UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Progettazione e sviluppo di una tecnica di smart development per l'implementazione automatica di codice sorgente

RELATORE

Prof. **Fabio Palomba**

Università degli studi di Salerno

CO-RELATORE

Prof. **Dario Di Nucci**

Università degli studi di Salerno

CANDIDATO

Pasquale Somma

Matricola: 0512107478

Anno Accademico 2021-2022

" Code is like humor. When you have to explain it, it's bad."

-Cory House

Sommario

La tesi presentata offre un'iniziale panoramica del mondo del Natural Language Processing, delle sue principali attività e dell'utilizzo di tecniche di Deep Learning per risolverle, le quali in poco tempo si sono rapidamente evolute. Tutto questo però per introdurre l'argomento centrale dell'eleborato, ovvero l'utilizzo di metodi di Deep Learning per la modellazione e la generazione di codice sorgente. Vengono così presentate tutte le attività proposte dal Big Code e le relative soluzioni di Deep Learning che sono state sviluppate negli anni, che hanno permesso di raggiungere lo stato dell'arte in alcune pratiche. Tra le varie tecniche, verrà analizzata principalmente quella dei Transformer, presentando le varie tipologie presenti, il loro utilizzo per analizzare e generare codice sorgente e le prospettive future che offrono. Infine, verrà utilizzato un tipo di Transformer per realizzare un modello in grado di completare snippet di codice, andando ad analizzarne le prestazioni relativamente alla comprensione del contesto, che si è dimostrata soddisfacente, e della correttezza del codice proposto, che invece ha mostrato delle lacune rispetto ad altri modelli dovute a diverse soluzioni adottate.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

Oggi sul web sono presenti enormi quantità di dati non strutturati e pesanti, come il testo, che per essere utilizzati hanno bisogno di un modo per essere elaborati in modo efficiente. È qui che si rivela utile il Natural Language Processing. Ovviamente, anche il codice fa parte dell'enorme quantità di dati non strutturati disponibili in repository Github, su StackOverflow e così via. Sarebbe quindi davvero un peccato non sfruttare il codice già a disposizione per cercare di facilitare il lavoro degli sviluppatori, che a volte si trovano a cimentarsi in nuovi linguaggi di programmazione, dover riscrivere pattern di codice ripetitivi, a risolvere gli stessi problemi già risolti da altri sviluppatori o a dover interpretare il codice per capire cosa effettivamente realizza, perdendo ulteriore tempo. Potremmo, invece, sfruttare questi archivi di Big Code per imparare automaticamente dal codice esistente per risolvere compiti quali la previsione di bug del programma, la previsione del comportamento del programma, la previsione dei nomi degli identificatori o la creazione automatica di nuovo codice, andando così a dare supporto al lavoro degli sviluppatori.

1.2 Motivazioni e Obiettivi

Il crescente interesse verso il NLP, in particolare applicato al codice, sta portando negli anni allo sviluppo di numerose tecniche che cercano di risolvere le diverse attività relative

al Big Code, che vedremo nella Sezione 2.3.5, con un livello di qualità sempre maggiore. Le motivazioni della stesura del seguente elaborato stanno nella necessità di raccogliere le principali tecniche realizzate fin'ora che utilizzano il Deep Learning e i risultati che hanno ottenuto, fornendo quindi una base per futuri sviluppi, nonchè provare a realizzare un modello descrivendo una delle possibili pipeline di Machine Learning da poter seguire e le tecnologie a disposizione degli sviluppatori che vogliono cimentarsi in questo ambito.

1.3 Risultati ottenuti

Le tecniche di Deep Learning utilizzate all'interno del framework encoder-decoder offrono un quadro completo sui moderni approcci di Intelligenza Artificiale che non solo vengono applicate su diverse piattaforme online o all'interno di ambienti di programmazione ma sono tutt'oggi in continua evoluzione. I risultati ottenuti non vogliono aprire nuovi orizzonti a differenti approcci sullo studio, ma fanno più da panoramica generale a differenti tecniche già esistenti. Inizialmente, sono state raccolte, per ogni attività di cui si compongono l'analisi e la generazione del codice, le principali tecniche di Deep Learning che hanno ottenuto i migliori risultati nel corso degli anni, talvolta, confrontandole tra loro e dimostrando come il loro utilizzo portasse numerosi vantaggi. Successivamente, si è passati alla tecnologia che al momento sta riscuotendo il maggior successo e su cui vengono riposte le più alte aspettative, ovvero i Transformer, che hanno concesso di raggiungere lo stato dell'arte in tantissime attività, superando molte delle tecniche realizzate precedentemente. Infine, è stato realizzato un modello di autocompletamento del codice, utilizzando proprio i Transformer, che è stato valutato ed anche confrontato con una versione ridotta del modello Codex di OpenAI, al fine di ricavarne le prestazioni e possibili miglioramenti.

1.4 Struttura della tesi

L'elaborato sarà strutturato in diversi capitoli ognuno con lo scopo di presentare e descrivere in modo approfondito diversi aspetti della ricerca effettuata e della tecnica sviluppata. Nel Capitolo 1 sono elencati gli obiettivi del seguente lavoro di ricerca e le motivazioni relative alla scelta di approfondire l'uso del Deep Learning nell'ambito del Big Code. Nel Capitolo 2 sarà analizzato il Background e lo Stato dell'Arte, ossia tutte le conoscenze e informazioni recuperate dalla letteratura esistente per cercare di realizzare una raccolta delle tecniche già esistenti per affrontare le problematiche del Big Code e il livello di qualità che si è raggiunto,

fornendo una guida per sviluppi futuri e, soprattutto, la base dell'implementazione del nostro lavoro. Nel Capitolo 3 è dove viene presentata la pipeline di Machine Learning impiegata per realizzare una tecnica di autocompletamento del codice, presentando le tecnologie utilizzate, le tecniche di Pre-Processing applicate ai dati e il possibile utilizzo del modello. Nel Capitolo 4 vengono presentati e discussi i Risultati Ottenuti, fornendo inoltre possibili accortenze da poter applicare per migliorare il modello realizzato. Infine, nel Capitolo 5 si giunge alle Conclusioni dove si considerano tutti i dati raccolti, si commentano i possibili sviluppi futuri e le implicazioni dell'utilizzo di queste tecniche nel lavoro degli sviluppatori.

CAPITOLO 2

Background e Stato dell'arte

2.1 Natural Language Processing

Il Natural Language Processing (NLP) è l'abilità di un programma di capire il linguaggio umano, sia scritto che parlato, indicato come linguaggio naturale. L'NLP è una componente di intelligenza artificiale ed utilizza proprio quest'ultima per prendere le parole reali date in input, processarle e dargli un senso in un modo che il computer possa capirle. Il computer ottiene le parole in input tramite programmi per leggerle e microfoni per raccogliere audio, che possono essere paragonati agli occhi e alle orecchie umane. E proprio come l'uomo che processa gli input esterni tramite il proprio cervello, il computer possiede un programma per processare i suoi input che, ad un certo punto dell'elaborazione, sarà convertito in codice comprensibile dal computer. Ci sono due fasi principali del natural language processing: data pre-processing e lo sviluppo di algoritmi. La fase di data pre-processing consiste nel preparare e pulire i dati testuali per renderli analizzabile dalla macchina. Il pre-processing rende i dati in una forma lavorabile e sottolinea le caratteristiche nel testo con cui l'algoritmo può lavorare. Ci sono molti modi con cui può essere fatto, tra i quali: [12]

- Tokenization, che divide il testo in unità più piccole con cui lavorare;
- Eliminazione delle parole non significative, il modo più comune con cui le parole vengono rimosse dal testo così da lasciare le parole più espressive;
- Lemmatization e stemming, per ridurre le parole alla loro forma base da processare;

- Part of speech tagging, questo è quando le parole sono marcate sulla base del ruolo che ricoprono nel discorso (come nomi, verbi, aggettivi,...).

Una volta che i dati sono stati pre-processati, vengono processati da un algoritmo. Ci sono molti differenti algoritmi per il natural language processing, ma principalmente ci sono due tipi usati comunemente: [12]

- Rules-base system. Questo sistema usa attente regole linguistiche progettate.
- Machine learning-base system. Gli algoritmi di machine learning usano metodi statistici. Imparano a eseguire task basati sui dati di training che gli vengono sottoposti e modificano i loro comportamento più sono i dati che vengono processati. Usando una combinazione di machine learning, deep learning e reti neurali, gli algoritmi di natural language processing perfezionano le loro proprie regole attraverso ripetuti processing e learning.

L'intera area di studio dell'NLP può essere suddivisa in due attività principali: l'elaborazione del linguaggio e la generazione del linguaggio.

2.1.1 Natural Language Understanding

Le aziende usano grandi quantità di dati testuali pesanti e non strutturati[1] e necessitano di un modo efficiente di processarli. La maggior parte delle informazioni create online e memorizzate nei database sono in linguaggio umano naturale e fino a poco tempo fa non potevano essere analizzati efficientemente. È proprio per questo che il natural language processing è importante e con i nuovi sviluppi nel deep learning e machine learning, gli algoritmi possono interpretare in modo sempre più efficiente le informazioni. L'analisi sintattica e semantica sono due tecniche principali usate con il natural language processing, in particolare vanno a formare un suo sottoinsieme, ovvero il natural language understanding (NLU). L'NLU aiuta la macchina a capire i dati. È usato appunto per interpretare i dati e capirne il significato così da essere processati correttamente. Lo fa capendo il contento, la semantica, sintassi e i sentimenti del testo. Quindi viene utilizzato per un'analisi dei dati più "umana". [13] La sintassi è la disposizione delle parole in una frase per ottenere un senso grammaticale. NLP usa la sintassi per valutare il significato da un linguaggio basato su regole grammaticali. Le tecniche di sintassi includono: [12]

[1] Dati non strutturati: possono essere rappresentati da qualsiasi tipo di file che non ricade nella categoria dei dati strutturati e sono più difficili da estrarre poiché richiedono degli strumenti ad-hoc.

- Parsing. Questa è l'analisi grammaticale di una frase, che può essere molto utile per problemi più complessi da risolvere.
- Word segmentation. Si tratta dell'azione di prendere una stringa di testo e derivare la forma delle parole da esso. Un esempio è una persona che scannerizza un documento e l'algoritmo sarà in grado di analizzare la pagina e riconoscere che le parole sono divise da spazi bianchi.
- Sentence breaking. Questa tecnica pone i limiti della frase in testi estesi. L'algoritmo riconosce ad esempio che due periodi sono divisi da un punto ".".
- Morphological segmentation. Divide le parole in parti più piccole chiamate morphemes. Specialmente utile nella traduzione automatica e nel riconoscimento vocale.
- Stemming. Divide le parole con inflessione nella loro forma radice. È utile se un utente sta analizzando un testo contente diverse coniugazione di un verbo, cosicchè l'algoritmo può vedere che sono esattamente la stessa parola anche se le lettere sono diverse.

La semantica prevede l'utilizzo del significato dietro le parole. NLP applica algoritmi per capire il significato e la struttura delle frasi. Le tecniche di semantica includono: [12]

- Word sense disambiguation. Deriva il significato di una parola basata sul contesto.
- Named entity recognition. Determina le parole che possono essere categorizzate in gruppi. Usando la semantica del testo, l'algoritmo è in grado di differenziare tra entità che sono visualmente le stesse e separarle nelle classi a cui appartengono.
- Natural language generation. Utilizza un dataset per determinare la semantica dietro parole e generare nuovo testo.

Un altro livello linguistico per capire i dati è l'analisi pragmatica, che aiuta a capire l'obiettivo o cosa il testo vuole raggiungere e ciò è supportato dall'analisi dei sentimenti. [13]

2.1.2 Natural language generation

La natural language generation (NLG) è l'uso dei programmi di intelligenza artificiale per produrre testi scritti o parlati da un dataset. NLG si occupa delle interazioni uomo-macchina e macchina-uomo, includendo computazione linguistica, natural language processing e natural language understanding. Le ricerche sull'NLG sono concentrate sul costruire programmi che forniscono dati con un contesto. I software più sofisticati possono analizzare grandi quantità

di dati, identificare dei pattern e condividere informazione in modo facile da capire per l'uomo. NLG prevede diversi passi, dove ognuno raffina i dati usati per produrre contenuti che sembrano in linguaggio naturale. I sei passi da seguire sono: [14]

1. Content analysis. I dati vengono filtrati per determinare cosa dovrebbe essere incluso nel prodotto finale del processo. Consiste nell'identificare gli argomenti principali nel documento e le relazioni tra essi.

2. Data understanding. I dati vengono interpretati, vengono identificati i pattern e inseriti nel contesto. In questa fase viene utilizzato il machine learning.

3. Document structuring. Un piano del documento viene creato e viene scelta la struttura narrativa sulla base del tipo di dati interpretati.

4. Sentence aggregation. Le frasi o le parti di frasi più rilevanti sono combinate in modo da sintetizzare accuratamente l'argomento.

5. Grammatical structuring. Le regole grammaticale sono applicate per generare testo naturale. Il programma deduce la struttura sintattica della frase e utilizza questa informazione per riscrivere la frase in modo che sia corretta grammaticalmente.

6. Language presentation. L'output finale è generato basandosi sul formato che l'utente ha selezionato

NLG è legato sia a NLU che al recupero di informazioni. È legato anche alla sintesi dei testi, alla sintesi vocale e alla traduzione automatica. La maggior parte delle ricerche in questo ambito si incorciano con la computazione linguistica[2].

2.1.3 Attività principali del NLP

All'interno delle due aree principali in cui può essere suddiviso il campo del Natural Language Processing, andiamo ad analizzare più nello specifico le varie attività che prevedono:

- Fill-mask si basa sulla modellazione di un linguaggio mascherato. Si procede col mascherare alcune parole di una frase e nel prevedere quali parole dovrebbero sostituire quelle mascherate. Questi modelli sono utili quando si vuole ottenere una comprensione

[2]La linguistica computazionale si concentra sullo sviluppo di formalismi descrittivi del funzionamento di una lingua naturale, che siano tali da poter essere trasformati in programmi eseguibili da computer.

statistica della lingua in cui il modello è stato addestrato. Viene impiegato anche nella risoluzione delle altre attività che vedremo nei punti successivi [15].

- Question answering, la risposta alle domande è uno dei problemi di ricerca più diffusi in NLP. Alcune delle sue applicazioni sono i chatbot, il recupero di informazioni, i sistemi di dialogo, ecc. Consente di rispondere automaticamente alle domande poste dall'uomo in linguaggio naturale, con l'aiuto di un database pre-strutturato o di una raccolta di documenti in linguaggio naturale. Alcuni modelli di risposta alle domande possono generare risposte senza contesto. Esistono diverse varianti di QA in base agli input e agli output:

  – QA estrattiva: Il modello estrae la risposta da un contesto (un testo fornito, una tabella, ...)

  – QA generativa aperta: Il modello genera testo libero direttamente in base al contesto.

  – QA generativa chiusa: in questo caso, non viene fornito alcun contesto. La risposta è completamente generata da un modello.

  È inoltre possibile differenziare i modelli di QA a seconda che siano a dominio aperto, cioè non limitato ad un ambito specifico, o a dominio chiuso [16].

- Sentence similarity riveste un ruolo importante nella ricerca e nelle applicazioni relative ai testi, in aree quali il text mining e i sistemi di dialogo. Questa tecnica si è dimostrata una delle migliori per migliorare l'efficacia del reperimento. I modelli convertono i testi in ingresso in vettori (embeddings) che catturano le informazioni semantiche e calcolano quanto sono vicini (simili) tra loro [17]. Esistono due varianti di questo compito:

  1. Il Passage Ranking è il compito di classificare i documenti in base alla loro rilevanza per una determinata query. Il compito è valutato in base al Mean Reciprocal Rank [3]. Questi modelli prendono in considerazione una query e più documenti e restituiscono documenti classificati in base alla pertinenza rispetto alla query.
  2. La Semantic Textual Similarity è un compito che consiste nel valutare la somiglianza tra due testi in termini di significato. Questi modelli prendono una frase

[3] Il Mean Reciprocal Rank (MRR), rank reciproco medio, è un indice statistico per valutare un processo che produce una lista di possibili risposte ad una interrogazione (query), ordinate per probabilità di correttezza.

di partenza e un elenco di frasi in cui cercare le somiglianze e restituiscono un elenco di punteggi di somiglianza. Un dataset di riferimento è il Semantic Textual Similarity Benchmark [18]. Il compito è valutato sulla correlazione di rango di Pearson. L'indice di correlazione di Pearson tra due variabili statistiche è un indice che esprime un'eventuale relazione di linearità tra esse. Ha un valore compreso tra +1 e -1, dove +1 corrisponde alla perfetta correlazione lineare positiva, 0 a un'assenza di correlazione lineare e -1 alla perfetta correlazione lineare negativa.

- Summarization è una tecnica che aiuta i lettori a cogliere i punti principali di un lungo documento con uno sforzo minore. È anche utile come fase di pre-elaborazione per alcuni compiti di text mining, come la classificazione dei documenti. Questo metodo può essere classificato in due diversi approcci: quello basato sull'astrazione e quello basato sull'estrazione. Un riassunto basato sull'estrazione include solo frasi estratte dal documento. Al contrario, in un riassunto basato sull'astrazione, si cerca di capire i concetti principali del testo e li si esprime usando anche parole e frasi che non compaiono nel documento originale [19].

- Text Classification è la tecnica di categorizzazione e analisi del testo in alcuni gruppi specifici. Questa tecnica supporta una valutazione comparativa dell'impatto delle informazioni linguistiche rispetto agli approcci basati sulla corrispondenza delle parole. La classificazione del testo è il compito di assegnare un'etichetta o una classe a un dato testo. Alcuni casi d'uso sono l'analisi del sentimento, l'inferenza del linguaggio naturale e la valutazione della correttezza grammaticale [20].

- Text Generation, che come già accennato prima, consente di generare nuovo testo. Come si può intuire, questo è una problematica molto generale, che viene affrontata anche nelle altre aree di NLP viste precedentemente [21]. Alcune sue varianti riguardano:

    1. Completion Generation Models prevede la parola successiva data una serie di parole. Parola per parola si forma un testo più lungo. I modelli più diffusi per questo compito sono quelli basati su GPT. Questi modelli vengono addestrati su dati privi di etichette, quindi per addestrare il proprio modello è sufficiente del testo semplice. È possibile addestrare i modelli GPT per generare un'ampia gamma di documenti, di diverse tipologie.
    2. Text-to-Text Generation Models sono modelli addestrati per imparare la mappatura tra una coppia di testi (ad esempio, la traduzione da una lingua all'altra). Le

varianti più diffuse di questi modelli sono T5, T0 e BART. I modelli Text-to-Text sono addestrati con capacità multi-tasking e possono svolgere un'ampia gamma di compiti, tra cui riassunto, traduzione e classificazione del testo.

Il testo generato può far riferimento a qualsiasi ambito, in particolare, vedremo nella Sezione 2.3.5.2 come questa categoria di modelli può essere applicata per la generazione di codice.

- Token Classification è un compito di comprensione del linguaggio naturale in cui viene assegnata un'etichetta ad alcuni token di un testo [22]. Possiamo individuare alcune sottoattività:

  1. Named Entity Recognition (NER) è il compito di riconoscere entità nominate in un testo. Queste entità possono essere nomi di persone, luoghi o organizzazioni. Il compito è formulato etichettando ogni token con una classe per ogni entità nominata e una classe denominata "0" per i token che non contengono alcuna entità. L'input per questo compito è il testo e l'output è il testo annotato con le entità nominate.
  2. Part-of-Speech (PoS) Tagging in cui il modello riconosce le parti del discorso, come nomi, pronomi, aggettivi o verbi, in un testo dato. Il compito è formulato come etichettatura di ogni parola con una parte del discorso.

- Translation è l'attività del tradurre da un linguaggio ad un altro. I modelli di traduzione possono essere utilizzati per costruire agenti conversazionali[4] in diverse lingue. Questo può essere fatto traducendo i dataset nella nuova lingua oppure traducendo in tempo reale l'input dell'utente nella lingua che il chatbot può processare e l'output del chatbot nella lingua originaria dell'utente.

2.1.4 Benefici del NLP

Come abbiamo visto il natural language processing permette di svolgere moltissime funzioni che trovano applicazione in diversi contesti concreti, sia nel quotidiano delle persone che in ambiti aziendali. Infatti è possibile utilizzare l'IA per analizzare le recensioni di un servizio da parte degli utenti e aiutare chi fornisce il servizio a capire sia i problemi principali riscontrati che i punti di forza. Oppure un'esperienza che viviamo ogni giorno è utilizzare il

[4] Un agente conversazionale è un programma software con intelligenza artificiale che interpreta e risponde alle dichiarazioni fatte dagli utenti in un linguaggio naturale normale.

riconoscimento vocale, del nostro smartphone ad esempio cosicchè possa capire esattamente cosa stiamo dicendo ed aiutarci a svolgere le nostre attività. In ambito accademico, inoltre risulta fondamentale durante le ricerche per analizzare grandi quantità di materiale basandosi non solo sui metadati del testo, ma anche sul testo stesso. L'elaboratore di testo permette anche di riconoscere casi di plagio o correggere la bozza di un documento. Può essere utilizzato anche in ambito medico, infatti l'analisi di cartelle cliniche, consente di predire, e idealmente prevenire, malattie [12]. Le ricerche fatte sul natural language processing ruotano intorno al cercare. Questo prevede l'avere una query di un utente da porre ad un dataset, però sotto forma di domanda che potrebbe essere fatta ad un'altra persona, non ad un computer solitamente. La macchina interpreta gli elementi più importanti della frase in linguaggio naturale, a cui corrispondono delle caratteristiche nel dataset, e restituisce una risposta. Il beneficio principale dell'NLP è che migliora il modo con cui le persone e i computer interagiscono l'un l'altro. Il modo più diretto per manipolare un computer è attraverso il codice, che il linguaggio proprio del computer. Facendo si che il computer capisca il linguaggio umano, interagire con il computer diventa molto più intuitivo per gli umani. D'altrocanto l'NLP può aiutare proprio i programmatori a comunicare col computer nel suo linguaggio, assistendoli e semplificando il modo in cui questi scrivono codice, come vedremo nel dettaglio successivamente.

2.2 Deep Learning

Gli approcci correnti al natural processing sono basati sul deep learning (DL), un tipo di IA che esamina e usa pattern nei dati per migliorare la capacità del programma di capire le informazioni. I modelli di deep learning richiedono grandi quantità di dati etichettati per allenare gli algoritmi di NLP e identificare le relazioni rilevanti; assemblare insieme tutta questa enorme quantità di dati è uno degli ostacoli principali per il natural language processing. Gli approcci iniziali al natural language processing prevedono un approccio più basato sulle regole, dove a più semplici algoritmi di machine learning veniva detto quali parole e frasi cercare nel testo e restituivano un risultato quando queste apparivano. Il deep learning è molto più flessibile, un approccio intuitivo in cui gli algoritmi imparano a identificare le intenzioni dell'oratore da molti esempi fornitigli [23].

Il deep learning cerca di imitare il modo con cui gli umani guadagnano conoscenza. Il deep learning è un importante elemento della data science, che include modelli statistici e predittivi. Visto nella sua forma più semplice, il deep learning può essere pensato come

un modo per effettuare l'analisi predittiva, che usa i dati correnti e passati per prevenire attività, comportamenti e trend. Contrariamente ai tradizionali modelli di machine learning che sono lineari, gli algoritmi di deep learning sono organizzati in una gerarchia di crescente complessità e astrattezza. Ogni algoritmo della gerarchia applica una trasformazione non lineare al suo input e usa quello che impara per creare un modello statistico da dare in output. L'iterazione continua finchè il modello in output non ha raggiunto un certo livello di accuratezza. Il nome deep deriva proprio dal numero di livelli attraverso i quali i dati devono passare [23].

Nei tradizionali modelli di machine learning, il processo di apprendimento è supervisionato, e il programmatore deve essere estremamente preciso nello specificare al computer che tipo di caratteristiche deve cercare in un'immagine per determinare cosa contiene. Questa fase è chiamata feature extraction, e la frequenza di successo del computer dipende interamente dall'abilità del programmatore di definire accuratamente le caratteristiche. Il vantaggio del deep learning è che il programma costruisce da sè l'insieme di caratteristiche senza supervisore, provando, ad esempio, più volte ad identificare da solo cosa contiene un immagine. L'apprendimento non supervisionato non solo è più veloce, ma anche più accurato di solito [23].

Immaginiamo che il programma debba stabilire se un'immagine contiene o meno un cane. Inizialmente, al programma viene fornito come dati di allenamento, un insieme di immagini etichettate dal programmatore con "cane" o "non cane" con i metadati. Il programma utilizza le informazioni ricevute dai dati di allenamento per creare l'insieme di caratteristiche per il "cane" e costruisce un modello predittivo. In questo caso, il primo modello creato dal computer potrebbe predire che tutto ciò che ha quattro gambe e una coda deve essere etichettato come "cane". Con ogni iterazione, il modello predittivo diventa sempre più complesso e più accurato, identificando nuove caratteristiche più specifiche. A un programma che utilizza algoritmi di deep learning può essere mostrato un training set di milioni di immagini, identificando accuratamente quali immagini hanno cani nel giro di pochi minuti. Poiché la programmazione in deep learning può creare modelli statistici complessi direttamente dai propri risultati iterativi, è in grado di creare modelli predittivi accurati da grandi quantità di dati non etichettati e non strutturati.
Esistono vari metodi per creare dei forti modelli di deep learning [23]:

- Il **Learning rate decay** è un hyperparameter (un attributo che definisce il sistema o un insieme di condizioni per le operazioni precedenti alla fase di apprendimento) che controlla qunato cambia l'esperienza del modello in risposta all'errore stimato ogni

volta che i pesi del modello vengono definiti. Il suo valore non deve essere nè troppo alto nè troppo basso e quindi il metodo del learning rate decay è il processo di adattamento del learning rate per aumentare le prestazioni e ridurre i tempi di training. Il metodo più semplice di adattare il learning rate durante l'allenamento include tecniche di riduzione nel corso del tempo.

- Il **Transfer learning** prevede l'addestramento prima di una rete di base su un set di dati e un compito di base e poi si riadattano le caratteristiche apprese, o le si trasferiscono, a una seconda rete di destinazione da addestrare su un set di dati e un compito di destinazione. Questo processo tende a funzionare solo se le caratteristiche sono generali, cioè adatte sia al compito di base che a quello di destinazione, invece che specifiche del compito di base. È un'ottimizzazione che consente di progredire rapidamente o di migliorare le prestazioni nella modellazione del secondo compito.

- **Training from scratch**. Permette ad uno sviluppatore di raccogliere grandi quantità di dati etichettati e di configurare un'architettura di rete da zero, che può imparare le caratteristiche e i modelli. È l'approccio meno usato, l'allenamento richiede molto tempo a causa dei dati eccessivi. Rispetto al precedente metodo, richiede più dati e più tempo di computazione per ottenere risultati comparabili.

- **Dropout**. Questo metodo tenta di risolvere il problema dell'overfitting nelle reti con una grande quantità di dati eliminando casualmente delle unità (i neuroni) e la loro connessione dalla rete neurale durante l'allenamento per ridurre il rumore dei milioni di nodi della rete. È stato provato che il dropout migliora le prestazioni della reti neurali in attività di apprendimento supervisionato.

2.2.1 Reti Neurali

Un tipo avanzato di algoritmo di machine learning, conosciuto come rete neurale artificiale, sostiene molti modelli di deep learning. Esistono diversi tipi di reti neurali, che hanno benefici diversi per specifici casi d'uso. Il loro funzionamento, tuttavia, è molto simile, ovvero dandogli in pasto dati e lasciando che il modello capisca da sè se ha interpretato correttamente o preso la decisione giusta per un un elemento dato in input. Le reti neurali entrano all'interno di un processo di tentativi ed errori, per questo necessitano di un grande quantitativo di dati per allenarsi. I dati usati durante l'allenamento del modello devono essere etichettati affinchè il modello posso capire se le sue ipotesi sono corrette. Quindi i dati

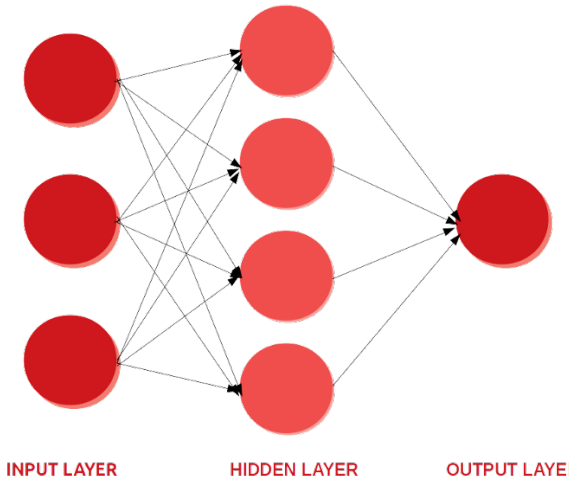

Figura 2.1: Rete Neurale - Architettura di base. Fonte [1].

non strutturati sono poco utili, possono essere solo analizzati dal modello di deep learning una volta che è stato allenato ed ha raggiunto un certo grado di accuratezza [24].

Una rete neurale artificiale coinvolge un grande numero di processori operanti in parallelo e organizzati in livelli. Il primo livello riceve le informazioni di input grezze ed esegue la pre-elaborazione dei dati, seguita dal passaggio dei dati filtrati al livello nascosto. Ogni livello successivo riceve l'output del livello precedente ad esso. L'ultimo livello produce l'output del sistema. Lo strato nascosto, ovvero i vari livelli intermedi, è costituito da reti neurali, algoritmi e funzioni di attivazione per recuperare informazioni utili dai dati. Una rete neurale funziona allo stesso modo in cui i neuroni più lontani dal nervo ottico ricevono segnali da quelli più vicini ad esso. Ogni nodo ha la sua piccola sfera di conoscenza, incluso cosa ha visto e le regole con cui è stato programmato o che ha sviluppato da sè. I livelli sono altamente interconnessi, cioè ogni nodo di un livello è connesso a più nodi appartenenti a quello successivo. Possono esserci uno o più nodi nel livello di output, dal quale la risposta che produce può essere letta. Le rete neurali sono degne di nota per la loro adattabilità, ovvero si modificano man mano che imparano dall'addestramento iniziale e gli utilizzi successivi forniscono maggiori informazioni sul mondo. Il modello di apprendimento più elementare è basato sul pesare l'input, che è come ogni nodo misura l'importanza dei dati in input ricevuti da ognuno dei suoi predecessori. Gli input che contribuiscono a ottenere una risposta giusta ricevono un peso più alto [24]. Ogni nodo o "neurone" della rete è dotato di una funzione di attivazione, che è una funzione matematica che collega l'input del neurone corrente all'output per il livello successivo. In pratica decide quando un neurone deve essere attivato o no. Le funzioni di attivazione possono essere lineari o non-lineari. Le prime, semplicemente, moltiplicano

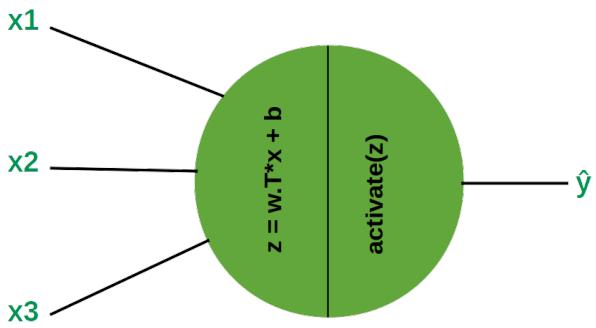

Figura 2.2: Un neurone. Fonte [1].

l'input ricevuto per il peso di ogni neurone e creano un segnale di output proporzionale all'input. La seconda tipologia consente di creare una corrispondenzza più complessa tra l'input e l'output, essenziale per gestire dati anch'essi complessi.

L'allenamento di una rete neurale consiste nel fornirgli input e dirle cosa l'output dovrebbe essere. Fornire le risposte consente al modello di adeguare le proprie ponderazioni interne per imparare come fare meglio il suo lavoro. Nel definire le regole e prendere decisioni, ovvero la decisione di ogni nodo su cosa inviare al livello successivo basandosi sull'input del livello precedente, le reti neurali utilizzano diversi principi tra i quali:

- L'allenamento basato sul gradiente. La gradient descent (discesa del gradiente) è un algoritmo di ottimizzazione comunemente utilizzato per addestrare modelli di apprendimento automatico e reti neurali. I dati di addestramento aiutano questi modelli ad apprendere nel tempo e la funzione di costo all'interno della gradient descent agisce specificamente come un barometro, misurando la sua precisione a ogni iterazione di aggiornamento dei parametri. Finché la funzione non è vicina o uguale a zero, il modello continuerà a regolare i suoi parametri per ottenere il minor errore possibile [25].

- La logica fuzzy è un approccio al calcolo basato su "gradi di verità" piuttosto che sulla consueta logica booleana "vero o falso" (1 o 0) su cui si basa il computer moderno. Viene utilizzata per imitare il ragionamento e la cognizione umana. Piuttosto che casi strettamente binari di verità, la logica fuzzy include 0 e 1 come casi estremi di verità, ma con vari gradi intermedi [26].

- Gli algoritmi genetici consistono in algoritmi che permettono di valutare diverse soluzioni di partenza (come se fossero diversi individui biologici) e che ricombinandole (analogamente alla riproduzione biologica sessuata) ed introducendo elementi di disordine (analogamente alle mutazioni genetiche casuali) producono nuove soluzioni

(nuovi individui) che vengono valutate scegliendo le migliori (selezione ambientale) nel tentativo di convergere verso soluzioni "di ottimo"; vengono utilizzati quindi per risolvere problemi di ottimizzazione [27].

- I metodi Bayesiani. La logica Bayesiana è una branca della logica applicata al processo decisionale e alla statistica inferenziale che si occupa dell'inferenza di probabilità: utilizzare la conoscenza di eventi precedenti per prevedere eventi futuri. Il Teorema di Bayes è un mezzo per quantificare l'incertezza. Basato sulla teoria della probabilità, il teorema definisce una regola per affinare un'ipotesi tenendo conto di ulteriori prove e informazioni di base, e porta a un numero che rappresenta il grado di probabilità che l'ipotesi sia vera [28].

Possono essere anche fornite regole di base sulle relazioni tra gli oggetti nei dati da modellare, che permettono di velocizzare l'allenamento e rendere il modello più potente. Tuttavia questo inserisce delle assunzioni sulla natura del problema, che possono rivelarsi irrilevanti o incorrette, rendendo cruciale la decisione sulle regole da inserire. Inoltre, le ipotesi che le persone fanno quando addestrano gli algoritmi fanno sì che le reti neurali amplifichino gli errori (o bias) culturali. Gli insiemi di dati distorti (ovvero che presentano bias) sono una sfida continua per l'addestramento di sistemi che trovano risposte da soli, riconoscendo pattern nei dati. Se i dati che alimentano l'algoritmo di machine learning non sono neutri, e quasi nessun dato lo è, l'algoritmo rischia di propagare gli errori [24].

Le reti neurali sono solitameante descritte in relazione alla loro profondità, incluso quanti livelli hanno tra l'input e l'output, o i cosiddetti livelli nascosti del modello. Questo è anche il motivo per cui il termine "rete neurale" è usato quasi come sinonimo di deep learning. Inoltre, può essere considerato anche il numero di nodi nascosti del modello oppure in termini di quanti input e output ogni nodo possiede. Le variazioni del design classico di rete neurale permette varie forme di propagazione dell'informazione avanti e indietro tra i vari livelli. I principali tipi di reti neurali artificiali includono le reti neurali feed-forward, le RNN, le CNN...che vedremo più nello specifico nella Sezione 2.3.4, applicate al contesto analizzato [24].

2.3 Big Code

Il codice sorgente è un tipo speciale di linguaggio naturale strutturato scritto dai programmatori, che può essere utilizzato dai modelli di Deep Learning visti precedentemente

nell'ambito del NLP. L'intelligenza artificiale che comprende e crea la struttura complessa del software ha molte applicazioni nell'ingegneria del software (SE). In particolare, Big Code è l'area di ricerca che utilizza Machine Learning e Deep Learning per la modellazione del codice sorgente. Per facilitare la costruzione di modelli di ML, la comunità di sviluppatori software offre dataset preziosi come le piattaforme di codice online come GitHub, i forum di risposta alle domande come Stack Overflow e la documentazione di vari software e strumenti di programmazione altamente strutturati e ricchi di contenuti. Rispetto ad altri domini, il DL per il Big Code è ancora in crescita e richiede più ricerca, compiti e dataset consolidati. Tra i compiti del Big Code, la generazione del codice sorgente è un campo importante per prevedere la struttura esplicita del codice o del programma da fonti di dati multimodali, come codice incompleto, programmi in un altro linguaggio di programmazione, descrizioni in linguaggio naturale o esempi di esecuzione. Gli strumenti di generazione del codice possono aiutare lo sviluppo di strumenti di programmazione automatica per migliorare la produttività della programmazione. Tuttavia, i modelli tradizionali di codice sorgente sono poco flessibili, lunghi da progettare per linguaggi/compiti specifici o incapaci di catturare le dipendenze a lungo termine delle dichiarazioni di codice. I modelli DL possono aiutare a risolvere i problemi sopra citati grazie alla loro capacità superiore di estrarre caratteristiche da vari formati di dati (ad esempio, linguaggio naturale e sequenze di simboli) e di catturare informazioni sintattiche e semantiche su varie scale [5].

2.3.1 Approcci tradizionali

Descriviamo quattro approcci tradizionali per gestire le sfaccettature sintattiche e semantiche del codice sorgente, tra cui (i) domain-specific language guided models (modelli guidati di linguaggio specifici per il dominio), (ii) probabilistic grammars (grammatiche probabilistiche), (iii) probabilistic language models (semplici modelli linguistici probabilistici, ad esempio, n-grammi) e (iv) simple neural language models (semplici modelli linguistici neurali). A questi approcci tradizionali si associano ancora diversi problemi seri, che possono essere gestiti efficacemente con i modelli di DL [5].

2.3.1.1 Domain-specific language guided models

I linguaggi specifici del dominio (DSL) sono spesso utilizzati per definire le regole di parametrizzazione e gli stati per generare la struttura di un programma. I modelli basati sui DSL creano regole grammaticali diverse per le dichiarazioni di codice comuni (ad esempio,

flusso di controllo, commenti e parentesi). Rispetto a un linguaggio di programmazione generico, la dimensione della grammatica di un DSL è solitamente inferiore, il che lo rende più efficiente per compiti specifici di generazione del codice [5]. Il modello basato su DSL è stato studiato in varie ricerche, che trattavano principalmente la sintesi di programmi. In particolare, Gulwani et al. [29] hanno posto la loro attenzione sul trovare un programma che soddisfi le richieste dell'utente, così da aiutare sia chi è privo di conoscenze riguardanti la programmazione sia i programmatori nel trovare in automatico problemi nei programmi, a capirli e a individuare nuovi algoritmi. Proprio per questo hanno ideato un sintetizzatore, che prende come input diverse forme di vincoli e restituisce in output un programma, cercandolo all'interno di uno spazio definito di programmi. Un sintetizzatore possiede tre dimensioni chiavi: le intenzioni dell'utente, espresse in forma di relazioni logiche input-output, dimostrazioni, linguaggio naturale; lo spazio di ricerca può comprendere la programmazione funzionale o imperativa, più ristretti modelli computazionali o rappresentazioni logiche concise; la tecnica di ricerca può essere basata sulla ricerca esaustiva, utilizzare tecniche di machine learning o tecniche di deduzione logica, solitamente composte da due fasi, ovvero la generazione dei vincoli e poi la risoluzione dei vincoli, usando risolutori Satisfiability Modulo Theory (SMT). Jha et al. [30], invece, si sono concentrati su due compiti specifici della sintesi di programmi, ovvero la scoperta di algoritmi non intuitivi e la deobfuscazione dei programmi. A differenza degli approcci tradizionali, che prevede una sintesi a partire da specifiche complete (come una formula appropriata o un programma più semplice), viene proposto un approccio alla sintesi dei programmi guidato da un oracolo, in cui un oracolo di I/O che mappa un dato input del programma verso l'output desiderato viene utilizzato come alternativa alla specifica completa, che spesso non è disponibile o più costosa. L'idea chiave dell'algoritmo consiste nell'interrogare l'oracolo di I/O su un input in grado di distinguere tra programmi non equivalenti che sono coerenti con l'interazione passata con l'oracolo di I/O. Il processo viene ripetuto fino a quando non si ottiene una semantica che si basa sull'input. Una caratteristica fondamentale di questo etodo è che è basato sui componenti, il che significa che sintetizza un programma eseguendo una composizione di componenti tratti da una determinata libreria di componenti. Può anche affrontare la sfida di identificare se l'insieme di componenti, scelto dall'utente in base al dominio, è insufficiente per sintetizzare il programma desiderato. Tale algoritmo di sintesi si basa su un nuovo approccio basato sui vincoli, che riduce il problema della sintesi alla risoluzione di due tipi di vincoli: il vincolo I/O-comportamentale, la cui soluzione produce un programma candidato coerente con l'interazione con l'oracolo I/O, e il vincolo di distinzione, la cui soluzione fornisce l'input che

distingue tra programmi candidati non equivalenti. Questi vincoli possono essere risolti con i solutori SMT, evitando una costosa ricerca combinatoria sullo spazio di tutti i programmi possibili. Gvero et al. [31], hanno pensato di aiutare i programmatori nel difficile compito di capire quali funzionalità, classi e metodi delle varie librerie esistenti utilizzare per comporre i propri programmi. La loro tecnica utilizza la type inhabitation per i termini del lamba calcolo in forma lunga normale. Viene utilizzata una rappresentazione concisa che fonde i tipi in classi equivalenti per ridurre lo spazio di ricerca, ricostruisce quindi un numero qualsiasi di soluzioni su richiesta. L'algoritmo realizzato è stato distribuito come plugin per Eclipse IDE per Scala col nome di InSynth. Lo strumento genera e suggerisce un elenco di espressioni che hanno il tipo desiderato e che considera tutti i valori dell'ambito corrente come i valori di partenza delle espressioni da sintetizzare. Trovare un frammento di codice del tipo dato ci porta direttamente al problema del type inhabitation: dato un tipo desiderato T e un ambiente di tipo Γ (una mappa dagli identificatori ai loro tipi), trovare un'espressione e di questo tipo T. Formalmente, trovare e tale che $\Gamma \vdash e : T$. Nella nostra implementazione, lo strumento calcola dalla posizione del cursore nel buffer dell'editor. Analogamente, cerca T esaminando il tipo dichiarato che appare a sinistra del cursore nell'editor. L'obiettivo dello strumento è trovare un'espressione e e inserirla nel punto corrente del programma, in modo da verificare il tipo complessivo del programma. Quando ci sono più soluzioni, lo strumento chiede allo sviluppatore di selezionarne una, come in scenari più semplici di completamento del codice. L'algoritmo utilizzando un calcolo di tipi succinti che produce classi di equivalenza di tipi che riducono lo spazio di ricerca nella ricerca diretta agli obiettivi, senza perdere la completezza. Inoltre, l'algoritmo genera una rappresentazione di tutte le soluzioni, da cui ne estrae un sottoinsieme. Tuttavia data la possibilità di un numero infinito di soluzioni di type inhabitation, per individuare la migliore introducono dei pesi per ogni dichiarazione di tipo per guidare la ricerca e classificare le soluzioni presentate. Un peso minore indica una formula più desiderabile. InSynth su un insieme di 50 benchmark si è dimostrato molto reattivo nel generare lo snippet previsto, che nella maggioranza dei casi, veniva restituito tra le prime decine di soluzioni. Tali ricerche hanno ridotto lo spazio di ricerca per il suggerimento di espressioni e poiché il processo di generazione è interpretabile dall'uomo, questo tipo di modello potrebbe essere un approccio promettente nella SE.

2.3.1.2 Grammatiche probabilistiche

Nei linguaggi formali, le regole di produzione possono essere utilizzate per definire tutte le possibili generazioni di stringhe (dichiarazioni di codice). Le grammatiche libere dal

contesto (Context-Free Grammars, CFG) sono un insieme di regole di produzione che possono essere applicate indipendentemente dal contesto [5]. Il CFG è un modo comune per definire la struttura di un linguaggio di programmazione, che può essere utilizzato per convertire il codice sorgente in alberi di sintassi astratta (AST[5]) [32]. La Probabilistic Context-Free Grammar (PCFG) è un'estensione della CFG, in cui le regole di produzione di un linguaggio context-free sono associate a un modello probabilistico. Bielik et al. [33] hanno generalizzato l'idea di PCFG al compito di completamento del codice in altri linguaggi non context-free come JavaScript. In seguito, Raychev et al. [34] hanno esteso il lavoro precedente sui PCFG apprendendo un albero decisionale per costruire un modello probabilistico utilizzando gli AST di un DSL proposto, TGen. Questi lavori hanno ottenuto ottimi risultati per alcuni compiti di completamento del codice. Un'altra estensione della CFG, la Tree Substitution Grammar (TSG), utilizza frammenti di albero invece di una sequenza di simboli per le regole di produzione. Tali regole della TSG sono definite da una Tree Adjoining Grammar [35], che può creare maggiore flessibilità e rappresentare meglio strutture linguistiche complesse[36]. Per limitare la complessità del modello e la grammatica rada, i ricercatori utilizzano spesso Bayes non parametrici per inferire[6] le distribuzioni [36]. Questi modelli sono adatti per il pattern mining, poiché la loro capacità di selezione automatica dei modelli consente di scoprire strutture più complesse. Tuttavia, i metodi bayesiani non parametrici sono spesso estremamente lenti da calcolare e difficili da scalare. Sebbene le grammatiche probabilistiche raggiungano prestazioni elevate per i linguaggi specifici del dominio, richiedono comunque regole progettate manualmente per modellare la localizzazione e il riutilizzo del codice.

2.3.1.3 Modelli linguistici basati su n-grammi

Un altro modello linguistico statistico semplice ma efficace è quello a n-grammi.
Più precisamente, questo modello presuppone che ogni token/parola sia condizionatamente dipendente dagli n - 1 token/parole precedenti, come descritto nella seguente equazione: $P(W) = \Pi_t P(w_t \mid w_{t-(n-1),\ldots,t-1})$, che può essere semplicemente calcolato contando le occorrenze di tutti gli n-grammi nell'insieme di addestramento. Un vantaggio diretto degli n-grammi rispetto ai modelli sintattici (ad esempio, le grammatiche probabilistiche e i modelli

[5]AST: una rappresentazione ad albero della struttura sintattica astratta del testo scritto in un linguaggio formale.Astratta perchè non rappresenta ogni dettaglio presente nella sintassi reale, ma solo quelli strutturali o relativi al contenuto. Ogni nodo dell'albero denota un costrutto che si verifica nel testo.

[6]L'inferenza è il procedimento per cui si inducono le caratteristiche di una popolazione dall'osservazione di una parte di essa (detta "campione"), selezionata solitamente mediante un esperimento casuale (aleatorio).

guidati da DSL) è che sono più facili da generalizzare, poiché le dipendenze e le regole dei linguaggi di programmazione vengono apprese automaticamente dal codice sorgente [5]. Hindle et al. [37] hanno preso l'iniziativa di utilizzare gli n-grammi per costruire un modello linguistico per il codice sorgente. Da allora, oltre al completamento del codice [38, 39], i modelli a n-grammi sono stati applicati anche ad altri compiti, come l'estrazione di idiomi [40], il rilevamento di errori di sintassi [41], l'analisi del codice sorgente [42] e l'offuscamento del codice [43]. Tuttavia, i modelli linguistici semplici come gli n-grammi non sono in grado di catturare i paradigmi di programmazione di alto livello. Per risolvere questo problema, una serie di lavori ha migliorato i modelli linguistici per renderli più adattabili alle informazioni locali. Bielik et al. [44] hanno utilizzato un programma di DSL, per apprendere una buona rappresentazione dei dati, comprendere dipendeze non banali e parametrizzare il modello basato su n-grammi. In questa ricerca hanno mostrato ottimi risultati empirici per la modellazione del linguaggio di programmazione. Il modello risultante era anche più efficiente nell'addestramento e nell'inferenza rispetto ai modelli neurali. Hellendoorn et al. [45] hanno aggiunto una cache locale di n-grammi e hanno unito le previsioni dei modelli locali e globali. Entrambi i modelli sono stati dichiarati superiori alle controparti DL (ad esempio, RNN o LSTM) all'epoca della loro pubblicazione; pertanto, questi due modelli sarebbero buoni punti di riferimento per la modellazione del codice sorgente. Si noti che gli n-gram non sono in grado di modellare le dipendenze a lungo termine tra i token, come anche gli altri modelli statistici. Il troncamento di n-gramma scarta le informazioni posizionali a lungo termine. Inoltre, un'altra limitazione di un modello a n-grammi è la scarsità della rappresentazione vettoriale della parola o del token di codice, causata dall'ampio vocabolario del codice sorgente. Questo problema di sparsità può essere risolto utilizzando la rappresentazione distribuita[7] dei modelli linguistici neurali.

2.3.1.4 Modelli semplici di programmi neurali

Invece di incorporare esplicitamente la frequenza di ogni token precedente, i modelli di embedding delle reti neurali con un solo strato nascosto convertono innanzitutto la codifica one-hot di una parola in un vettore intermedio di word-embedding di lunghezza molto inferiore rispetto alla dimensione del vocabolario [5]. Questa idea è nota anche come rappresentazione distribuita delle parole. Nel lavoro condotto da Schwenk et al. [46], le incorporazioni di parola di un massimo di n - 1 token precedenti rispetto alla parola corrente

[7] Rappresentazione distribuita: le informazioni all'interno di una rete neurale sono distribuite per tutti i vari nodi della rete e non in un "posto" singolo.

sono state inserite in una rete neurale completamente connessa con uno strato nascosto. Allo strato di uscita è stata applicata una funzione softmax[8] per calcolare la probabilità della parola successiva. Tuttavia, un grave inconveniente di questo modello originale è l'elevato costo computazionale dello strato nascosto. Pertanto, è stato presentato [47] il Deep Learning for Source Code Modeling and Generation 5 per affrontare questa sfida, sostituendo le attivazioni non lineari con una matrice di contesto per determinare il vettore di contesto rispetto alla parola corrente. Quindi, veniva calcolata la somiglianza tra il vettore di contesto dei token precedenti e la parola corrente. In seguito, sono stati proposti diversi metodi per accelerare i tempi di addestramento e di predizione utilizzando un'architettura gerarchica [48]. In questo lavoro precedente, è stato dimostrato che il modello log-bilineare supera le tradizionali reti neurali completamente connesse e i modelli linguistici a n-grammi per il compito di modellare APNews. In seguito, l'idea del modello di incorporazione della rete neurale è stata adottata dai ricercatori per la modellazione del codice sorgente, che può essere definito come modello semplice di programma neurale. In particolare, Maddison et al. [49] hanno combinato il log-bilineare con una tecnica di ricerca ad albero (Logbilinear Tree-Traversal models) per attraversare l'AST, generando codice sorgente comprensibile all'uomo. Il modello log-bilineare, dato il contesto $w_1 : n - 1$, predice innanzitutto la rappresentazione della parola successiva w_n combinando linearmente le rappresentazioni delle parole del contesto. Quindi la distribuzione della parola successiva viene calcolata in base alla somiglianza tra la rappresentazione prevista e le rappresentazioni di tutte le parole del vocabolario. Allamanis et al. [50] hanno esteso l'approccio di Maddison et al. per recuperare frammenti di codice sorgente da query in linguaggio naturale e viceversa. Allamanis et al. [51] hanno anche utilizzato un modello log-bilineare per raccomandare i nomi dei metodi e delle classi per la programmazione orientata agli oggetti in Java e questo modello ha superato il modello a n-grammi in entrambi i compiti. Come i modelli a n-grammi, la conoscenza dei modelli log-bilineari è limitata agli n - 1 token precedenti. Pertanto, i lavori precedenti hanno richiesto la definizione esplicita del contesto globale e locale per i modelli log-bilineari, al fine di catturare le dipendenze a breve e lungo termine e le proprietà sequenziali del codice sorgente. L'elenco dei contesti è ancora umano e incompleto, il che limita le sue applicazioni in nuovi domini. Tuttavia, grazie alla loro semplicità, i modelli di programmi neurali semplici vengono utilizzati come caratteristiche di input (pre-addestrate) per vari compiti di Big Code. Un'estensione di questo modello è la Feed-forward neural network (Multi-Layer Perceptron),

[8] La funzione softmax è, di solito, l'ultima funzione di attivazione della rete neurale per normalizzare l'output di una rete a una probabilità distribuita per predirre la classe dell'output.

una delle più semplici varianti di rete neurale. Le informazioni vengo passate tra i livelli in un'unica direzione, attraverso i vari nodi di input, fino al nodo di output. La rete può avere come anche no dei livelli di nodi nascosti, rendendo il loro funzionamento più interpretabile. Tuttavia, neanche questa evoluzione riesce a carpire le dipendenze e le proprietà sequenziali delle sequenze di parole e non è molto usata, se non combinata con altre tecniche [14].

2.3.2 Vantaggi nell'utilizzo del DL

Il framework encoder-decoder [52] con i modelli DL può essere utilizzato per catturare efficacemente le dipendenze e le proprietà sequenziali di una sequenza di input. Più precisamente, i modelli DL sono adatti per la modellazione e la generazione di codice, poiché sono in grado di soddisfare i seguenti quattro aspetti importanti: (i) generazione automatica di caratteristiche, (ii) cattura delle dipendenze a lungo termine e delle proprietà sequenziali, (iii) apprendimento end-to-end e (iv) generalizzabilità. I modelli esistenti devono trovare un compromesso tra queste quattro proprietà. Ad esempio, i modelli a n-grammi possono estrarre automaticamente le caratteristiche dal codice sorgente, ma non possono catturare bene le dipendenze a lungo termine a causa dell'esplosione combinatoria dei termini. Allo stesso modo, i semplici modelli di programmi neurali (ad esempio, i modelli completamente connessi o log-bilineari) richiedono ancora regole progettate dall'uomo per catturare le dipendenze e la struttura del codice sorgente, il che limita la loro formazione e generalizzabilità end-to-end. Al contrario, con una profonda conoscenza del dominio incorporata, i modelli guidati da DSL e le grammatiche probabilistiche possono catturare efficacemente le dipendenze e le proprietà sequenziali, ma le regole strettamente definite rendono i modelli più difficili da generalizzare e automatizzare il processo di apprendimento in nuovi domini [5].

2.3.3 Framework encoder-decoder

Approfondiamo ora il framework encoder-decoder che utilizza modelli di DL per la modellazione di sequenze/codici. In NLP, l'obiettivo principale è quello di elaborare una grande quantità di linguaggio naturale, per lo più sotto forma di testo o di voce umana. Il codice - un mezzo di comunicazione per gli sviluppatori - è simile al linguaggio naturale [53, 37], che ha strutture sintattiche e significato semantico. In particolare, esistono diversi compiti nell'ambito Big Code (cfr. Sezione 2.3.5) il cui input è una sequenza (ad esempio, codice sorgente e/o linguaggio naturale) e il target di predizione può essere una sequenza o un semplice valore numerico/categoriale. Ispirandosi al campo dell'NLP, tali compiti

di Big Code possono essere formulati in un quadro di encoder-decoder. Se sia l'input che l'output sono sequenze, il framework encoder-decoder può essere chiamato anche seq2seq [52]. La Figura 2.3 illustra le fasi principali di un framework encoder-decoder. I componenti delle fasi 1 e 3 sono denominati rispettivamente encoder e decoder, che in genere sono due modelli di Deep Learning (cfr. Sezione 2.3.4). Nella ricerca condotta [52], lo stato interno finale dell'encoder viene utilizzato come vettore di contesto, che contiene informazioni sulla sequenza di input. In lavori più recenti, la fase 2 è spesso gestita da un meccanismo di attenzione [54] (cfr. Sezione 2.3.4.3) o da memorie esterne [55] (cfr. Sezione 2.3.4.4) con una sequenza di contesto più ricca e sensibile alla posizione [5]. Il vettore di contesto viene passato al decoder che lo usa per generare la sequenza di destinazione.

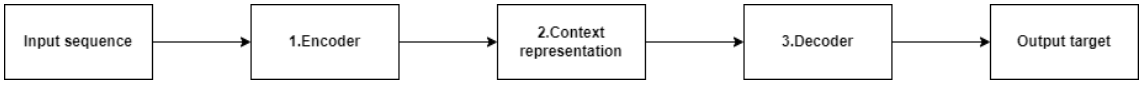

Figura 2.3: Framework encoder-decoder

2.3.4 Modelli di DL utilizzati

Questa sezione è importante per due motivi: (i) i modelli/tecniche deep presentati possono essere estesi al codice sorgente e (ii) solo una piccola parte di tali modelli è stata utilizzata per la modellazione del codice sorgente [5]. I modelli di deep learning utilizzati all'interno del framework encoder-decoder si possono dividere in due classi principali, ovvero le reti neurali recurrent e non-recurrent. A questi modelli si aggiungono poi delle tecniche per renderli più robusti: il meccanismo dell'attention, la memoria esterna e la beam search [5].

2.3.4.1 Recurrent Neural Networks

L'idea principale delle Recurrent Neural Network (RNN) è quella di elaborare in modo efficiente i dati sequenziali, grazie alla capacità di memorizzare l'input grazie a una memoria interna associata [56]. Questo è fondamentale per prevedere i risultati in modo più preciso, soprattutto in ambito di NLP, dato che il testo è un tipo di dato sequenziale. In genere, una rete neurale tradizionale elabora l'input e passa al successivo senza considerare alcuna sequenza. I dati sequenziali, invece, vengono elaborati seguendo un ordine specifico, necessario per comprenderli in modo distinto. Una rete feed-forward non è in grado di comprendere la sequenza, in quanto ogni ingresso è considerato individuale. Al contrario, per i dati delle serie temporali, ogni ingresso dipende dall'ingresso precedente. Un'altra differenza tra le reti feed-forward e le RNN, come possiamo vedere dalla Figura 2.4, è che nella prima le informazioni

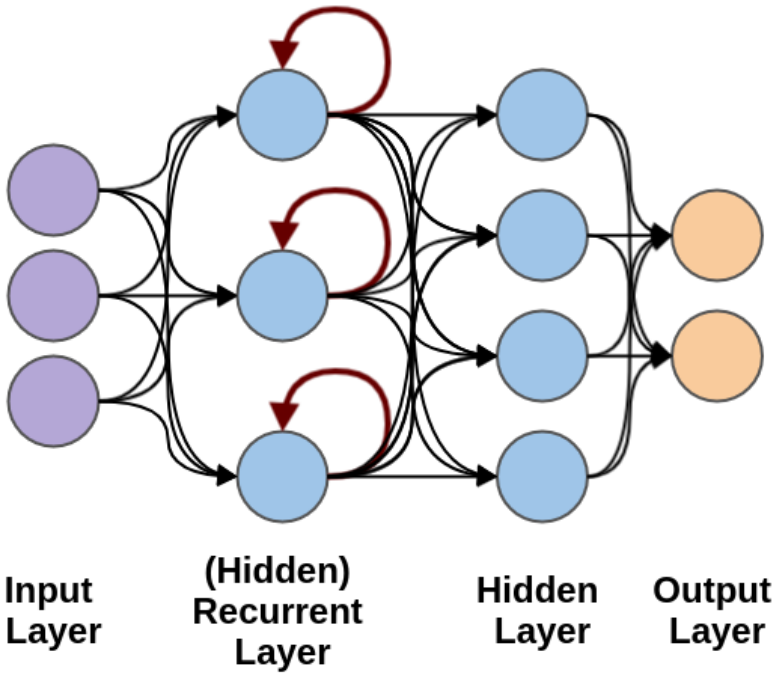

Figura 2.4: Recurrent Neural Network - Architettura. Fonte [2].

possono muoversi in una sola direzione, cioè, dallo strato di ingresso allo strato nascosto e quindi allo strato di uscita, mentre nella seconda, le informazioni che passano attraverso l'architettura attraversano un ciclo. Ogni input dipende dal precedente per prendere decisioni. La RNN assegna lo stesso peso e lo stesso bias a ciascuno degli strati della rete. Pertanto, tutte le variabili indipendenti vengono convertite in variabili dipendenti. I loop della RNN garantiscono la conservazione delle informazioni nella sua memoria. Questo meccanismo viene denominato retropropagazione (backpropagation) ed è stato una grande aggiunta alla procedura di addestramento. L'obiettivo dell'utilizzo della retropropagazione è quello di ripercorrere la rete neurale in modo da identificare qualsiasi derivata parziale dell'errore rispetto ai pesi. Questo ci permette di rimuovere tali valori dai pesi. Le derivate vengono utilizzate dalla gradient descent per minimizzare una determinata funzione di perdita (ctr. Sezione 2.2.1). I pesi vengono regolati in modo da ridurre il tasso di errore.

Le RNN possono essere usati per trasferire informazioni da un sistema all'altro, come tradurre frasi da una lingua ad un'altra. Sono anche usati per identificare pattern nei dati che possono aiutare nell'identificazione di immagini [5]. Un RNN può essere allenato a riconoscere differenti oggetti in un'immagine o per identificare le varie parti di un discorso in una frase. Tuttavia, le RNN semplici sono difficili da addestrare [57], a causa dei problemi dell'exploding gradient e vanishing gradient. Il problema dell'exploding gradient si riferisce

al grande aumento della norma[9] del gradiente durante l'addestramento e rende impossibile modificare i pesi dei parametri della rete. Il problema del vanishing gradient si riferisce al comportamento opposto, quando la norma del gradiente è molto piccola e l'aggiornamento dei pesi è inefficace, rendendo impossibile per il modello apprendere la correlazione tra eventi temporalmente lontani. Inoltre, non sono in grado di conservare le informazioni passate su diverse scale temporali.

Le RNN gated, come le Long Short-Term Memory (LSTM) [58] e le Gated Recurrent Units (GRU) [59], permettono di rimediare ai problemi riguardanti il gradiente, modellano i meccanismi di mantenimento e dimenticanza in modo esplicito con attivazioni sigmoidali, ovvero porte, per imparare dipendenze a lungo termine. Le reti LSTM usano un meccanismo di gating per limitare il numero di precedenti fasi che possono influenzare la fase corrente. Un LSTM ha tre porte per controllare rispettivamente l'ingresso, l'uscita e la dimenticanza. Inoltre, c'è una cella di memoria per generare gli stati nascosti. Le reti LSTM sono comunemente usate nelle attività di NLP perché possono imparare il contesto richiesto per processare sequenze di dati. Un GRU, invece, è dotato di due porte per gestire l'aggiornamento, ovvero quante informazioni devono essere passate al prossimo stato (mantenendo tutte le informazioni è possibile eliminare il rischio di vanishing gradient), e il reset, per decidere quante informazioni del passato è necessario trascurare, cioè decide se lo stato precedente della cellula è importante o meno. Le GRU urilizzano meno memoria rispetto alle LSTM e sono più veloci, dal momento che usano meno parametri. Tuttavia, le LSTM sono più accurate quando si utilizzano dataset di grandi dimensioni con sequenze più lunghe.

Le unità RNN possono essere rese profonde, ovvero con più livelli, per codificare transizioni più complesse [60]. Per stabilizzare i gradienti di addestramento nelle RNN ricorrenti sono stati introdotti gli Highway Layer [61], che permettono di addestrare più facilmente reti neurali molto profonde. L'architettura introdotta utilizza delle piccole unità di regolazione, che, appunto, imparano a regolare il flusso di informazioni attraverso la rete. Queste reti, con centinaia di livelli, possono così essere allenate utilizzando il metodo dello stochastic gradient descent e una grande varietà di funzioni di attivazione. Per catturare le dipendenze a lungo termine nelle serie temporali e rappresentare informazioni gerarchiche, gli strati delle RNN possono essere impilati con diverse frequenze di aggiornamento [62]. Le RNN gated feedback [63] consentono alla rete di apprendere le proprie frequenze di clock utilizzando porte aggiuntive. Lo stato dell'arte delle RNN per il modello linguistico è il modello Fast-Slow RNN [64] che incorpora i punti di forza delle RNN profonde, che cercano di risolvere la

[9] La norma è il quadrato del modulo di un vettore, ovvero della lunghezza del suo segmento.

relazione poco profonda tra gli stati nascosti e tra gli stati nascosti e l'output per aumentarne l'efficienza, e di quelle multiscala, che raggruppano le unità nascoste in più moduli di diversa scala temporale per modellare la rappresentazione gerarchica e temporale tipica del cervello umano ed imparare in modo più efficiente.

2.3.4.2 Non-Recurrent Neural Network

Le convolutional neural networks (CNN) sono uno dei modelli più utilizzati oggi. Questa rete neurale utilizza una variante dei percettori multistrato e contiene uno o più strati convoluzionali che possono essere interamente connessi o raggruppati.

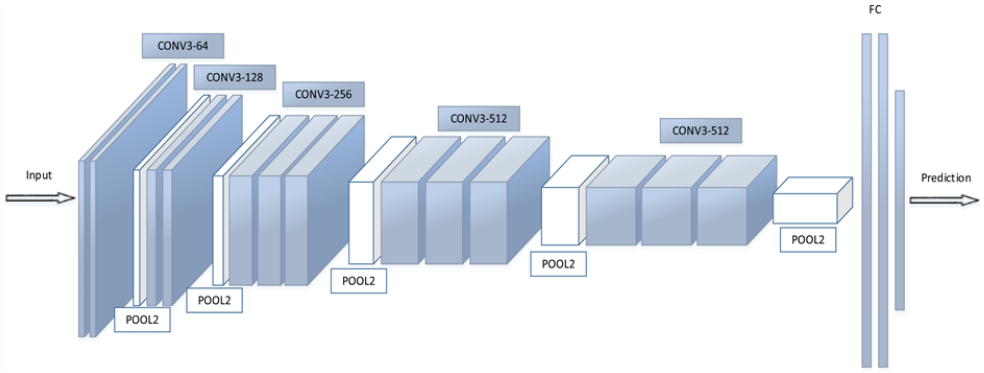

Figura 2.5: Convolutional Neural Network - Architettura. Fonte [3].

Come possiamo vedere nella Figura 2.5, le CNN hanno tre tipi di livelli [65]:

- Strato convoluzionale (CONV). È il primo livello in una CNN. Prende come input una matrice di dimensioni [h1 * w1 * d1]. Successivamente, presenta diversi kernel (o filtri). Un kernel è una matrice di dimensioni [h2 * w2 * d1] ed ha quindi lo stesso numero di canali dell'input. Per ogni strato convoluzionale, vi sono più kernel impilati uno sopra l'altro (con d2 indichiamo il numero di kernel). Ogni kernel ha un proprio errore (bias), che è una quantità scalare. Infine vi è il livello di output, che è una matrice di dimensione [h3 * w3 * d2]. Per ogni posizione del kernel, ogni numero sul kernel viene moltiplicato con il numero corrispondente sulla matrice di ingresso e poi tutti vengono sommati, tenendo conto anche del bias del kernel per ottenere il valore nella posizione corrispondente nella matrice di uscita. Con d1 > 1, la stessa cosa avviene per ciascuno dei canali.

- Strato di pooling (POOL). Viene utilizzato epr ridurre il numero di parametri dell'input aiutando a ridurre l'overfitting, estrarre le caratteristiche più importanti dall'input e ridurre la computazione. I principali metodi sono il Max Pooling e l'Average Pooling.

Nel primo, un kernel di dimensione nxn viene spostato sulla matrice e per ogni posizione viene preso il valore massimo e inserito nella posizione corrispondente nella matrice di output; nel secondo, invece, l'unica differenza è che viene preso il valore medio tra tutti i valori della matrice. Questo viene ripetuto per ogni canale.

- Stato totalmente connesso (FC). Si tratta semplicemente di una feed forward neural network e costituisce gli ultimi strati delle CNN. L'input è la matrice di output del pooling finale o del livello convoluzionale, che viene "appiattita", ovvero tutti i suoi valori vengono srotolati per passare da una matrice tridimensionale ad un vettore. Questo vettore è poi connesso ad alcuni strati totalmente connessi, che sono come Reti Neurali Artificiali ed eseguono le stesse operazioni matematiche. Ogni livello di questa rete esegue il seguente calcolo $g(Wx + b)$, con g che è la funzione di attivazione, x il vettore di input di dimensioni [p_l, 1], W è la matrice dei pesi di dimensione [p_l[10], n_l[11]] e b è il vettore di errore di dimensione [p_l, 1]. Questo calcolo è ripetuto per ogni livello. Infine, viene utilizzata una funzione softmax per ottenere le probabilità dell'input di appartenere ad una particolare classe (classificazione).

Il modello CNN è particolarmente popolare nel campo del riconoscimento delle immagini, la digitalizzazione del testo e l'elaborazione del linguaggio naturale. Le reti neurali convoluzionali (CNN) sono state utilizzate in diversi compiti di modellazione delle frasi. Nel 2013, Kalchbrenner e Blunsom [66] hanno utilizzato una CNN come encoder e una RNN come decoder per la generazione di dialoghi. Un anno dopo, Blunsom et al. hanno proposto Dynamic CNN [67] per la modellazione semantica delle frasi, in cui la lunghezza variabile e la scoperta delle relazioni erano abilitate dal max pooling. Tuttavia, questi lavori precedenti non sono riusciti a raggiungere prestazioni simili a quelle delle LSTM. Recentemente, sono emerse nuovamente strutture non ricorrenti con prestazioni simili a quelle delle RNN, ma più veloci da calcolare. Gli strati di convoluzione mascherati sono utilizzati come decodificatori in un sistema di traduzione automatica neurale [68]. Gehring et al. [69] hanno proposto ConvS2S che ha introdotto la connessione di salto [70] e l'attenzione [54] (cfr. Sezione 2.3.4.3) nella modellazione delle frasi e ha raggiunto lo stato dell'arte della traduzione. Anche la combinazione di unità ricorrenti e convoluzionali è utile. He et al. [71] hanno rafforzato la correlazione ingresso-uscita aggiungendo convoluzioni cross-layer alle RNN impilate.

Vaswani et al. [72] hanno proposto un modello di attenzione multitesta (Multi-Head attention) chiamato Transformer che si basa sull'autoattenzione e sulla codifica posizionale

[10] p_l è il numero di neuroni nel livello precedente della rete.

[11] n_l è il numero di neuroni nel livello corrente della rete.

per calcolare le rappresentazioni della sequenza. I transformer sono un particolare tipo di rete neurale che ha iniziato a prendere piede grazie ai miglioramenti che ha apportato in effecienza e accuratezza nella gestione di task riguardanti il natural language processing. Transformer consente al decodificatore di prestare attenzione a informazioni arbitrariamente distanti e riduce significativamente il tempo di addestramento senza perdita di qualità. Come la CNN, la struttura causale viene mantenuta mascherando l'output successivo per la fattorizzazione autoregressiva. Queste reti si spingono oltre al trovare dei pattern all'interno di dati o all'imparare ripetendo le stesse azioni, ma possono imparare dal contesto e creare nuove informazioni.

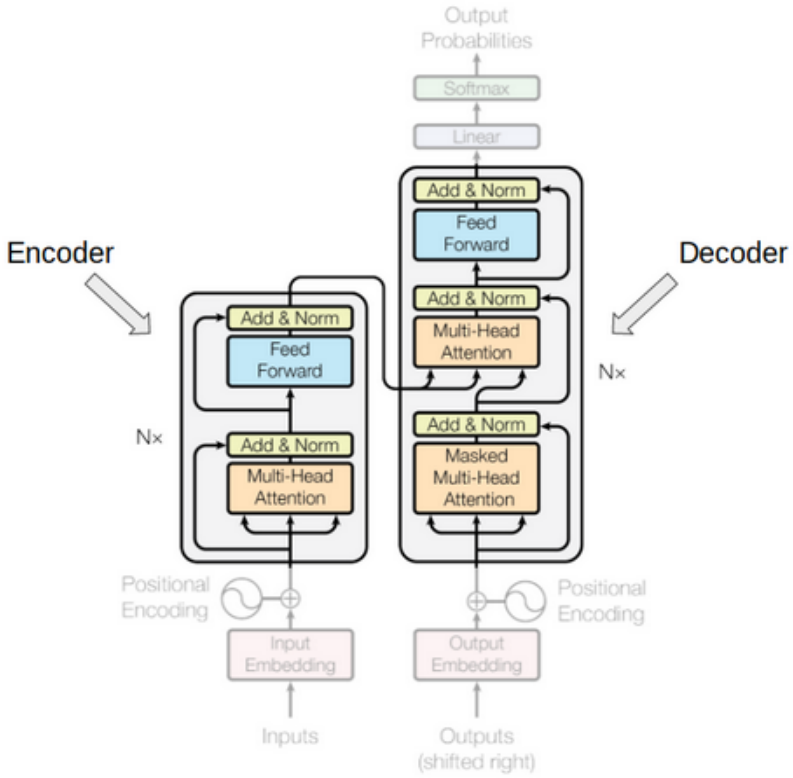

Figura 2.6: Transformer - architettura del modello. Fonte [4].

I transformer, come vediamo nella Figura 2.6, riprendono la struttura delle RNNs e delle reti convoluzionali utilizzando strati di auto-attenzione e di reti feed-forward completamente connesse sia per l'encoder che per il decoder. L'encorder mappa la sequenza di sinboli in input $(x_1,...,x_n)$ in una sequenza di rappresentazioni continue $\mathbf{z}=(z_1,...,z_n)$. Data la sequenza $\mathbf{z}$, il decoder genera una sequenza di output $(y_1,...,y_m)$ di simboli un elemento alla volta. Ad ogni passo il modello è auto-regressivo ed utilizza i simboli generati precedentemente come un input aggiuntivo per produrre il prossimo [72].

I transformer combinano alcuni dei benefici tradizionalmente visti con le CNN e le RNN, due delle più comuni architetture di reti neurali usate nel deep learning [73]. CNN è usata maggiormente per riconoscere oggetti nelle immagini. È più facile elaborare diversi pixel in parallelo per distinguere linee, forme e oggetti interi. Ma hanno difficoltà con flussi di input continui come il testo. D'altra parte le RNN vengono usate comunemente per l'NPL, perchè sono buone nel valutare flussi continui di input che includono stringhe di testo. La tecnica funziona bene quando si analizza la relazione tra parole che sono vicini tra loro, ma perde accuratezza nel modellare la relazione tra parole alla fine di una lunga frase o di un paragrafo. Per superare queste limitazioni, i ricercatori si sono rivolti ad altre tecniche di elaborazione delle reti neurali, come long short-term memory, per aumentare i feedback tra neuroni. Le nuove tecniche miglioravano le prestazioni degli algoritmi, ma non molto quelle dei modelli usati per tradurre testi lunghi. I ricercatori hanno quindi iniziato a esplorare altri modi per collegare i processori della rete neurale per rappresentare la forza della connessione tra le parole. La modellazione matematica della forza di quanto fortemente due parole sono connesse è stata chiamata attenzione (attention), che vedremo più in dettaglio nella Sezione 2.3.4.3. All'inizio, l'idea era che l' "attention" fosse qualcosa di aggiunto come passo supplementare per aiutare a organizzare un modello per un'ulteriore elaborazione da parte di un modello RNN. I ricercatori di Google hanno scoperto che potevano ottenere risultati migliori eliminando del tutto le RNN e migliorando il modo in cui i transfomer modellavano le relazioni tra le parole. Ciò conferisce ai trasformatori due vantaggi chiave rispetto ad altri modelli. In primo luogo, possono essere più precisi perché riescono a comprendere la relazione tra elementi sequenziali distanti tra loro. In secondo luogo, sono più veloci nell'elaborare una sequenza perché prestano maggiore attenzione alle sue parti più importanti.

I transformer hanno cambiando considerevolmente il modo con cui lavoriamo con i dati testuali, tuttavia presentano alcune limitazioni. In particolare, l'attenzione può gestire solo stringhe con una lunghezza fissata, quindi il testo viene diviso in un certo numero di segmenti prima di essere immesso nel sistema come input. Proprio questo troncamento del testo può causare una frammentazione del contesto, dividendo parti cruciali del testo. Queste limitazioni hanno portato all'introduzione di una nuova architettura, denominata Transformer-XL, dove gli stati nascosti ottenuti nei segmenti precedenti vengono riutilizzati come fonte di informazioni per il segmento corrente. Ciò consente di modellare una dipendenza a lungo termine, poiché le informazioni possono fluire da un segmento all'altro [74]. Un esempio di questa tipologia di trasformer è XLNet. Si tratta di una rete neurale artificiale allenata su un insieme di dati, che identifica i pattern che sono usati per costruire conclusioni

logiche. Un motore NLP può estrarre informazioni da una semplice query in linguaggio naturale. XLNet mira ad insegnare a se stesso per essere capace di leggere e interpretare testo e usa queste conoscenze per scrivere nuove frasi. XLNet ha due parti: un codificatore e decodificatore. Il codificatore usa le regole sintattiche dei linguaggi per convertire frasi in rappresentazioni basate sui vettori; il decodificatore usa queste regole per convertire la rappresentazione vettoriale nuovamente in una frase sensata.

2.3.4.3 Meccanismo di attenzione

Un problema del sistema originale di codifica-decodifica è che il decodificatore può accedere a un solo vettore di contesto. L'uomo comprende le righe di testo prestando ripetutamente attenzione a diverse parti di una sequenza. Per simulare questo comportamento, Bahdanau et al. [54] hanno utilizzato una sequenza come contesto e hanno proposto un meccanismo di attenzione (attention) per adattare i pesi del contesto associato a un certo stadio di uscita e imporre un allineamento esplicito tra i token di ingresso e quello di uscita. Per attenzione ci si riferisce alla descrizione matematica di come le cose (ad esempio, le parole) si relazionano, si completano e si modificano a vicenda. Una funzione di attenzione può essere descritta come la corrispondenza tra una query (Q) e un insieme di coppie chiave(K)-valore(V) a un output, dove tutti sono in forma di vettori. Più precisamente, con una sequenza codificata F e a ogni passo t, lo stato nascosto h_t viene calcolato utilizzando un modello RNN con il vettore sorgente c_t generato dal meccanismo di attenzione come input aggiuntivo: h_t = RNN(h_{t-1}, [e_{t-1}; c_t]). Questo modo di incorporare l'attenzione è noto come early binding. In alternativa, l'attenzione può essere considerata appena prima di generare il token di uscita. Una tipica attenzione morbida simile a quella di Bahdanau et al. [54] può essere calcolata come segue:

1. Con lo stato nascosto precedente h_{t-1}, l'energia di attenzione viene calcolata con una funzione di punteggio basata sul contenuto u_t = score(F, h_{t-1}).

2. Esponenzia e normalizza u_t a 1: a_t = softmax(u_t).

3. Calcolo del vettore sorgente in ingresso c_t = Fa_t .

Il modo più semplice per definire il punteggio è il prodotto scalare: score(F, h_{t-1}) = $F^T h_{t-1}$. Oppure si può definire un embedding di ingresso atteso V in modo che score(F, h_{t-1}) = $F^T V h_{t-1}$. Nell'articolo originale [54], l'energia di attenzione è calcolata con un percettrone multistrato (MLP) come segue: score(F, h_{t-1}) = v^T tanh(WF + Vh_{t-1}). L'energia di attenzione

basata sul contenuto viene calcolata assegnando un punteggio a ogni elemento separatamente, il che rende difficile discriminare gli elementi con contenuto simile ma posizioni diverse. L'attenzione sensibile alla posizione [75] ha superato questa limitazione generando le attenzioni in modo autoregressivo. Tuttavia, lo scorrimento attraverso un'altra RNN durante la retropropagazione può aumentare notevolmente il tempo di calcolo. Vaswani et al. [72] hanno presentato un'autoattenzione multitesta per rappresentare il contesto rilevante di ogni parola in una sequenza di input in posizioni diverse. Combinando la modellazione della sequenza e il meccanismo di attenzione, sono stati raggiunti risultati all'avanguardia per la traduzione automatica neurale [54, 76, 77, 78]. Inoltre abbiamo osservato come sia uno dei meccanismi alla base del funzionamento dei Transformer, che hanno consentito di raggiungere grandi risultati nel risolvere i task previsti dal Big Code.

2.3.4.4 Reti neurali ad alta intensità di memoria

L'attenzione è strettamente legata alla memoria esterna. Insieme, sono diventati importanti elementi costitutivi di una rete neurale. Le memorie esterne sono utilizzate come stati interni, che possono essere aggiornati dal meccanismo di attenzione per la lettura e l'aggiornamento in modo selettivo. La classica rete di memoria (MemNN) [55] ha cercato di imitare una memoria ad accesso casuale (RAM) e di utilizzare l'attenzione morbida come versione differenziabile dell'indirizzamento. Una rete di memoria di solito riceve i seguenti input:

- Una query q è l'ultimo enunciato pronunciato dal parlante in un contesto di dialogo generale o la domanda in un contesto di AQ.
- Un vettore di memoria m è la storia del dialogo del modello. La conoscenza può essere grande quanto l'intera base di codice o la documentazione, se il modello è sufficientemente potente.

Questo tipo di rete ha i seguenti moduli per gestire gli input:

- Un codificatore converte q in un vettore utilizzando una RNN [79] o un più semplice word embedding [80].
- Un modulo di memoria M trova la parte migliore di m relativa a q. Questa è la fase di indirizzamento.
- Un modulo di controllo C invia q a M e legge la memoria pertinente, aggiungendola allo stato corrente. In pratica, questo processo viene sempre ripetuto per consentire ragionamenti complessi.

- Un decodificatore genera l'output dagli stati finali.

L'addestramento di MemNN è completamente supervisionato, in cui l'etichetta della parte migliore della memoria viene fornita in ogni fase dell'indirizzamento della memoria. Il suo seguito, End-to-End Memory Network [81], utilizza la soft-attention per l'indirizzamento della memoria per addestrare l'attenzione in retropropagazione e rilassare la supervisione solo sull'output. Si nota che l'uso di un'unità di memoria per soddisfare sia l'interrogazione che lo stato limita l'espressività. Suddividendo la memoria in una coppia chiave-valore, Millor et al. [82] hanno codificato la conoscenza pregressa e ottenuto risultati migliori. Recurrent Entity Network [83] dimostra come migliorare l'unità di memoria lasciando che l'agente impari a leggere e scrivere la memoria per tenere traccia dei fatti. Weston [84] ha generalizzato questo modello all'apprendimento non supervisionato, aggiungendo un nuovo stadio per generare risposte e prevedere le risposte.

2.3.4.5 Beam search

La ricerca del risultato meglio decodificato con la massima probabilità è computazionalmente intrattabile. In altre parole, può esistere un numero esponenzialmente elevato di frasi generate in NLP o di codice sorgente in Big Code. Una soluzione potrebbe essere quella di scegliere la parola/token con la più alta probabilità di uscita dopo ogni passo temporale durante il processo di decodifica. Tuttavia, questo processo greedy probabilmente porterà a un risultato subottimale [5]. Pertanto, nella traduzione automatica, la beam search è ampiamente adottata come tecnica di ricerca euristica [52]. Invece di prendere direttamente la parola successiva con la più alta possibilità, viene conservato un elenco delle precedenti traduzioni parziali più probabili e le parole/token selezionati vengono estesi a ogni traduzione al passo corrente e riordinati. La lunghezza dell'elenco a ogni passo è nota come dimensione del fascio. Questo metodo spesso migliora la traduzione, ma le prestazioni dipendono strettamente dalla dimensione del fascio [85].

2.3.5 DL nell'ambito del Big Code

Per esaminare il Deep Learning (DL) per le applicazioni Big Code, classifichiamo i formati di input e di output dei diversi compiti in analisi del codice sorgente e generazione di programmi. Per l'analisi del codice sorgente, l'input è il codice sorgente e l'output può assumere diverse forme, come il linguaggio naturale, frammenti di codice/pattern o un intero programma. Per la generazione di programmi, tutti i compiti hanno lo stesso output nel formato

del codice sorgente, ma input diversi (ad esempio, codice e linguaggio naturale). Gli input dell'analisi del codice sorgente e della generazione di programmi nella nostra tassonomia sono sequenze, che possono essere modellate da modelli DL nell'ambito del framework "encoder-decoder". Quindi, i modelli DL adatti possono essere selezionati di conseguenza per risolvere tali compiti. La DL per il Big Code sta crescendo molto rapidamente; vengono portati avanti tanti lavori per ogni applicazione possibile. Inoltre, un confronto diretto delle prestazioni tra i vari modelli per ciascuna applicazione potrebbe non essere sempre possibile a causa dei diversi set di dati utilizzati e/o della mancanza di risultati riportati nei lavori originali [5].

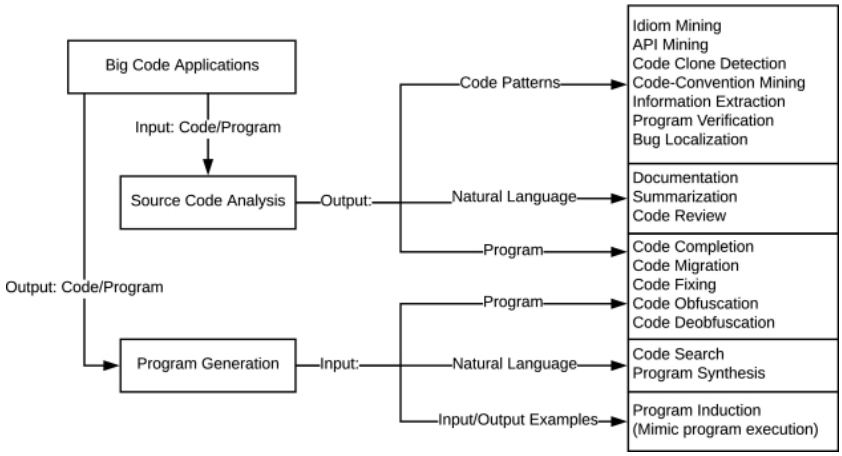

Figura 2.7: Divisione input/output dei task di Big Code. Fonte [5].

2.3.5.1 Analisi del codice sorgente

Le attività di analisi del codice sorgente prendono il codice sorgente come contesto e generano output in un altro formato. L'analisi del codice sorgente utilizza la distribuzione appresa da un'ampia raccolta di codice ed esegue vari tipi di previsioni. In primo luogo, viene presentato il caso in cui gli output sono modelli/elementi di codice.

L'**Idiom mining** estrae i segmenti di codice che ricorrono nei vari progetti. Gli idiomi di codice[12] più comuni spesso descrivono concetti di programmazione importanti e possono essere riutilizzati nei vari progetti. Un compito correlato all'estrazione di idiomi è la previsione dei nomi di classi/metodi in base al loro contesto/corpo, che può essere generalizzato alla classificazione del codice sorgente. Uno dei primi lavori di DL sull'elaborazione dei linguaggi di programmazione è stato proposto da Mou et al. [86]. Questo studio ha proposto

[12]Gli idiomi di codice sono frammenti di codice, che possono essere sintatticamente diversi, ma che ricoprono lo stesso ruolo semantico.

una CNN ad albero (TBCNN) con pooling dinamico appresa direttamente dall'AST di un programma. In questa tecnica gli AST vengono rappresentati come vettori, ai quali si applica un rilevatore di sottoinsiemi di caratteristiche, chiamato Tree-Based Convolution, che percorre l'AST per estrarre informazioni strutturali di un programma. Il pooling dinamico viene utilizzato per riunire le informazioni da parti diverse dell'albero, viene utilizzato quindi per campionare le caratteristiche in modo proporzionale alla dimensione dell'input. A questo punto le caratteristiche estratte sono completamente connesse ad uno stato nascosto e inserite nel livello di output (softmax) per la classificazione supervisionata del codice. Gli autori hanno dimostrato che i vettori di caratteristiche dei token del programma appresi con la TBCNN potevano essere raggruppati in termini di funzionalità. Tali rappresentazioni si sono dimostrate più efficaci dei metodi a n-grammi (Bag-of-words) per identificare i compiti di programmazione e rilevare i modelli di bubble-sort. In seguito, diversi studi che hanno utilizzato diverse informazioni strutturali del codice (percorsi degli AST [87] o informazioni sul flusso di dati/controllo [88]) hanno ottenuto buone prestazioni anche per questi compiti.

L'**Application Programming Interface (API) mining** e la **Code detection** testimoniano molti usi dei modelli DL. In particolare, DeepAPI [89] è stato concepito per apprendere una rappresentazione distribuita utilizzando un modello RNN seq2seq profondo sia per le query degli utenti che per le API associate. Questo strumento prende in input una query in linguaggio naturale relativa alle API e restituisce una sequenza di utilizzo di quest'ultime che la soddisfa. Per estrarre la sequenza da restituire in output dalla raccolta di codice (fornita anche di annotazioni) con cui DeepAPI è stato allenato, analizza i vari file di codice sorgente in forma AST. L'algoritmo di estrazione parte da tutte le dipendenze all'interno di un repository di progetto, analizza le classi, registrando le dichiarazioni dei campi insieme ai loro binding di tipo, ed inizia ad estrarre la sequenza di API dai singoli metodi attraversando l'AST del corpo del metodo. Permette anche di annotare in linguaggio naturale la sequenza estratta, utilizzando la prima frase della documentazione relativa. Utilizza la beam search (cfr. Sezione 2.3.4.5) per valutare la migliore sequenza da consigliare. Questo lavoro è risultato migliore dell'approccio bag-of-word per la generazione di API. Successivamente, molti lavori di DL che modellano gli AST del codice sorgente hanno ottenuto alti tassi di rilevamento dei cloni di codice (ad esempio, RtNN [90], Tree-LSTM [91], ASTNN [92]).

La **Code-convention mining** individua le pratiche di codice raccomandate da uno specifico linguaggio di programmazione ma non applicate dai compilatori. Queste convenzioni di codifica (ad esempio, indentazione, convenzioni di denominazione, spazi bianchi) aiutano a migliorare la leggibilità e la manutenibilità del codice sorgente. Un lavoro pionieristico che

utilizza l'apprendimento automatico in questa direzione è stato proposto da Allamanis et al. [93] nel 2014, utilizzando caratteristiche di n-grammi combinate con un modello di Support Vector Machine. Tuttavia, da allora non sono stati segnalati altri lavori che utilizzassero direttamente il DL per l'estrazione delle convenzioni del codice. Tuttavia, si osserva che i recenti modelli DL che catturano efficacemente i contesti d'uso del codice sorgente possono essere studiati per questo compito.

L' **Information extraction** mira a identificare l'esistenza di frammenti di codice, programmi o artefatti relativi al software dal linguaggio naturale, dalle immagini o dai video. Con i recenti progressi dei modelli DL (ad esempio, CNN) nella computer vision, stanno emergendo altri lavori [94, 95] in questa direzione per individuare elementi di codice/software da immagini e video. Parallelamente alle CNN, le RNN sono state utilizzate anche per separare frammenti di codice dal linguaggio naturale del forum Stack Overflow.

La **Program verification** predice la presenza di bug o problemi di sicurezza in un programma. La localizzazione dei bug è strettamente correlata alla verifica dei programmi, con la differenza che oltre al codice vengono identificate anche le posizioni di specifici tipi di bug. Oltre ai metodi formali [96] e agli approcci ML tradizionali, è stato dimostrato che la DL si comporta abbastanza bene per questo compito. DeepBugs ha rappresentato più di 150.000 file di codice sorgente JavaScript utilizzando word2vec [97] e ha poi addestrato una rete neurale feed-forward per distinguere tra codice buggy, cioè che presenta bug, e non buggy. Questo approccio ha raggiunto un'accuratezza superiore al 90% ed è potenzialmente in grado di funzionare in tempo reale. Un altro studio [98], che utilizza un'architettura CNN con incorporazione di parole ("word embedding") sia per le segnalazioni di bug che per i file di codice sorgente, ha superato i lavori DL esistenti per la localizzazione dei bug. Inoltre, VulDeePecker ha utilizzato un modello LSTM bidirezionale (bi-LSTM) con "code gadget" come input per identificare diversi tipi di vulnerabilità nel codice sorgente. Esiste una rassegna completa sull'analisi e la scoperta delle vulnerabilità dal codice sorgente utilizzando tecniche ML e DL. I modelli DL esaminati per questo caso hanno utilizzato una rappresentazione del codice sequenziale e/o strutturale (ad esempio, AST). La maggior parte dei modelli DL è risultata migliore di quelli non DL, mentre i modelli strutturali sembrano avere prestazioni migliori rispetto alle controparti sequenziali.

In altri scenari di analisi del codice sorgente, gli output possono essere in linguaggio naturale, il che porta ai seguenti compiti:

- La **documentazione** è un importante artefatto incorporato all'interno di un programma software per aiutare ad annotare i requisiti, il funzionamento e gli usi del codice sor-

gente e per facilitare la manutenzione del software. Ispirandosi alla Neural Machine Translation [54], Barone et al. [99] hanno utilizzato lo stesso modello per creare una base per la generazione della documentazione. Successivamente, Hu et al. [100] hanno proposto un modello LSTM con attention, DeepCom, per generare automaticamente la documentazione del codice Java. Recentemente, è stato proposto un nuovo modello, code2seq [101], con un decodificatore, che utilizza anch'esso l'attention, per selezionare percorsi compositivi ottimali di un AST per rappresentare frammenti di codice. Code2seq ha ottenuto prestazioni migliori di DeepCom per la documentazione del codice.

- La **sintetizzazione** è un'attività secondaria della documentazione, in cui viene descritta brevemente la funzionalità principale del codice sorgente o di una funzione. Nel 2015, Rush et al. [102] hanno proposto SUM-NN (cioè un modello di attenzione neurale) per la sintesi del codice e hanno ottenuto prestazioni superiori al metodo di recupero delle informazioni esistente (cioè la minimizzazione della distanza del coseno tra il codice e il riassunto corrispondente) e ai sistemi basati sulle frasi. Tuttavia, questo modello tendeva a generare descrizioni brevi. Per superare questo problema, Iyer et al. [103] hanno introdotto un modello di attenzione neurale migliorato, CODE-NN, sostituendo la rete neurale feed-forward con LSTM per il decodificatore in un modello seq2seq. Chen et al. [104] hanno presentato un modello bimodale che utilizza due "Variational AutoEncoders" (uno per il linguaggio naturale e uno per il codice sorgente) per supportare il recupero e la sintesi del codice per i linguaggi C# e SQL. Per sfruttare la conoscenza delle API per riassumere il codice sorgente, Hu et al. [100] hanno combinato la rappresentazione del modello con l'attenzione. Come per la documentazione del codice, il modello DL strutturale code2seq [101] ha recentemente superato altri lavori esistenti per il compito di riassunto del codice.

- La **revisione del codice** è una fase importante nello sviluppo del software, poiché aiuta a identificare le cattive pratiche di codifica che possono portare a difetti del software. Tuttavia, questa fase è per lo più eseguita manualmente, il che richiede molto tempo ed è soggetta a errori. Per automatizzare questo processo, DeepCodeReviewer [105] ha cercato di trovare recensioni rilevanti per i frammenti di codice/programma corrente. In particolare, sono stati addestrati quattro modelli LSTM separati con embeddings word2vec per diverse parti del codice sorgente e recensioni. Poi, i risultati sono stati combinati utilizzando una rete neurale feed-forward per determinare la pertinenza

della recensione corrente. Non c'è ancora molto lavoro di DL in quest'area, il che può stimolare gli sforzi futuri per migliorare le prestazioni dei modelli di revisione profonda del codice.

Possiamo concludere quindi che molti algoritmi di DL moderni provenienti dall'elaborazione neurale del linguaggio naturale sono stati applicati a questi compiti rigurdanti codice, il che dimostra la somiglianza tra questi due campi [5]. Alcune tecniche recenti, come le incorporazioni strutturali e i meccanismi di attenzione, sono state integrate nei modelli DL, il che consente ai modelli di apprendere dalla conoscenza massiva precedente e di essere più espressivi per catturare la struttura sottostante dei dati.

2.3.5.2 Generazione di programmi

Le attività di generazione di programmi richiedono l'inferenza sul codice del programma o sulla struttura del codice. Classifichiamo le applicazioni di generazione di programmi in tre categorie in base ai loro input: unconditional program generation (generazione di programmi incondizionata), program trasduction (trasduzione di programmi) e multimodal program generation (generazione di programmi multimodale).

Nella **generazione incondizionata di programmi**, l'input è costituito solo da un corpus di codice. L'obiettivo è generare i prossimi token più probabili o disegnare campioni simili all'input corrente.

Il **completamento/suggerimento del codice** (code completion) è un compito tipico di questa categoria. Il completamento del codice è una funzione utile e comune che molti editor di codice offrono. Sebbene la maggior parte degli strumenti di completamento del codice incorporati negli ambienti di sviluppo integrati (IDE) siano per lo più basati su "regole", il completamento del codice basato sull'apprendimento è stato un campo di studio attivo. Sono stati sviluppati algoritmi di completamento per diversi linguaggi con elevata precisione e flessibilità [106, 31, 39]. Questi algoritmi aiutano a migliorare gli IDE con un migliore suggerimento del codice, delle chiamate API o dei componenti in base al contesto corrente. I modelli che generano programmi sono un caso estremo di completamento in cui non viene dato alcun input e i token vengono campionati dal primo alla fine come una catena di Markov [49, 107, 38]. I modelli di completamento profondo del codice hanno utilizzato le pratiche recenti, tra cui (i) informazioni strutturali (ad esempio, AST [108, 109]), (ii) vocabolario aperto (ad esempio, caratteristiche a livello di carattere [110]) e (iii) meccanismo di attenzione (ad esempio, rete di puntatori [111]).

Nella **traduzione di programmi**, l'obiettivo è convertire il codice sorgente in un'altra forma di codice. Le seguenti applicazioni rientrano in questa categoria.

La **migrazione del codice** (code migration) aiuta gli sviluppatori a trasferire progetti in un linguaggio a un altro [112]. È un caso comune di aggiornamento del codice sorgente per una versione superiore di un linguaggio o di un framework. I trasduttori automatici per aggiornare le API e la struttura del codice sono molto utili per lo sviluppo e la distribuzione. Una di queste migrazioni di API da Java a C# è stata effettuata utilizzando un modello seq2seq, ovvero DeepAM [113]. Un altro modo simile, per migrare le API da Java a C#, è stato proposto da Nguyen et al. [114], conservando comunque le informazioni semantiche attraverso l'apprendimento degli embeddings word2vec delle API a coppie.

Il **Code fixing** o **Code repair** è la fase successiva alla verifica del programma e alla localizzazione dei bug, in cui i bug devono essere corretti. Nel 2016 è stato proposto un modello sk_p che utilizza il framework seq2seq basato su LSTM per correggere sette compiti Python in corsi online aperti e di massa (MOOC), ovvero MITx, e questo modello ha raggiunto una media del 29% di accuratezza per la correzione degli errori. SynFix [115] e DeepFix [116], rispettivamente con LSTM e GRU, sono stati addestrati sui compiti degli studenti per correggere gli errori di sintassi, ottenendo una percentuale di correzione completa del 30% circa. Un altro lavoro [117] ha addestrato un modello combinato di LSTM e n-gram su un corpus Java di grandi dimensioni proveniente da GitHub; è interessante notare che il modello a 10-gram supera leggermente la controparte LSTM nella correzione degli errori di sintassi. Questo dato suggerisce che un modello DL più sofisticato (ad esempio, che incorpori la struttura sintattica del codice) e meglio sintonizzato può essere utilizzato per migliorare ulteriormente il risultato [118].

L'**offuscamento del codice** (code obfuscation) impedisce a persone non autorizzate di analizzare e rubare il codice sorgente, proteggendo così la proprietà intellettuale. Esistono alcuni generatori di codice off-the-shelf offuscato, come Allatori4[13] e ProGuard5[14]. Per questo compito è stato utilizzato anche un modello linguistico statistico ML [43], ma i modelli DL non sono stati molto esplorati, se non per alcuni casi semplici come contrastare gli strumenti di reverse engineering[15] incluso il concolic testing (una tecnica ibrida di verifica del software che esegue l'esecuzione simbolica, che tratta le variabili del programma come variabili simboliche e con un risolutore di vincoli genera man mano nuovi input concreti, lungo un percorso di

[13] http://www.allatori.com/

[14] https://www.guardsquare.com/en/products/proguard

[15] Il reverse engineering, in ambito software, è un processo il cui scopo è quello di riprodurre il codice di un programma esistente.

esecuzione concreta, che prevede test su input particolari) [119]. Tuttavia, questo compito si adatta bene al framework encoder-decoder, poiché l'input è una sequenza di codici e l'output è una sequenza di testo offuscata di tale codice. Un compito diverso ma correlato, l'identificazione del codice offuscato, ha visto un maggior numero di applicazioni dei modelli DL [120, 121]. Le caratteristiche generate automaticamente da questi lavori possono fornire alcuni spunti per la progettazione di metodi (DL) efficaci per l'offuscamento del codice.

La **deoffuscamento del codice** (code deobfuscation) è il processo opposto all'offuscamento, in cui la deobfuscazione recupera la versione originale del codice sorgente da quella offuscata. Esistono strumenti di deobfuscazione per JavaScript [122, 123, 124] e per le applicazioni Android. Un approccio basato su DL [125], Context2Name, è stato proposto per recuperare i nomi naturali delle variabili nel codice minificato[16]. Per prima cosa ha utilizzato un autocodificatore sequenziale profondo per estrarre gli embedding degli identificatori dai loro contesti d'uso. Tali incorporazioni sono state poi inserite in una RNN per dedurre i nomi naturali delle variabili. È stato dimostrato che questo approccio supera lo stato dell'arte degli strumenti di deobfuscazione Javascript, come JSNice[17] e JSNaughty[18]. Ulteriori ricerche sui modelli DL per la deobfuscation del codice possono essere condotte poiché la DL per la deblurring, la demosaicizzazione e l'inpainting delle immagini è oggetto di studio attivo [126]. Per ottenere una generazione più accurata dei programmi, il completamento del codice di solito prende in input sequenze di nodi AST. Anche i meccanismi di attenzione e di copia migliorano le prestazioni dei modelli generativi di codice. Tuttavia, una trasduzione del codice di alta qualità è difficile da ottenere con il trasferimento frase per frase, poiché le differenze nelle proprietà e nella progettazione dei linguaggi di programmazione possono richiedere la modifica della struttura del codice. In questi casi possono essere utili la retro-traduzione e i modelli generativi, come nel caso del trasferimento di immagini [127].

Una categoria più impegnativa è la generazione di programmi multimodali, in cui il tipo di input non è limitato. È possibile utilizzare il linguaggio naturale (ad esempio, documentazione e commenti), le schermate della GUI [128] e il parlato. Le applicazioni correlate sono elencate di seguito.

La **ricerca di codice** (code search) restituisce i frammenti di codice sorgente esistenti meglio corrispondenti in base a query in linguaggio naturale. È stato utilizzato un modello linguistico neurale log bilineare per consentire il recupero del codice sorgente con il lin-

[16]Un codice minificato è un codice compresso ottenuto eliminando elementi inutili al compilatore o accorporando più file per ridurne le dimensioni.

[17]http://www.jsnice.org/

[18]http://jsnaughty.org/

guaggio naturale e viceversa [50]. Come menzionato prima, nella Sezione 2.3.1, il modello log-bilineare da solo non è tuttavia in grado di catturare le dipendenze del codice lungo, che è essenziale per la ricerca del codice, soprattutto per i segmenti di codice lunghi. In seguito, Gu et al. hanno proposto un modello ricorrente profondo, CODEnn (cioè unità bi-LSTM combinate con uno strato di max-pooling) [129]. Questo modello ha dimostrato di superare il precedente stato dell'arte di CodeHow [130] (un modello booleano esteso) per la ricerca di codici.

La **sintesi dei programmi** (program synthesis) estende il completamento del codice generando codice basato su molte forme di informazione, come il linguaggio naturale, le immagini e il parlato. In precedenza, le applicazioni erano limitate alla ricerca DSL [50]. Con la DL, ora è possibile affrontare la sintesi di programmi di uso generale per vari compiti. Per generare programmi sintatticamente corretti, molti studi (ad esempio, [131, 132, 133, 134]) hanno proposto di utilizzare decodificatori basati su AST anziché sequenziali. Tali decodificatori predicevano i nodi AST in modo sequenziale utilizzando RNN (ad esempio, LSTM), che potevano essere computazionalmente costose. In seguito, Sun et al. [135] hanno proposto una CNN strutturale basata sulla grammatica con meccanismi di attenzione per sostituire le RNN nel decodificatore per la generazione di codice dalla descrizione del linguaggio naturale. Il decodificatore strutturale basato su CNN genera una regola grammaticale (sequenza di token) per ogni fase, anziché token per token, rendendo il processo di decodifica più compatto ed efficiente. È stato inoltre dimostrato che questo approccio ha raggiunto lo stato dell'arte nella generazione di codice Python utilizzando il dataset di benchmark HeartStone [136]. Recentemente, Brockschmidt et al. [137] hanno esteso la rappresentazione del codice basata su grafi [88] alla generazione di codice, aumentando l'AST con nodi ereditati e sintetizzati per catturare le grammatiche degli attributi [138] durante il processo di decodifica. Il modello Graph2Graph di questo lavoro ha generato campioni di codice C# più accurati rispetto ai metodi esistenti. Con le CNN, i programmi grafici possono essere dedotti dai loro disegni in uscita. Ellis et al. [139] hanno addestrato una rete neurale gerarchica per convertire semplici disegni a mano in un DSL, che viene poi convertito in codice LaTeX con un algoritmo di ricerca bias-ottimale [140]. Beltramelli et al. [128] hanno utilizzato un'architettura encoder-decoder per generare codice front-end dalle schermate della GUI.

L'**induzione del programma** (program induction) cerca di adattare una data coppia di esempi di input/output per imitare l'esecuzione del programma, in cui la correttezza dell'esecuzione è più importante della leggibilità del codice sorgente. A causa della complessità dello spazio dei problemi, i programmi mirati sono spesso limitati ad alcune forme di DSL

o solo ad alcuni problemi semplici. Balog et al. [141] hanno utilizzato un modello seq2seq, ovvero DeepCoder, per prevedere le probabili funzioni DSL necessarie per mappare gli input e gli output dati, riducendo lo spazio di ricerca e rendendo il processo di induzione del programma 10 volte più veloce rispetto alle corrispondenti controparti basate sulla ricerca. Un'altra classe di modelli è quella degli interpreti differenziabili, in cui le grammatiche predefinite di un DSL sono continuamente parametrizzate con reti neurali. Tale parametrizzazione consente al modello di ricercare il programma per una data coppia ingresso-uscita in modo più efficiente. Evans e Grefenstette [142] hanno reso differenziabile la programmazione logica induttiva utilizzando una rete neurale per eseguire inferenze date le clausole generate, i loro pesi e la valutazione degli assioni. Il modello è addestrabile end-to-end e generalizzabile su piccoli insiemi di dati. Tuttavia, questo metodo richiede molta memoria. Riedel et al. [143] hanno proposto un interprete differenziabile per il linguaggio di programmazione Forth che utilizza esempi di input-output per creare un programma completo. Gli approcci differenziabili hanno spesso prestazioni peggiori rispetto ai metodi basati sulla ricerca per i linguaggi di programmazione di basso livello (ad esempio, Assembly) [144]. Inoltre, gli interpreti differenziabili sono ancora limitati a risolvere solo problemi semplici (ad esempio, accesso, ordinamento o copia di elementi di un array) [145]. Anche le macchine astratte neurali sono adatte all'induzione di programmi. Ad esempio, apprendendo direttamente dalle ricorsioni, Cai et al. [146] hanno esteso le capacità del Neural Programmer-Interpreter [147] e hanno fornito una prova di generalizzazione sul comportamento complessivo del sistema. Esiste anche una recente rassegna [148] sull'induzione dei programmi.

Possiamo notare come la maggior parte dei lavori recenti che rivendicano lo stato dell'arte su vari compiti di generazione di Big Code hanno utilizzato alcune varianti di RNN. Inoltre, gli embeddings estratti dai token del codice e dagli AST sono scelte comuni per l'input di questi modelli DL. Tuttavia, i modelli DL proposti sono stati studiati solo per alcune applicazioni. Mancano ancora studi approfonditi sull'ablazione[19] per testare la generalizzabilità di questi modelli a diversi compiti di Big Code.

2.4 Transformer per Big Code

Nella Sezione 2.3.4.2 è già stato accennato, in linea generale, il funzionamento dei Transformer e quanto siano stati rivoluzionari nell'ambito del NLP e, di conseguenza, per le attività

[19] L'ablazione è la rimozione di un componente da un sistema di IA per comprendere il contributo che apporta al sistema complessivo

del Big Code. Nel corso di pochi anni sono nati tantissimi modelli ispirati a questa struttura, ma ora concentriamoci su tre modelli di rappresentazione linguistica profonda a cui le varie ricerche hanno portato, ovvero Google Bert, GPT-3 di Open'AI e T5, e, in particolare, su come questi possono essere addestrati per lavorare col codice sorgente come visto nella sezione 2.3.5. Ci concentreremo su questi per poter avere un'idea generale di come funzionano e come vengono utilizzati e dei vari passi in avanti che sono stati fatti in poco tempo.

2.4.1 BERT

BERT, che sta per Bidirectional Encoder Representations from Transformers, utilizza un nuovo addestramento bidirezionale per i Transformer. Bidirezionale indica che BERT acquisisce informazioni da entrambi i lati del contesto di un token (parola) durante l'allenamento. In questo modo, J. Devlin et al. [6] hanno cercato di superare quello che per loro era un limite dei modelli linguistici standard, ovvero l'unidirezionalità, tipica di GPT come vedremo nella Sezione 2.4.2. L'unidirezionalità limitano la potenza delle rappresentazioni pre-addestrate, soprattutto per gli approcci di ottimizzazione. BERT a raggiunto lo stato dell'arte in undici compiti di NLP, come la classificazione/tagging di frasi, la risposta a domande e il riconoscimento di entità denominate. In seguito, BERT è stato esteso alla modellazione del linguaggio [149] e alla generazione del linguaggio [150], affrontando i limiti dei contesti a lunghezza fissa e della natura bidirezionale, rispettivamente. Si noti che i trasformatori possono essere sfruttati per l'incorporazione contestuale, ovvero lo stesso token ma con usi diversi, del codice sorgente.

2.4.1.1 Architettura e funzionamento

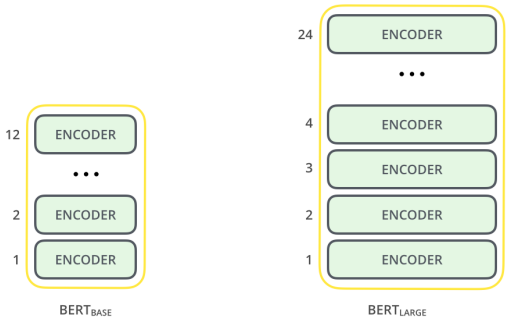

Figura 2.8: BERT - architettura modello Base e Large. Fonte [4].

Basandoci sull'architettura dei Transformer vista nella Figura 2.6, BERT presenta una struttura costituita da una pila di encoder. Entrambe le versioni di BERT hanno un grande

numero di livelli di encoder (12 per quello Base e 24 per il Large). Hanno inoltre reti feed-forward più larghe (786 e 1024 stati nascosti rispettivamente) e più moduli di attenzione (12 e 16 rispettivamente) rispetto all'architettura di default dell'implementazione dei Transformer.

BERT è in grado di "capire il linguaggio" con cui ha a che fare e questo costituisce la sua forza, che gli ha permesso di risolvere in modo adeguato diversi compiti di NLP. BERT viene prima pre-addestrato su una grande quantità di testo per capire il linguaggio e poi ottimizzato (fine-tuning) per imparare una specifica attività. Queste due fasi costituisco l'addestramento di BERT, che ora andremo a vedere più nel dettaglio:

1. Il pre-addestramento permette a BERT di "imparare il linguaggio" addestrandolo su due attività non supervisionate contemporaneamente: il Masked Language Model (MLM), che consiste nell'attività di fill-mask, vista nella Sezione 2.1.3, e la Next Sentence Prediction (NSP), dove BERT prende in input due frasi e determina se la seconda segue effettivamente la prima. NSP consiste quindi in un compito di classificazione binaria, ma che consente al modello di capire il contesto tra le diverse frasi. La Figura 2.9 ci mostra

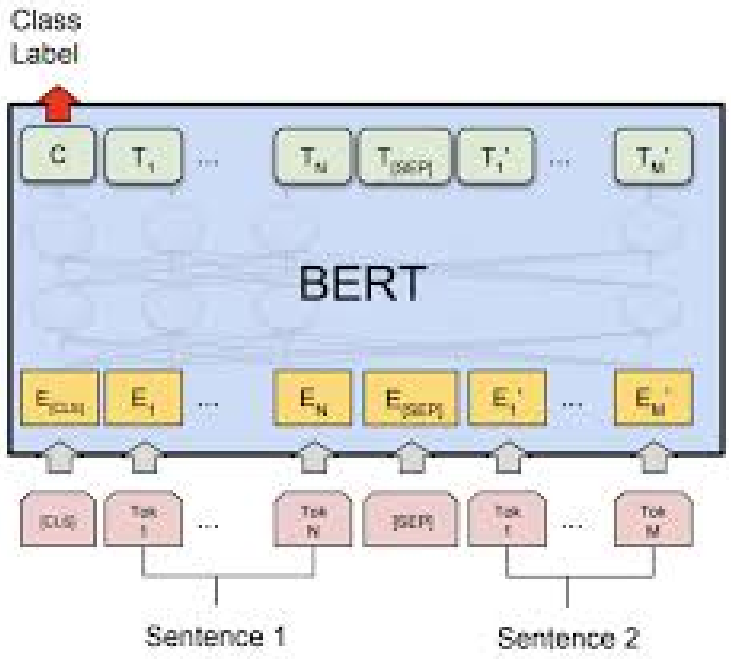

Figura 2.9: BERT - fase di pre-addestramento. Fonte [6].

come le due attività avvengano simultaneamente. I vari token Tok_i sono le parole della frase, che vengono convertiti in embeddings. GLi embeddings vengono costruiti da tre vettori: i Token embeddings, è l'embeddings ottenuto dal pre-addestramento, ottenuto nel documento originale da un vocabolario di 30k parole; i Segment embeddings, è semplicemente la frase a cui la parola appartiene; i Position embeddings, è la posizione

della parola all'interno della frase sotto forma di vettore.

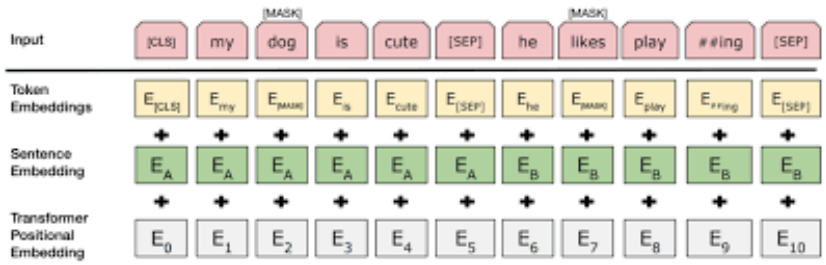

Figura 2.10: Embeddings iniziale. Fonte [6].

Nella parte riguardante l'output, la C indica l'output binario per la NSP (1 se la seconda frase segue la prima, 0 altrimenti), mentre le T_i sono vettori di parole che corrispondono all'output del MLM e sono nello stesso numero delle parole di input, sono tutti della stessa dimensione e sono generati contemporaneamente. Ogni T_i viene poi passato all'interno di una rete totalmente connessa, il cui numero di neuroni è uguale al numero di parole del vocabolario (30k), dove viene applicata una funzione softmax. In questo modo viene convertito il vettore della parola in una distribuzione per allenare BERT ad usare la funzione Cross Entropy Loss[20] e predirre il token originale che è stato mascherato.

2. Il fine-tuning permette di allenare BERT su un'attività di NLP (e di conseguenza anche per quelle relative al Big Code, viste nella Sezione 2.3.5) molto specifica. Questa fase prevede che venga aggiunto un nuovo livello di output (uno strato di classificazione) e modificati i parametri del modello in base allo specifico compito da assolvere. Così siamo in grado di eseguire un allenamento supervisionato usando un dataset etichettato relativo al compito specifico. Il tempo di allenamento è molto poco in questa fase, poichè solo i parametri del nuovo livello di output saranno imparati da zero, mentre quelli del modello saranno leggermente ottimizzati. Nella Figura 2.11, ad esempio, BERT viene ottimizzato per eseguire l'attività di QA (ctr. Sezione 2.1.3) e quindi viene addestrato modificando l'input, passando la domanda e il testo dove si trova la risposta, e l'output, restituiendo la parola iniziale e finale che incapsulano la risposta.

3. Un'alternativa al fine-tuning, è l'approccio basato sulle caratteristiche (feature-based), dove caratteristiche fisse vengono estratte dal modello pre-addestrato. Questo ap-

[20] Cross Entropy Loss è una delle più importanti funzioni di costo, utilizzata per ottimizzare i modelli di classificazione, e misura la differenza tra due distribuzioni di probabilità per una data variabile casuale o un insieme di eventi.

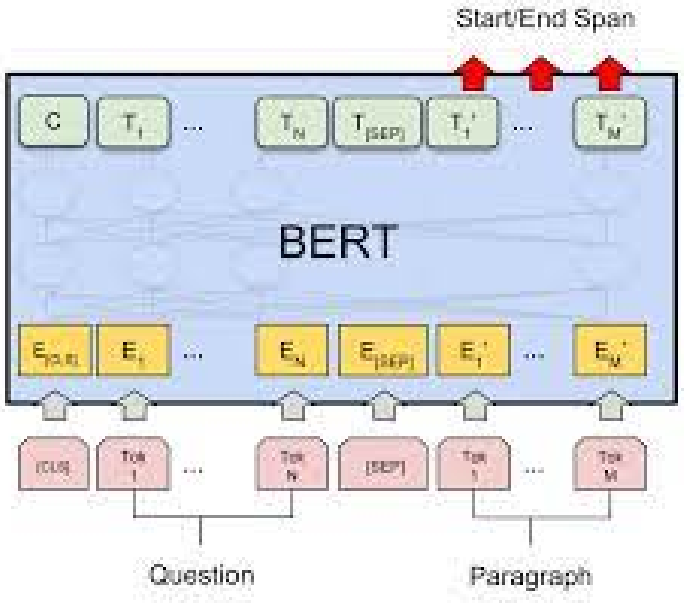

Figura 2.11: BERT - fase di fine-tuning. Fonte [6].

proccio è utile quando non tutti i compiti possono essere facilmente rappresentati da un'architettura di codifica Transformer e richiedono quindi l'aggiunta di un'architettura di modello specifica per il compito. Inoltre, il fatto di precompilare una volta una rappresentazione costosa dei dati di addestramento e di eseguire poi molti esperimenti con modelli più economici sulla base di questa rappresentazione presenta notevoli vantaggi computazionali. I test svolti [6], dimostrano che BERT è adatto ad entrambi i metodi.

2.4.1.2 Limitazioni

La maggior parte degli svantaggi del BERT può essere collegata alle sue dimensioni. Se da un lato l'addestramento dei dati su un corpus di grandi dimensioni aiuta in modo significativo il modo in cui il computer predice e impara, dall'altro c'è anche un altro aspetto. Questi includono [151]:

- Il modello è grande a causa della struttura di addestramento e del corpus.
- È lento da addestrare perché è grande e ci sono molti pesi da aggiornare.
- È costoso. Richiede più calcoli a causa delle sue dimensioni, il che ha un costo.
- È stato progettato per essere inserito in altri sistemi (non è un programma autonomo) e per questo motivo deve essere messo a punto per i compiti a valle, il che può essere difficile.

Per cercare di superare queste limitazioni sono state realizzati diversi modelli che si basano su BERT, apportando però alcune modifiche, come roBERTa [152], ALBERT [153], ecc.

2.4.1.3 Implementazioni orientate al Big Code

Molte ricerche sono state condotte su BERT e sui vari modelli da lui discendenti per poter utilizzarli in compiti riguardanti il Big Code. Andiamo a vedere alcune.

CodeBERT [154] è un modello bimodale pre-addestrato per i linguaggi di programmazione (LP) e il linguaggio naturale (LN). CodeBERT apprende le relazioni tra semantica LP e LN, così da supportare applicazioni quali la ricerca di codice in linguaggio naturale, la generazione di documentazione relativa al codice e tante altre di cui abbiamo parlato prima (ctr. Sezione 2.3.5.2). Supporta, inoltre, le trasformazioni codice-codice, codice-testo, codice-testo e testo-testo in sei linguaggi di programmazione. È sviluppato seguendo BERT e utilizzando la struttura trasformatore multistrato [72], adottato nella maggior parte dei modelli pre-addestrati di grandi dimensioni. Per sfruttare sia le istanze "bimodali" delle coppie NL-PL sia la grande quantità di codici "unimodali" disponibili, addestriamo CodeBERT con una funzione obiettivo ibrida, che include la modellazione standard del linguaggio mascherato [6] e il rilevamento di token sostituiti [155], dove i codici unimodali aiutano ad apprendere generatori migliori per produrre token alternativi migliori per il secondo obiettivo. CodeBERT è stato allenato su repository di codice di Github in 6 linguaggi di programmazione, dove i dati bimodali sono codici che si accompagnano con documentazione in linguaggio naturale a livello di funzione [156]. CodeBERT è stato valutato su due applicazioni di LP-LN, regolando con precisione i parametri del modello. I risultati mostrano che CodeBERT raggiunge lo stato dell'arte sia nella ricerca di codice in linguaggio naturale sia nella generazione di documentazione di codice. Inoltre, per indagare sul tipo di conoscenza appresa da CodeBERT, hanno costruito un dataset di banchmark apposito, CodeXGLUE [157], per compiti di NLP orientati al codice e hanno effettuato la valutazione in un'impostazione a zero-shot in cui i parametri dei modelli preaddestrati sono fissi. I risultati mostrano che CodeBERT si comporta meglio dei precedenti modelli pre-addestrati sul probing [21] NLPL. I contributi apportati da questa ricerca sono stati fondamentali, in quanto è stato il primo modello preaddestrato NL-PL di grandi dimensioni per più linguaggi di programmazione, e, inoltre, i risultati empirici dimostrano che CodeBERT è efficace sia nella ricerca di codice che

[21] Il compito di probing è utilizzare la rappresentazione prodotta da un modello di encoder (BERT) per allenare un classificatore per predirre una proprietà dell'input. In questo modo verifichiamo se la prappresentazione prodotta è corretta e utile.

nella generazione di codice-testo. Inoltre, gli studi su di esso hanno portatoa ad un set di dati che è il primo a studiare la capacità di probing dei modelli preaddestrati basati sul codice.

GraphCodeBERT [158] è un modello preaddestrato per il linguaggio di programmazione che considera la struttura intrinseca del codice. I modelli precedenti utilizzano solo il codice sorgente per il pre-training, ignorando la struttura intrinseca del codice. Tale struttura fornisce utili informazioni semantiche sul codice, che possono favorire il processo di comprensione del codice. GrapghCodeBERT, invece di prendere in considerazione la struttura del codice a livello sintattico, come l'abstract syntax tree (AST), nella fase di pre-addestramento utilizza il flusso di dati, che è una struttura a livello semantico del codice, in cui i nodi rappresentano le variabili e gli archi rappresentano la relazione "da dove viene il valore" tra le variabili. Tale struttura a livello semantico è meno complessa e non comporta una gerarchia inutilmente profonda di AST, la cui proprietà rende il modello più efficiente. GraphCodeBERT è stato sviluppato seguendo BERT come struttura, quindi si basa sui Transformer [72]. Oltre a utilizzare il compito di modellazione del linguaggio mascherato, per imparare la rappresentazione del codice dal codice sorgente e dalla struttura del codice, sono stati introdotti due compiti di pre-training consapevoli della struttura. Uno è quello di prevedere i bordi della struttura del codice e l'altro è quello di allineare le rappresentazioni tra codice sorgente e struttura del codice. Il modello è stato implementato in modo efficiente con una funzione di attenzione mascherata guidata dal grafo per incorporare la struttura del codice. Abbiamo pre-addestrato GraphCodeBERT sul dataset CodeSearchNet [156], che comprende 2,3 milioni di funzioni di sei linguaggi di programmazione abbinate a documenti in linguaggio naturale. Per valutare il modello sono stati utilizzati quattro compiti, tra cui la ricerca di codice, il rilevamento di cloni, la traduzione di codice e il perfezionamento di codice. I risultati mostrano che la struttura del codice e i nuovi compiti di pre-formazione introdotti possono migliorare GraphCodeBERT e raggiungere lo stato dell'arte nei quattro compiti a valle. Dimostriamo inoltre che il modello preferisce le attenzioni a livello di struttura rispetto a quelle a livello di token nel compito di ricerca del codice.

2.4.2 GPT-3

GPT-3, ovvero la terza generazione di Generative Pre-trained Transformer, di OpenAI è un modello di apprendimento automatico a rete neurale addestrato utilizzando grandi quantità di dati utilizzando cloud computing per generare qualsiasi tipo di testo. GPT-3 è stato realizzato dall'azienda di ricerca e sviluppo OpenAI ed è messa a disposizione degli utenti come API. GPT-3 è un modello di previsione linguistica che prende in input vettori di

parole, delle sequenze, e in base al contesto produce in output stime sulla probabilità della parola successiva. Va specificato che tale modello è autoregressivo, ovvero ogni token della frase ha solo il contesto delle parole precedenti [159].

La natura autoregressiva di GPT-3 rende possibile utilizzare una struttura di soli decoder, invece dell'intera struttura encoder-decoder solita dei Transformer illustrata nella Figura 2.6, per adottare l'apprendimento per trasferimento (transfer learning, ctr. Sezione 2.2) e un modello linguistico ottimizzabile per compiti NLP (ctr. Sezione 2.1.3). Il decodificatore è una buona scelta perché è una scelta naturale per la modellazione del linguaggio (previsione della parola successiva), in quanto è costruito per mascherare i token futuri, una caratteristica preziosa quando si tratta di generare una traduzione parola per parola.

2.4.2.1 Architettura e funzionamento

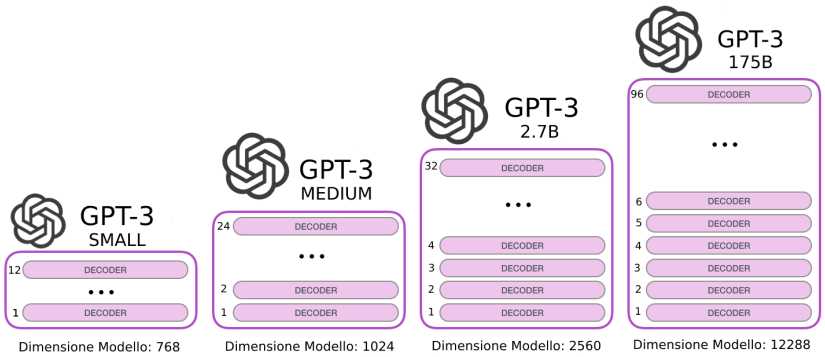

Figura 2.12: GPT-3 - architettura.

Il modello GPT-3, come possiamo vedere nella Figura 2.12, è costituito da una pila di strati di Decoder, il cui numero varia tra le varie versioni del modello. Il decodificatore è una buona scelta perché è una scelta naturale per la modellazione del linguaggio (previsione della parola successiva), in quanto è costruito per mascherare i token futuri, una caratteristica preziosa quando si tratta di generare una traduzione parola per parola.

Nella Tabella 2.1, oltre al numero dei livelli, possiamo notare anche il diverso numero di parametri della rete neurale di apprendimento profondo, che per la versione più grande di GPT-3, che è quella a cui ci riferiremo, raggiungono i 175 miliardi. Questo valore è ampiamento maggiore rispetto a tutti gli altri precedenti modelli realizzati, sia di GPT che degli altri competitor, e costituisce la maggiore novità introdotta dal modello, nonchè determina la maggior parte dei suoi vantaggi [160]. I parametri vengono ordinati all'interno di matrici all'interno del modello.

Nome modello	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B o "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Tabella 2.1: Varianti dimensioni del modello GPT-3. Fonte [9].

La fase di pre-addestramento non supervisionato di GPT-3 inizia con un modello non allenato, i cui 175B di parametri sono impostati su valori casuali. Il modello viene allenato su un dataset di 300 miliardi di token e il compito che dovrà risolvere è quello di predirre la parola successiva in una frase che gli viene fornita. I token appartenenti alla sequenza di input, vengono presi uno alla volta, così come quelli di output vengono generati uno alla volta. La predizione della parola successiva avviene moltiplicando tutti i parametri del modello per ogni token che prende in input. Ad ogni predizione, il modello andrà a calcolare l'errore tra la parola predetta e l'effettiva parola che ci aspettavamo, andando a modificare i pesi e i valori dei parametri per effettuare migliori predizioni.

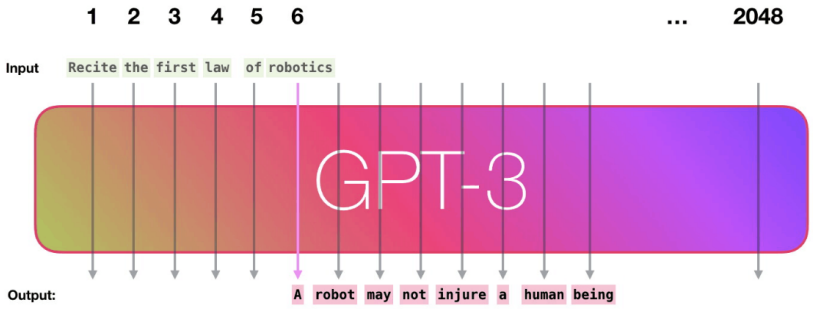

Figura 2.13: GPT-3 - Esempio predizione della prossima parola. Fonte [7].

GPT-3 ha una larghezza di 2048 token. Questa è la sua "finestra di contesto". Ciò significa che ha 2048 tracce lungo le quali vengono elaborati i token. Quindi i token di input e output devono rientrare in questa finestra, anche se si può fare in modo che il modello vado oltre questo numero di token, ma è sconsigliato e vedremo il perchè nella Sezione 2.4.2.4. Dall'esempio in Figura 2.13, potremmo domandarci "Com'è possibile che robot da come input A?". I passaggi che avvengono all'interno di GPT-3, ad alto livello, sono tre:

1. Converte la parola in un vettore rappresentante la parola (Word2Vec).

2. Elabora la predizione all'interno dei 96 livelli di decoder, ognuno dotato di 1.8B di parametri per effettuare i calcoli e che costituisce la profondità del modello.

3. Converte il vettore risultante in una parola.

Possiamo notare come ogni token fluisce attraverso l'intera pila di livelli, ma non ci interessa l'output delle prime parole. Quando l'input è terminato, iniziamo a preoccuparci dell'output. Ogni parola predetta viene poi reinserita nel modello, andando a costituire l'input per le parole successive [7].

2.4.2.2 Valutazione metodi di ottimizzazione

GPT-3 è stato valutato su oltre due dozzine di set di dati. Per ognuno di questi compiti, viene valutato per tre impostazioni: zero-shot, one-shot e few-shot. Vedremo quali sono queste impostazioni e le confronteremo con l'approccio fine-tuning [161].

Nell'approccio fine-tuning, prima si preaddestra un modello, come abbiamo visto precedentemente, che aiuta il modello a catturare gli schemi generali della lingua; poi, lo si ri-addestra separatamente per compiti specifici di NLP a valle. Lo svantaggio principale di questo approccio è la necessità di disporre di grandi set di dati per i singoli compiti. Inoltre, la messa a punto su un set di dati potrebbe non fornire una buona generalizzazione su altri per lo stesso compito. Sebbene i risultati del fine-tuning siano forti, il confronto con le prestazioni umane non è corretto.

Nell'impostazione zero-shot, dopo aver pre-addestrato il modello (con l'apprendimento in contesto), forniamo direttamente al modello l'input per il compito, senza alcun addestramento speciale per quel compito. Basta dire al modello "cosa fare" insieme all'input. Questa è l'impostazione più impegnativa e, in alcuni casi, può essere "ingiustamente difficile". Ad esempio, per input come "crea una tabella dei record mondiali dei 200 metri", il formato di output è ambiguo.

Per quanto riguarda l'approccio one-shot, forniamo: (i) "cosa fare", (ii) esattamente un esempio (one-shot) del compito e poi (iii) l'input. L'esempio ha il solo scopo di condizionare, cioè di fornire un contesto al compito. Può essere visto come una sorta di analogia con il modello. Abbiamo visto l'esempio "ingiustamente difficile" nell'impostazione a zero colpi. In compiti come questi, l'input diventa più plausibile da rispondere se viene fornita almeno una dimostrazione del compito.

Infine, nell'impostazione a few-shot, l'input comprende 1) "cosa fare", 2) alcuni esempi (pochi scatti) e poi 3) l'input. Questa impostazione consente di condizionare meglio l'input affinché il modello possa prevedere l'output. In genere, all'input vengono aggiunti K esempi, dove K è compreso tra 10 e 100. Il modello supporta una lunghezza del contesto di 2048, quindi approssimativamente, al massimo, K = 100 esempi possono essere inseriti nella finestra del contesto. L'impostazione a pochi scatti riduce notevolmente la quantità di dati necessari rispetto al fine-tuning. Tuttavia, non si può negare che sia necessaria almeno una certa quantità di dati specifici per l'attività. Lo svantaggio principale di questa impostazione è che finora i risultati ottenuti sono stati molto peggiori rispetto allo stato dell'arte. Tuttavia, GPT-3 è riuscito a ottenere risultati molto simili allo stato dell'arte su molti compiti.

2.4.2.3 Dataset per il pre-addestramento

Dataset	Quantità (token)	Peso nell'addestramento	Epoche trascorse durante l'addestramento per 300B di token
Common Crawl	410 miliardi	60%	0.44
WebText2	19 miliardi	22%	2.9
Books1	12 miliardi	8%	1.9
Books2	55 miliardi	8%	0.43
Wikipedia	3 miliardi	2%	3.4

Tabella 2.2: Dataset usati in GPT-3. Fonte [9].

Come la maggior parte dei modelli linguistici, anche GPT-3 viene addestrato sul dataset CommonCrawl [161]. Sono stati raccolti 41 frammenti di CommonCrawl mensili che coprono il periodo 2016-2019 con 45 TB di dati. Tuttavia, i dati non filtrati o leggermente filtrati di CommonCrawl tendono ad avere una qualità inferiore rispetto ai dataset filtrati. Gli autori hanno quindi adottato tre misure per filtrarli:

- Hanno preso alcuni corpora di alta qualità e, in base alla somiglianza con questi corpora, hanno filtrato CommonCrawl.
- La de-duplicazione fuzzy viene utilizzata per rimuovere le ridondanze all'interno e tra i set di dati. In questo modo si garantisce anche l'integrità, cioè il modello non si addestra sui dati di validazione.

- Infine, vengono aggiunti dati di riferimento noti e di alta qualità per aumentare la diversità del set di dati.

Il risultato è 570GB di dati, ovvero 400B byte-pair token.

2.4.2.4 Limitazioni

Nonostante i grandi vantaggi, il GPT-3 presenta alcuni limiti [161]:

- Sebbene la qualità della generazione del testo sia ottima, in alcuni casi il modello genera testo ripetitivo. Durante la generazione di documenti lunghi, può perdere la coerenza, contraddirsi e talvolta perdere completamente il contesto.

- Sappiamo che per alcuni compiti come "riempire gli spazi vuoti", i modelli bidirezionali come BERT, hanno superato i modelli autoregressivi, come appunto GPT-3.

- Poiché il modello è addestrato in modo generico, non ha alcun pregiudizio specifico per il compito. Il modello pesa tutti i token allo stesso modo. Pertanto, alcune applicazioni del mondo reale, come gli assistenti virtuali, possono trarre vantaggio dagli approcci di fine-tuning, in quanto forniscono un risultato più orientato agli obiettivi piuttosto che alle semplici previsioni.

- Un modello della portata di GPT-3 avrebbe ovviamente difficoltà nell'inferenza. È costoso e scomodo fare inferenza. Ciò solleva dubbi sull'applicabilità pratica del modello. Questo apre anche dei filoni di ricerca per lo sviluppo di versioni semplificate del modello.

- Questa è una limitazione per la maggior parte dei modelli di deep learning: le decisioni prese da un modello si basano sui dati su cui è stato addestrato. Quindi, in ultima analisi, c'è una forte possibilità che il modello abbia opinioni stereotipate e preconcette.

Inotre GPT-3 non è ancora open-source. Per questo la community di eleuther.ai ha pensato di cercare di emulare i suoi pesi e risultati. Da questo lavoro è nato GPTNeo, che implementeremo nella Sezione 3. A differenza dell'originale, quindi, GPTNeo è facilmente accessibile ed in base ai test effettuati sui compiti di generazione titoli e classificazione di sentimenti sembra migliore rispetto al modello GPT-3 più piccolo.

2.4.2.5 Implementazioni orientate al Big Code

Codex è un modello di linguaggio GPT messo a punto su codice pubblicamente disponibile su GitHub, di cui sono state studiate le capacità di scrittura di codice Python. Una versione di produzione distinta di Codex alimenta GitHub Copilot e i modelli Codex nell'API OpenAI. L'operazione di fine-tuning è stata effettuata sui modelli GPT a 12B di parametri, utilizzando 54 milioni di repository software pubblici ospitati su GitHub, contenenti 179 GB di file Python unici sotto 1 MB. Hanno filtrato i file probabilmente autogenerati, con lunghezza media delle righe superiore a 100, con lunghezza massima superiore a 1000, o che contenevano una piccola percentuale di caratteri alfanumerici. Dopo il filtraggio, il set di dati finale ammontava a 159 GB. Su HumanEval[22], un nuovo set di valutazione per misurare la correttezza funzionale della sintesi di programmi a partire da docstring, il modello risolve il 28,8% dei problemi, rispetto a GPT che ne risolve lo 0%. Il modello effettua un campionamento ripetuto, una strategia sorprendentemente efficace per produrre soluzioni funzionanti a richieste difficili. Utilizzando questo metodo, è stato risolto il 70,2% dei problemi posti con 100 campioni per problema.

Mark Chen et al. [162] effettuando una serie di prime indagini su GPT-3, hanno rivelato che era in grado di generare semplici programmi a partire da docstring Python. Pur essendo rudimentale, questa capacità era interessante perché GPT-3 non era stato addestrato esplicitamente per la generazione di codice. Dato il notevole successo dei modelli linguistici di grandi dimensioni in altre modalità e l'abbondanza di codice disponibile pubblicamente, hanno ipotizzato che un modello GPT specializzato, chiamato Codex, potesse eccellere in una varietà di compiti di codifica.

In questo lavoro, si sono concentrati sul compito di generare funzioni Python standalone da docstring e di valutare automaticamente la correttezza dei campioni di codice attraverso i test unitari. Ciò è in contrasto con la generazione in linguaggio naturale, dove i campioni sono tipicamente valutati da euristiche o da valutatori umani. Per effettuare un benchmark accurato del modello, hanno realizzato appositamente un set di 164 problemi di programmazione originali con test unitari. Questi problemi valutano la comprensione del linguaggio, gli algoritmi e la matematica semplice, e alcuni sono paragonabili a semplici domande di intervista sul software. Per risolvere un problema del set di test, vengono generati più campioni dai modelli e viene verificato se uno di essi supera i test unitari. Con un solo campione, un Codex a 12B parametri risolve il 28,8% dei problemi e un Codex a 300M

[22] Il dataset HumanEval può essere trovato al sito https://www.github.com/openai/human-eval.

parametri risolve il 13,2% dei problemi.

Per migliorare le prestazioni del modello nel compito di sintesi di funzioni a partire da docstring, è stato messo a punto Codex su funzioni autonome e correttamente implementate. Il modello risultante, Codex-S, risolve il 37,7% dei problemi con un singolo campione. In 100 campioni, Codex-S è in grado di generare almeno una funzione corretta per il 77,5% dei problemi. Questo risultato suggerisce che i campioni di codice accurati possono essere selezionati tramite una classificazione euristica invece di valutare completamente ogni campione, cosa che potrebbe non essere possibile o pratica in fase di distribuzione.

I modelli hanno ottenuto ottime prestazioni su un set di problemi scritti da esseri umani con un livello di difficoltà paragonabile a quello di un facile colloquio. Le prestazioni del modello potrebbero essere migliorate con l'addestramento su una distribuzione più simile all'insieme di valutazione e con la produzione di più campioni di un modello. Inoltre, hanno anche scoperto che è semplice addestrare un modello per completare il compito inverso e che le prestazione di questi modelli sono simili. Un'attenta analisi del modello, tuttavia, ne rivela i limiti, tra cui la difficoltà nel descrivere lunghe catene di operazioni e nel legare le operazioni alle variabili.

2.4.3 T5

L'apprendimento per trasferimento (transfer learning, ctr. Sezione 2.2) è emerso come una tecnica potente nell'elaborazione del linguaggio naturale. L'efficacia dell'apprendimento per trasferimento ha dato origine a una varietà di approcci, metodologie e pratiche. Nel loro lavoro, C. Raffel et al. [11] hanno esplorato il panorama delle tecniche di apprendimento per trasferimento in NLP introducendo una struttura unificata che converte tutti i problemi linguistici basati sul testo in un formato testo-testo. Tale struttura prende il nome di Text-to-Text Transfer Transformer o, più semplicemente, T5.

2.4.3.1 Architettura e funzionamento

Il T5 si basa anch'esso sulla struttura base dei Transformer, ma mantiene sia l'encoder che il decoder, a differenza degli altri due modelli visti precedentemente. Questa struttura è in grado di eseguire diversi compiti: traduzione automatica, compiti di classificazione, compiti di regressione (per esempio, prevedere quanto sono simili due frasi), altri compiti da sequenza di testo ad un'altra sequenza come la sintesi di documenti.

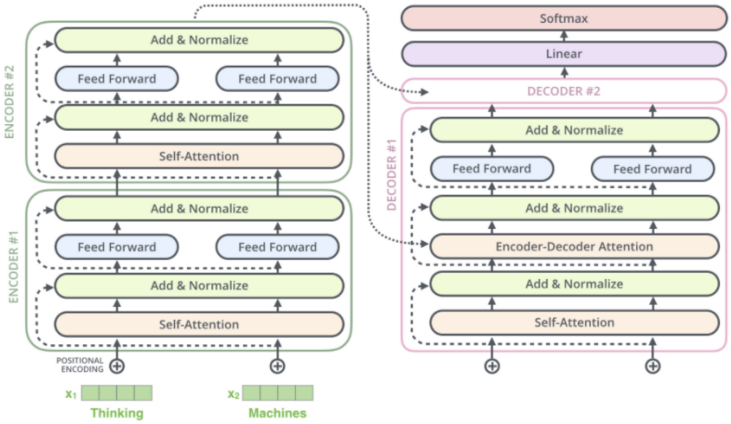

Figura 2.14: T5 - architettura. Fonte [8].

Per quanto riguarda la fase di pre-addestramento non supervisionato sono stati valutati tre obiettivi per fare imparare al modello dai dati non etichettati: predirre la parola successiva, il Masked Language Model (come in BERT) e il deshuffling (che consiste nel mescolare l'input in modo casuale e cercare di prevedere il testo originale). Dai risultati è emerso che la strategia MLM ottiene i risultati migliori, applicandogli però alcune modifiche. Infatti i sono 3 opzioni per le strategie di corruzione: mascherare solo i token e non scambiarli; mascherare i token e sostituirli con un singolo token sentinella; rimuovere i token. Le prestazioni mostrano che la strategia di "sostituzione degli intervalli corrotti" funziona meglio. Inoltre, sono stati applicati diversi tassi di corruzione e il risultato mostra che, a meno che non venga adottato un tasso di corruzione elevato (ad esempio il 50%), questa impostazione non è sensibile alle prestazioni, poiché il modello ha prestazioni simili con tassi di corruzione del 10%, 15% e 25%. Dato un buon tasso di corruzione, ora bisogna capire quanto deve essere lungo l'intervallo da eliminare. Anche questa impostazione non influenza molto le prestazioni, a meno che non venga abbandonato un ampio intervallo di token (ad esempio 10 token). Il risultato di questo esperimento sugli obiettivi è che gli obiettivi di corruzione tendono a funzionare meglio. I modelli tendono a funzionare in modo abbastanza simile con tassi di corruzione diversi e, alla luce di questo fatto, si suggerisce di utilizzare obiettivi che producono sequenze di target brevi, in quanto la sequenza più corta è più economica per il preaddestramento. Le parole vengono eliminate in modo indipendente e uniformemente casuale. Il modello viene addestrato a prevedere sostanzialmente i token sentinella per delineare il testo eliminato.

2.4.3.2 Varianti addestramento

Invece di eseguire un preallenamento e una messa a punto non supervisionati, il modello viene addestrato su più compiti. Esistono diverse strategie di combinazione. La prima opzione è quella di addestrare i compiti allo stesso modo equamente. L'opzione successiva consiste nel pesare ciascun set di dati in base al numero di esempi presenti in esso, in quanto non si vuole sovraadattare (overfit) il set di dati piccolo e sottoadattare (underfit) il set di dati grande. Poiché il compito non supervisionato è così grande, è necessario fissare un limite artificiale alla quantità di addestramento su quel compito. L'addestramento uguale su più compiti porta a prestazioni peggiori. Impostando correttamente le soglie e le temperature, si otterrebbero prestazioni simili a quelle dell'impostazione preaddestrata e ottimizzata. Soprattutto sui dataset di benchmark GLUE [163], SQUAD [164] e superGLUE [165], si registra un calo significativo delle prestazioni quando si cerca di addestrare un singolo modello su tutti questi compiti contemporaneamente. Oltre all'addestramento multi-task, esistono anche altre strategie di addestramento. Una è il preaddestramento su compiti non supervisionati e la messa a punto su ogni singolo compito a valle. Un'altra opzione è l'addestramento su una combinazione multitask e poi la messa a punto su ogni singolo compito. Questa strategia colma il divario tra l'apprendimento non supervisionato e il fine-tuning. L'addestramento multi-tasks con esclusione è l'addestramento su una combinazione multi-tasks e poi il fine-tuning su un compito che non viene utilizzato durante l'addestramento. Il risultato mostra che questa strategia genera ancora un modello preaddestrato piuttosto buono. L'ultima opzione prevede l'afferenza solo su compiti supervisionati e la messa a punto sullo stesso insieme di compiti supervisionati, in modo del tutto simile a quanto fa la computer vision. Tuttavia, questa strategia riduce in modo significativo le prestazioni.

2.4.3.3 Scalatura del modello

Allenare il modello più a lungo migliora le prestazioni e questa è una delle cose principali che RoBERTa [152] ha fatto (far lavorare il modello più a lungo). Anche l'ingrandimento del modello, rendendolo più profondo e più largo e allenandolo per due volte più a lungo, produce guadagni piuttosto significativi. In questo consiste la scalatura (scaling) del modello.

Dopo aver combinato le varie idee viste precedentemente e averle scalate, gli autori dell'articolo orginale [11] hanno addestrato 5 varianti: modello piccolo, modello di base, modello grande e modelli con 3 miliardi e 11 miliardi di parametri (rendendo i livelli feed-forward più ampi). Hanno ottenuto buone prestazioni, tranne che per la traduzione.

Strategia addestramento	GLUE	CNNDM	SQUAD	SGLUE	EnDe	EnFr
Pre-addestramento non supervisonato + fine-tuning	83.28	19.24	80.88	71.36	26.98	39.82
Addestramento multi-task	81.42	19.24	79.78	67.30	25.21	36.30
Addestramento multi-task + fine-tuning	83.11	19.12	80.26	71.03	27.08	39.80
Addestramento multi-task con esclusione	81.98	19.05	79.97	71.68	26.93	39.79
Pre-addestramento multi-task supervisonato	79.93	18.96	77.38	65.36	26.81	40.13

Tabella 2.3: Confronto varianti di addestramento T5. EnDe e EnFr indicano rispettivamento le performace nella traduzione Inglese-Tedesco e Inglese-Francese. CNNDM indica il dataset CNN/Daily Mail [10] per il riassunto del testo. Fonte [11]

Nome modello	n_{params}	n_{layers}	d_{model}	d_{ff}	d_{kv}	n_{heads}
Small	60M	6	512	2048	64	8
Base	220M	12	768	3072	64	12
Large	770M	24	1024	4096	64	16
3B	3B	24	1024	16384	128	32
11B	11B	24	1024	65536	128	128

Tabella 2.4: Varianti dimensioni del modello T5. Fonte [11].

2.4.3.4 Dataset per il pre-addestramento

T5 utilizza come dataset per il pre-addestramento il Common Crawl[23], ovvero testo estratto dal web. Gli autori applicano un filtro euristico piuttosto semplice. T5 rimuove tutte le righe che non terminano con un segno di punteggiatura terminale. Rimuove anche le righe con la parola javascript e tutte le pagine che presentano una parentesi graffa (poiché spesso compare nel codice). Deduplica il set di dati prendendo una finestra scorrevole di 3 frasi e eliminando i doppioni in modo che solo una di esse appaia nel set di dati. Il risultato è 750 gigabyte di testo inglese pulito. Il dataset è stato nominato C4 ed è disponibile pubblicamente[24]. Oltre al dataset C4 pulito, sono stati provati anche gli stessi dati senza alcun filtro. Il risultato mostra che il filtraggio aiuta il modello a ottenere prestazioni migliori.

[23] http://commoncrawl.org

[24] https://www.tensorflow.org/datasets/catalog/c4

Sono stati applicati anche altri set di dati con un ordine di grandezza inferiore (cioè su alcuni domini vincolati). I risultati mostrano che il pre-addestramento sui dati del dominio aiuta le prestazioni del compito a valle. Tuttavia se il dataset è troppo limitato e inizia ad essere ripetuto per un certo numero di volte durante l'allenamento, il modello può iniziare a memorizzare il set di dati di pertinenza e ciò causa un calo significativo delle prestazioni.

2.4.3.5 Implementazioni orientate al Big Code

I metodi di pre-training del codice esistenti, e visti precedentemente nelle Sezioni 2.4.1.3 e 2.4.2.5, presentano due limitazioni principali. In primo luogo, spesso si basano su un modello di solo codificatore simile a BERT o su un modello di solo decodificatore come GPT, che è subottimale per i compiti di generazione e comprensione. Ad esempio, CodeBERT richiede un decodificatore aggiuntivo quando viene applicato al compito di riassunto del codice, dove questo decodificatore non può beneficiare del pre-addestramento. In secondo luogo, la maggior parte dei metodi attuali adotta semplicemente le tecniche convenzionali di pre-addestramento NLP sul codice sorgente, considerandolo una sequenza di token come il linguaggio naturale (NL). Ciò ignora in larga misura le ricche informazioni strutturali del linguaggio di programmazione (PL), che sono fondamentali per comprendere appieno la semantica del codice. Per ovviare a queste limitazioni, è stato creato CodeT5 [166], un modello unificato di codificatore-decodificatore preaddestrato e consapevole degli identificatori. CodeT5 raggiunge prestazioni all'avanguardia in diversi compiti a valle relativi al codice e compiti di generazione in varie direzioni, tra cui PL-NL, NL-PL e PL-PL.

CodeT5 si basa su un'architettura simile a quella di T5, ma incorpora conoscenze specifiche sul codice per dotare il modello di una migliore comprensione del codice. Prende in input il codice e i commenti che lo accompagnano come sequenza. CodeT5 viene pre-addestrato ottimizzando alternativamente prima i seguenti obiettivi (a-c) e poi l'obiettivo (d):

- Obiettivo (a): Masked Span Prediction (MSP) maschera casualmente gli intervalli di lunghezza arbitraria e richiede al decodificatore di recuperare l'input originale. Cattura le informazioni sintattiche dell'input NL-PL e apprende robuste rappresentazioni multilingue, grazie al pre-training su più PL con un modello condiviso.

- Obiettivo (b): Identifier Tagging (IT) applicato solo all'encoder, che distingue se ogni token di codice è un identificatore (ad esempio, variabili o nomi di funzioni) o meno. Funziona come la funzione di evidenziazione della sintassi in alcuni strumenti di sviluppo.

- Obiettivo (c): Masked Identifier Prediction (MIP), a differenza di MSP, maschera solo gli identificatori e utilizza lo stesso segnaposto per tutte le occorrenze di un identificatore unico. Funziona come la deobfuscazione nell'ingegneria del software ed è un compito più impegnativo che richiede al modello di comprendere la semantica del codice sulla base del codice offuscato.

- Obiettivo (d): La generazione duale bimodale (dual-gen) ottimizza congiuntamente la conversione dal codice ai commenti e viceversa. Favorisce un migliore allineamento tra le controparti NL e PL.

CodeT5 viene pre-addestrato sul dataset CodeSearchNet [156], che consiste in sei PL con dati unimodali e bimodali. Inoltre, sono stati raccolti due set di dati di C/C# da BigQuery[25] per garantire che tutti i compiti a valle abbiano PL sovrapposti ai dati di pre-training. In totale, sono stati utilizzati circa 8,35 milioni di istanze per il preaddestramento. Per ottenere le etichette degli identificatori dal codice, si utilizza il tree-sitter[26] per convertire il PL in un AST e quindi estrarre le informazioni sul tipo di nodo. Per ogni PL viene filtrata le parole chiave riservate dall'elenco degli identificatori.

CodeT5 raggiunge lo stato dell'arte (SOTA) su quattordici compiti secondari in un benchmark di intelligenza del codice CodeXGLUE [157]. Supera in modo significativo il precedente modello SOTA di vari modelli (tra cui CodeBERT, GPT visti prima) in tutti i compiti di generazione, tra cui la sintesi del codice, la generazione da testo a codice, la traduzione da codice a codice e il perfezionamento del codice. Per quanto riguarda i compiti di comprensione, offre una migliore accuratezza nel rilevamento dei difetti e risultati comparabili nel rilevamento dei cloni. Inoltre, osserviamo che la generazione bimodale doppia favorisce soprattutto i compiti NL-PL come la sintesi del codice e la generazione da testo a codice. In Salesforce, si può utilizzare CodeT5 per costruire un assistente di codifica dotato di intelligenza artificiale per gli sviluppatori Apex. Questa assistente di codifica alimentato da CodeT5 può migliorare la produttività degli sviluppatori generando codice in base alla descrizione in linguaggio naturale, effettuando l'autocompletamento del codice e generando il riassunto di una funzione nella descrizione in linguaggio naturale.

[25]https://console.cloud.google.com/marketplace/details/github/github-repos?pli=1
[26]https://tree-sitter.github.io/tree-sitter/

CAPITOLO 3

Implementazione

In questo capitolo ci si addentrerà in una delle possibili pipeline da seguire per poter utilizzare i Transformer visti nella Sezione 2.4 per svolgere le attività che l'ambito del Big Code prevede. Il compito su cui ci soffermeremo è il CLM (Casual Language modeling), che predice il token successico ad una data sequenza i token. In questa situazione, quindi il nostro modello farà riferimento solo al contesto sinistro della sequenza, per questo andremo ad utilizzare una variante di GPT, che come abbiamo visto nella Sezione 2.4.2 è di natura auto-regressiva. Quindi dovremmo costruire un modello che prenda in input una stringa contente commenti, snippet di codice, ... e restituisca una sequenza di token per completare l'input ricevuto. Per fare ciò ci baseremo sul corso generico offerto da Hugging Face[1] sul come ottimizzare un Transformer[2], andandolo però ad adattare al nostro contesto, utilizzando il modello che maggiormente soddisfa i nostri obiettivi e risorse, un dataset contente codice, che andremo a pre-processare in modo specifico per ottimizzare il modello. Inoltre, realizzeremo il nostro modello da zero. Questo approccio è consigliato quando si devono utilizzare grandi quantità di dati, che sono diversi dai dati su cui il modello è stato pre-addestrato.

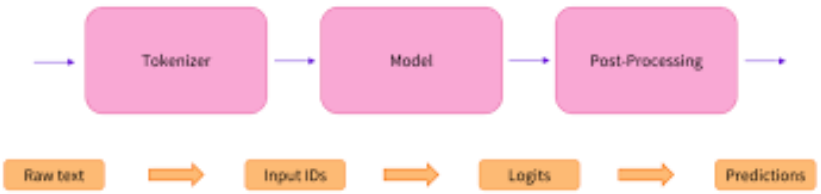

Figura 3.1: Pipeline implementazione

[1]https://huggingface.co/
[2]https://huggingface.co/course/chapter1/1

3.1 Raccolta dati

Il dataset che utilizzeremo per ottimizzare il modello pre-addestrato sul compito di code search è disponibile sul sito HuggingFace nella libreria "dataset" tra altri numerosissimi dataset orientati alle attività di NLP (ctr. Sezione 2.1.3). Il dataset è CodeSearchNet [156], che contiene codice e documentazione per 6 diversi linguaggi di programmazione [167], perchè il nostro modello dovrà essere in grado di proporre soluzioni scritte in diversi linguaggi. Ogni data point[3] quindi è costituito dal codice di una funzione, dalla relativa documentazione e da anche dei metadati riguardo la funzione.

Campo	Significato
id	Un numero arbitrario
repository_name	Il nome della repository su Github
func_path_in_repository	Percorso del file che contiene la funzione nella repository
func_name	Nome della funzione nel file
whole_func_string	Codice + documentazione della funzione
language	Linguaggio di programmazione in cui la funzione è scritta
func_code_string	Codice della funzione
func_code_tokens	Token della funzione prodotti da Treesitter
func_documentation_string	Documentazione della funzione
func_documentation_string_tokens	Token della documentazione prodotti da Treesitter
split_name	Nome della divisione alla quale l'esempio appartiene (il dataset è già diviso in training, test e validation)
func_code_url	URL al codice della funzione su Github

Tabella 3.1: Campi del dataset CodeSearchNet

Nel dataset non sono presenti funzioni senza la rispettiva documentazione. Quindi le coppie codice e documentazione vengono sottoposte a delle attività di pre-processing che prevedono:

- La documentazione è troncata al primo paragrafo per eliminare descrizioni troppo minuziose sugli argomenti e i valori restituiti dalla funzione
- Le coppie in cui la lunghezza della documentazione è minore di tre token vengono rimosse
- Le coppie in cui l'implementazione della funzione è minore di tre token vengono rimosse
- I costruttori e i metodi di estensione standard (come toString() in Java) vengono rimossi
- Le funzioni duplicate e quasi duplicate vengono rimosse, per mantenere una sola versione della funzione.

[3] I data point sono lo stato atomico dei dati. Concettualmente, possono essere considerati come una cella di una tabella di dati, o un'informazione su un'osservazione, in un determinato momento.

3.2 Moduli necessari

I moduli necessari a realizzare il modello ci vengono offerti da Hugging Face tramite le librerie Pytorch, quella che useremo, e Tensorflow. Innanzitutto sarà necessario installare il pacchetto datasets per poter scaricare il dataset su cui lavoreremo, ovvero CodeSearchNet. Per lavorare sul dataset è necessario installare dal pacchetto transformers il tokenizer GPT2Tokenizer adatto al modello che useremo, ovvero GPTNeo (ctr. Sezione 2.4.2.4), perchè nonostante le sue dimensioni ridotte ha ottenuti buoni risultati in compiti di generazione del testo. Il nostro obiettivo è realizzare da zero il nostro modello, per cui utilizzeremo la classe GPTNeoHeadModel ddel pacchetto transformers, che ci permette, appunto di dargli una configurazione personalizzata, tramite la classe AutoConfig dello stesso pacchetto. Infine, per l'addestramento del modello sarà necessario utilizzare il DataCollatorForLanguageModelling e il Trainer sempre dal pacchetto transformer.

3.3 Tokenizer

Il primo passo sarà quello di preparare i dati, in modo da poterli utilizzare per l'addestramento. Abbiamo quindi bisogno di un tokenizer, che si occupa di: normalizzare i dati con una pulizia generale, come la rimozione degli spazi bianchi inutili, delle lettere minuscole e/o degli accenti..; dividere le sequenze in token, che converte poi in input_ids; aggiungere e gestire token speciali. Possiamo addestrare un tokenizer specifico per il codice, in modo che divida bene i token del codice. Possiamo prendere un tokenizer esistente, il GPT2Tokenizer, e addestrarlo direttamente sul nostro set di dati con il metodo train_new_from_iterator().

Poiché il nostro obiettivo è quello di autocompletare principalmente chiamate di funzioni brevi, possiamo mantenere la dimensione del contesto relativamente piccola. Questo ha il vantaggio di poter addestrare il modello molto più velocemente e di richiedere una quantità di memoria significativamente inferiore. Ovviamente se avessimo più risorse a disposizione potremmo ampliare il contesto, ottenendo così prestazioni più alte in caso di autocompletare funzioni più complesse e grandi. Nel nostro caso quindi manterremo un contesto di 128 token. La maggior parte dei documenti, così come anche il codice in questo caso, contiene molti più di 128 token, quindi troncare semplicemente gli input alla lunghezza massima eliminerebbe una grande frazione del nostro set di dati. Invece, useremo l'opzione return_overflowing_tokens del Tokenizer realizzato per tokenizzare l'intero input e dividerlo in diversi pezzi. Spesso l'ultimo pezzo sarà più piccolo della dimensione del contesto, e ci sbarazzeremo di questi pezzi per evitare problemi di padding; non ne abbiamo davvero bisogno, visto che abbiamo comunque molti dati.

Ovviamente andremo ad applicare questo procedimento al campo func_code_string di tutti i record del nostro dataset, che fanno parte della divisione dedicata al training. Una volta applicato, il tokenizer restituirà un dizionario con tutti gli argomenti necessari per il corrispettivo modello per lavorare. In questo caso avremo solamente i cosiddetti input_ids: si tratta di indici di token, rappresentazioni numeriche dei token che costruiscono le sequenze che saranno usate come input dal modello.

3.4 Inizializzare il nuovo modello

Il primo passo consiste nell'inizializzare un modello GPTNeo. Per il nostro modello useremo la stessa configurazione del modello piccolo GPTNeo-125M, quindi carichiamo la configurazione preaddestrata, ci assicuriamo che la dimensione del tokenizer corrisponda alla dimensione del vocabolario del modello e passiamo gli ID dei token bos ed eos (inizio e fine sequenza). Con questa configurazione, andiamo ora a caricare un nuovo modello con GPTNeoHeadModel. Il nostro modello ha 125M di parametri che dovremo mettere a punto. Prima di iniziare l'addestramento, dobbiamo impostare un data collator che si occupi di creare i batch, usando gli elementi del dataset come input. Possiamo usare il DataCollatorForLanguageModeling scaricato, che è stato progettato specificamente per la modellazione linguistica. Oltre a impilare e riempire i batch, si occupa anche di creare le etichette del modello linguistico: nella modellazione causale del linguaggio anche gli input servono come etichette e questo collatore di dati le crea al volo durante l'addestramento, evitando di duplicare gli input_ids. Si noti che DataCollatorForLanguageModeling supporta sia la modellazione linguistica mascherata (MLM) che quella causale (CLM). Per impostazione predefinita, prepara i dati per MLM, ma è possibile passare a CLM impostando l'argomento mlm=False. A questo punto non ci resta che allenare effettivamente il nostro modello. Per far ciò è necessario inanzitutto devinire i parametri per il Trainer, in particolare useremo il cosine learning rate schedule con un warmup abbastanza alto, intorno al 1.000, visto che è molto utile per gestire bene il meccanismo dell'attenzione. Per sopperire a mancanze di memoria impostiamo il gradient accumulation ad 8, in quanto viene utilizzato quando un singolo batch non è sufficiente per la memoria e costruisce il gradiente in modo incrementale attraverso diversi passaggi avanti/indietro.

Infine, il modello inizializzato, il tokenizer, il data collator, i parametri stabiliti e i dataset di training e di valutazione vengono passati al Trainer che utilizzerà il tutto per addestrare il nostro modello, che è ora pronto per assolvere al compito per il quale l'abbiamo ottimizzato.

3.5 Utilizzo della tecnica

La tecnica realizzata potrebbe essere integrata in un qualsiasi IDE e funziona molto similmente all'AutoPilot di HitHub. Quindi qualsiasi sviluppatore mentre sta scrivendo il proprio codice potrebbe visualizzare a schermo i vari suggerimenti proposti dal modello per completare il suo lavoro, risparmiando così tempo. Il modello prenderebbe man mano come input ciò che lo sviluppatore scrive, quindi successione di token, e in base a ciò gli viene chiesto di generare i token più probabili che potrebbero continuare la sequenza presa in input. Supponiamo che il programmatore fin'ora abbia scritto il seguente codice:

```
def average_int(numbers):
"""Compute the average of numbers"""
```

In questi caso, il modello prende tale sequenza in input, la elabora e restituisce la funzione completa:

```
def average_int(numbers):
"""Compute the average of numbers"""
return sum(numbers) / len(numbers)
```

CAPITOLO 4

Valutazione preliminare

Un'operazione molto importante nella progettazione di un modello è la sua valutazione, l'unico modo con cui possiamo farci un'idea di cosa è realmente in grado di fare.
Per valutare il nostro modello, realizzato come descritto nella Sezione 3, non utilizzeremo mai lo stesso dataset utilizzato per l'addestramento, perchè valutare un modello sugli stessi dati su cui è stato effettuato l'addestramento è un grave errore che porta ad un fenomeno noto come data leakage, che si verifica quando un modello lavora bene in fase di training ma non al rilascio. Inoltre lavorare con dati che non sono stati processati, a differenza di quelli di training, consente di valutare il modello con dati che non ha mai visto e che potrebbero essergli dati effettivamente in input durante il normale utilizzo. Infine, sceglieremo come metriche quelle che ci consentono di capire se il modello realizzato ha compreso il contesto su cui è stato addestrato, il codice quindi, e se l'output prodotto è effettivamente ciò che volevamo, ovvero se il codice realizzato è concorde all'input .

4.1 Metriche

Le metriche che utilizzeremo sono due e vengono utilizzate solitamente per valutare attività di generazione del testo non supervisionata, e in particolare per i modelli autoregressivi:

- Perplexity [168]. Supponendo che il nostro dataset di testing sia composto per lo più da frasi grammaticalmente corrette, un modo per misurare la qualità del nostro modello linguistico è calcolare le probabilità che assegna alla parola successiva in tutte le frasi del dataset stesso. Le probabilità elevate indicano che il modello non è "sorpreso" o "perplesso" dagli esempi non visti e suggeriscono che ha appreso gli schemi grammaticali di base della lingua.
  Esistono varie definizioni matematiche di perplessità, ma quella che useremo noi la definisce come l'esponenziale della perdita di entropia incrociata. In questo caso verrà usata la divisione

di validazione passata al Trainer come argomento. Ovviamente più è basso il punteggio, meno il modello ha acquisito conoscenza del contesto.

- Pass @k [169]. Misura la probabilità che almeno un programma superi l'unit-test di un problema di programmazione, date k generazioni candidate del modello. Naturalmente, la metrica aumenta con k, poiché un maggior numero di candidati può potenzialmente risolvere la sfida di codifica. Per questa metrica, utilizziamo il dataset OpenAI's HumanEval, introdotto nel documento [162] e disponibile su Hugging Face. Utilizziamo questo dataset perchè è costituito da 164 problemi di programmazione originali, utilizzati per misurare la correttezza funzionale per la sintesi di programmi, ovvero contiene uno unit test legato ad ogni problematica con cui si può andare a valutare la qualità del codice proposto per risolverla. Se si ottiene un valore pari ad 1 allora la funzione è corretta. 0,5 indica che la funzione è parzialmente corretta. Infine, 0 determina che non vi è alcuna corrispondenza tra predizione ed unit test.

Sono state escluse altre metriche tipiche per le attività di NLP, perchè o fanno riferimento ad attività supervisionate o perchè non si adeguano correttamente al nostro contesto e non sono così in grado di fornire una valutazione affidabile. Ad esempio, BLEU, che è una delle più conosciute, non è in grado di catturare le caratteristiche semantiche specifiche del codice.

Entrambe le metriche possono essere scaricate da Hugging Face. La Perplexity prenderà come input il modello e una lista di snippet di codici presi dal dataset di testing, intervallati da un simbolo separatore. Per la Pass @k verranno date in input le k predizioni del modello effettuate su una problematica del dataset di riferimento e il relativo unit test che ne andrà a valutare la correttezza. Ovviamente la metrica Pass @k verrà applicata al tutto il dataset di testing, ottenendo una percentuale dei test andati a buon fine al variare del numero k di predizioni considerate.

4.2 Risultati

Valutando il nostro modello sulla base delle metriche descritte nella Sezione 4.1, otteniamo per quanto riguarda la Perplexity un punteggio di 14.62%, il che è ancora un valore abbastanza alto, ma ci fa capire che il modello ha compresso bene il dominio dei linguaggi di programmazione.

Mentre per la metrica Pass @k, variando il valore k, abbiamo ottenuto i risultati:

Modello	k = 1	k = 10	k = 100
GPTNeo-125M fine-tuned	4.34%	9.58%	16.40%
Codex-300M	13.17%	20.37%	36.27%

Tabella 4.1: Risultati metrica Pass @k sul dataset HumanEval di OpenAI

I risultati per quest'ultima valutazione possono sembrare al quanto deludenti, soprattutto se comparati, ad esempio, con una versione di Codex neanche tanto grande. Ma ciò era prevedibile. Le varie scelte prese durante l'implementazione per andare incontro alle risorse a disposizione, hanno condizionato i risultati del nostro modello. Un modo per migliorare le prestazioni del nostro

modello potrebbe essere quello di utilizzare per il tokenizer un modo più efficiente per preparare i dati, unendo tutti i campioni tokenizzati in un unica sequenza con un token eos_token_id come divisore, e poi eseguire il chunking sulle sequenze concatenate. Si potrebbe utilizzare un modello più grande rispetto al GPTNeo 125M, come, ad esempio, il GPTNeo 2.7B oppure il GPTJ 6B, che mettono a disposizione tantissimi parametri in più da ottimizzare. Inoltre, invece di addestrare il nostro modello con il Trainer, si potrebbe utilizzare un Accellerate, sempre messo a disposizione da Hugging Face, che consente di controllare pienamente il ciclo di addestramento, personalizzando la funzione di loss, dividendo l'addestramento tra più GPU e molto altro. Inoltre, si potrebbe provare variando il dataset utilizzare per l'addestramento, usandone magari specializzato su un singolo linguaggio, invece che molteplici, così che il modello possa comprenderlo meglio. Ad esempio, utilizzare per l'addestramento il dataset CodeParrot, che presenta solo codice Python estratto da GitHub per provare a migliorare la Pass@k, che viene valutata sul dataset OpenAI's HumanEval, che utilizza unit test che fanno riferimento a Python principalmente. Ci sono molte migliorie che, con le adeguate risorse, potrebbero essere introdotte per migliorare il nostro modello, ma ciò che abbiamo realizzato è già un buon inizio.

CAPITOLO 5

Conclusioni e sviluppi futuri

L'utilizzo di tecniche di Intelligenza Artificiale e Deep Learning è sempre più presente per svolgere attività di modellazione del codice sorgente. Questo consente agli sviluppatori di accellerare l'implementazione e ridurre il ricorso a risorse esterne. Inoltre consentono una documentazione più rapida e una manutenzione più semplice del software. In questo lavoro, ci si concentra in particolare sul framework encoder-decoder e del suo utilizzo nelle applicazioni di Big Code. Ci si è concentrati sull'immenso potenziale dei modelli di Deep Learning realizzati fin'ora, ma anche sulle loro limitazioni. In particolare ci si è soffermati su un modello che segue il framework endoer-deocder, ovvero i Transformer e di come il loro sviluppo potrà essere fondamentale per raggiungere lo stato dell'arte in tantissime attività di Natural Language Processing, ma, in particolare, di analisi e generazione del codice sorgente. Proprio una tipologia di Transformer, il GPTNeo-125M, è stata utilizzata per realizzare un modello di autocompletamento del codice, simile all'AutoPilot di Github. Il Transformer è stato, quindi, addestrato e ottimizzato nel compito di Casual Language Modeling sul dataset Code Search Net, contente repository di Github con codice scritto in 6 diversi linguaggi di programmazione. Il modello è stato poi valutato seguendo due metriche: Perplexity e Pass@k, mostrando una buona comprensione del contesto, ma non una grandissma correttezza del codice proposto, rispetto ad altri tools già presenti in commercio. Le motivazioni per spiegare queste prestazioni sono molteplici, ma la principale è soprattutto l'utilizzo di un Transformer dalle dimensioni limitate, dal momento che questi trovano la loro forza nei miliardi di parametri da andare ad ottimizzare, che però a causa delle risorse limitate non è stato possibile utilizzare a pieno. Realizzare questo modello, però, ha concesso di comprendere l'evoluzione che questo settore sta avendo e il grande supporto offerto da librerie e piattaforme disponibili sul web, come, ad esempio, Hugging Face. Col tempo sarà sempre più semplice realizzare modelli del genere, che potranno assistere gli sviluppatori nel loro lavoro raggiungendo un alto grado di qualità del codice o della documentazione proposta. E, anche quando raggiungeremo questo livello, non ci sarà da preoccuparsi, i modelli non ruberanno il lavoro dei programmatori, non vi sarà una competizione tra i due, ma una collaborazione che

renderà la realizzazione del codice più facile e di migliore qualità. Tuttavia, non siamo ancora a questo momento e la ricerca dovrà affrontare alcune problematiche prima di raggiungerlo, come: gli errori dei dataset, che sono costituiti da repository prese da Github, disponibili pubblicamente e che ovviamente vanno ad influenzare il modello con gli errori e i pregiudizi di chi le ha realizzate; i costi computazionali, dal momento che il consumo, in termini monetari e di CO2 prodotta, è molto alto per addestrare i Transformer; gli sviluppatori non possono fare eccessivo affidamento sugli output generati dal modello, perchè possono produrre funzioni che superficialmente sembrano corrette, ma che in realtà non corrispondono alle intenzioni dello sviluppatore, quindi richiedono ulteriori controlli di correttezza e sicurezza; a causa della natura non deterministica dei modelli di generazione, potrebbero produrre del codice vulnerabile che potrebbe danneggiare il software e persino favorire lo sviluppo di malware avanzati se deliberatamente utilizzato in modo improprio.

Ringraziamenti

Quando ho iniziato il mio percorso universitario mi è capitato diverse volte di pensare a questo giorno, ma avendo davanti a me tre anni di esami ed esperienze mi appariva molto molto lontano. Tuttavia, esame dopo esame, esperienza dopo esperienza, i tre anni sono passati in fretta e finalmente il giorno della mia laurea è giunto. Mentre scrivo queste parole ripenso a tutte le giornate di studio fino a notte tarda, all'ansia per gli esami, alle persone che ho conosciuto e quelle che già conoscevo che mi sono state accanto, mi hanno trasmesso la loro forza nei momenti più infelici e hanno gioito con me nei miei successi.

Vorrei, così, innanzitutto ringraziare il mio relatore, il Prof. Fabio Palomba, e il mio co-relatore, il Prof. Dario di Nucci, che si sono dimostrati da subito molto entusiasti dell'argomento scelto per la mia tesi, fornendomi già dal primo giorno tutto l'aiuto e i chiarimenti necessari per superare questo ultimo grande passo del mio percorso universitario. Sono sempre stati molto disponibili e gentili e spero lo siano anche in futuro nel proseguio della mia carriera universitaria.

Un grazie speciale va ai miei compagni di corso Alessandro, Gennaro, Marco, Raffaele, Sabatino, Valentina e Daniele. Sono state le prime persone che ho conosciuto all'università e da cui non mi sono più separato, che hanno reso i corsi e i progetti indimenticabili.

Il mio perscorso universitario, però, mi ha permesso di conoscere anche altre stupende e particolari persone, come Matteo, Luca, Giacomo e Gerardo. Il vostro umorismo e la vostra euforia hanno davvero contribuito a rendere indimenticabile questa esperienza. Le nottate su Discord durante la pandemia, i gruppi studio e le serate a Fisciano sono i ricordi più belli che porterò con me, grazie ragazzi. Un grazie speciale anche a Sissio, che dal lancio di crostini alle superiori ad oggi, si è dimostrato una delle persone più gentili e divertenti che conosca.

Non ho neanche ricordo del giorno in cui siamo diventati amici, per chissà quale motivio. Caro Tony ci sei sempre stato per me, dalla foto che abbiamo insieme fuori l'asilo a quelle che

faremo il giorno della mia e della tua laurea, che anche se saranno in giorni diversi va bene lo stesso. L'amicizia che ci lega è una di quelle che basta uno sguardo per capirci e iniziare a ridere, e forse proprio perchè è meglio non dire ad alta voce ciò che pensiamo. Averti accanto anche in questi tre anni non poteva che rendere il nostro legame ancora più forte e spero davvero non termini mai, grazie davvero.

Non posso dimenticare di ringraziare la mia comitiva, Antonio, Kekko, Matteo, Peppe, Vincenzo, Cuginetto e Gigi. Siete le persone più in gamba che conosco e sono felice di far parte di questo meraviglioso gruppo. Ho condiviso tantissime esperienze con voi, serate, momenti di felicità, passioni (soprattutto con chi mi sono spaccato in palestra), che mi hanno permesso di distrarmi per un po' dai miei problemi. Grazie per essere con me in questo momento speciale, non offrirò per sempre approfittatene adesso.

Un grande ringraziamento lo devo ai miei nonni, Pasquale, Anna, Felice e Anna, ai miei cuginetti, Vale, Benito, Lory e Genni, e ai miei zii, Maria e Gennaro, che fin da quando ero bambino hanno creduto in me, riempiendomi di amore, divertimento e lezioni di vita. Devo questo traguardo anche voi, spero di avervi resi fieri di me.

Molta della mia felicità e dell'amore che ho ricevuto in questi tre anni lo devo alla mia fidanzata Irene. Non avrei mai potuto immaginare di giungere a questo importante traguardo della mia vita con una persona altrettanto importante al mio fianco. Senza di te, il tuo sostegno, il tuo entusiamo e il tuo amore, questi anni non sarebbero stati completi, ma d'altronde è per questo che sei la mia metà. Grazie per avermi confortato ogni volta che venivo a lamentarmi da te, per avermi regalato un sorriso e gioia ad ogni mio successo e avermi reso l'uomo più felice del mondo, sei unica. Sii sempre la stupenda persona che sei, ti amo amore mio.

Le prime persone a cui devo dire grazie per questo traguardo sono i miei genitori, fonte di sostegno e di coraggio, che mi hanno trasmesso la passione per lo studio e la voglia di raggiungere questo traguardo più di qualsiasi altra cosa. Queste parole non sono abbastanza per ringraziavi per tutti i sacrifici e l'amore che mi avete dimostrato fin dal giorno in cui sono nato. Tutto ciò che sono lo devo a voi, alla gentilezza e alla pazienza di mamma, alla precisione e al romanticismo di papà. Siete e sarete sempre il mio esempio di vita, il più grande amore e la più grande forza che abbia mai visto. Senza di voi, non avrei avuto la possibilità di realizzare tutto ciò, quindi questo traguardo è anche vostro. Grazie, infine, per avermi fatto il regalo più grande che aveste mai potuto, il mio fratellino. Caro fratello, ti ringrazio di vero cuore per tutto ciò che fai per me, nonostante i litigi e il mio essere burbero, sei sempre stato pronto ad augurarmi buona fortuna prima di un esame e a tifare per me. Ti voglio un bene dell'anima, sarò sempre fiero di te.

Bibliografia

[1] "Immagine architettura rete neurale," https://medium.com/@snaily16/what-why-and-which-activation-functions-b2bf748c0441, online; accessed 1 Settembre 2022. (Citato alle pagine iv, 14 e 15)

[2] "Immagine architettura RNN," https://medium.datadriveninvestor.com/recurrent-neural-network-with-keras-b5b5f6fe5187, online; accessed 26 Agosto 2022. (Citato alle pagine iv e 25)

[3] H. El Khiyari and H. Wechsler, "Face recognition across time lapse using convolutional neural networks," *Journal of Information Security*, vol. 7, no. 3, pp. 141–151, 2016. (Citato alle pagine iv e 27)

[4] "Immagine architettura Transformer," https://www.analyticsvidhya.com/blog/2019/06/understanding-transformers-nlp-state-of-the-art-models/, online; accessed 28 Agosto 2022. (Citato alle pagine iv, 29 e 43)

[5] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications and challenges," *CoRR*, vol. abs/2002.05442, 2020. [Online]. Available: https://arxiv.org/abs/2002.05442 (Citato alle pagine iv, 17, 18, 20, 21, 23, 24, 25, 33, 34 e 38)

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018. (Citato alle pagine iv, 43, 44, 45, 46 e 47)

[7] "Spiegazione funzionamento GPT-3," https://jalammar.github.io/how-gpt3-works-visualizations-animations/, online; accessed 3 Settembre 2022. (Citato alle pagine iv, 50 e 51)

[8] "Immagine architettura T5," https://www.projectpro.io/article/bert-nlp-model-explained/558#mcetoc_1fr22q9rko, online; accessed 12 Settembre 2022. (Citato alle pagine iv e 56)

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. J. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *ArXiv*, vol. abs/2005.14165, 2020. (Citato alle pagine v, 50 e 52)

[10] R. Nallapati, B. Zhou, C. N. dos Santos, Çaglar Gülçehre, and B. Xiang, "Abstractive text summarization using sequence-to-sequence rnns and beyond," in *CoNLL*, 2016. (Citato alle pagine v e 58)

[11] C. Raffel, N. M. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *ArXiv*, vol. abs/1910.10683, 2020. (Citato alle pagine v, 55, 57 e 58)

[12] "Natural Language Processing," https://www.techtarget.com/searchenterpriseai/definition/natural-language-processing-NLP, online; accessed 7 Maggio 2022. (Citato alle pagine 4, 5, 6 e 11)

[13] "Natural Language Understanding," https://www.xenonstack.com/blog/difference-between-nlp-nlu-nlg, online; accessed 8 Maggio 2022. (Citato alle pagine 5 e 6)

[14] "Natural Language Generation," https://www.techtarget.com/searchenterpriseai/definition/natural-language-generation-NLG, online; accessed 9 Maggio 2022. (Citato alle pagine 7 e 23)

[15] "Fill-Mask," https://huggingface.co/tasks/fill-mask, online; accessed 10 Maggio 2022. (Citato a pagina 8)

[16] "Question Answering," https://huggingface.co/tasks/question-answering, online; accessed 10 Maggio 2022. (Citato a pagina 8)

[17] "Sentence Similarity," https://huggingface.co/tasks/sentence-similarity, online; accessed 10 Maggio 2022. (Citato a pagina 8)

[18] "Semantic Textual Similarity Benchmark," http://ixa2.si.ehu.eus/stswiki/index.php/STSbenchmark, online; accessed 10 Maggio 2022. (Citato a pagina 9)

[19] "Summarization," https://huggingface.co/tasks/summarization, online; accessed 10 Maggio 2022. (Citato a pagina 9)

[20] "Text Classsification," https://huggingface.co/tasks/text-classification, online; accessed 10 Maggio 2022. (Citato a pagina 9)

[21] "Text Generation," https://huggingface.co/tasks/text-generation, online; accessed 11 Maggio 2022. (Citato a pagina 9)

[22] "Token Classification," https://huggingface.co/tasks/token-classification, online; accessed 11 Maggio 2022. (Citato a pagina 10)

[23] "Deep Learning," https://www.techtarget.com/searchenterpriseai/definition/deep-learning-deep-neural-network, online; accessed 20 Giugno 2022. (Citato alle pagine 11 e 12)

[24] "Rete Neurale," https://www.techtarget.com/searchenterpriseai/definition/neural-network, online; accessed 20 Giugno 2022. (Citato alle pagine 14 e 16)

[25] "Gradient Descent," https://www.ibm.com/cloud/learn/gradient-descent, online; accessed 28 Agosto 2022. (Citato a pagina 15)

[26] "Fuzzy Logic," https://www.techtarget.com/searchenterpriseai/definition/fuzzy-logic, online; accessed 28 Agosto 2022. (Citato a pagina 15)

[27] "Algoritmi genetici," https://it.wikipedia.org/wiki/Algoritmo_genetico, online; accessed 28 Agosto 2022. (Citato a pagina 16)

[28] "Logica Bayesiana," https://www.techtarget.com/whatis/definition/Bayesian-logic, online; accessed 28 Agosto 2022. (Citato a pagina 16)

[29] S. Gulwani, "Dimensions in program synthesis," in *PPDP '10 Hagenberg, Austria*, January 2010. [Online]. Available: https://www.microsoft.com/en-us/research/publication/dimensions-program-synthesis/ (Citato a pagina 18)

[30] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, 2010, pp. 215–224. (Citato a pagina 18)

[31] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 27–38. (Citato alle pagine 19 e 38)

[32] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001. (Citato a pagina 20)

[33] P. Bielik, V. Raychev, and M. Vechev, "Phog: Probabilistic model for code," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2933–2942. [Online]. Available: https://proceedings.mlr.press/v48/bielik16.html (Citato a pagina 20)

[34] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 731–747. (Citato a pagina 20)

[35] A. K. Joshi and O. Rambow, "A formalism for dependency grammar based on tree adjoining grammar," 2003. (Citato a pagina 20)

[36] T. Cohn, P. Blunsom, and S. Goldwater, "Inducing tree-substitution grammars," *The Journal of Machine Learning Research*, vol. 11, pp. 3053–3096, 2010. (Citato a pagina 20)

[37] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016. (Citato alle pagine 21 e 23)

[38] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 532–542. (Citato alle pagine 21 e 38)

[39] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428. (Citato alle pagine 21 e 38)

[40] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th working conference on mining software repositories (MSR)*. IEEE, 2013, pp. 207–216. (Citato a pagina 21)

[41] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 252–261. (Citato a pagina 21)

[42] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, "Using web corpus statistics for program analysis," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 49–65. (Citato a pagina 21)

[43] H. Liu, "Towards better program obfuscation: Optimization via language models," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 680–682. (Citato alle pagine 21 e 39)

[44] P. Bielik, V. Raychev, and M. Vechev, "Program synthesis for character level language modeling," 2016. (Citato a pagina 21)

[45] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773. (Citato a pagina 21)

[46] H. Schwenk and J.-L. Gauvain, "Connectionist language modeling for large vocabulary continuous speech recognition," in *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1. IEEE, 2002, pp. I–765. (Citato a pagina 21)

[47] A. Mnih and G. Hinton, "Three new graphical models for statistical language modelling," in *Proceedings of the 24th international conference on Machine learning*, 2007, pp. 641–648. (Citato a pagina 22)

[48] A. Mnih and G. E. Hinton, "A scalable hierarchical distributed language model," *Advances in neural information processing systems*, vol. 21, 2008. (Citato a pagina 22)

[49] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*. PMLR, 2014, pp. 649–657. (Citato alle pagine 22 e 38)

[50] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *International conference on machine learning*. PMLR, 2015, pp. 2123–2132. (Citato alle pagine 22 e 41)

[51] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 38–49. (Citato a pagina 22)

[52] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014. (Citato alle pagine 23, 24 e 33)

[53] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018. (Citato a pagina 23)

[54] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014. (Citato alle pagine 24, 28, 31, 32 e 37)

[55] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *arXiv preprint arXiv:1410.3916*, 2014. (Citato alle pagine 24 e 32)

[56] "Recurrent Neural Network," https://dataaspirant.com/how-recurrent-neural-network-rnn-works/, online; accessed 1 Settembre 2022. (Citato a pagina 24)

[57] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*. PMLR, 2013, pp. 1310–1318. (Citato a pagina 25)

[58] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997. (Citato a pagina 26)

[59] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014. (Citato a pagina 26)

[60] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks," *arXiv preprint arXiv:1312.6026*, 2013. (Citato a pagina 26)

[61] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *arXiv preprint arXiv:1505.00387*, 2015. (Citato a pagina 26)

[62] J. Koutnik, K. Greff, F. Gomez, and J. Schmidhuber, "A clockwork rnn," in *International Conference on Machine Learning*. PMLR, 2014, pp. 1863–1871. (Citato a pagina 26)

[63] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Gated feedback recurrent neural networks," in *International conference on machine learning*. PMLR, 2015, pp. 2067–2075. (Citato a pagina 26)

[64] A. Mujika, F. Meier, and A. Steger, "Fast-slow recurrent neural networks," *Advances in Neural Information Processing Systems*, vol. 30, 2017. (Citato a pagina 26)

[65] "Convolutional Neural Network," https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05, online; accessed 26 Agosto 2022. (Citato a pagina 27)

[66] N. Kalchbrenner and P. Blunsom, "Recurrent continuous translation models," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1700–1709. (Citato a pagina 28)

[67] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," *arXiv preprint arXiv:1404.2188*, 2014. (Citato a pagina 28)

[68] N. Kalchbrenner, L. Espeholt, K. Simonyan, A. v. d. Oord, A. Graves, and K. Kavukcuoglu, "Neural machine translation in linear time," *arXiv preprint arXiv:1610.10099*, 2016. (Citato a pagina 28)

[69] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *International conference on machine learning*. PMLR, 2017, pp. 1243–1252. (Citato a pagina 28)

[70] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778. (Citato a pagina 28)

[71] Z. He, S. Gao, L. Xiao, D. Liu, H. He, and D. Barber, "Wider and deeper, cheaper and faster: Tensorized lstms for sequence learning," *Advances in neural information processing systems*, vol. 30, 2017. (Citato a pagina 28)

[72] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017. (Citato alle pagine 28, 29, 32, 47 e 48)

[73] "Transformer," https://www.techtarget.com/searchenterpriseai/feature/Transformer-neural-networks-are-shaking-up-AI, online; accessed 9 Maggio 2022. (Citato a pagina 30)

[74] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," *ArXiv*, vol. abs/1901.02860, 2019. (Citato a pagina 30)

[75] J. K. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, "Attention-based models for speech recognition," *Advances in neural information processing systems*, vol. 28, 2015. (Citato a pagina 32)

[76] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado *et al.*, "Google's multilingual neural machine translation system: Enabling zero-shot translation," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 339–351, 2017. (Citato a pagina 32)

[77] G. Lample, A. Conneau, L. Denoyer, and M. Ranzato, "Unsupervised machine translation using monolingual corpora only," *arXiv preprint arXiv:1711.00043*, 2017. (Citato a pagina 32)

[78] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016. (Citato a pagina 32)

[79] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014. (Citato a pagina 32)

[80] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013. (Citato a pagina 32)

[81] S. Sukhbaatar, J. Weston, R. Fergus *et al.*, "End-to-end memory networks," *Advances in neural information processing systems*, vol. 28, 2015. (Citato a pagina 33)

[82] A. Miller, A. Fisch, J. Dodge, A.-H. Karimi, A. Bordes, and J. Weston, "Key-value memory networks for directly reading documents," *arXiv preprint arXiv:1606.03126*, 2016. (Citato a pagina 33)

[83] M. Henaff, J. Weston, A. Szlam, A. Bordes, and Y. LeCun, "Tracking the world state with recurrent entity networks," *arXiv preprint arXiv:1612.03969*, 2016. (Citato a pagina 33)

[84] J. E. Weston, "Dialog-based language learning," *Advances in Neural Information Processing Systems*, vol. 29, 2016. (Citato a pagina 33)

[85] P. Koehn and R. Knowles, "Six challenges for neural machine translation," *arXiv preprint arXiv:1706.03872*, 2017. (Citato a pagina 33)

[86] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI conference on artificial intelligence*, 2016. (Citato a pagina 34)

[87] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019. (Citato a pagina 35)

[88] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017. (Citato alle pagine 35 e 41)

[89] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 631–642. (Citato a pagina 35)

[90] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98. (Citato a pagina 35)

[91] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040. (Citato a pagina 35)

[92] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794. (Citato a pagina 35)

[93] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 281–293. (Citato a pagina 36)

[94] J. Ott, A. Atchison, P. Harnack, A. Bergh, and E. Linstead, "A deep learning approach to identifying source code in images and video," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 376–386. (Citato a pagina 36)

[95] J. Ott, A. Atchison, P. Harnack, N. Best, H. Anderson, C. Firmani, and E. Linstead, "Learning lexical features of programming languages from imagery using convolutional neural networks," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 336–3363. (Citato a pagina 36)

[96] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about javascript promises," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–24, 2017. (Citato a pagina 36)

[97] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013. (Citato a pagina 36)

[98] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Information and Software Technology*, vol. 105, pp. 17–29, 2019. (Citato a pagina 36)

[99] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *arXiv preprint arXiv:1707.02275*, 2017. (Citato a pagina 37)

[100] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010. (Citato a pagina 37)

[101] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018. (Citato a pagina 37)

[102] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," *arXiv preprint arXiv:1509.00685*, 2015. (Citato a pagina 37)

[103] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083. (Citato a pagina 37)

[104] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 826–831. (Citato a pagina 37)

[105] A. Gupta and N. Sundaresan, "Intelligent code reviews using deep learning," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018. (Citato a pagina 37)

[106] P. Bielik, V. Raychev, and M. Vechev, "Phog: probabilistic model for code," in *International Conference on Machine Learning*. PMLR, 2016, pp. 2933–2942. (Citato a pagina 38)

[107] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 858–868. (Citato a pagina 38)

[108] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017. (Citato a pagina 38)

[109] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, "Neural code completion," 2016. (Citato a pagina 38)

[110] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks, 2015," http://karpathy.github.io/2015/05/21/rnn-effectiveness, 2016. (Citato a pagina 38)

[111] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel, "Learning python code suggestion with a sparse pointer network," *arXiv preprint arXiv:1611.08307*, 2016. (Citato a pagina 38)

[112] K. Aggarwal, M. Salameh, and A. Hindle, "Using machine translation for converting python 2 to python 3 code," PeerJ PrePrints, Tech. Rep., 2015. (Citato a pagina 39)

[113] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deepam: Migrate apis with multi-modal sequence to sequence learning," *arXiv preprint arXiv:1704.07734*, 2017. (Citato a pagina 39)

[114] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 438–449. (Citato a pagina 39)

[115] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *arXiv preprint arXiv:1603.06129*, 2016. (Citato a pagina 39)

[116] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI conference on artificial intelligence*, 2017. (Citato a pagina 39)

[117] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 311–322. (Citato a pagina 39)

[118] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018. (Citato a pagina 39)

[119] H. Ma, X. Ma, W. Liu, Z. Huang, D. Gao, and C. Jia, "Control flow obfuscation using neural network to fight concolic testing," in *International Conference on Security and Privacy in Communication Networks*. Springer, 2014, pp. 287–304. (Citato a pagina 40)

[120] P. Skolka, C.-A. Staicu, and M. Pradel, "Anything to hide? studying minified and obfuscated code in the web," in *The world wide web conference*, 2019, pp. 1735–1746. (Citato a pagina 40)

[121] Y. Wang, W.-d. Cai, and P.-c. Wei, "A deep learning approach for detecting malicious javascript code," *Security and Communication Networks*, vol. 9, no. 11, pp. 1520–1534, 2016. (Citato a pagina 40)

[122] S. Aebersold, K. Kryszczuk, S. Paganoni, B. Tellenbach, and T. Trowbridge, "Detecting obfuscated javascripts using machine learning," in *ICIMP 2016 the Eleventh International Conference on Internet Monitoring and Protection, Valencia, Spain, 22-26 May 2016*, vol. 1. Curran Associates, 2016, pp. 11–17. (Citato a pagina 40)

[123] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from" big code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015. (Citato a pagina 40)

[124] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 683–693. (Citato a pagina 40)

[125] R. Bavishi, M. Pradel, and K. Sen, "Context2name: A deep learning-based approach to infer natural variable names from usage contexts," *arXiv preprint arXiv:1809.05193*, 2018. (Citato a pagina 40)

[126] R. McPherson, R. Shokri, and V. Shmatikov, "Defeating image obfuscation with deep learning," *arXiv preprint arXiv:1609.00408*, 2016. (Citato a pagina 40)

[127] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232. (Citato a pagina 40)

[128] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2018, pp. 1–6. (Citato alle pagine 40 e 41)

[129] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944. (Citato a pagina 41)

[130] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 260–270. (Citato a pagina 41)

[131] L. Dong and M. Lapata, "Language to logical form with neural attention," *arXiv preprint arXiv:1601.01280*, 2016. (Citato a pagina 41)

[132] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016. (Citato a pagina 41)

[133] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," *arXiv preprint arXiv:1704.07535*, 2017. (Citato a pagina 41)

[134] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017. (Citato a pagina 41)

[135] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7055–7062. (Citato a pagina 41)

[136] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016. (Citato a pagina 41)

[137] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," *arXiv preprint arXiv:1805.08490*, 2018. (Citato a pagina 41)

[138] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968. (Citato a pagina 41)

[139] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum, "Learning to infer graphics programs from hand-drawn images," *Advances in neural information processing systems*, vol. 31, 2018. (Citato a pagina 41)

[140] J. Schmidhuber, "Optimal ordered problem solver," *Machine Learning*, vol. 54, no. 3, pp. 211–254, 2004. (Citato a pagina 41)

[141] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016. (Citato a pagina 42)

[142] R. Evans and E. Grefenstette, "Learning explanatory rules from noisy data," *Journal of Artificial Intelligence Research*, vol. 61, pp. 1–64, 2018. (Citato a pagina 42)

[143] M. Bošnjak, T. Rocktäschel, J. Naradowsky, and S. Riedel, "Programming with a differentiable forth interpreter," in *International conference on machine learning*. PMLR, 2017, pp. 547–556. (Citato a pagina 42)

[144] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, and D. Tarlow, "Terpret: A probabilistic programming language for program induction," *ArXiv*, vol. abs/1608.04428, 2016. (Citato a pagina 42)

[145] J. K. Feser, M. Brockschmidt, A. L. Gaunt, and D. Tarlow, "Differentiable functional program interpreters," *arXiv: Programming Languages*, 2016. (Citato a pagina 42)

[146] J. Cai, R. Shin, and D. X. Song, "Making neural programming architectures generalize via recursion," *ArXiv*, vol. abs/1704.06611, 2017. (Citato a pagina 42)

[147] S. E. Reed and N. de Freitas, "Neural programmer-interpreters," *CoRR*, vol. abs/1511.06279, 2016. (Citato a pagina 42)

[148] N. Kant, "Recent advances in neural program synthesis," *ArXiv*, vol. abs/1802.02353, 2018. (Citato a pagina 42)

[149] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," *arXiv preprint arXiv:1901.02860*, 2019. (Citato a pagina 43)

[150] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H.-W. Hon, "Unified language model pre-training for natural language understanding and generation," *Advances in Neural Information Processing Systems*, vol. 32, 2019. (Citato a pagina 43)

[151] "Spiegazione BERT," https://www.projectpro.io/article/bert-nlp-model-explained/558#mcetoc_1fr22q9rko, online; accessed 5 Settembre 2022. (Citato a pagina 46)

[152] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019. (Citato alle pagine 47 e 57)

[153] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019. (Citato a pagina 47)

[154] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020. (Citato a pagina 47)

[155] K. L. Clark, M. Le, Q. Manning, and C. ELECTRA, "Pre-training text encoders as discriminators rather than generators," *Preprint at https://arxiv. org/abs/2003.10555*, 2020. (Citato a pagina 47)

[156] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019. (Citato alle pagine 47, 48, 60 e 62)

[157] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code

understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021. (Citato alle pagine 47 e 60)

[158] S. L. Z. F. D. T. S. L. L. Z. N. D. A. S. S. F. M. T. S. K. D. C. C. D. D. N. S. J. Y. D. J. M. Z. Daya Guo, Shuo Ren, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020. (Citato a pagina 48)

[159] "Generative Pre-trained Transformer," https://www.techtarget.com/searchenterpriseai/definition/GPT-3, online; accessed 7 Luglio 2022. (Citato a pagina 49)

[160] "Confronto tra modelli di linguaggio," https://www.theaidream.com/post/openai-gpt-3-understanding-the-architecture#:~:text=Introduction,based%20artificial%20intelligence%20research%20laboratory., online; accessed 5 Settembre 2022. (Citato a pagina 49)

[161] "Spiegazione GPT-3," https://towardsdatascience.com/gpt-3-explained-19e5f2bd3288, online; accessed 5 Settembre 2022. (Citato alle pagine 51, 52 e 53)

[162] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. W. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, I. Babuschkin, S. A. Balaji, S. Jain, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *ArXiv*, vol. abs/2107.03374, 2021. (Citato alle pagine 54 e 66)

[163] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," in *BlackboxNLP@EMNLP*, 2018. (Citato a pagina 57)

[164] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," in *EMNLP*, 2016. (Citato a pagina 57)

[165] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Superglue: A stickier benchmark for general-purpose language understanding systems," *ArXiv*, vol. abs/1905.00537, 2019. (Citato a pagina 57)

[166] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *ArXiv*, vol. abs/2109.00859, 2021. (Citato a pagina 59)

[167] "Scheda di CodeSearchNet su HuggingFace," https://huggingface.co/datasets/code_search_net, online; accessed 21 Settembre 2022. (Citato a pagina 62)

[168] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, "Perplexity—a measure of the difficulty of speech recognition tasks," *The Journal of the Acoustical Society of America*, vol. 62, no. S1, pp. S63–S63, 1977. (Citato a pagina 65)

[169] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. (Citato a pagina 66)