conclusione della loro scrittura. Non smettete mai di andare a fondo nelle cose.

BIBLIOGRAFIA

[1] DeepSeek AI. *DeepSeek R1: Scaling Up Reinforcement Learning with Language Feedback*. 2024. URL: https://deepseek.com/blog/deepseek-r1.

[2] DeepSeek AI. *DeepSeek-V2: Bridging the Gap between Retrieval and Generation*. 2023. URL: https://arxiv.org/abs/2312.01362.

[3] DeepSeek AI. *DeepSeek-V3: Towards Language Agents with Higher General Intelligence*. 2024. URL: https://deepseek.com/blog/deepseek-v3.

[4] Mistral AI. *Mistral 7B*. Official release. 2023. URL: https://mistral.ai/news/introducing-mistral-7b/.

[5] Stability AI. *Stable Diffusion 3: Open multimodal generative models with state-of-the-art performance*. Technical Report. 2024. URL: https://stability.ai/news/stable-diffusion-3.

[6] Jimmy Lei Ba, Jamie Ryan Kiros e Geoffrey E. Hinton. «Layer Normalization». In: *arXiv preprint arXiv:1607.06450* (2016). URL: https://arxiv.org/abs/1607.06450.

[7] Dzmitry Bahdanau, Kyunghyun Cho e Yoshua Bengio. «Neural Machine Translation by Jointly Learning to Align and Translate». In: *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. 2015. URL: https://arxiv.org/abs/1409.0473.

[8] Peter W. Battaglia et al. «Relational inductive biases, deep learning, and graph networks». In: *arXiv preprint arXiv:1806.01261* (2018).

[9] Mikhail Belkin e Partha Niyogi. «Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering». In: *Advances in Neural Information Processing Systems*. Vol. 14. 2001, pp. 585–591.

[10] Anthony J. Bell e Terrence J. Sejnowski. «An Information-Maximization Approach to Blind Separation and Blind Deconvolution». In: *Neural Computation* 7.6 (1995), pp. 1129–1159.

[11] Richard Bellman. *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

[12] Yoshua Bengio, Patrice Simard e Paolo Frasconi. «Learning Long-Term Dependencies with Gradient Descent is Difficult». In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166.

[13] Nils Bjorck et al. «Towards Deeper Understanding of Batch Normalization». In: *Journal of Machine Learning Research* 22.70 (2021), pp. 1–42. URL: https://www.jmlr.org/papers/volume22/19-1068/19-1068.pdf.

[14] Nils Bjorck et al. «Understanding Batch Normalization». In: *arXiv preprint arXiv:1806.02375* (2018). URL: https://arxiv.org/abs/1806.02375.

[15] Sid Black, Stella Biderman, Eric Hallahan et al. *GPT-NeoX-20B: An Open-Source Autoregressive Language Model*. EleutherAI. 2022. URL: https://github.com/EleutherAI/gpt-neox.

[16] Jane Bromley et al. «Signature verification using a "Siamese" time delay neural network». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 6. 1993.

[17] Tom B Brown et al. «Language models are few-shot learners». In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[18] Ricky T. Q. Chen et al. «Neural Ordinary Differential Equations». In: *Advances in Neural Information Processing Systems*. arXiv preprint arXiv:1806.07366. 2018, pp. 6571–6583. URL: https://arxiv.org/abs/1806.07366.

[19] Ting Chen et al. «A Simple Framework for Contrastive Learning of Visual Representations». In: *International Conference on Machine Learning (ICML)*. PMLR. 2020, pp. 1597–1607.

[20] Xinlei Chen e Kaiming He. «Exploring simple siamese representation learning». In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021, pp. 15750–15758.

[21] Kyunghyun Cho et al. «Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation». In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1724–1734. URL: https://arxiv.org/abs/1406.1078.

[22] Sumit Chopra, Raia Hadsell e Yann LeCun. «Learning a similarity metric discriminatively, with application to face verification». In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2005.

[23] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin et al. *PaLM: Scaling Language Modeling with Pathways*. Google Research. 2022. URL: https://arxiv.org/abs/2204.02311.

[24] Kevin Clark et al. «ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators». In: *arXiv preprint arXiv:2003.10555* (2020).

[25] Thomas M. Cover. «Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition». In: *IEEE Transactions on Electronic Computers* EC-14.3 (1965), pp. 326–334.

[26] Google DeepMind. *Gemma: Lightweight, Open Models by Google DeepMind*. Official announcement. 2024. URL: https://ai.google.dev/gemma.

[27] Michaël Defferrard, Xavier Bresson e Pierre Vandergheynst. «Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2016. URL: https://arxiv.org/abs/1606.09375.

[28] Jacob Devlin et al. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *arXiv preprint arXiv:1810.04805* (2018).

[29] Alexey Dosovitskiy et al. «An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale». In: *International Conference on Learning Representations (ICLR)*. 2021. URL: https://openreview.net/forum?id=YicbFdNTTy.

[30] John Duchi, Elad Hazan e Yoram Singer. «Adaptive Subgradient Methods for Online Learning and Stochastic Optimization». In: *Proceedings of the 24th Annual Conference on Learning Theory (COLT 2011)*. JMLR Workshop e Conference Proceedings, 2011, pp. 257–269. URL: https://jmlr.org/proceedings/papers/v12/duchi11a.html.

[31] Jeffrey L. Elman. «Finding Structure in Time». In: *Cognitive Science* 14.2 (1990), pp. 179–211. DOI: 10.1207/s15516709cog1402_1.

[32] Hady Elsahar, Tarek Penedo et al. *Falcon LLM: Scaling Up and Distilling Knowledge*. Technology Innovation Institute. 2023. URL: https://falconllm.tii.ae/.

[33] Justin Gilmer et al. «Neural Message Passing for Quantum Chemistry». In: *Proceedings of the 34th International Conference on Machine Learning*. PMLR. 2017, pp. 1263–1272.

[34] Xavier Glorot e Yoshua Bengio. «Understanding the difficulty of training deep feedforward neural networks». In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Vol. 9. PMLR. 2010, pp. 249–256.

[35] Ian Goodfellow et al. «Generative Adversarial Nets». In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680.

[36] Raia Hadsell, Sumit Chopra e Yann LeCun. «Dimensionality Reduction by Learning an Invariant Mapping». In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 2. IEEE. 2006, pp. 1735–1742.

[37] Kaiming He et al. «Momentum contrast for unsupervised visual representation learning». In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 9729–9738.

[38] Geoffrey E. Hinton e Ruslan R. Salakhutdinov. «Reducing the Dimensionality of Data with Neural Networks». In: *Science* 313.5786 (2006), pp. 504–507. DOI: 10.1126/science.1127647.

[39] Jonathan Ho, Ajay Jain e Pieter Abbeel. «Denoising Diffusion Probabilistic Models». In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 6840–6851.

[40] Sepp Hochreiter. «Untersuchungen zu dynamischen neuronalen Netzen». Tesi di dott. Technische Universität München, 1991.

[41] Sepp Hochreiter e Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[42] Harold Hotelling. «Analysis of a Complex of Statistical Variables into Principal Components». In: *Journal of Educational Psychology* 24.6 (1933), pp. 417–441.

[43] Aapo Hyvärinen e Erkki Oja. «Independent Component Analysis: Algorithms and Applications». In: *Neural Networks* 13.4–5 (2000), pp. 411–430.

[44] Sergey Ioffe e Christian Szegedy. «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift». In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. PMLR. 2015, pp. 448–456.

[45] Sergey Ioffe e Christian Szegedy. «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift». In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. PMLR, 2015, pp. 448–456. URL: https://arxiv.org/abs/1502.03167.

[46] Diederik P. Kingma e Jimmy Ba. «Adam: A Method for Stochastic Optimization». In: *Proceedings of the 3rd International Conference for Learning Representations (ICLR 2015)*. arXiv preprint arXiv:1412.6980. 2015. URL: http://arxiv.org/abs/1412.6980.

[47] Thomas N. Kipf e Max Welling. «Semi-Supervised Classification with Graph Convolutional Networks». In: *International Conference on Learning Representations (ICLR)*. arXiv preprint arXiv:1609.02907, 2016. 2017. URL: https://arxiv.org/abs/1609.02907.

[48] Yann LeCun et al. «Gradient-Based Learning Applied to Document Recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791. URL: https://doi.org/10.1109/5.726791.

[49] Yiheng Li et al. «DiT: Self-supervised Pre-training for Document Image Transformers». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2023). DOI: 10.1109/TPAMI.2023.3246421.

[50] Yinhan Liu et al. «RoBERTa: A Robustly Optimized BERT Pretraining Approach». In: *arXiv preprint arXiv:1907.11692* (2019).

[51] Ilya Loshchilov e Frank Hutter. «Decoupled Weight Decay Regularization». In: *Proceedings of the 5th International Conference on Learning Representations (ICLR 2017)*. 2017. URL: https://openreview.net/forum?id=Bkg3g0EFw.

[52] A. A. Markov. «Rasprostranenie zakona bol'shikh chisel na velichiny, zavisyashchiye drug ot druga». In: *Izvestiya Fiziko-Matematicheskogo Obshchestva pri Kazanskom Universitete* 15 (1906). Translated as "Extension of the law of large numbers to dependent quantities", pp. 135–156.

[53] Tomas Mikolov et al. «Efficient Estimation of Word Representations in Vector Space». In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2013. URL: https://arxiv.org/abs/1301.3781.

[54] Marvin Minsky e Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969.

[55] Ishan Misra e Laurens van der Maaten. «Self-supervised learning of pretext-invariant representations». In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 6707–6717.

[56] John F. Nash. «Non-Cooperative Games». In: *Annals of Mathematics* 54.2 (1951), pp. 286–295. DOI: 10.2307/1969529.

[57] Yurii Nesterov. «A Method for Solving the Convex Programming Problem with Convergence Rate O(1/k$\hat{2}$)». In: *Soviet Mathematics Doklady* 27 (1983), pp. 372–376.

[58] Jorge Nocedal e Stephen J. Wright. *Numerical Optimization*. 2nd. Chapter 3: Line Search Methods. Springer, 2006. ISBN: 978-0387303031. DOI: 10.1007/978-0-387-40065-5. URL: https://doi.org/10.1007/978-0-387-40065-5.

[59] Long Ouyang et al. «Training language models to follow instructions with human feedback». In: *arXiv preprint arXiv:2203.02155* (2022).

[60] Karl Pearson. «On Lines and Planes of Closest Fit to Systems of Points in Space». In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572.

[61] Jeffrey Pennington, Richard Socher e Christopher D. Manning. «GloVe: Global Vectors for Word Representation». In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: https://aclanthology.org/D14-1162/.

[62] Matthew E. Peters et al. «Deep Contextualized Word Representations». In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics. New Orleans, Louisiana, 2018, pp. 2227–2237. DOI: 10.18653/v1/N18-1202. URL: https://aclanthology.org/N18-1202.

[63] Boris T. Polyak. «Some Methods of Speeding Up the Convergence of Iteration Methods». In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[64] Xinbao Qiao et al. «Hessian-Free Online Certified Unlearning». In: *arXiv preprint arXiv:2306.01047* (2023). URL: https://arxiv.org/abs/2306.01047.

[65] Lawrence R. Rabiner. «A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition». In: *Proceedings of the IEEE* 77.2 (1989), pp. 257–286. DOI: 10.1109/5.18626.

[66] Alec Radford et al. *Improving Language Understanding by Generative Pre-Training*. Rapp. tecn. OpenAI, 2018. URL: https://openai.com/research/language-unsupervised.

[67] Alec Radford et al. «Language models are unsupervised multitask learners». In: *OpenAI Blog* 1.8 (2019).

[68] Colin Raffel et al. «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer». In: *Journal of Machine Learning Research*. Vol. 21. 140. 2020, pp. 1–67. URL: http://jmlr.org/papers/v21/20-074.html.

[69] Colin Raffel et al. «Exploring the limits of transfer learning with a unified text-to-text transformer». In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67.

[70] Martin A. Riedmiller e Heinrich Braun. «A Direct Adaptive Method for Faster Backpropagation Learning: the RPROP Algorithm». In: *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*. IEEE, 1993, pp. 586–591. DOI: 10.1109/ICNN.1993.298623.

[71] Olaf Ronneberger, Philipp Fischer e Thomas Brox. «U-Net: Convolutional Networks for Biomedical Image Segmentation». In: *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)* (2015), pp. 234–241. URL: https://arxiv.org/abs/1505.04597.

[72] David E. Rumelhart, Geoffrey E. Hinton e Ronald J. Williams. «Learning representations by back-propagating errors». In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0.

[73] Victor Sanh et al. «DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter». In: *arXiv preprint arXiv:1910.01108* (2019).

[74] Shibani Santurkar et al. «How Does Batch Normalization Help Optimization? (No, It Is Not About Internal Covariate Shift)». In: *Advances in Neural Information Processing Systems (NeurIPS 2018)*. 2018, pp. 2483–2493. URL: https://arxiv.org/abs/1805.11604.

[75] Mike Schuster e Kuldip K. Paliwal. «Bidirectional Recurrent Neural Networks». In: *IEEE Transactions on Signal Processing*. Vol. 45. 11. 1997, pp. 2673–2681. DOI: 10.1109/78.650093.

[76] Rico Sennrich, Barry Haddow e Alexandra Birch. «Neural Machine Translation of Rare Words with Subword Units». In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*. Introduce a BPE-based subword segmentation for open-vocabulary NMT. Association for Computational Linguistics, 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162. URL: https://aclanthology.org/P16-1162/.

[77] Hidetoshi Shimodaira. «Improving Predictive Inference under Covariate Shift by Weighting the Log-Likelihood Function». In: *Journal of Statistical Planning and Inference*. Vol. 90. 2. Elsevier, 2000, pp. 227–244. DOI: 10.1016/S0378-3758(00)00115-4.

[78] Yang Song et al. «Score-Based Generative Modeling through Stochastic Differential Equations». In: *International Conference on Learning Representations*. 2021.

[79] Alexandra Souly et al. *Poisoning Attacks on LLMs Require a Near-constant Number of Poison Samples*. 2025. arXiv: 2510.07192 [cs.LG]. URL: https://arxiv.org/abs/2510.07192.

[80] Nitish Srivastava et al. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958.

[81] Jianlin Su et al. «RoFormer: Enhanced Transformer with Rotary Position Embedding». In: *arXiv preprint arXiv:2104.09864* (2021). URL: https://arxiv.org/abs/2104.09864.

[82] Ilya Sutskever et al. «On the Importance of Initialization and Momentum in Deep Learning». In: *Proceedings of the 30th International Conference on Machine Learning (ICML)*. PMLR. 2013, pp. 1139–1147.

[83] Yaniv Taigman et al. «DeepFace: Closing the gap to human-level performance in face verification». In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2014, pp. 1701–1708.

[84] Joshua B. Tenenbaum, Vin de Silva e John C. Langford. «A Global Geometric Framework for Nonlinear Dimensionality Reduction». In: *Science* 290.5500 (2000), pp. 2319–2323.

[85] Tijmen Tieleman e Geoffrey Hinton. *Lecture 6.5 – RMSProp: Divide the Gradient by a Running Average of Its Recent Magnitude*. Coursera: Neural Networks for Machine Learning (Technical Report), University of Toronto. Unpublished lecture slides; often cited as origin of RMSprop. 2012.

[86] Hugo Touvron, Guillaume Lample et al. *LLaMA 3: Open Foundation Models*. Meta AI. 2024. URL: https://ai.meta.com/blog/meta-llama-3/.

[87] Hugo Touvron, Thibaut Lavril, Gautier Izacard et al. *LLaMA 2: Open Foundation and Fine-Tuned Chat Models*. Meta AI. 2023. URL: https://ai.meta.com/llama/.

[88] Alan M. Turing. «Computing Machinery and Intelligence». In: *Mind* 59.236 (1950), pp. 433–460.

[89] Dmitry Ulyanov, Andrea Vedaldi e Victor Lempitsky. «Instance Normalization: The Missing Ingredient for Fast Stylization». In: *arXiv preprint arXiv:1607.08022*. arXiv:1607.08022. 2016. URL: https://arxiv.org/abs/1607.08022.

[90] Ashish Vaswani et al. «Attention is all you need». In: *Advances in neural information processing systems* 30 (2017).

[91] Alex Wang et al. «GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding». In: *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Association for Computational Linguistics, 2018, pp. 353–355. DOI: 10.18653/v1/W18-5446. URL: https://aclanthology.org/W18-5446/.

[92] Paul J. Werbos. «Backpropagation Through Time: What It Does and How to Do It». In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: 10.1109/5.58337.

[93] Yuxin Wu e Kaiming He. «Group Normalization». In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 3–19. URL: https://arxiv.org/abs/1803.08494.

[94] Yilun Xu et al. «Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow». In: *arXiv preprint arXiv:2305.08891* (2023). Accessed: 2025-06-23. URL: https://arxiv.org/abs/2305.08891.

[95] Matthew D. Zeiler. «ADADELTA: An Adaptive Learning Rate Method». In: *CoRR* abs/1212.5701 (2012). URL: http://arxiv.org/abs/1212.5701.

[96] Tian Zhang et al. «MM-DetDit: A Multimodal Document Layout Analysis Benchmark and Method». In: *arXiv preprint arXiv:2306.00989* (2023). URL: https://arxiv.org/abs/2306.00989.

[97] Yukun Zhu et al. «Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books». In: *The IEEE International Conference on Computer Vision (ICCV)*. 2015.