Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Multimodal Side-Tuning for Code Snippets Programming Language Recognition

Relatore:
Prof.
Maurizio Gabbrielli

Presentata da:
Salvatore Visaggi

Correlatore:
PhD.
Francesca Del Bonifro

Sessione III
Anno Accademico 2019/2020

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Identificare in modo automatico il linguaggio di programmazione di una porzione di codice sorgente è uno dei temi che ancora oggi presenta diverse difficoltà. Il numero di linguaggi di programmazione, la quantità di codice pubblicato e reso *open source*, e il numero di sviluppatori che producono e pubblicano nuovo codice sorgente è in continuo aumento[1] [2]. Le motivazioni che richiedono la necessità di disporre di strumenti in grado di riconoscere il tipo di linguaggio per *snippet* di codice sorgente sono svariate. Ad esempio, tali strumenti trovano applicazione in ambiti quali: la ricerca di codice sorgente; la ricerca di possibili vulnerabilità nel codice; la *syntax highlighting*; o semplicemente per comprendere il contenuto di progetti *software* e il *trend* di adozione dei linguaggi di programmazione da parte degli sviluppatori [1].

Per avere una stima del numero di linguaggi di programmazione usati dagli sviluppatori, basti considerare *GitHub*, la più grande piattaforma di hosting di progetti software, che ospita codice sorgente scritto in più di 500 linguaggi di programmazione[3].

In letteratura è possibile trovare approcci che considerano il *task* di identificazione del linguaggio per interi *file* di codice sorgente, a volte avvalendosi dell'estensione dei *file*, in quanto offre senz'alcun dubbio un'informazione decisiva per l'identificazione del contenuto. Altri approcci si concentrano invece sull'identificazione del linguaggio per porzioni di codice molto piccole, dell'ordine di poche linee di codice.

[1] https://insights.stackoverflow.com/survey/2020

[2] https://octoverse.github.com/

[3] https://github.com/github/linguist/blob/master/lib/linguist/languages.yml

Ed è in questo scenario che entrano in gioco piattaforme e forum di condivisione di codice tra sviluppatori come *StackOverflow* (SO). Il principio alla base di SO è la condivisione della conoscenza tra gli sviluppatori attraverso un sistema di *Q&A*. Di conseguenza, su SO è possibile trovare centinaia di migliaia di *snippet* di codice sorgente scritti nei linguaggi più usati dagli sviluppatori, rendendolo il luogo ideale da cui estrarre *snippet* per la risoluzione del *task* proposto.

A partire da tale premessa, è possibile trovare in letteratura lavori che tentanto di classificare il linguaggio di programmazione di *snippet* di codice estratti da SO. Tuttavia, l'estrazione di *snippet* etichettati in modo adeguato presenta diverse sfide: i *dataset* presenti in letteratura coprono un numero di linguaggi esiguo e presentano una quantità non trascurabile di *snippet* etichettati in modo non adeguato.

Nasce così l'esigenza di estrarre un *dataset* allineato di *snippet-linguaggio* avvalendosi delle potenzialità offerte da SO. Nel lavoro di tesi svolto si è dedicata molta attenzione a tale problematica, iterando sull'approccio scelto al fine di ottenere una metodologia che ha permesso l'estrazione di un *dataset* adeguato. Si sono così ottenuti due corpus di *snippet* che comprendono rispettivamente 61 e 69 linguaggi di programmazione.

Successivamente, considerando quanto realizzato in letteratura, è possibile fare riferimento a tecniche di classificazione che si basano sull'estrazione di *features* testuali, o sul riconoscimento di immagini.

Nel lavoro di tesi svolto si fa uso di un approccio *multimodale* (considerando rappresentazioni testuali e di immagini degli *snippet*), prendendo in esame la tecnica innovativa di *side-tuning* (basata sull'adattamento incrementale di una rete neurale pre-addestrata), al fine di risolvere il *task* dell'identificazione del linguaggio per gli *snippet* estratti da SO.

I risultati ottenuti sono confrontabili con lo stato dell'arte e in alcuni casi migliori, in considerazione della difficoltà del *task* affrontato nel caso di *snippet* di codice sorgente che presentano poche linee di codice. Nel dettaglio, l'approccio proposto raggiunge un'accuratezza superiore al 91% sui corpus

di *snippet* estratti da SO.

Nel prossimo capitolo si descriveranno gli approcci presenti in letteratura, stilando una rassegna sintetica ed enfatizzando le peculiarità di ogni approccio esaminato. Nel capitolo 2 si descriveranno i dettagli della metodologia adottata per l'estrazione degli *snippet* e la costruzione del corpus, descrivendo le sfide affrontate e le scelte effettuate. Nel capitolo 3 si porrà maggiore attenzione ai dettagli implementativi dell'approccio usato, descrivendo la tecnica innovativa di *side-tuning multimodale*. Infine, nel capitolo 4 si descriverà la metodologia di addestramento del modello e i risultati ottenuti, cercando di esporre un confronto con le architetture di base proposte.

Capitolo 1

Stato dell'arte

Il linguaggio di programmazione della maggior parte del codice sorgente prodotto e pubblicato viene specificato manualmente dagli sviluppatori, solitamente attraverso la definizione dell'estensione del *file*. Esistono strumenti *open-source*, come ad esempio *Linguist*[1] di *GitHub*, che permettono di identificare il linguaggio di programmazione considerando principalmente l'estensione dei *file*. In particolare, *Linguist* considera inizialmente l'estensione dei *file*, rendendolo di fatto veloce e adatto nella maggior parte delle situazioni; nei casi in cui l'estensione può rappresentare codice scritto in diversi linguaggi (ad esempio *file* con estensione *.h* possono contenere codice scritto nei linguaggi *C*, *C++* o *Objective-C*), viene usato un classificatore *Bayesiano* per risolvere eventuali ambiguità. L'accuratezza di *Linguist* raggiunge l'84% su oltre 300 linguaggi di programmazione [14]. Nei casi in cui non siano presenti le estensioni dei *file*, le prestazioni di *Linguist* calano notevolmente, rendendolo di fatto inadeguato per *snippet* di codice sorgente [14].

Data l'importanza ricoperta dal *task* della classificazione del linguaggio di programmazione di codice sorgente, in letteratura vi sono molteplici contributi in cui gli autori cercano di identificare il linguaggio escludendo le informazioni fornite dalle estensioni dei *file*, avendo così a disposizione una moltitudine di approcci e metodologie da cui prendere spunto per costruire

[1] https://github.com/github/linguist

metodi sempre più prestanti. Al fine di comprendere lo stato dell'arte per il *task* che si vuole affrontare, sono stati esaminati alcuni dei lavori più rilevanti, cercando di mettere in evidenza le caratteristiche principali di ognuno.

Il primo aspetto da considerare per la catalogazione degli approcci presenti in letteratura, consiste nel distinguere tra tecniche che si basano sull'estrazione di *features* testuali da codice sorgente e tecniche basate sulla classificazione di immagini. Come si vedrà in seguito, la maggior parte degli approcci esaminati si basa sull'estrazione di *features* testuali, mentre resta tutt'oggi poco esplorato l'uso di tecniche di *computer vision* per l'identificazione del linguaggio di programmazione.

Il secondo aspetto da considerare per ogni approccio esaminato, consiste nell'effettuare una distinzione che tenga conto della natura dei dati di *training* presi in esame dagli autori. I risultati ottenuti con approcci che considerano dati estratti da piattaforme quali *GitHub*, devono essere spiegati in modo differente rispetto ad approcci che considerano dati estratti da piattaforme quali *StackOverflow* (SO). Ciò è dovuto alla natura del codice sorgente estraibile dalle due piattaforme. Nel dettaglio, *GitHub* è una piattaforma di *hosting* di progetti *open-source* e il codice sorgente ospitato da tale piattaforma consta di *file* etichettati, in linea teorica, in modo adeguato. SO è invece una piattaforma di *questions-and-answers* per sviluppatori e il codice sorgente estraibile consta di *snippet* di poche linee non sempre etichettati con il linguaggio corretto.

Ciò giustifica ulteriormente la necessità di disporre di strumenti accurati per l'identificazione del linguaggio di programmazione per *snippet* di codice sorgente, in quanto ciò permetterebbe, ad esempio, una corretta catalogazione del codice pubblicato su piattaforme quali *SO*. L'automazione di tale *task* permetterebbe inoltre di effettuare una corretta *syntax highlighting* del codice; faciliterebbe il tracciamento dei *trend* di adozione dei linguaggi da parte degli sviluppatori; permetterebbe una migliore scansione del codice per la ricerca di eventuali vulnerabilità; infine, garantirebbe un miglioramento dei processi di produzione del software, facilitando lo sviluppo di nuove estensioni

per gli ambienti di sviluppo integrati (*IDE*) del codice.

Si procederà dunque ad illustrare in modo sintetico le tecniche presenti in letteratura suddividendole in approcci basati sull'estrazione di *features* testuali e approcci basati sulla classificazione di immagini. Si cercherà di illustrare le caratteristiche dei metodi proposti, ponendo enfasi sui dati usati dagli autori e descrivendone i risultati ottenuti.

1.1 Approcci basati su Features Testuali

Tra gli strumenti che si occupano di classificare il codice sorgente attraverso l'estrazione di *features* testuali occorre menzionare *GuessLang*[2]. Si tratta di uno strumento *open-source* che, secondo quanto dichiarato dagli autori, supporta 30 linguaggi di programmazione ed è in grado di raggiungere un'accuratezza del 93.82%. *GuessLang* usa una rete neurale profonda che opera al livello dei caratteri. Tale rete è stata addestrata su un corpus di codice sorgente composto da circa 1 milione di *file* estratti da circa 100 mila *repository open-source* presenti su *GitHub*.

In [2] gli autori propongono un classificatore *Multinomial Naive Bayes* (*MNB*) per la classificazione del codice sorgente per *snippet* estratti da SO. Vengono estratti i *post* contenenti *snippet* per 21 diversi linguaggi di programmazione. La costruzione del corpus etichettato avviene prendendo in esame i *tag* associati ai post da cui si estraggono gli *snippet*, e il codice sorgente viene pre-processato ottenendo una rappresentazione vettoriale di tipo *bag-of-word*. Il modello proposto permette di raggiungere un'accuratezza del 75% sui dati di test.

Nella versione successiva dell'approccio appena descritto [3], gli autori propongono di combinare le informazioni testuali e gli *snippet* di codice sorgente contenuti nei post estratti da SO per migliorare la classificazione del linguaggio di programmazione. Considerando 21 linguaggi, viene addestrato un classificatore *XGBoost* e un classificatore *Random Forests* (RF). Nel caso

[2]https://guesslang.readthedocs.io

in cui le informazioni testuali sono combinate con le *features* estratte dagli *snippet*, si è raggiunta un'accuratezza dell'88, 9% con *XGBoost*, e 87, 7% con *RF*. Considerando invece la classificazione solo per gli *snippet* di codice contenuti nei post, gli autori raggiungono un'accuratezza del 77, 4% usando *XGBoost*, e del 78, 1% usando *RF*.

Un approccio simile era stato proposto precedentemente in [7], in cui i *post* vengono estratti da SO considerando i *tag* associati a 18 diversi linguaggi di programmazione. L'approccio proposto si basa sull'addestramento di un classificatore basato sulle informazioni testuali contenute nei post e negli *snippet* di codice. Nel dettaglio, viene appreso un classificatore *Support Vector Machine* (SVM), raggiungendo un'accuratezza del 44, 6%, nel caso in cui si considerano solo le informazioni testuali degli *snippet*, e del 60, 9% considerando le *features* estratte dal testo in linguaggio naturale.

Un'ulteriore tecnica esplorata in letteratura consiste nel derivare automaticamente un'unica grammatica dal corpus dei *file* di codice sorgente, in modo da poter essere usata per effettuare successivamente il *parsing* di ogni *file* nel *training* e *test set* [45]. L'ipotesi sostenuta dagli autori è quella secondo cui la costruzione di una grammatica comune si basi sulla condivisione da parte della maggioranza dei linguaggi di programmazione di una struttura sintattica comune, in particolare, se si considera la costruzione di blocchi quali parole chiavi, identificatori, costanti o dichiarazioni. Viene così definita una grammatica lessicale usando la libreria *ANTLR*[3]. Il *dataset* utilizzato dagli autori è stato estratto da *GitHub* considerando 29 linguaggi e circa 10 mila *file* per linguaggio. Usando un classificatore *Maximum Entropy*, e un classificatore *Bayesiano*, gli autori hanno raggiunto un'accuratezza del 99% e 96% rispettivamente.

Tra gli altri approcci che fanno uso di classificatori *Bayesiani* vi è quanto proposto in [23]. Il *dataset* utilizzato è stato estratto considerando oltre 100 *repository open-source* su *GitHub* e consta di 20 mila *file* scritti in 10 diversi linguaggi di programmazione. L'approccio proposto si basa sull'estrazione

[3]https://www.antlr.org/

di 1056 parole chiave per tutti i linguaggi considerati. Viene poi appreso un classificatore *MNB* che raggiunge un'accuratezza del $93,48\%$ sui dati di test.

Oltre alle tecniche di *machine learning* come MNB, SVM e RF, in letteratura è possibile trovare anche approcci che fanno uso di *Reti Neurali Artificiali* (ANN) e *Convolutional Neural Network* (CNN).

Ad esempio, in [10] gli autori propongono l'uso di una semplice *ANN* come classificatore, raggiungendo un'accuratezza superiore all'85% su 133 linguaggi di programmazione. Il *dataset* usato è stato estratto da *GitHub*. Il codice sorgente è stato pre-processato effettuando l'estrazione di *token* e, prendendo spunto dalle applicazioni di elaborazione del linguaggio naturale, sono stati costruiti *bi-grams* e *tri-grams* di *token*.

Anche in [15] si fa ricorso a tecniche di elaborazione del linguaggio naturale per la classificazione del codice sorgente. Il codice viene processato al fine di ottenere una rappresentazione vettoriale numerica dei *token*, costruendo in altri termini dei *word embeddings*. In seguito viene appresa una *CNN* che ha come input una matrice di *word embeddings* ottenuta dai *file* di codice sorgente. I dati usati sono stati estratti da *GitHub* e comprendono circa 1 milione di *file* di codice sorgente. Il corpus finale comprende 60 linguaggi di programmazione. L'accuratezza raggiunta dal metodo proposto è del 97% circa.

Reyes J. et al [35] affrontano il *task* della classificazione del linguaggio usando reti neurali ricorrenti basate su *Long Short-Term Memory* (*LSTM*) con *word embeddings*. I corpus di *file* di codice sorgente sono stati estratti da *GitHub* e *Rosetta Code*[4]. Considerando 10 linguaggi di programmazione, vengono estratte le *features* da codice in modo da considerare l'ordine in cui i *token* appaiono. La rete addestrata raggiunge un'accuratezza superiore al 99%.

Ciò che si evince dai metodi illustrati in modo sintetico è la mancanza di un corpus comune qualitativamente adatto ad affrontare il *task* in esame,

[4]http://www.rosettacode.org

oltre ad una chiara tendenza verso l'uso di tecniche di *machine learning* sempre più sofisticate che permettono di ottenere prestazioni di volta in volta migliori. Le soluzioni esaminate trattano l'estrazione di *features* dagli *snippet* in modo differente, ma la maggior parte si basa su tecniche ampiamente usate per l'elaborazione del linguaggio naturale. I classificatori addestrati spaziano da SVM, MNB, XGBoost e RF, fino a ANN, CNN ee LSTM-RNN. Inoltre, come accennato, la maggior parte degli studi proposti si concentra sulla classificazione del linguaggio per *file* di codice sorgente estratti da *GitHub*, ottenendo in alcuni casi ottime prestazioni, in particolare quando il numero di linguaggi non è eccessivamente ampio.

Nella sezione seguente si descriveranno brevemente i lavori presenti in letteratura che fanno uso di tecniche di *computer vision* o che semplicemente considerano il codice sorgente sotto forma di *rendering* del testo in immagini.

1.2 Approcci basati su Immagini

Nella sezione precedente, tra i lavori esaminati, alcuni adottavano delle CNN per la classificazione del linguaggio trattando il codice sorgente come input testuale. In letteratura sono presenti anche tentativi di classificazione del linguaggio usando CNN che operano su immagini ottenute dal *rendering* del testo degli *snippet*.

In [26] è possibile esaminare un interessante confronto tra un approccio basato su testo e uno basato su immagini. Lo studio proposto è il primo nel suo genere e cerca di esplorare le prestazioni delle CNN considerando un *dataset* di immagini e testo. Gli esperimenti sono stati effettuati considerando tre *dataset* differenti estratti da GitHub e contenenti 8 linguaggi di programmazione. L'idea alla base di tale confronto è quella di verificare l'impatto che ha la rappresentazione del codice sorgente (immagine o testo) per la classificazione del linguaggio di programmazione. Il codice sorgente in forma testuale viene sottoposto ad una fase di analisi lessicale e costruzione di *word embedding* per ottenere le *features* testuali, e viene renderizzato per

generare immagini di dimensione 224×224. Vengono proposte due architetture CNN per la risoluzione del *task*, raggiungendo il $99,38\%$ nel caso di input di immagini e il $98,81\%$ per input testuali.

Tra i lavori esaminati, in [32] viene esplorata la possibilità di usare CNN con architetture più profonde, prendendo in esame reti come la *VGG*[37]. Tali architetture rappresentano lo stato dell'arte nella classificazione e riconoscimento delle immagini, e sono in grado di ottenere ottimi risultati in competizioni quali la *ImageNet Large Scale Visual Recognition Challenge* (*ILSVRC*)[5]. Dunque, il suo utilizzo consiste nell'apprendere *feaures* lessicali di diversi linguaggi di programmazione classificando *snippet* di codice sorgente presenti in video tutorial online. Gli autori hanno estratto ed etichettato manualmente circa 17 mila video *frames* che contengono esempi di codice scritti in *Java* e *Python* oppure informazioni testuali e non testuali. I risultati ottenuti evidenziano la validità dell'idea proposta, raggiungendo un'accuratezza del $98,73\%$. Indubbiamente il numero di linguaggi considerato è estremamente esiguo, ma ciononostante il lavoro svolto pone le basi per l'esplorazione di ulteriori architetture.

In [20], gli autori prendono in esame le architetture *ResNet* [17] e *AlexNet* [27] per la classificazione di *snippet* di codice estratti da SO e *GitHub*. L'idea è dunque quella di usare una CNN per la classificazione delle immagini come fatto nei lavori visti precedentemente. Un primo *dataset* consiste in *snippet* di codice estratti da SO e comprende 10 linguaggi di programmazione. L'estrazione è avvenuta considerando solo *snippet* che presentano una dichiarazione esplicita del linguaggio associato. Il secondo *dataset* consiste in *snippet* di codice contenenti funzioni scritte in 5 diversi linguaggi, estratti da *GitHub*. L'estrazione di questi ultimi avviene effettuando il *parsing* del codice e costruendo l'albero sintattico astratto (AST). Le immagini sono state renderizzate con dimensione 384×384 e considerando al più 48 caratteri per linea. Sia l'architettura *AlexNet* che *ResNet* raggiungono un'accuratezza del 92% circa sul *dataset* di *snippet* di codice estratti da SO. Mentre, sul *dataset*

[5]http://www.image-net.org/challenges/LSVRC/

di funzioni, le due architetture raggiungono un'accuratezza del 99%. Inoltre, gli autori sottolineano come nel *dataset* di *snippet* estratto da SO vi sia del disturbo che sicuramente influenza negativamente le prestazioni dei modelli appresi.

Da quanto esaminato emerge chiaramente che il numero di linguaggi di programmazione considerati è esiguo. Tuttavia, i risultati motivano la necessità di esplorare l'utilizzo di tali architetture in contesti differenti e per la classificazione di un numero maggiore di linguaggi. Altra necessità emersa riguarda la costruzione di un *dataset* di *snippet* di codice etichettati in modo corretto. Ad esempio, alcuni approcci si concentrano sul filtraggio del codice mettendo in pratica analisi lessicale e *parsing*. Ciò è impraticabile in modo efficace per porzioni di codice dell'ordine di poche linee, e il numero di linguaggi trattato richiederebbe l'accesso ad analizzatori lessicali e compilatori per ognuno dei linguaggi in esame. Dall'altra parte, etichettare manualmente una mole di dati considerevole non risulta fattibile, a meno di prevedere, ad esempio, un progetto di lungo termine e che sia supportato da *community* di sviluppatori online, come ad esempio SO. L'obiettivo del lavoro di tesi svolto, come si vedrà nei capitoli successivi, è quello di estrarre un *dataset* di *snippet* di codice sorgente da SO che comprenda un ampio numero di linguaggio ed applicare un'architettura che combini rappresentazioni testuali e di immagini per la classificazione degli *snippet*.

Tabella 1.1: *Confronto dei metodi presenti in letteratura per l'identificazione del linguaggio di programmazione da codice sorgente. Vengono mostrate le differenze relative al metodo usato e alle features considerate.*

Pubblicazione	Anno	Metodo	Features
Khasnabish et al. [23]	2014	MNB, NB	Estrazione delle parole chiave e conversione degli *snippet* in vettori di indici
Reyes et al. [35]	2016	LSTM - RNN	Costruzione di *word embeddings* dei token
Baquero et al. [7]	2017	SVM	Costruzione di *word embeddings* per l'estrazione di features testuali e *n-grams* per l'estrazione di *features* da codice
Zevin e Holzem [45]	2017	Maximum Entropy, NB	Costruzione di *n-grams* di sequenze di token lessicali attraverso la definizione di una grammatica comune
Gilda [15]	2017	CNN	Estrazione di token e costruzione di matrice di *word embeddings* per ogni *snippet*
Ott et al. [32]	2018	CNN (VGG)	*Features* sintattiche e contestuali
Alrashedy et al. [2]	2018	MNB	Vettori numerici costruiti da *snippet* di codice sorgente
Alrashedy et al. [3]	2019	RF, XGBoost	Informazioni testuali e *features* estratte da *snippet* di codice
Hong et al. [20]	2019	CNN (AlexNet, ResNet)	*Rendering* di immagini da *snippet* e di funzioni estratte con costruzione di *AST*
Kiyak et al. [26]	2020	CNN	Costruzione di *word embeddings* ed estrazione di *features* attraverso *layer* convoluzionali
Del Bonifro et al. [10]	2021	ANN	*Tokenization* e costruzione di *bi-grams* e *tri-grams* di sequenze di token lessicali
GitHub Linguist[a]	-	NB	Estensione del *file* e disambiguazione con contenuto testuale
GuessLang[b]	2020	DNN	Costruzione di *n-grams* di caratteri

[a]https://github.com/github/linguist
[b]https://guesslang.readthedocs.io

Tabella 1.2: *Confronto dei metodi presenti in letteratura per l'identificazione del linguaggio di programmazione da codice sorgente. Vengono mostrate le differenze relative alla fonte dei dati usati, il numero di linguaggi considerati e i valori di accuratezza raggiunti.*

Pubblicazione	**Anno**	**Linguaggi**	**Tipo Dati**	**Origine Dati**	**Accuracy (%)**
Khasnabish et al. [23]	2014	10	Testo	GitHub	93.48
Reyes et al. [35]	2016	10	Testo	Rosetta Code e GitHub	80.22 - 99.85
Baquero et al. [7]	2017	18	Testo	SO	60.88 testo 44.61 codice
Zevin e Holzem [45]	2017	29	Testo	GitHub	99 - 96.5
Gilda [15]	2017	60	Testo	GitHub	97
Ott et al. [32]	2018	2	Immagini	Youtube	85.60 - 98.73
Alrashedy et al. [2]	2018	21	Testo	SO	75
Alrashedy et al. [3]	2019	21	Testo	SO	88.90
Hong et al. [20]	2019	10, 5	Immagini	SO e GitHub	92 - 99
Kiyak et al. [26]	2020	8	Testo e Immagini	GitHub	98.81 testo 99.38 immagini
Del Bonifro et al. [10]	2021	133	Testo	GitHub	85
GitHub Linguist[a]	-	$\sim$ 300	Testo	GitHub	84
GuessLang[b]	2020	30	Testo	GitHub	93.82

[a] https://github.com/github/linguist
[b] https://guesslang.readthedocs.io

Capitolo 2

Estrazione del Dataset

Determinare il linguaggio di programmazione di codice sorgente è un *task* esplorato ampiamente in letteratura. Nel capitolo precedente sono stati illustrati in sintesi alcuni degli approcci usati, evidenziandone peculiarità e prestazioni. Emerge chiaramente una tendenza che vede nell'adozione delle tecniche di *Machine Learning* e di elaborazione del linguaggio naturale un approccio efficace per la risoluzione del *task* in esame. La maggior parte dei lavori presenti in letteratura si concentra sull'identificazione del linguaggio di programmazione per interi file di codice sorgente, mentre sono ancora pochi i tentativi di classificazione del linguaggio per porzioni di codice di dimensioni ridotte. Inoltre, in letteratura, i lavori che affrontano lo stesso problema prendono in esame un numero esiguo di linguaggi. Ciò rende i *dataset* di *snippet* presenti in letteratura (quando accessibili pubblicamente) non adatti all'obiettivo che si vuole raggiungere. Nel lavoro di tesi svolto si vuole affrontare il problema della classificazione del linguaggio di programmazione per *snippet* con poche linee di codice sorgente, considerando allo stesso tempo un ampio numero di linguaggi.

La grande quantità di codice sorgente disponibile, reso possibile anche grazie alla proliferazione di *software open source*, alla crescita di piattaforme come *GitHub*, o di forum e piattaforme di condivisione di domande e rispo-

ste tra sviluppatori, come *StackOverflow* (SO)[1], rende possibile l'accesso a centinaia di migliaia di potenziali *snippet* di codice sorgente.

In particolare, SO ha una *community* di circa 14 milioni di utenti, e presenta più di 21 milioni di domande e più di 31 milioni di risposte, con un tasso del 70% di domande con risposta[2]. Considerando SO, sono stati dunque estratti due nuovi corpus di *snippet* di codice sorgente. Si può affermare che gli *snippet* di codice estratti da SO hanno dimensioni molto minori (in termini di linee di codice) rispetto al codice sorgente estraibile da *GitHub*. Inoltre, come si vedrà in seguito, uno dei problemi maggiori affrontati è stata l'etichettatura con il linguaggio corretto degli *snippet* estratti. Tale problema non sussiste per il codice estratto da *GitHub*, dove si può fare affidamento sull'estensione del file, o su strumenti quali *Linguist*[3] per la risoluzione di eventuali ambiguità.

A tal fine si procederà analizzando SO, fornendo informazioni chiave sui meccanismi che regolano la piattaforma e utili a comprendere l'approccio usato per l'estrazione dei *dataset*. Si eseguirà poi una rapida analisi dei *dataset* di *snippet* di codice sorgente estratti da SO nei lavori esaminati in letteratura e disponibili pubblicamente. Come si vedrà, la maggior parte di tali *dataset* sono stati estratti per risolvere *task* diversi dalla classificazione del linguaggio (ad esempio, per affrontare il *task* della *code search* attraverso l'uso di *query* in linguaggio naturale).

Si proporrà infine la metodologia adottata nel lavoro di tesi svolto per l'estrazione di un corpus di *snippet* allineati con il relativo linguaggio di programmazione, dettagliando le euristiche adottate.

[1] https://stackoverflow.com/

[2] https://stackexchange.com/sites

[3] https://github.com/github/linguist

2.1 Analisi di StackOverflow

StackOverflow è un sito di domande e risposte per programmatori professionisti ed appassionati. Nato nel 2008, l'idea alla base di SO è la possibilità per qualsiasi utente di sottoporre alla *community* un quesito in merito ad un determinato argomento relativo alla programmazione. Ogni utente della *community* può dunque contribuire aggiungendo commenti e/o risposte ad una determinata domanda, proponendo, il più delle volte, anche uno o più esempi di codice sorgente per la risoluzione del quesito. Ed è da questo semplice meccanismo che SO conta oggi milioni di potenziali *snippet* di codice sorgente, aggiunti nel corso degli anni dagli utenti.

Altro aspetto interessante di SO è l'uso di meccanismi di *gamification* per incentivare gli utenti ad avere un ruolo attivo nella *community*: ogni risposta data può essere votata dagli utenti (attraverso un sistema di voti *up* e *down*), facendo guadagnare punti all'autore della risposta[4]. Una risposta con molti voti *up* sale in cima, fino a piazzarsi idealmente come prima risposta del post. Ciò incentiva gli utenti ad accumulare punti, in modo da acquisire *badges* che ne incrementino la reputazione e permettano di acquisire maggiori privilegi.

Altro aspetto interessante è dato dalla licenza d'uso dei post e delle risposte pubblicate su SO: ogni contenuto creato dagli utenti, in puro stile *open-source*, è rilasciato con licenza *Creative Common Attribute-Share Alike* 2.5, 3.0 o 4.0[5], rendendo di fatto qualsiasi contenuto presente su SO usabile ed adattabile in qualsiasi modo, anche per fini commerciali[6].

Per avere un quadro più dettagliato del ruolo che gioca SO tra gli sviluppatori di tutto il mondo, è possibile fare affidamento ai *survey* annuali che la piattaforma realizza[7], i quali forniscono informazioni demografiche sugli utenti della piattaforma, oltre che informazioni interessanti sulle tecnologie e i linguaggi usati dalla *community*. In particolare, si evince come su un

[4]https://stackoverflow.com/tour
[5]https://stackoverflow.com/help/licensing
[6]https://creativecommons.org/licenses/by-sa/4.0
[7]https://insights.stackoverflow.com/survey/2020

campione di 65000 sviluppatori, il 67.7% programma in *JavaScript*, a seguire il 63.1% scrive codice in *HTML/CSS*, il 54.7% adotta *SQL*, il 44.1% *Python*, il 40.2% *Java*, e così via, fino ad arrivare, ad esempio, ad avere solo lo 0.9% di programmatori appartenenti al campione considerato che scrivono codice in *Julia*.

A partire da tali informazioni, è possibile formulare una semplice domanda: quanti linguaggi di programmazione è possibile trovare su SO? In altri termini, si vuole comprendere per quali linguaggi di programmazione è possibile estrarre *snippet* di codice sorgente da SO.

Prima di rispondere a tale domanda, occorre entrare nel dettaglio, descrivendo la struttura di SO, la quale si basa su un meccanismo di *post* e risposte. Gli elementi che costituiscono un *post* sono il titolo (che rappresenta il quesito posto dall'autore); un contesto, per chiarire l'*intent* del titolo, scritto in linguaggio naturale; ed eventualmente uno o più *snippet* di codice sorgente. Un utente può commentare un *post*, oppure pubblicare una risposta. La risposta ad un *post* è a sua volta un ulteriore *post*, senza titolo, che contiene un contesto in linguaggio naturale (la spiegazione della risposta), ed eventualmente uno o più *snippet* di codice sorgente.

Ogni *post* (che sia dunque la domanda o una delle risposte) ha uno *score*, che può essere positivo o negativo, determinato dagli altri utenti nella *community*. Ad esempio, una risposta inadeguata, può ricevere diversi voti *down* e trovarsi in fondo alla lista delle risposte; viceversa, ricevendo voti *up*, una risposta può salire in cima. Anche una domanda può ricevere *up votes* o *down votes*, a seconda del contributo che aggiunge nella *community*.

Infine, l'autore di una domanda, può contrassegnare un risposta come accettata, specificando dunque la risposta che soddisfa la domanda sottoposta, indipendentemente dai voti assegnati a tale risposta dagli altri utenti nella *community*.

Un ulteriore aspetto da tenere in considerazione per i *post*, riguarda l'uso dei *tag* per determinare il linguaggio di programmazione, il *framework*/libreria, o più in generale l'argomento del post. Ad ogni post possono essere assegnati

uno o più *tag*, ed è partire dai *tag* che nella fase successiva di estrazione degli *snippet* si cercherà di comprendere il linguaggio di programmazione in cui sono scritti.

Per poter estrarre gli *snippet*, tutto il contenuto di SO è disponibile e viene periodicamente aggiornato sulla piattaforma *Google BigQuery*[8]. Si tratta di una copia di *SOTorrent* [5], il progetto che si occupa di tenere traccia dei contenuti e dell'evoluzione dei post su SO. Ciò rende semplice l'accesso alle informazioni appena elencate, attraverso semplici *query* in linguaggio *SQL*.

Tornando al quesito iniziale, SO contiene più di 60 mila *tag* (dato aggiornato al 2 Dicembre 2020). Tra tutti i *tag* presenti, occorre determinare quali si riferiscono ad un linguaggio di programmazione, così da comprendere in quali post cercare potenziali *snippet* di codice sorgente scritti in un determinato linguaggio.

Considerando la lista di linguaggi di programmazione usata in *Linguist* (che comprende più di 500 tra linguaggi e tipologie di formato dei file)[9], è stato effettuato un semplice confronto con la lista dei *tag* usati su SO (aggiungendo opportunamente possibili nomenclature che possono differire per alcuni linguaggi noti), ottenendo inizialmente una lista di 367 *tag* che identificano potenziali linguaggi di programmazione. Successivamente, considerando solo *tag* che compaiono in almeno 1000 post, il numero di linguaggi usabili è sceso a 153. Tuttavia, analizzando i linguaggi etichettati come tali da *Linguist*, ci si rende conto che molti non sono linguaggi ma piuttosto nomi di file, di librerie o *frameworks* (ad esempio *numpy* o *nginx*) o comunque non linguaggi ma piuttosto estensioni di file (ad esempio *csv*).

Operata tale fase di filtraggio, è possibile rispondere al quesito affermando che su SO è possibile estrarre *snippet* usabili per 78 potenziali linguaggi. Analizzando la distribuzione dei post per linguaggio, come mostrato in Fig. 2.1 è possibile osservare che per *JavaScript*, ad esempio, vi sono più di 2 milioni di post con score maggiore o uguale di 0. A seguire è vi sono *Java* e

[8]https://cloud.google.com/bigquery

[9]https://github.com/github/linguist/blob/master/lib/linguist/languages.yml

Python con rispettivamente più di 1.7 e 1.5 milioni di post, fino ad arrivare ad *Objective-C++* per cui su SO vi sono poco più di 1000 post. Come si può intuire, il problema principale è dovuto alla presenza di post per cui non vi sono risposte, per cui il numero di post per *tag* fornisce una stima teorica del numero di *snippet* estraibili.

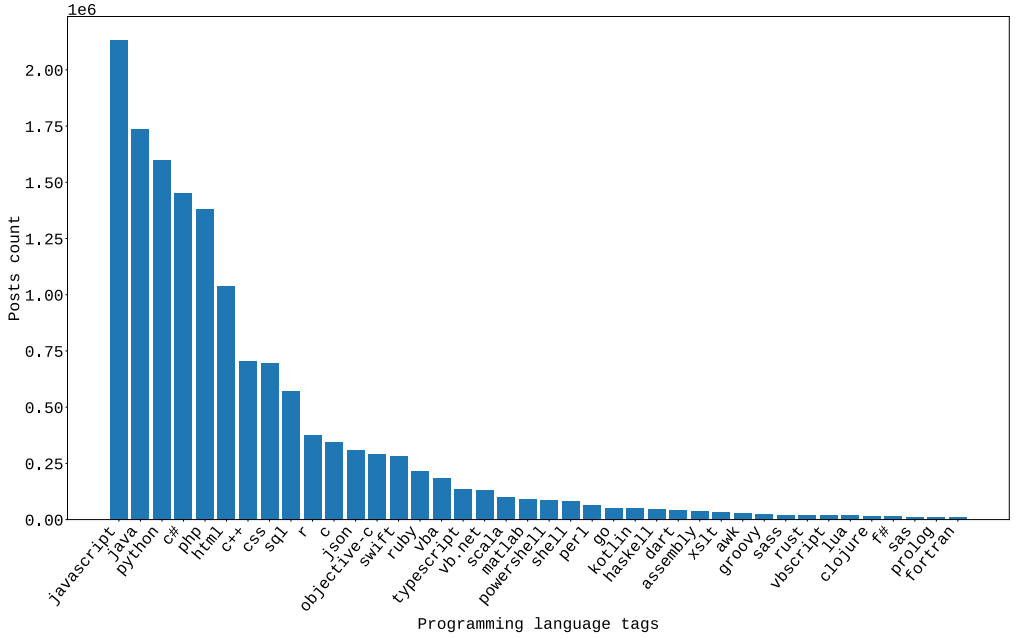


Figura 2.1: *Distribuzione del numero di post per linguaggio su SO. Vengono mostrati solo i primi* 40 *linguaggi.*

Compresi quali e quanti linguaggi possono essere considerati, si vuole identificare il metodo più efficace per estrarre un corpus allineato correttamente di *snippet*-linguaggio. Dunque, si vuole comprendere quali proprietà concorrono a determinare la qualità di uno *snippet* estratto da un post o da una risposta.

Prendendo in esame l'analisi degli *snippet* proposta in [41], su un totale di 3 milioni di *snippet* di codice per 4 linguaggi (*C#, Java, Javascript e Python*), gli autori hanno provato a comprendere quanti di questi *snippet* sono usabili. Il loro obiettivo è stato comprendere quanti *snippet*, per i linguaggi in esame, superano la fase di *parsing* e successivamente la fase di compilazione.

Essi arrivano alla conclusione che *Python* e *JavaScript* hanno una percentuale di *snippet* usabili (che dunque sono anche compilabili) pari al 65.66% e 20% rispettivamente. Se invece si considerano solo gli *snippet* che superano la fase di *parsing*, tale percentuale sale al 76.22% e al 25.61% rispettivamente. Discorso diverso invece per *Java* e *C#*, per cui solo il 3.89% e il 16% rispettivamente superano la fase di *parsing*, e solo l'1% e lo 0.12% rispettivamente superano la fase di compilazione. Inevitabilmente, usare un approccio di estrazione basato sulla verifica dell'usabilità degli *snippet* è impraticabile.

Altra considerazione fondamentale, sottolineata anche dagli autori dello studio citato [41], riguarda la possibile presenza di post che il più delle volte possono presentare *snippet* scritti in *pseudocodice*. Ciò risulta utile per comprendere l'*intent* della domanda, ma sicuramente rappresenta un problema per il corpus che si vuole estrarre.

Per l'estrazione degli *snippet*, si potrebbe allora assumere che nelle risposte migliori vi siano gli *snippet* migliori. Si vuole, in altri termini, rispondere alla domanda: come identificare le migliori risposte? Una semplice euristica consiste nel considerare il punteggio attribuito alla risposta, oltre che nel dare più importanza alle risposte contrassegnate come accettate dall'autore della domanda a cui fanno riferimento [40], [33]. L'approccio usato segue inevitabilmente queste semplici euristiche.

Fissati i linguaggi di programmazione identificati dai *tag* dei post, si cerca di estrarre gli *snippet* dalle risposte considerandone il punteggio e l'eventuale presenza del *flag* di risposta accettata. Prima di procedere alla descrizione della metodologia usata, si vuole proporre in modo sintetico una rassegna dei dataset di *snippet* di codice sorgente resi pubblici in letteratura e gli approcci usati dagli autori per l'estrazione da SO.

2.2 Dataset di Snippet di Codice Sorgente in Letteratura

Si procederà ora esaminando brevemente i *dataset* di *snippet* di codice più significativi estratti da SO in letteratura. Prendendo in esame il lavoro svolto in [43], gli autori hanno ottenuto un corpus allineato di *snippet* di codice sorgente e *intent* per la risoluzione del *task* della *code search* attraverso *query* in linguaggio naturale. Gli autori esplorano i limiti dell'uso di euristiche, applicato però al caso di allineare correttamente l'*intent* di tali *snippet*. Seppure sia un *task* diverso da quello proposto nel lavoro di tesi, si vuole descrivere una delle possibili metodologie per l'estrazione e l'etichettatura di un corpus di *snippet* di codice.

Per l'estrazione vengono prima costruite delle *features* estratte manualmente, considerando la struttura degli *snippet* estratti, e successivamente, usando un modello probabilistico, si estraggono ulteriori *features* per catturare la correlazione tra *snippet* e *intent* in linguaggio naturale. In seguito, è stato addestrato un classificatore per l'estrazione dell'intero corpus di *snippet* da SO. A partire da circa 1000 esempi annotati manualmente, il *dataset* estratto (CoNaLa[10]) contiene circa 600 mila esempi annotati ed estratti automaticamente da SO per i linguaggi *Python* e *Java*. Il corpus include anche un *dataset* minore, di circa 2800 esempi estratti automaticamente e poi controllati manualmente.

Un altro esempio di dataset di *snippet* estratto da SO in letteratura è *StaQC*[11]. Si tratta di un ulteriore corpus di *snippet* allineati con l'*intent* estratti dalle risposte di SO [42]. Gli autori propongono una rete neurale per catturare le relazioni tra *snippet* e *intent* in linguaggio naturale. Anche in questo caso sono stati considerati solo due linguaggi, *Python* e *SQL*. Gli autori hanno chiesto a 4 studenti (che avessero familiarità con i linguaggi in esame) di annotare le coppie *intent-snippet* di circa 7000 esempi. Il *dataset*

[10] http://conala-corpus.github.io/

[11] https://github.com/LittleYUYU/StackOverflow-Question-Code-Dataset

estratto automaticamente da SO comprende circa 260 mila coppie codice-domanda.

Se si volesse replicare tali approcci per l'estrazione di un corpus di *snippet* allineati con il linguaggio di programmazione, si potrebbe pensare di etichettare manualmente un piccolo corpus estratto da SO. Si potrebbe poi considerare il contesto in linguaggio naturale, oltre che i *tag* associati, per addestrare un modello in grado di estrarre *features* che mettono in correlazione il contenuto del contesto dello *snippet* e il linguaggio di programmazione dello *snippet* stesso. Tuttavia, tale approccio è stato scartato per via del limite al numero di linguaggi: occorre disporre comunque di esperti in grado di annotare correttamente un numero di linguaggi considerevoli. Tale approccio potrebbe essere esplorato in lavori futuri.

In [2] e [3] viene reso disponibile pubblicamente un ulteriore *dataset* di *snippet* di codice sorgente. Si tratta di un corpus di circa 250 mila *snippet* di almeno 2 linee di codice per 21 linguaggi di programmazione estratti da SO. In tale approccio gli autori assumono che il *tag* identifichi correttamente il contenuto dello *snippet*, filtrando gli *snippet* che presentano più di un *tag* riferito ad un linguaggio di programmazione. Tuttavia, viene considerato anche il contenuto dei post, oltre che delle risposte e tale *dataset* presenta un numero esiguo di linguaggi di programmazione.

2.3 Estrazione di Snippet da StackOverflow

Come descritto precedentemente, l'estrazione del corpus di *snippet* di codice sorgente etichettati con il rispettivo linguaggio di programmazione è avvenuta prendendo in esame delle euristiche per determinare la qualità delle risposte. Nel dettaglio, è possibile riassumere come segue i criteri usati per l'estrazione degli *snippet*:

- I *tag* associati al post identificano il linguaggio di programmazione degli *snippet* nelle risposte al post.
- Lo *score* del post e gli *score* delle risposte forniscono una misura dell'approvazione da parte delle *community*. Risposte con uno score più alto hanno *snippet* di codice qualitativamente migliori.
- Una risposta accettata risolve l'*intent* del post, dunque, gli *snippet* eventualmente presenti nella risposta sono qualitativamente migliori.

Figura 2.2: *Esempio di post e risposta accettata su StackOverflow. Sono evidenziati gli elementi chiave del post e della risposta.*

Come descritto precedentemente, sia il post che le relative risposte possono comprendere *snippet* di codice. Si è scelto di considerare solo il contenuto delle risposte ai post, supponendo che gli *snippet* presenti nei post (come evidenziato anche in [41]) possano essere scritti in *pseudocodice*, o possano comunque essere di bassa qualità.

La prima fase di estrazione, tenendo in considerazione la lista di *tag* che identificano i linguaggi ottenuta come descritto precedentemente, consiste nell'eseguire diverse *query* al database di SO su *BigQuery* per ottenere tutte le risposte che contengono almeno uno *snippet* di codice. Si estraggono le risposte con score maggiore o uguale di 0, per i soli post che hanno a loro volta score maggiore o uguale di 0.

Il contenuto delle risposte estratte contiene la formattazione *HTML* originale, dunque, è abbastanza semplice filtrare quelle risposte che al loro interno contengono i *tag* <pre><code> per identificare eventuali *snippet* presenti.

Successivamente si è passati alla risoluzione di eventuali ambiguità nei *tag* associati ai post delle risposte estratte. Ad esempio, se si considera il *tag* *JavaScript*, non è insolito trovare per lo stesso post *tag* quali *HTML* e *CSS*. Di conseguenza, avendo più di un *tag* che identifica un possibile linguaggio, occorre scartare tali risposte per risolvere l'ambiguità. Si è praticata un'operazione intensiva per confrontare tutti *tag* per le risposte estratte, considerando la lista iniziale dei linguaggi presenti sia su *Linguist* che su SO.

Ad esempio, considerando *JavaScript*, come visto precedentemente, vi sono più di 2 milioni di post su SO. Nella prima fase di estrazione vengono estratte circa 830 mila risposte che contengono almeno uno *snippet* di codice e *score* maggiore o uguale di 0; nella successiva fase di disambiguazione il numero di risposte estratte scende a circa 520 mila.

In alcuni casi, l'autore dello *snippet* può specificare esplicitamente il linguaggio di programmazione in cui è scritto uno *snippet* al fine di avere una migliore *syntax highlighting* su SO. Sebbene ciò possa rappresentare un'informazione importante per l'estrazione di *snippet* etichettati in modo corretto, su SO la maggior parte degli utenti non specifica esplicitamente il linguag-

gio dello *snippet*. Inoltre, per i linguaggi *HTML*, *JavaScript* e *CSS*, su SO è possibile definire degli *snippet* interattivi ed eseguibili nel browser. Per tali linguaggi si ha dunque a disposizione una grande quantità di *snippet* etichettati esplicitamente.

L'estrazione successiva è stata messa in pratica iterando sulle risposte e tenendo conto dei criteri fissati per la scelta degli *snippet*. In particolare, i criteri fissi ad ogni iterazione sono il numero minimo di linee di codice, fissato a 2, e il numero minimo di *token*, fissato a 10. Tale scelta è obbligata in quanto *snippet* molto piccoli non forniscono abbastanza informazioni per identificarne correttamente il linguaggio di programmazione.

In seguito, si procede prendendo in esame ogni linguaggio e, fissato k il numero di *snippet* etichettati che si vuole estrarre, si procede considerando prima eventuali risposte che contengono *snippet* etichettati in modo esplicito. Se dopo questa prima iterazione non sono stati estratti k *snippet*, si procede considerando prima le risposte accettate ed iterando sullo *score* delle risposte.

Su SO, considerando i post che hanno score maggiore o uguale a 0, l'84% circa di questi ha una risposta accettata. L'iterazione sullo score avviene considerando prima le risposte con *score* maggiore o uguale di 10, poi considerando risposte con score compreso tra 5 e 9, in seguito si rilassa tale intervallo tra 2 e 4, e se ancora non è stato raggiunto il numero di *snippet* fissato, si considerano risposte con score pari a 1 ed infine pari a 0. La scelta di tale intervallo non è banale, essendo stata dettata dalla quantità di risposte presenti nell'intervallo fissato. Ad esempio, considerando tutte le risposte su SO con *score* maggiore o uguale di 0 (sempre per i soli post con a loro volta score maggiore o uguale a 0) si ottengono circa 17 milioni di potenziali risposte. Di queste, solo il 36% circa ha score maggiore o uguale di 2, e solo il 5% circa hanno score maggiore o uguale di 10.

Infine, se considerando solo le risposte accettate non si è raggiunto il numero fissato di *snippet*, si considerano le risposte non accettate iterando nuovamente sugli intervalli di *score* descritti. Tale approccio permette l'estrazione di *snippet* per un numero ampio di linguaggi, ma penalizza sia quei

linguaggi per cui non vi sono molti *snippet* validi che quei pochi linguaggi per cui si hanno centinaia di migliaia di *snippet* validi. Si ha infatti *under-sampling* per la maggior parte dei linguaggi, dal momento che viene fissato un limite per il numero di *snippet* da considerare.

Di seguito si riportano i criteri descritti, in ordine di priorità data durante l'estrazione:

- Dichiarazione esplicita del linguaggio: presente, non presente
- Risposta accettata: si, no
- Score risposta: $[10, +\infty)$, $[5, 9]$, $[2, 4]$, 1, 0

Prendendo in esame i criteri elencati, per meglio chiarire la loro adozione, si consideri una tupla composta dai valori di {linguaggio esplicito, risposta accettata, intervallo *score*}. Si inizia ad iterare sui dati per l'estrazione degli *snippet* considerando, in ordine di priorità, le possibili combinazioni dei criteri descritti. Dunque, inizialmente si avrà una tupla del tipo {presente, si, [10,∞)}, successivamente si avrà la tupla {presente, si, [5,9]}, fino ad avere {presente, no, 0}. Se il numero di *snippet* estratti è inferiore al valore fissato, si procede considerando i criteri {non presente, si, [10,+∞)}, e così via, fino all'ultimo criterio che sarà {non presente, no, 0}.

Ulteriori considerazioni da fare riguardano l'eventuale disturbo presente nei *dataset* estratti: il metodo proposto non permette sicuramente un filtraggio accurato degli *snippet*, tuttavia, per quei linguaggi che presentano una grande quantità di *snippet* etichettati esplicitamente, o per quelli per cui non si è reso necessario rilassare fino in fondo i criteri usati, è possibile sostenere che l'approccio usato permette l'estrazione di *snippet* qualitativamente adatti per il *task* che si vuole risolvere.

Ad esempio, per linguaggi quali *Rust*, *CSS*, *Dart* o *HTML* (per citarne alcuni), sono stati estratti solo *snippet* di codice per cui vi era la presenza esplicita del relativo *tag*. Come si vedrà, le prestazioni ottenute nella classificazione di tali linguaggi sono promettenti. Differentemente, per linguaggi

quali *HiveQL* o *GLSL* si è dovuto rilassare i criteri quasi fino in fondo per ottenere il numero fissato di *snippet*.

Un ulteriore problema affrontato riguarda la presenza di *snippet* duplicati su SO[12]. In [6] gli autori cercano di affrontare tale problema, fornendo una visione dell'ampiezza del fenomeno presente su SO. In particolare, volendo riportare un esempio, gli autori citano la presenza di uno *snippet* duplicato, il più delle volte dallo stesso autore, ben 31 volte all'interno di risposte diverse su SO. Una delle motivazioni di tale pratica potrebbe risiedere nel meccanismo di *gamification* alla base dell'interazione tra gli utenti della *community*, spingendo un utente a postare più volte la stesso *snippet* (invece di creare un collegamento alla risposta originale) per guadagnare punti.

Ciò ha portato, durante la fase di estrazione, a mettere in pratica un meccanismo di filtraggio per evitare *snippet* duplicati.

A conclusione del processo di estrazione, per affrontare il *task* proposto, sono stati costruiti due corpus di *snippet*:

- 122000 snippet per 61 linguaggi di programmazione.
- 103500 snippet per 69 linguaggi di programmazione.

Aumentando il numero di *snippet* per linguaggio è possibile ottenere corpus più grandi, riducendo però il numero di linguaggi che possono essere considerati. In alternativa si potrebbe considerare l'idea di usare un *dataset* non bilanciato. Tuttavia, dal momento che l'obiettivo è stato ottenere un *dataset* che comprenda un numero di linguaggi il più ampio possibile, si è trovato in 2000 e 1500 il numero ideale di *snippet* per linguaggio considerando due dataset perfettamente bilanciati.

[12]https://meta.stackoverflow.com/questions/375761

Statistiche Dataset Estratti

Per i *dataset* in esame sono state calcolate delle statistiche per comprendere la natura degli *snippet* estratti. In entrambi i *dataset*, il numero medio di *token* per *snippet* di codice è pari 55 (il processo di estrazione dei *token* verrà descritto nel capitolo successivo). Inoltre, circa il 50% degli *snippet* estratti ha un numero di *token* al di sotto del valore medio. Gli *snippet* estratti hanno inoltre un numero medio di linee di codice pari a 11, e circa il 63% degli *snippet* ha un numero di linee di codice al di sotto del valore medio. In particolare, circa il 93% degli *snippet* estratti hanno meno di 25 linee di codice. Tale dato sarà utile per comprendere le scelte fatte per il *rendering* delle immagini degli *snippet* estratti. In Fig. 2.3 e 2.4 è possibile esaminare i grafici che mettono in evidenza la distribuzione del numero di linee di codice e di *token* per gli *snippet* estratti.

Infine, i *dataset* sono stati suddivisi in tre parti, ognuna riservata per l'addestramento, la validazione e il *testing* dei modelli testati. Nel dettaglio, si è scelto di suddividere i *dataset* in parti che comprendono per ogni linguaggio il 65%, 14% e 21% degli *snippet*. Dunque, per il corpus da 61 linguaggi, per ogni linguaggio si hanno 1300, 280 e 420 *snippet* per *train*, *validation* e *test set* rispettivamente. Per il dataset da 69 linguaggi, per ogni linguaggio si hanno invece 975, 210 e 315 *snippet* rispettivamente.

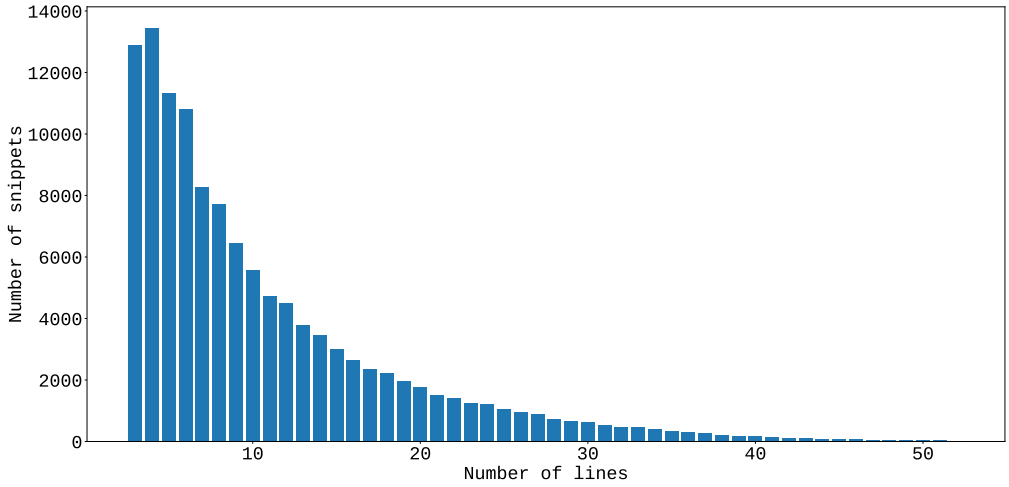


Figura 2.3: *Distribuzione del numero di linee di codice per gli snippet estratti. Vengono mostrati solo i valori per snippet di lunghezza inferiore a* 50 *linee di codice.*

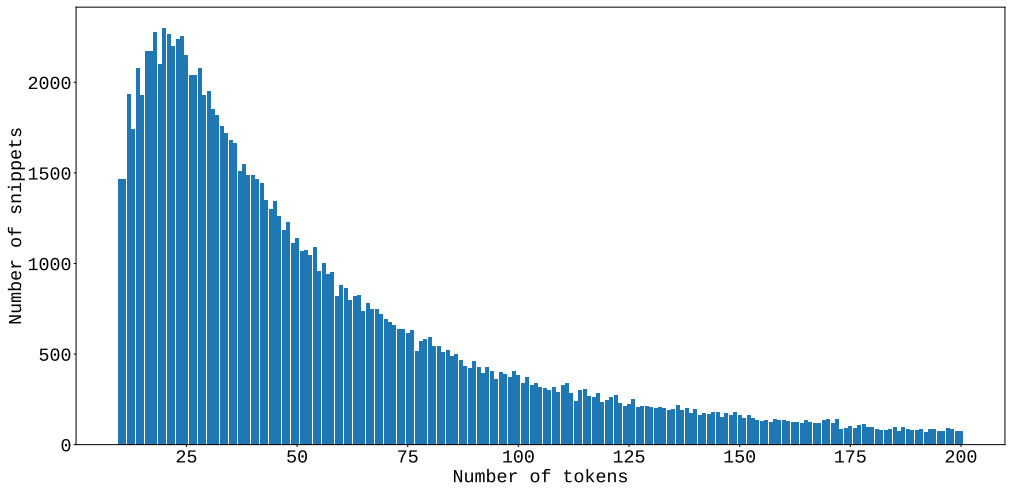


Figura 2.4: *Distribuzione del numero di token per gli snippet estratti. Vengono mostrati solo i valori per snippet di lunghezza inferiore a* 200 *token.*

2.4 Rendering delle Immagini

Come accennato, l'obiettivo è implementare un approccio *multimodale* per la classificazione del linguaggio di programmazione degli *snippet*. A partire dal corpus estratto sono state renderizzate le immagini relative ad ogni *snippet*. Nel generare le immagini sono state fatte alcune scelte dettate anche dalla natura del corpus a disposizione.

Si è scelto di generare delle immagini con sfondo nero e testo bianco in formato *RGB*, dunque, viene generato un canale e duplicato per tre volte. Tale scelta, come si vedrà nella sezione dedicata alle architetture utilizzate, è dettata dall'input richiesto dalla rete che mappa l'input di immagini.

Il passo successivo è stato determinare la dimensione ideale delle immagini da generare. Prima di tutto, si è scelto di usare il *font RobotoMono*[13] e dimensione fissata a 11 pt. In seguito, considerando una piccola porzione del corpus, sono state generate immagini di diverse dimensioni. Naturalmente, immagini più grandi comprendono potenzialmente più informazioni, tuttavia, si è sperimentato che immagini più grandi non aggiungono informazioni utili al miglioramento delle prestazioni del modello. Inoltre, considerando immagini troppo grandi, la maggior parte di esse presenta poco testo in relazione alla dimensione. Infatti, come visto nella sezione precedente, circa il 93% degli *snippet* ha un numero di linee di codice inferiore a 25, e circa il 63% ha meno di 10 linee di codice.

Tra le configurazioni provate, quella che è risultata essere il giusto compromesso, ha dimensione 294×294 *pixels*. Per queste immagini è possibile avere rappresentate un numero massimo di 24 linee di codice per *snippet* e un numero massimo di 42 caratteri per linea. Inoltre, a differenza delle operazioni di pre-processamento messe in pratica per la rappresentazione testuale degli *snippet* (come si vedrà nel capitolo successivo), per il *rendering* delle immagini il contenuto degli *snippet* non è stato processato in alcun modo.

[13] https://fonts.google.com/specimen/Roboto+Mono

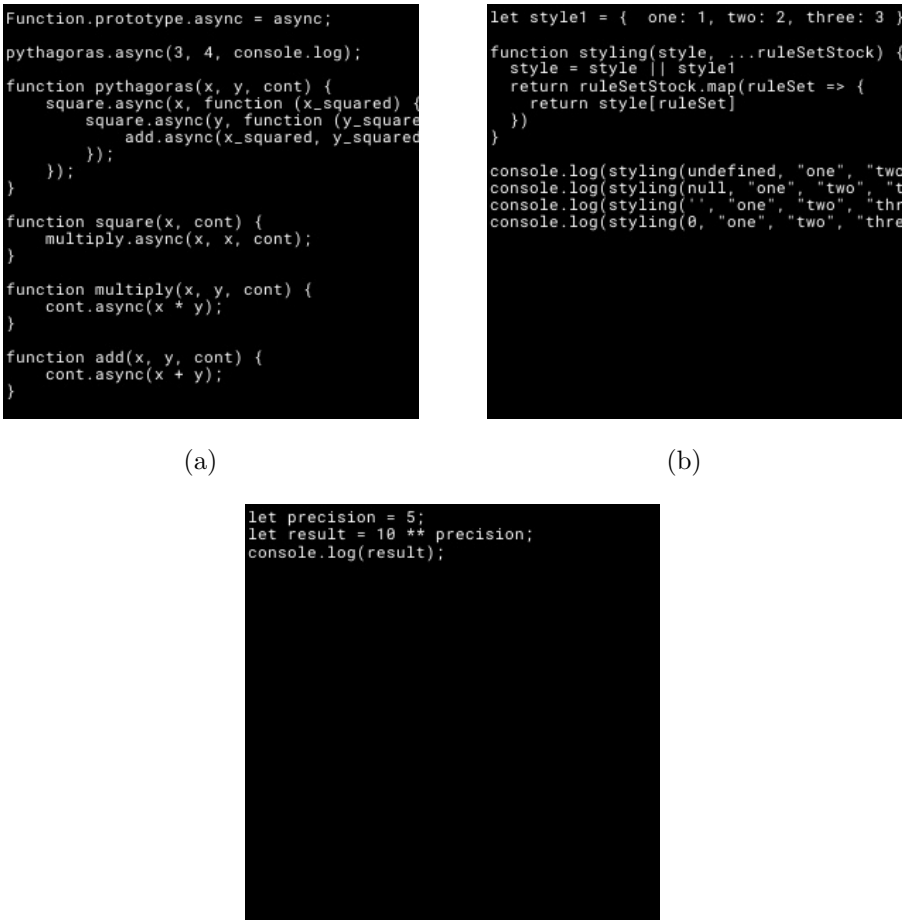


Figura 2.5: *Esempi di rendering delle immagini per snippet di codice scritti in JavaScript. Vengono riportati tre esempi di snippet con numero di linee di codice differente.*

Capitolo 3

Approccio

I dataset esportati comprendono per ogni *snippet* di codice sorgente una rappresentazione testuale e l'immagine renderizzata a partire dal testo. Prima di procedere nella descrizione dell'approccio usato per la classificazione del linguaggio di programmazione degli *snippet*, occorre dedicare l'attenzione sul lavoro svolto per processare ulteriormente gli *snippet* testuali, al fine di trasformarli in rappresentazioni vettoriali numeriche che possano essere usate come input per una rete neurale. Si procederà descrivendo brevemente il processo di trasformazione per la costruzione di *word embeddings*, per poi descrivere le architetture di base scelte per mappare gli input testuali e le immagini; seguiranno i dettagli dell'approccio innovativo di *side-tuning* e l'implementazione realizzata.

3.1 Pre-processamento e Word Embeddings

Ogni *snippet* testuale di codice sorgente deve essere trasformato in valori numerici comprensibili per una rete neurale. Per fare ciò, è possibile far riferimento alle tecniche di elaborazione del linguaggio naturale (*NLP*) che permettono di ottenere una rappresentazione vettoriale in grado di esprimere relazioni semantiche e sintattiche tra parole attraverso misure di similarità. Prima di tutto, si procede ad identificare le componenti significative più

piccole che compongono uno *snippet*; in altri termini, si vogliono identificare le parole che compongono uno *snippet*, estraendo così un vettore di *token* per ogni *snippet*. Tale processo è stato realizzato usando una semplice espressione regolare. Prima dell'estrazione dei *token*, ogni stringa compresa tra apici è stata sostituita con un *token* "*strv*" [15]. Ciò è stato fatto supponendo che il contenuto tra apici il più delle volte rappresenta frasi in linguaggio naturale inserite dall'autore dello *snippet* per aggiungere espressività al codice, per stampare a video o per eventuali *log*, oppure per includere codice sorgente scritto in linguaggio diverso.

L'espressione regolare usata non cattura i caratteri numerici e i caratteri speciali quali *carriage-return*, *line-feed* e tabulazioni. Diversamente da quanto accade in *NLP*, i caratteri di punteggiatura vengono considerati come *token* indipendenti, in quanto sono significativi per l'espressività di alcuni linguaggi di programmazione. Inoltre, l'espressione regolare usata permette di catturare simboli come += come un unico *token*, piuttosto che considerare + e = come *token* indipendenti.

Per ottenere una rappresentazione vettoriale numerica possono essere usati diversi approcci. Una semplice tecnica consiste in due fasi: si costruisce prima un vocabolario ordinato dei *token* e si assegna ad ogni *token* un indice, che banalmente rappresenta la posizione del *token* nel vocabolario. Si definisce poi ogni *token* come un vettore a lunghezza fissa dove tutti i valori sono fissati a 0, eccezion fatta per il valore la cui posizione nel vettore rappresenta la posizione del *token* nel dizionario, che viene fissato a 1. Tale rappresentazione prende il nome di *One-Hot Encoding* e soffre di molti limiti, tra cui l'impossibilità di definire nozioni naturali di similarità tra due vettori *one-hot*. Come accennato, si vogliono ottenere *word embeddings*[9] a lunghezza fissa su cui è possibile definire operazioni di similarità.

Si introducono di conseguenza i modelli *Word2Vector* [29]. Il successo di questi modelli è dovuto alla loro semplicità in relazione ad un'alta efficacia nel rappresentare adeguatamente il significato semantico delle parole [30]. In

modo semplicistico, un modello *Word2Vector* è una rete neurale con un solo *layer* intermedio che viene addestrata per ricostruire il contesto linguistico di una parola. Per apprendere gli *embedding* con un modello *word2vec* si procede come segue: si prepara un corpus che consiste in un insieme di frasi e un vocabolario contenente le parole di cui apprendere la rappresentazione vettoriale; si fissa una finestra temporale, definendo l'ampiezza del contesto di una parola e si procede, per ogni frase nel corpus, nell'estrazione di coppie di parole facendo scorrere la finestra temporale sull'intera frase. Ad ogni passo, viene fissata una parola target e si procede considerando tutte le possibili coppie di parole nel contesto definito.

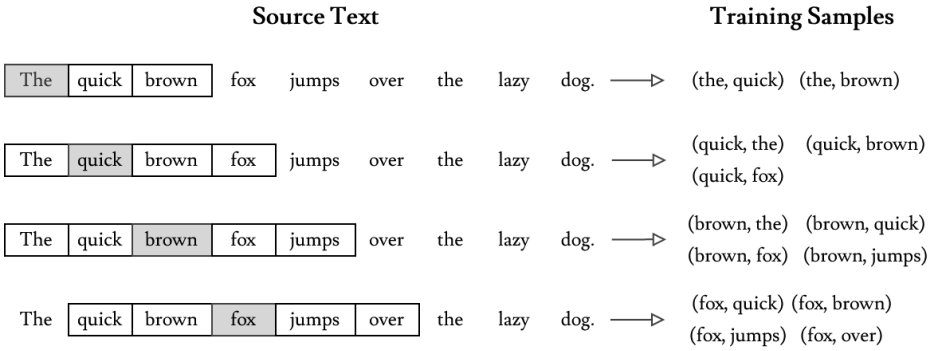


Figura 3.1: *Esempio di scorrimento della finestra temporale con dimensione fissata a* 2*.*

Il *layer* intermedio della rete ha dimensione data dalla lunghezza fissata degli *embedding* e dal numero di parole nel corpus. Le parole vengono passate in input alla rete come *one-hot encoding* e ad ogni iterazione, durante l'addestramento, i pesi del *layer* intermedio vengono aggiornati cercando di approssimare una funzione di perdita che definisce la distanza tra le probabilità dell'output ottenuto e quello target. Alla fine del processo di training, il *layer* intermedio sarà una matrice dove ogni riga rappresenta i *word embeddings* appresi.

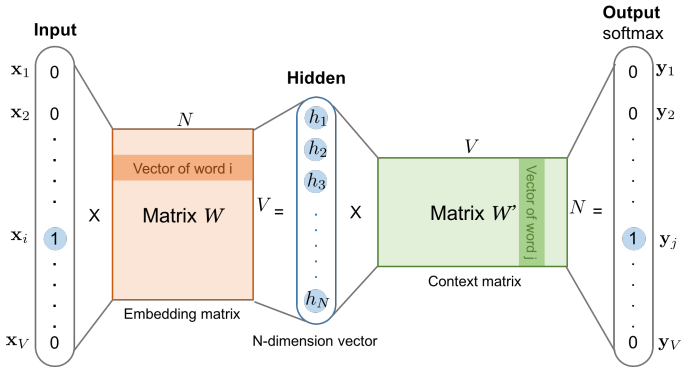
Mikolov et al. [29] descrivono due approcci per l'apprendimento di *word embeddings*: un modello *Continuous Bag-of-Word (CBOW)* e un modello *Skip-Gram*.

- Un modello *CBOW* cerca di prevedere la probabilità di una parola a partire da un contesto. Statisticamente un modello *CBOW* riesce ad estrarre maggiori informazioni distribuzionali, trattando dunque il contesto come un'unica osservazione e perciò, teoricamente, tale modello risulta migliore per ottenere *word embeddings* quando si ha a disposizione un dataset di dimensioni non eccessive.

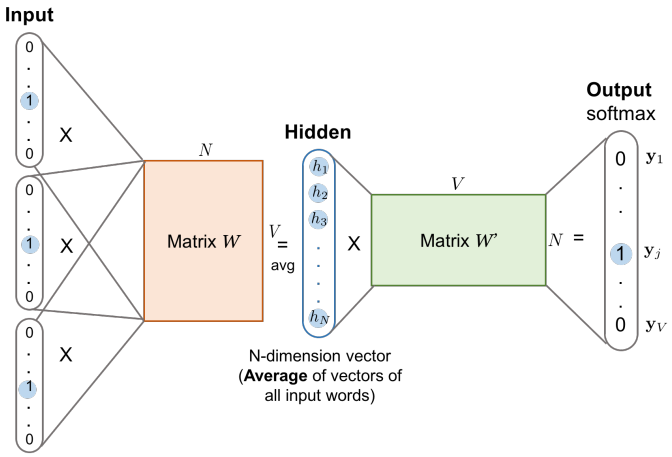
- Un modello *Skip-Gram*, al contrario, a partire da una parola target, cerca di prevedere la distribuzione di probabilità delle parole che ne compongono il contesto. Statisticamente, un modello *skip-gram* tratta ogni coppia contesto-target come una nuova osservazione, rendendolo più adeguato per dataset più grandi.

La scelta dei parametri e la dimensione del corpus giocano un ruolo fondamentale nella costruzione dei *word embeddings*. Inevitabilmente, anche la scelta del modello (*CBOW* o *Skip-Gram*), la scelta della dimensione degli *embeddings* e la dimensione della finestra che determina il contesto di una parola, definiscono la qualità dei *word embeddings* ottenuti.

Nel lavoro di tesi svolto, al fine di ottenere dei *word embeddings* in grado di esprimere al meglio le informazioni sintattiche e semantiche di uno *snippet*, sono stati addestrati diversi modelli *word2vector*, testando sia modelli *skip-gram* che *cbow*, e con parametri diversi. Successivamente, i *word embeddings* ottenuti, sono stati usati come input per una *Convolutional Neural Network* (*CNN*), prendendo in esame nelle fasi successive gli *embeddings* prodotti dal modello *word2vector* per cui la rete addestrata ha raggiunto un accuratezza maggiore sui dati di test.



(a) *Skip-Gram*



(b) *CBOW*

Figura 3.2: *Rappresentazione dei modelli Skip-Gram e CBOW. Per il modello Skip-Gram sia il vettore in input x che il vettore in output y sono rappresentazioni di tipo one-hot. Il layer intermedio rappresenta l'embedding dell'input di dimensione N. Per il modello CBOW viene invece calcolata la media dei vettori one-hot delle parole in input.*

Usando l'implementazione *word2vector* messa a disposizione dalla libreria *Gensim*[1], sono stati appresi *word embeddings* con un modello *Skip-Gram* addestrato per 10 epoche per ognuno dei dataset estratti, considerando una finestra pari a 5 e dimensione degli *embedding* pari a 300.

[1]https://radimrehurek.com/gensim/models/word2vec.html

Altra considerazione fatta riguarda il numero minimo di occorrenze per *token*: sono stati inclusi nel vocabolario ordinato solo *token* che compaiono almeno 10 volte nel corpus degli *snippet*. Ciò ha portato ad una riduzione del numero totale di *token* a circa il 10% del totale, e ad una riduzione al 95% dei *token* totali nel corpus. In altri termini, la riduzione della dimensione del vocabolario è stata significativa, perdendo il 90% circa dei *token*, tuttavia ciò ha impattato in modo non aggressivo sul numero totale di *token* nel corpus (perdendo meno del 5% di *token* sul totale). Ciò ha inevitabilmente introdotto possibili *snippet* che presentano *token Out-Of-Vocabulary* (OOV). Se in *NLP* ciò rappresenta un problema affrontato da diversi studi, in quanto ogni parola porta con sé un significato semantico utile [16], nel caso della classificazione degli *snippet* ciò ha permesso l'eliminazione di quei *token* che non aggiungono alcuna informazione utile per l'identificazione del linguaggio di programmazione. Occorre però menzionare che in letteratura è possibile trovare sforzi atti ad avere una corretta rappresentazione di *token* OOV per il codice sorgente [11], utile soprattutto in contesti in cui il task richiede una buona rappresentazione del codice sorgente per comprenderne il significato semantico.

Successivamente la dimensione del vettore dei *token* di ogni *snippet* è stata limitata a 100, e al posto dei *token* nel vettore è stato inserito l'indice del rispettivo *token* nel vocabolario. Dal momento che circa l'86% degli *snippet* presenta un numero di *token* inferiore a 100, è stato introdotto un *padding* a 0 per avere *snippet* della stessa dimensione, e troncamento della lunghezza per gli *snippet* che eccedono il limite fissato (meno del 15% sul totale).

3.2 Architetture di base

Avendo definito come sono stati processati gli input testali, e considerando le corrispondenti immagini renderizzate a partire dalla rappresentazione testuale degli *snippet* (come descritto nel capitolo precedente) si procederà ora ad illustrare nel dettaglio le architetture dei modelli base usati per la classificazione. Si cercherà di descrivere le peculiarità e i dettagli implementativi della *CNN* che opera sul dataset testuale e la *CNN* che opera sul dataset di immagini. Successivamente si descriverà la tecnica di *side-tuning multimodale* che opera considerando come input coppie testo-immagine del dataset ottenuto, e si esplicheranno i dettagli della rete finale implementata.

Occorre precisare che al livello implementativo è stato usato il *framework Pytorch* [2], per cui si farà riferimento all'implementazione delle reti considerando le classi implementate dalla libreria.

3.2.1 CNN per la classificazione del testo

La *baseline* implementata per la classificazione delle rappresentazioni testuali degli *snippet* è una *CNN* non profonda e prende spunto dall'architettura implementata in [24] e [4] per la classificazione di documenti, e dalle architetture proposte in [31], [15] e [8] per la classificazione di codice sorgente. Si procede realizzando una *lookup table* contenente per ogni riga gli *embedding* dei *token* nel vocabolario. Uno *snippet* viene dato in input alla rete come vettore di 100 valori numerici, dove ogni valore rappresenta l'indice del corrispettivo *token* nella *lookup table*.

Il primo *layer* della rete è un *layer Embedding* che trasforma l'input di dimensione 100 in un output di dimensione 100×300, mappando ogni valore con il corrispettivo *embedding* nella *lookup table*. La rappresentazione degli *embedding* non viene aggiornata durante l'addestramento della rete. L'output viene poi passato ad un *layer* convoluzionale 1D (*conv1d*)[3]. Si può

[2] https://pytorch.org/

[3] https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html

interpretare l'applicazione di un *layer conv1d* come una convoluzione temporale che scorre sull'input applicando un filtro sugli *embedding* nella finestra temporale. Nell'implementazione realizzata è stato fissato un *kernel* pari a 3 e un numero di filtri pari a 1280. Sull'input descritto, il filtro si muove 98 volte (con *stride* fissato a 1 e *padding* fissato a 0), producendo un output con dimensione 98×1280. La scelta del numero di filtri pari a 1280 non è banale e verrà esplicata successivamente nella descrizione dell'architettura che implementa il *side-tuning*. Inoltre, con diversi test sperimentali effettuati, sono state testate diverse reti con numeri di filtri differenti: la scelta di un numero di filtri pari a 1280 ha permesso di ottenere un *accuracy* media più alta.

Sull'output ottenuto viene applicata una funzione di attivazione *ReLU* $f(x) = max(0, x)$ e successivamente l'output risultante viene passato ad un *layer MaxPool1D*[4] per ottenere un output di dimensione 1280. Il *layer MaxPool1D* permette di ridurre la dimensione dell'input considerando il valore massimo di ogni filtro. In genere, i *layer* di *pooling*, oltre ad una riduzione della dimensione dell'input, permettono di controllare possibili *overfitting* e ridurre il numero di parametri della rete da addestrare.

Subito dopo è stato posto un *layer* densamente connesso (*Linear*) che sull'input di dimensione 1280 applica una trasformazione lineare e restituisce un output di dimensione 512. Per regolarizzare la rete durante l'addestramento [18] è stato posto un *layer Dropout* con probabilità 0.5. Un *layer Dropout* assegna semplicemente valore 0 ad alcuni valori dell'input con probabilità specificata. Infine un *layer Linear* produce in output la probabilità delle classi degli *snippet* applicando una funzione di attivazione *Softmax*.

[4]https://pytorch.org/docs/stable/generated/torch.nn.MaxPool1d.html

Tabella 3.1: *Struttura della CNN per gli input testuali. k rappresenta il numero di classi del dataset.*

Text CNN	
Layer	**Dimensioni Output**
Embedding	100×300
Conv1D	98×1280
MaxPooling1D	1280
Linear	512
Dropout	512
Linear	k

3.2.2 MobileNetV2

In letteratura è possibile fare affidamento su diverse *Deep Convolutional Neural Networks* (*DCNN*) per la classificazione di immagini [38], e ogni architettura porta con se peculiarità e innovazioni introdotte dagli autori per migliorarne le prestazioni. Il dataset di riferimento per l'addestramento e la valutazione di tali modelli è senz'alcun dubbio *ImageNet*[12]. *ImageNet* è un database di immagini organizzato secondo la gerarchia di *WordNet*[5], in cui ad ogni nodo della gerarchia appartengono centinaia e migliaia di immagini. Attualmente per ogni nodo vi sono in media più di 500 immagini, per un totale di 1000 categorie e circa 1.2 milioni di immagini[6].

Per la classificazione delle immagini si è scelto di usare una rete pre-addestrata su *ImageNet*. In letteratura è possibile trovare svariati riscontri sulle effettive potenzialità offerte dall'uso di reti pre-addestrate, adattandole per la risoluzione di task specifici e mettendo dunque in pratica il *trasfer-learning* [44]. Si tralascerà per ora il concetto di *transfer-learning* che verrà approfondito nella sezione successiva.

[5]https://wordnet.princeton.edu

[6]http://image-net.org/about-stats

Come architettura base per la classificazione delle immagini degli *snippet* di codice sorgente è stata scelta la *MobileNetV2* [36]. Introdotta da *Google* nel 2018, tale rete è stata pensata per poter essere eseguita in modo performante su dispositivi *mobile*. Il numero di parametri inferiore rispetto ad altre reti che offrono prestazioni simili se confrontate sullo stesso task[7], ne determina una minore richiesta di risorse oltre che un tempo minore per l'addestramento, rendendola la candidata ideale per il task che si vuole risolvere nel lavoro di tesi svolto.

Le peculiarità della *MobileNetV2* consistono nell'adozione di *layer* convoluzionali *Deepthwise Separable* [21], *Linear Bottlenecks* e *Inverted Residual Blocks* [36].

In un layer convoluzionale 2d standard (*conv2d*) il *kernel* (fissandone opportunamente le dimensioni, e fissando anche i valori di *stride* e *padding*) opera in larghezza e altezza. Nel complesso, impilando i *kernel* che operano ognuno su un canale diverso dell'input (il numero di canali determina la profondità dell'input), si ottiene un filtro 3d che si muove su due dimensioni. Ad esempio, nel caso di convoluzioni 2d, se ad un input di dimensione 7×7 con tre canali si applicano 128 filtri di dimensione $3 \times 3 \times 3$, si ottiene un output di 128 canali con dimensione $5 \times 5 \times 1$. Impilando i canali ottenuti dall'applicazione di ogni filtro si ha come risultato una riduzione della dimensione spaziale e un aumento della profondità. Tuttavia, tale approccio è costoso in quanto opera molte trasformazioni sull'input.

L'idea alla base delle convoluzioni *Deepthwise separable* è quella di applicare prima convoluzioni *deepthwise* che riducono la dimensione spaziale in modo efficiente applicando un singolo filtro convoluzionale per canale, e in seguito, applicare delle convoluzioni 1×1 (chiamate anche *pointwise*) che hanno il compito di aumentare la profondità dell'input operando una combinazione lineare sui canali. Per meglio esplicarne il funzionamento, se si considera l'input dell'esempio precedente di dimensione $7 \times 7 \times 3$, usando dunque 3 filtri $3 \times 3 \times 1$ che operano sui canali, si ottiene per ogni filtro un

[7] https://paperswithcode.com/sota/image-classification-on-imagenet

output di dimensione $5 \times 5 \times 1$. Impilando gli output ottenuti, si ottiene una nuova rappresentazione dell'input di dimensione $5 \times 5 \times 3$. In seguito, applicando delle convoluzioni *pointwise* con filtri di dimensione $1 \times 1 \times 3$, si produrrà un *mapping* verso un output $5 \times 5 \times 1$. Dunque, applicando 128 filtri, si ha un output di dimensione $5 \times 5 \times 128$ gestendo un numero di moltiplicazioni nettamente minori rispetto ad un *layer conv2d* [36].

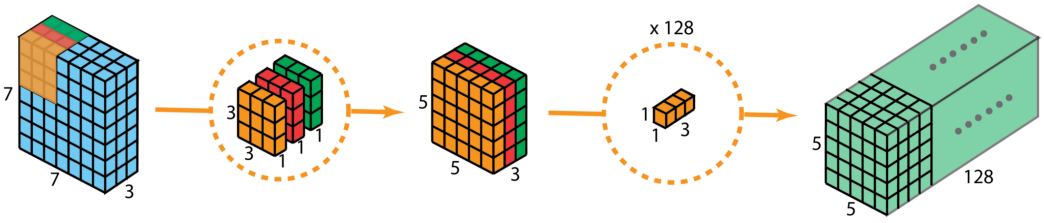


Figura 3.3: *Esempio di applicazione della Depthwise Separable Convolution.*

Per quanto riguarda i *bottleneck layer*, si procede considerando un input con k canali su cui vengono applicate prima convoluzioni *pointwise* per espanderlo e mappandolo verso uno spazio dimensionale più ampio; viene poi applicata una funzione di attivazione *ReLU6* (una semplice *ReLU* in cui il valore massimo è fissato a 6). In seguito si applica una convoluzione *deepthwise* con *kernel* di dimensione 3×3 seguita da una *ReLU6*. Successivamente si riduce nuovamente la dimensione applicando un ulteriore convoluzione *pointwise*. Infine, l'aggiunta di una *residual connection* tra *features* della stessa dimensione, permette di migliorare la velocità di training. Come possiamo intuire, si parla di *inverted residual blocks* poiché, a differenza dei classici *residual block* in cui si segue un approccio *wide-narrow-wide* [17], con i *bottleneck layer* si ha un approccio *narrow-wide-narrow*.

Per sintetizzare, l'intuizione è che i *bottleneck layer* codificano gli input e gli output intermedi della rete, mentre i *layer* interni catturano l'abilità del modello di trasformare *pixels* in *features* che descrivono meglio proprietà delle immagini. Infine, le *inverted residual connection* tra i blocchi permettono di migliore le performance e la velocità di *training*.

3.3 Side-Tuning Multimodale

Nel lavoro di tesi svolto si propone la combinazione di due metodologie in un unico *framework* per la classificazione del linguaggio di programmazione di *snippet* di codice sorgente. Come menzionato, il *framework* proposto è in grado di operare su input eterogenei, addestrando un modello che ha in input immagini e rappresentazioni testuali.

Come descritto nel capitolo precedente, per la gestione degli input sono state definite le architetture delle reti base. Successivamente, analizzando quanto proposto in [47], è stato sperimentato l'approccio innovativo di *Side-Tuning*, introdotto inizialmente da Zhang et al. [46].

Prima di porre l'attenzione ai dettagli di tale approccio, occorre fare una breve digressione sulle tecniche di *Transfer Learning*. In generale, si identificano come *transfer learning* l'insieme di tecniche che prevedono l'addestramento di una rete su un dataset molto ampio (ad esempio *ImageNet*), per poi considerare quanto appreso dalla rete per adattarla nella risoluzione di un nuovo *task*, usando un dataset differente, solitamente di ordini di dimensioni inferiore rispetto al dataset di addestramento iniziale. Si prendono in esame due scenari di *transfer learning* utili al fine di comprendere il lavoro di tesi svolto: l'uso di una *DCNN* come *fixed features extractor*, e il *fine-tuning* di una *DCNN*.

Nello specifico, considerando una *DCNN* pre-addestrata, l'ultimo *layer* della rete (o gli ultimi *layer*, a seconda delle architetture considerate) comporrà il classificatore che restituisce in output le probabilità delle k classi del dataset iniziale. Si opera dunque una rimozione del classificatore, che viene poi sostituito con un nuovo *layer Linear* al fine di ottenere le probabilità delle classi del nuovo dataset. Successivamente si procede ad addestrare la rete sul nuovo dataset (trasformandone opportunamente gli input per rispettare le proprietà del dataset iniziale).

Una volta operata tale modifica, si può scegliere se usare la rete pre-addestrata come *fixed-features extractor* o se riaddestrarla mettendo in pratica il *fine-tuning*.

Nel caso *fixed-features extractor*, si procede addestrando solo il *layer* finale di classificazione. Ciò vuol dire che durante l'addestramento sul nuovo dataset, i pesi intermedi appresi in precedenza non vengono aggiornati [34].

Nel caso del *fine-tuning* della *DCNN*, oltre ad apprendere i pesi dell'ultimo *layer*, si aggiornano i pesi dei *layer* intermedi. Per l'aggiornamento dei pesi intermedi possono essere adottati diversi approcci: si può sbloccare l'intera rete, facendo si che tutti i pesi siano aggiornabili durante il *training*, oppure si possono bloccare i *layer* più profondi aggiornando solo i pesi dei *layer* più in superficie [34], [44].

Avendo chiare le modalità di adattamento di una rete pre-addestrata per la risoluzione di un nuovo task, si cercherà ora di descrivere l'approccio *Side-Tuning* e successivamente i dettagli implementativi del *framework* adoperato.

Il *Side-Tuning* prevede l'uso di una rete base pre-addestrata come *fixed feature extractor* $B : \mathbb{X} \to \mathbb{Y}$, a cui viene affiancata e appresa una rete *side* $S : \mathbb{X} \to \mathbb{Y}$ attraverso una operazione di combinazione lineare.

$$R(x) \triangleq B(x) \oplus S(x)$$

L'operatore $\oplus$, che mette in pratica la combinazione delle due reti, può essere implementato in diversi modi e gli autori del *side-tuning* propongono l'uso del semplice operatore di *alpha-blending*, dove α viene trattato come un parametro della rete da apprendere e la sua scelta è determinata dalla sua correlazione con il *task* in esame [46]. Considerando dunque $B(x) \oplus S(x) \triangleq \alpha B(x) + (1 - \alpha)S(x)$, la scelta di α permette di passare da *fine-tuning* ($\alpha = 0$) a *fixed feature extractor* ($\alpha = 1$).

L'idea alla base del *side-tuning* è che l'uso della rete base come *black box*, permetta il suo adattamento al nuovo *task* senza degradare le performance acquisiste nella risoluzione del *task* originale. Infatti, gli autori sostengono la tesi dell'apprendimento incrementale, dove si cerca di aggiungere una rete più leggera alla rete base (in modo che siano indipendenti) per la risoluzione di nuovi *task*. Tralasciando i dettagli dell'apprendimento incrementale, possiamo riassumere i punti di forza del *side-tuning* come segue:

- Limitazione del degradamento delle performance della rete base.
- Riduzione della rigidità della rete base e supporto all'apprendimento incrementale, migliorandone l'adattabilità per nuovi *task*.
- Riduzione dell'*overfitting* nel caso di dataset di piccole dimensioni.
- Riduzione del rischio di perdita di informazioni nel caso di adattamento su dataset di grandi dimensioni.

Prendendo in esame l'implementazione proposta in [47], è possibile sfruttare la flessibilità offerta dal *side-tuning* per risolvere il *task* della classificazione di documenti usando un approccio multimodale. Nel lavoro di tesi svolto, è stata dunque presa in considerazione la validità di tale *framework*, ed è stato adattato per il task della classificazione del linguaggio di programmazione degli *snippet* usando un approccio multimodale.

Si vuole dunque usare una *DCNN* base pre-addestrata (che mappa l'input di immagini) come *fixed feature extractor* e affiancarvi una replica della rete base, i cui pesi vengono però aggiornati, mettendo in pratica una sorta di *fine-tuning*. Alle due reti base viene affiancata un'ulteriore *CNN*, con pochi *layer*, e dunque con un numero di parametri inferiore, che si occupa di mappare l'input testuale. Gli autori dell'articolo originale [46] propongono l'uso di tecniche di distillazione [19] della rete base per inizializzare i pesi della rete *side*, tuttavia, dal momento che la rete *side* adottata è molto leggera, l'addestramento richiede poco tempo e risorse, perciò si è scelto di addestrarla da zero.

Partendo dall'approccio innovativo descritto, si è deciso di far uso delle architetture base descritte precedentemente per mettere in pratica il *side-tuning multimodale*. L'*alpha blending* descritto in precedenza viene messo in pratica usando i coefficienti α_i che definiscono, volendo usare un'espressione informale, l'importanza assegnata a ciascuna rete. Sia B la rete base, S_1 la replica della rete base e S_2 la *CNN* che mappa l'input testuale, ed R la rappresentazione della rete complessiva, la combinazione avviene in modo

semplice secondo la formula seguente:

$$R(x) = \alpha_0 B(x) + \alpha_1 S_1(x) + \alpha_2 S_2(x) \tag{3.1}$$

Si può notare come assegnando valori diversi agli α_i, sia possibile usare la rete come estrattore di *features* dalle immagini ($\alpha_0 = 1, \alpha_1 = 0, \alpha_2 = 0$), come *fine-tuning* della rete sulle immagini ($\alpha_0 = 0, \alpha_1 = 1, \alpha_2 = 0$), o semplicemente usando la rete *side* che viene addestrata sul testo ($\alpha_0 = 0, \alpha_1 = 0, \alpha_2 = 1$). La scelta dei coefficienti α_i influisce inevitabilmente sulle prestazioni del modello, perciò, come verrà descritto nel capitolo successivo, sono state testate diverse combinazioni di α_i.

Dal punto di vista implementativo, per ogni rete sono stati rimossi i *layer* finali di classificazione. L'output della rete base ha dunque dimensione 1280. L'output del *layer* di *pooling* della *CNN* che mappa le rappresentazioni testuali ha a sua volta dimensione 1280, permettendo la combinazione degli output secondo l'operazione di *alpha-blending* riportato dalla formula 3.1.

Sull'output della combinazione lineare viene applicato un *Dropout layer* con probabilità 0.5 per regolarizzare la rete. Successivamente si applica un *layer Linear* che mappa l'input di dimensione 1280 verso un output di dimensione 512. Il parametro che definisce la dimensione di tale output prende il nome di *side_fc* e per la sua scelta sono state effettuate diverse prove sperimentali, confermando che la scelta di una dimensione pari a 512 permette di ottenere performance migliori oltre che la riduzione del numero di parametri da apprendere. Infine, si applica un ulteriore *Dropout layer* con probabilità 0.5 ed un *layer* di classificazione *Linear* con funzione di attivazione *Softmax*.

Nel prossimo capitolo si procederà ad illustrare i parametri scelti per l'addestramento del modello, in particolare si descriverà l'approccio usato per l'aggiornamento del *learning rate* durante il *training* e si discuteranno i risultati ottenuti sui dataset estratti.

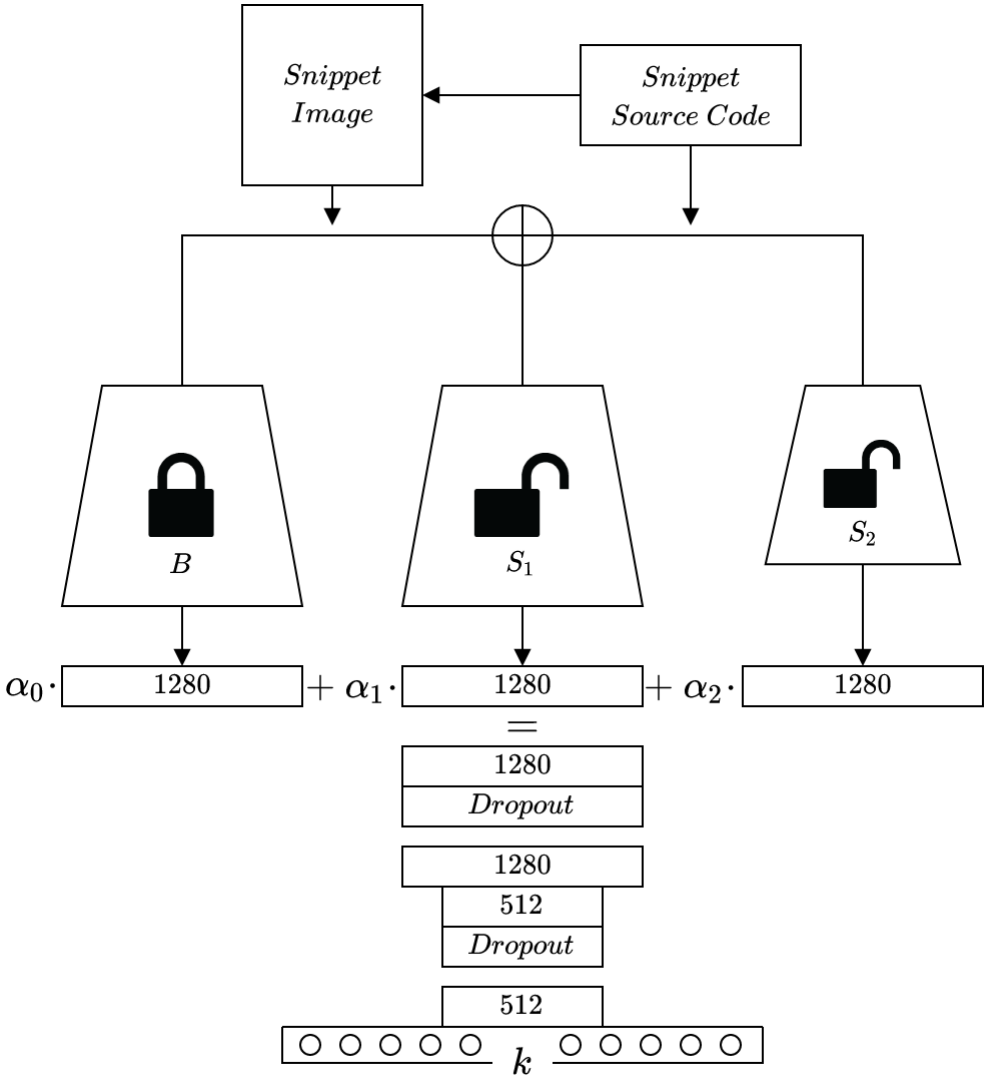


Figura 3.4: *Schema riassuntivo del framework di side-tuning multimodale, con k che rappresenta il numero di linguaggi. Si può notare come la somma degli* α_i *sia* 1*, dunque* $\alpha_2 = 1 - (\alpha_0 + \alpha_1)$.

Capitolo 4

Risultati Sperimentali

In questo capitolo si procederà ad illustrare i risultati ottenuti dall'addestramento del modello di *side-tuning* descritto in precedenza. Si descriverà brevemente l'approccio usato per l'addestramento, dettagliando e motivando la scelta degli iperparametri usati. Si forniranno successivamente una descrizione formale delle metriche di validazione usate. Infine, si discuteranno i risultati cercando di offrire un confronto tra l'approccio proposto e le architetture di base descritte nel capitolo precedente, mettendo in relazione i risultati ottenuti sui *dataset* estratti.

4.1 Addestramento e Iperparametri

Definito il *task* che si vuole affrontare, i dati di *training* e di *test*, e l'architettura del modello, la scelta degli iperparametri rappresenta senz'altro uno dei punti cruciali per l'addestramento di un modello che sia in grado di mappare in modo adeguato i dati in input con l'output corretto.

Come descritto nei capitoli precedenti, sono stati estratti e processati due *dataset* di *snippet* allineati con il rispettivo linguaggio di programmazione. In seguito, si processano tali *snippet* per ottenere una rappresentazione di *word embeddings* e una rappresentazione di immagini. Coppie di immagini-

testo rappresentano l'input del modello di *side-tuning* descritto nel capitolo precedente.

Riepilogando, per i due *dataset* estratti, si hanno 122 *mila snippet* per 61 linguaggi di programmazione e 103500 *snippet* per 69 linguaggi rispettivamente. Per ognuno dei *dataset* si è scelto di usare il 65% per il *training* delle reti, il 14% per la validazione durante il *training*, e il 21% è stato riservato per la valutazione del modello alla fine della fase di addestramento. Inoltre, durante il *training*, ad ogni epoca i dati di *training* e validazione vengono mescolati. Ciò permette di avere *mini-batch* che ad ogni epoca presentano dati differenti, permettendo al modello di generalizzare meglio e fornendo un contributo per evitare possibili *overfitting*.

Uno dei problemi affrontati è stato cercare di ottenere un modello che fosse in grado di generalizzare il più possibile sui dati di *training*, e che dunque fosse in grado di offrire buone prestazioni sui dati di test senza specializzarsi sui dati di *training*. A tal fine, la scelta degli *iperparametri* ha giocato un ruolo chiave per l'addestramento di un buon modello.

La scelta degli iperparametri è stata effettuata considerando una piccola porzione del *dataset* e testando diverse configurazioni. Nel dettaglio, sono state testate diverse combinazioni di *batch size* per i dati di *training* e diversi *learning rate*. Sono stati anche testati diversi *scheduler* per determinare l'aggiornamento del *learning rate* durante la fase di *training*. Ovviamente, al crescere del numero di *iperparametri*, il numero delle possibili configurazioni cresce in modo ingestibile, perciò si è principalmente fatto riferimento a quanto proposto in letteratura per la loro scelta.

Tra gli iperparametri da configurare, il *batch size* determina il numero di esempi da sottoporre al modello durante il *training* prima di calcolare la funzione di perdita e applicare la funzione di ottimizzazione per l'aggiornamento dei pesi del modello. La scelta del *batch size* determina inoltre la memoria e il tempo di addestramento richiesti. Dunque, fissato k il numero di esempi nel *training set*, e b il *batch size*, ad ogni epoca la rete verrà addestrata per k/b iterazioni.

Prendendo in esame quanto proposto in [22], gli autori considerano diversi *batch size*, giungendo alla conclusione che la scelta di un *batch size* troppo grande può portare ad una scarsa generalizzazione sui dati di *training*. Allo stesso modo, la scelta di *batch size* molto piccolo, influisce negativamente sul modello, portando ad un degradamento delle prestazioni. Effettuando diversi test con *batch size* differenti, si è scelto di adoperare un *batch size* pari a 64.

La funzione di perdita, calcolata alla fine di ogni iterazione, misura la distanza (*loss*) tra la previsione della rete e l'output corretto. All'inizio del *training* si avranno valori di *loss* molto grandi. Se durante il *training* la rete apprende in modo corretto, i valori *loss* avranno una tendenza decrescente. Inoltre, il confronto dell'andamento della curva dei valori di *loss* sul *training* e sul *validation set* permette di evidenziare la presenza di un eventuale *overfitting* in fase di addestramento. Dal momento che il modello in esame ha come output la probabilità di una delle classi del *dataset*, si è fatto ricorso alla *Cross Entropy Loss*[1].

Successivamente occorre determinare quale funzione di ottimizzazione risulta essere adeguata per i modelli addestrati. Tale funzione determina il modo in cui i pesi del modello vengono ottimizzati in modo tale da minimizzare la funzione di perdita. Sostanzialmente, i valori di *loss* vengono retropropagati verso i *layer* precedenti della rete, al fine di modificarne i pesi applicando la funzione di ottimizzazione scelta. Inoltre, per controllare l'applicazione della funzione di ottimizzazione, occorre definire il valore del *learning rate*. In sintesi, la scelta del *learning rate*, assieme alla scelta della funzione di ottimizzazione, determinano la capacità del modello di apprendere più o meno velocemente e in modo adeguato dai dati di *training*.

Per l'addestramento delle reti si è scelto di usare l'algoritmo di ottimizzazione *Adam* (*Adaptive moment estimation*) [25]. Tale funzione è un'estensione dell'algoritmo di discesa stocastica del gradiente (*SGD*) e la sua adozione è cresciuta sostanzialmente negli ultimi anni. I benefici apportati dall'uso di

[1] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

Adam, secondo gli autori, sono molteplici, e, per citarne alcuni, riguardano una maggiore efficienza computazionale; una minore richiesta di memoria; una maggiore adattabilità per problemi che presentano grandi quantità di dati e/o parametri; una facile interpretazione e la presenza di poche configurazioni da effettuare. *Adam*, a differenza dell'algoritmo *SGD* che applica lo stesso *learning rate* per l'aggiornamento dei parametri del modello, sfrutta un meccanismo di *learning rate* adattivo, permettendo di trovare il *learning rate* adatto per ogni parametro.

Come accennato, la scelta del *learning rate* adeguato assieme alla funzione di ottimizzazione è di fondamentale importanza per l'addestramento del modello. Per ridurre in modo decisivo i tempi richiesti per l'addestramento, facendo si che la rete possa convergere velocemente, ma allo stesso regolarizzandone i parametri, si è scelto di far uso dell'approccio *One-Cycle Policy* [39]. L'idea di tale approccio consiste nel fissare un *learning rate* massimo sufficientemente grande, facendolo variare durante il *training* da un valore minimo al valore massimo fissati, per poi scendere al di sotto del valore minimo iniziale. L'aggiornamento del *learning rate* avviene di conseguenza dopo ogni *batch* (dunque, ad ogni iterazione).

Oltre ad offrire una rapida convergenza del modello, l'uso della *One-Cycle Policy* offre una migliore convergenza del modello anche in presenza di una quantità ridotta di dati e permette di regolarizzare l'aggiornamento dei parametri durante il *training* grazie all'uso di un *learning rate* più grande. Inevitabilmente, l'uso di tale approccio comporta il bilanciamento con altre forme di regolarizzazione usate per migliorare il *training* del modello.

Nel dettaglio, come mostrato in fig. 4.1, è stato usato un *learning rate* massimo pari a $max\_lr = 0.001$ per tutte le tre architetture addestrate. Il *learning rate* iniziale è pari a $initial\_lr = max\_lr/25$ e l'evoluzione ha un andamento cosinusoidale fino al 30% del numero totale delle epoche, per poi scendere fino al *learning rate* finale $min\_lr = initial\_lr/1e4$.

Per le architetture addestrate, come si vedrà nelle sezioni successive, l'ad-

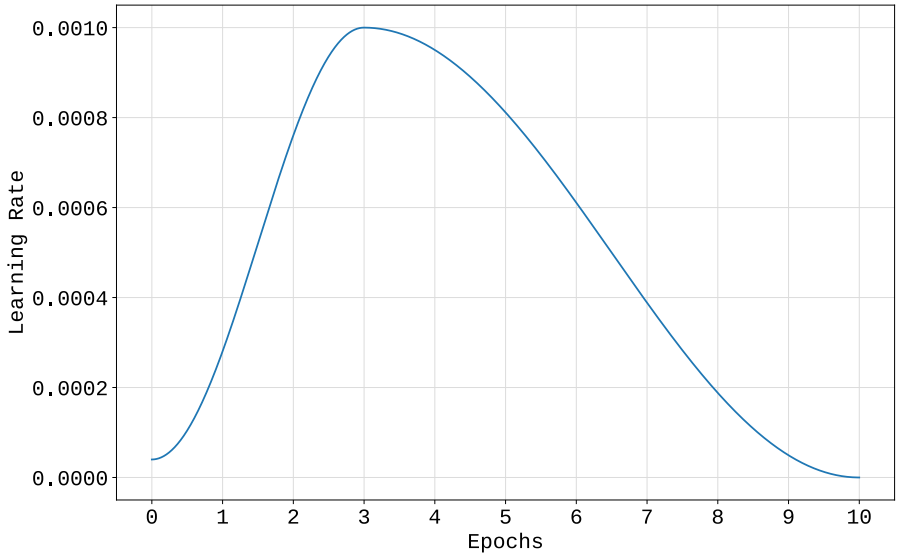


Figura 4.1: *Grafico della curva di evoluzione del learning rate durante l'addestramento con One-Cycle Policy per* 10 *epoche.*

destramento per 10 epoche ha permesso ottenere una buona convergenza dei modelli usando gli *iperparametri* descritti.

Di seguito si riportano in sintesi gli *iperparametri* adottati per l'addestramento dei modelli:

- *Optimizer: Adam*
- *Learning Rate Scheduler: One-Cycle Policy* con $max\_lr = 0.001$
- *Batch-Size* : 64
- Epoche: 10

Per quanto riguarda l'architettura di *side-tuning*, sono state testate 12 possibili configurazione per i valori α_0, α_1 e α_2. Ogni α_i può assumere i valori compresi nell'intervallo [0.2, 0.3, 0.4, 0.5] in modo tale che la somma degli α_i sia 1.

Per effettuare le sperimentazioni è stata usata la piattaforma *Google Colaboratory* (*Colab*[2]). Si tratta di una piattaforma gratuita che permette di scrivere ed eseguire codice *Python* attraverso un browser. *Colab* è basato sul progetto *open source Jupyter*[3]. L'idea alla base di tale piattaforma è di facilitare la scrittura e l'esecuzione di codice attraverso le potenzialità offerte dai *Notebook Jupyter* sul *Cloud* di *Google*. Si è scelto di usare *Colab* poiché offre la possibilità di disporre di un *runtime* pronto all'uso (senza la necessità di configurare e installare le librerie più comuni) oltre al vantaggio di poter avere accesso gratuito a *GPU* con 12 *Gib* di *ram*. Dal momento che le risorse assegnate da *Colab* sono variabili a seconda della disponibilità, il *training* delle reti ha richiesto tempo variabile a seconda della *GPU* assegnata: in alcuni casi l'addestramento e il test dell'architettura di *side-tuning* su 122000 esempi ha richiesto circa 2 ore, in altri casi quasi 5 ore per 10 epoche, con una media che oscilla tra gli 11 minuti e i 30 minuti per epoca. Di conseguenza, non si riporteranno i tempi di addestramento accurati in quanto i dati a disposizione non sono uniformati allo stesso hardware.

[2]https://colab.research.google.com
[3]https://jupyter.org/

4.2 Discussione dei Risultati

4.2.1 Metriche di Validazione

Per comprendere le effettive prestazioni del modello addestrato occorre definire le metriche da utilizzare per quantificare il tasso di successo raggiunto nel classificare i dati di test. Come metrica di base viene usata l'accuratezza (*accuracy*) che fornisce semplicemente la percentuale di previsioni corrette sul numero totale di previsioni attese nei dati di test. Per il task di classificazione in esame, dal momento che si vuole comprendere le prestazioni del modello per ogni singolo linguaggio di programmazione, viene calcolata l'*accuracy* per ogni classe. Oltre all'*accuracy*, per avere maggiori informazioni sui risultati ottenuti, sono state calcolate le metriche quali *Precision*, *Recall* e *F1-Score*, oltre che la costruzione di matrici di confusione.

Se si suppone di avere solo due classi, una *Positive* e una *Negative*, una matrice di confusione, come mostrato in figura 4.2, non è altro che una matrice in cui per ogni classe vengono riportati:

		Actual	
		Positive	**Negative**
Predicted	**Positive**	*True Positive*	*False Positive*
	Negative	*False Negative*	*True Negative*

Figura 4.2: *Esempio matrice di confusione.*

- *True Positive*: i *Positive* identificati correttamente come *Positive*.
- *True Negative*: i *Negative* identificati correttamente come *Negative*.
- *False Positive*: i *Negative* ed identificati erroneamente come *Positive*.
- *False Negative*: i *Positive* ma identificati erroneamente come *Negative*.

Una matrice di confusione permette di identificare ad un primo sguardo quelle classi per cui il modello genera confusione. L'obiettivo di un buon modello sarà dunque quello di massimizzare i *TP* e *TN* e minimizzare i *FP* e *FN*.

L'*accuracy* può essere definita in funzione dell'esempio riportato come $Accuracy = (TP + TN)/(TP + FP + TN + FN)$.

A partire dai concetti di *TP* e *FP* descritti, è possibile definire un'ulteriore metrica di performance, la *Precision*. Tale metrica è data dal rapporto tra il numero di *TP* e la somma di *TP* e *FP*. Considerando il caso in cui si hanno k classi, sia C la matrice di confusione, è possibile formalizzare la *Precision* dell'i-esima classe nella matrice di confusione come segue.

$$P_i = \frac{C_{ii}}{\sum_{j=0}^{k} C_{ji}} \tag{4.1}$$

Considerando invece il rapporto tra *TP* e la somma di *TP* e *FN* si ottiene la metrica *Recall*.

$$R_i = \frac{C_{ii}}{\sum_{j=0}^{k} C_{ij}} \tag{4.2}$$

Al fine di ottenere una stima più accurata delle prestazioni del modello viene calcolata la *F1-Score* data dalla media armonica tra *Precision* e *Recall* sui dati di test.

$$F_1 = \frac{2P_iR_i}{P_i + R_i} \tag{4.3}$$

4.2.2 Risultati Architetture di base

Secondo quanto descritto nelle sezioni precedenti, le architetture di base (*Text CNN* e *MobileNetv2*) sono state addestrate per 10 epoche sui *dataset* estratti. Per comodità, nella discussione dei risultati, i due *dataset* verranno identificati attraverso il numero di linguaggi considerati in ognuno (*Dataset* 61, *Dataset* 69).

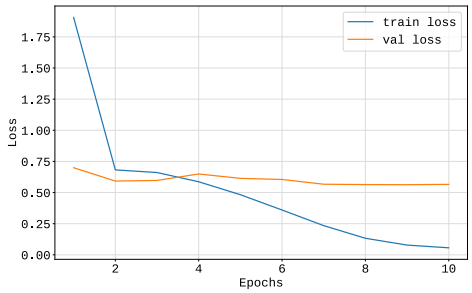
Prendendo in esame la *Text CNN*, si è ottenuta un'*accuracy* del $90,4\%$ sui dati di test del Dataset 61, e un'accuracy dell'$89,2\%$ sui dati di test del Dataset 69. L'addestramento di tale rete, per via del numero ridotto di parametri rispetto alla *MobileNetv2* e alla rete *side-tuning*, ha richiesto dai 25 ai 32 secondi ad epoca.

Analizzando il processo di *training*, come si può facilmente osservare in fig. 4.3, i valori della funzione di perdita (*loss*) sui dati di *training* hanno una tendenza decrescente per l'intero periodo di addestramento, mentre i valori di *loss* sui dati di validazione hanno un'evoluzione decrescente molto meno accentuata. Ciò dimostra che la rete non riesce a generalizzare in modo adeguato sui dati di *training*, specializzandosi su di essi. Si verifica dunque *overfitting*. A dimostrazione di ciò, è possibile esaminare il grafico di evoluzione dell'*accuracy* sui dati di *training* e validazione durante l'addestramento: dopo circa 5 epoche, l'*accuracy* sui dati di *training* continua a crescere in modo più rapido rispetto all'*accuracy* raggiunta sui dati di validazione, fino a raggiungere valori superiori al 98% alla fine del processo di addestramento.

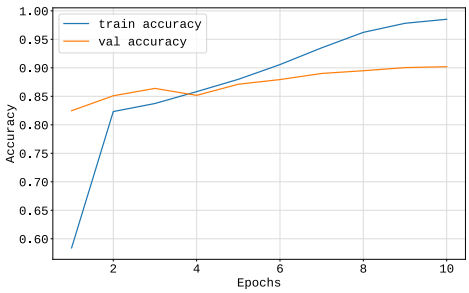
La rete *MobileNetv2* è stata addestrata mettendo in pratica il *fine-tuning*, come descritto nei capitoli precedenti. Sul *Dataset* 61 la *MobileNetv2* raggiunge un'*accuracy* media dell'$86,9\%$, mentre sul *Dataset* 69 l'*accuracy* raggiunta è dell'$85,2\%$.

Analizzando i grafici riportati in fig. 4.4, si può notare come per tale rete, a differenza della *Text CNN*, vi sia un andamento della *loss* sul *training* e sul *validation set* abbastanza regolari. Allo stesso modo, l'*accuracy* cresce in

egual modo per le prime 8 epoche. Durante le ultime due epoche le curve di *loss* e *accuracy* divergono in modo significativo, a dimostrazione della perdita di generalizzazione sui dati di *training*. Il *fine-tuning* della rete ha richiesto un tempo variabile tra 10 e 25 minuti per epoca.

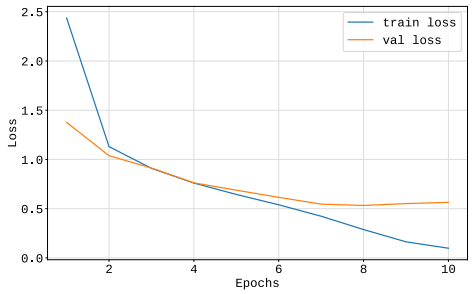


(a) Loss *Text CNN*

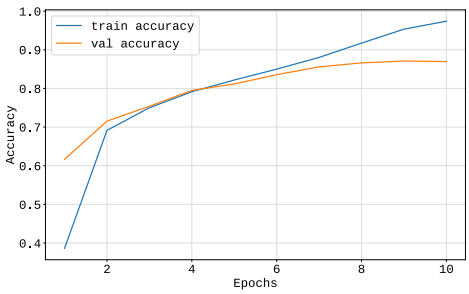


(b) Accuracy *Text CNN*

Figura 4.3: *Grafico dell'andamento dei valori di loss e accuracy della Text-CNN per i dati di training e di validation sul dataset contenente* 61 *linguaggi.*



(a) Loss *MobileNetv2*



(b) Accuracy *MobileNetv2*

Figura 4.4: *Grafico dell'andamento dei valori di loss e accuracy della MobileNetv2 per i dati di training e di validation sul dataset contenente* 61 *linguaggi.*

Considerando la differenza di risultati raggiunti da entrambe le reti sui due dataset, è possibile giustificare tale fenomeno prendendo in esame la differenza data dal numero di esempi per classe (2000 per il Dataset 61 e 1500

per il Dataset 69) e dal numero di classi: l'eventuale disturbo presente nei dataset, nel caso di un maggior numero di classi e di un numero minore di esempi per classe, influisce in modo più significativo sulle prestazioni finali del modello, in quanto la presenza di eventuali snippet etichettati in modo errato o che presentano informazioni disturbate assume un peso maggiore. Inoltre, la *Text CNN* si è dimostrata maggiormente accurata nonostante la sua semplice struttura e la presenza di un forte *overfitting* durante l'addestramento. L'ipotesi alla base di tali risultati, come si esaminerà nel dettaglio nella sezione successiva, è che per la classificazione del linguaggio di programmazione per *snippet* di codice sorgente, si ottengono risultati migliori nel caso in cui si da maggiore importanza all'estrazione di *features* testuali. Altra considerazione da fare riguarda le proprietà delle immagini generate: dal momento che più della metà degli snippet hanno un numero di linee di codice inferiore a 11, le immagini potrebbero non presentare abbastanza informazioni tali da permettere un adeguato addestramento della *MobileNetv2*.

4.2.3 Risultati Side-Tuning Multimodale

L'architettura di *side-tuning* multimodale proposta è stata addestrata utilizzando i parametri specificati nelle sezioni precedenti, per 10 epoche. Sono stati addestrati diversi modelli considerando le combinazioni degli α_i descritti in precedenza, e tali prove hanno evidenziato alcune caratteristiche interessanti.

Prendendo in esame il caso in cui venga dato peso maggiore al parametro α_2 (comportando implicitamente l'assegnazione di un peso maggiore alla rete *Text CNN*), si può notare un generale incremento delle prestazioni finali del modello. Nel grafico riportato in fig. 4.5 vengono riportati i valori di *accuracy* ottenuti sui due dataset dalle diverse configurazione usate. Si può facilmente osservare che al diminuire dei valori di α_0 ed α_1, l'*accuracy* ottenuta cresce in modo sostanziale, fino a raggiungere, per valori $\alpha_0 = 0.2$, $\alpha_1 = 0.3$ e $\alpha_2 = 0.5$, il 91.5% sui dati di test del Dataset 61 e il 90.4% sui dati di test del Dataset 69.

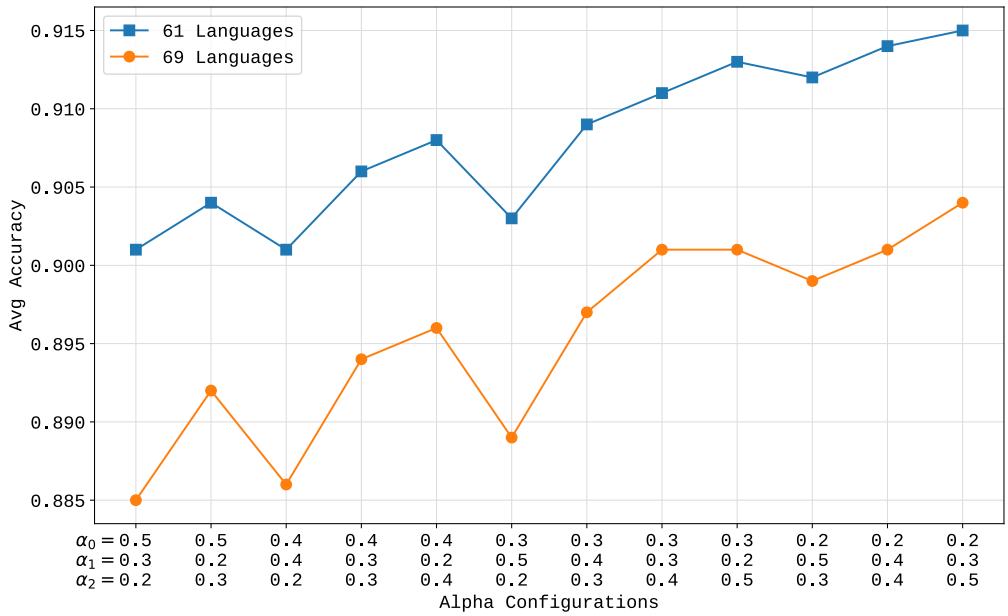


Figura 4.5: *Grafico dell'evoluzione dell'accuracy al variare dei parametri α_i sui due dataset estratti. Si può notare come i risultati migliori sono stati ottenuti dando maggiore importanza alle features estratte dalla Text CNN (α_2).*

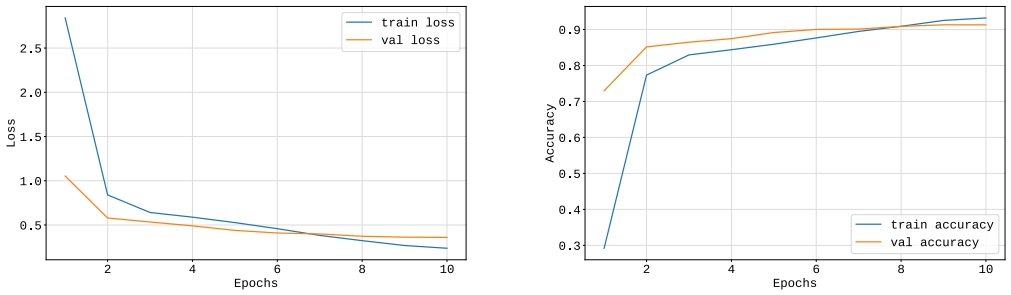
Tale fenomeno permette l'affermazione dell'importanza delle *features* testuali per la corretta classificazione del linguaggio per *snippet* di codice sorgente. Prendendo in esame l'architettura della rete testuale, si può osservare che nell'architettura di *side-tuning* essa consta solo di un *layer* convoluzionale, su cui viene applicato un *layer max pooling*, e il cui output viene combinato con le altre due reti. Un'ulteriore spiegazione del fenomeno può essere riconducibile alla qualità dei *word embedding* ottenuti, a dimostrazione della loro validità per un'adeguata espressione del contenuto degli *snippet*.

Considerando la rete addestrata con i valori α_i che hanno permesso di ottenere i risultati migliori, durante l'addestramento i valori di *loss* sui dati di *training* e di *validation* hanno avuto un'evoluzione decrescente abbastanza stabile. Allo stesso modo, come riportato in fig. 4.6, l'*accuracy* sui dati di *validation* cresce in modo adeguato durante l'addestramento. Nonostante si sia data maggiore importanza alla *Text CNN*, la rete *side-tuning* mostra un

Tabella 4.1: *Tabella riassuntiva dell'accuracy media raggiunta dai modelli addestrati sui dataset estratti.*

	Side-Tuning	Text CNN	MobileNetv2
Dataset 61	0.915	0.904	0.869
Dataset 69	0.904	0.892	0.852

andamento più regolare durante l'addestramento, oltre che l'ottenimento di risultati migliori su entrambi i dataset come mostrato in tabella 4.1. Ciò può essere giustificato affermando che la combinazione con le reti base, permette l'estrazione di *features* che compensano in modo significativo le *features* testuali estratte dalla *Text CNN*.



(a) Loss *Side-Tuning* (b) Accuracy *Side-Tuning*

Figura 4.6: *Grafico dell'andamento dei valori di loss e accuracy dell'architettura side-tuning per i dati di training e di validation sul dataset contenente* 61 *linguaggi.*

Sebbene i risultati ottenuti siano interessanti, l'uso di architetture come quella proposta richiede maggiori risorse e tempi di addestramento più lunghi rispetto all'uso della semplice *Text CNN*. Infatti, l'addestramento della rete ha richiesto un tempo variabile tra 11 e 30 minuti ad epoca, contro i 30 secondi medi per epoca richiesti dalla *Text CNN*.

Analizzando i risultati ottenuti per singola classe dall'architettura proposta (riportati nelle tabelle 4.3 e 4.4), si può facilmente notare come vi siano linguaggi per cui si hanno prestazioni molto soddisfacenti (ad esempio, con-

siderando il Dataset 61, per *GO*, *Haskell*, *NetLogo*, o *R*, per citarne alcuni, si raggiunge un *F1-score* superiore al 96%. Testo a parte fa *Rust*, per cui si raggiunge un *F1-score* del 99%), mentre per altri si raggiungono prestazioni leggermente inferiori.

Per comprendere nel dettaglio per quali linguaggi la rete crea maggiore confusione sui dati di test, sono state generate delle matrici di confusioni per i due dataset, considerando i risultati ottenuti dalle reti con *accuracy* media più alta.

Prendendo in esame i risultati ottenuti sul Dataset 61, come mostrato in fig. 4.7, si possono facilmente notare quali sono i linguaggi classificati in modo errato dalla rete. Nel dettaglio, alcuni *snippet* etichettati come *C++* vengono classificati come *C*. Ciò non è insolito, in quanto codice scritto in *C++* può essere codice valido per *C*, oltre al fatto che i due linguaggi presentano una struttura sintattica molto simile. Tra il linguaggi che presentano caratteristiche sintattiche simili e che vengono confusi dalla rete, è possibile evidenziare nella matrice di confusione i linguaggi *Sass* e *CSS*, dove *Sass* è un'estensione del *CSS*; oppure, *snippet* scritti in *SQL* vengono erroneamente classificati come *HiveQL*, e viceversa; proseguendo, *snippet* etichettati come *TypeScript* sono confusi con *JavaScript*, e così via.

Si può facilmente osservare che gli errori commessi dalla rete sono dovuti alla natura stessa dei linguaggi, i quali il più delle volte condividono strutture sintattiche e parole chiave simili se non identiche. Per una migliore consultazione, si riportano in fondo al capitolo le tabelle riassuntive dei risultati ottenuti utilizzando le tre architetture descritte (tabelle 4.3, 4.4) e le matrici di confusione ottenute sui dati di test dei due dataset (fig. 4.7 e fig. 4.8).

Per confermare ulteriormente la validità del modello proposto, è stato preso in considerazione il *tool GuessLang* che vanta un'accuratezza superiore al 93% su 30 linguaggi di programmazione. Prendendo in esame i linguaggi presenti nei dataset estratti, è stato effettuato un confronto utilizzando 25

linguaggi e la rete *side-tuning* che ha ottenuto i risultati migliori sul Dataset 61. Emerge chiaramente che *GuessLang* non è in grado di classificare, nella maggior parte dei casi, snippet di codice in modo adeguato. Nella tabella 4.2 si riportano i risultati ottenuti sui dati di test del Dataset 61. Gli unici linguaggi per cui *GuessLang* ottiene risultati migliori sono *CSS* e *SQL*. La spiegazione delle scarse prestazioni di *GuessLang* sui dati di test considerati, può essere ricondotta alla natura dei dati su cui è stato addestrato. *GuessLang* è stato addestrato considerando il contenuto di più di 1 milione di file di codice sorgente estratti da GitHub, dunque, l'*accuracy* dichiarata dagli autori si riferisce alla classificazione di porzioni di codice più lunghe di quelle considerate nel lavoro di tesi svolto, a dimostrazione della necessità di esplorare ulteriormente lo sviluppo di *tool* accurati per la classificazione di porzioni ridotte di codice sorgente.

In conclusione, nonostante i risultati promettenti ottenuti, occorrerebbe approfondire il *task* della classificazione di codice sorgente dando maggiore attenzione alle informazioni testuali, a dimostrazione della validità della rete *Text CNN* proposta, la quale, nella sua estrema semplicità, ottiene risultati di poco inferiori rispetto ad architetture più complesse. Il *trade-off* da considerare quando si trattano *features* testuali riguarda la scelta di una rappresentazione numerica adeguata del testo, e perciò, la costruzione di *embedding* di codice offre senz'altro spunti interessanti per la costruzione di *tool* efficaci.

Tabella 4.2: *Confronto tra GuessLang e la rete side-tuning. I linguaggi in comune e supportati da GuessLang sono 25. I valori di side-tuning sono* $\alpha_0 = 0.2$, $\alpha_1 = 0.3$, $\alpha_2 = 0.5$ *per la rete addestrata sul Dataset 61. Viene mostrata l'accuracy raggiunta per ogni linguaggio.*

Linguaggi	Side-Tuning	GuessLang
c	**0.926**	0.488
c#	**0.943**	0.519
c++	**0.793**	0.312
css	0.957	**0.976**
erlang	**0.945**	0.545
go	**0.948**	0.702
haskell	**0.960**	0.898
html	**0.971**	0.331
java	**0.933**	0.455
javascript	**0.931**	0.640
lua	**0.905**	0.621
matlab	**0.936**	0.674
objective-c	**0.914**	0.448
perl	**0.883**	0.514
php	**0.971**	0.557
powershell	**0.952**	0.686
python	**0.917**	0.598
r	**0.983**	0.852
ruby	**0.890**	0.645
rust	**0.988**	0.876
scala	**0.931**	0.714
shell	**0.869**	0.693
sql	0.917	**0.948**
swift	**0.952**	0.667
typescript	**0.829**	0.690

Tabella 4.3: *Valori di Precision (P), Recall (R) e F1-score (F1) per le tre reti addestrate sul dataset contenente* 61 *linguaggi. Sono rappresentati i valori per ogni singola classe, e la Macro Average delle metriche.*

		Side-Tuning			Text CNN			MobileNev2		
	Linguaggio	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
1	actionscript	0.926	0.888	0.906	0.932	0.883	0.907	0.862	0.845	0.853
2	ada	0.957	0.907	0.932	0.930	0.919	0.925	0.883	0.867	0.875
3	applescript	0.966	0.933	0.949	0.972	0.919	0.945	0.945	0.900	0.922
4	assembly	0.878	0.874	0.876	0.836	0.838	0.837	0.822	0.779	0.800
5	autohotkey	0.959	0.955	0.957	0.954	0.929	0.941	0.942	0.929	0.935
6	awk	0.780	0.862	0.819	0.795	0.876	0.834	0.751	0.810	0.779
7	c	0.760	0.926	0.835	0.736	0.888	0.805	0.734	0.879	0.800
8	c#	0.857	0.943	0.898	0.843	0.905	0.873	0.825	0.898	0.860
9	c++	0.898	0.793	0.842	0.846	0.826	0.836	0.826	0.748	0.785
10	clojure	0.940	0.931	0.935	0.946	0.919	0.932	0.916	0.881	0.898
11	common-lisp	0.928	0.924	0.926	0.898	0.960	0.928	0.871	0.888	0.880
12	coq	0.973	0.948	0.960	0.973	0.950	0.961	0.956	0.929	0.942
13	css	0.885	0.957	0.920	0.896	0.945	0.920	0.892	0.907	0.900
14	dart	0.940	0.900	0.920	0.924	0.902	0.913	0.855	0.860	0.857
15	elixir	0.967	0.914	0.940	0.948	0.917	0.932	0.869	0.840	0.855
16	erlang	0.925	0.945	0.935	0.900	0.917	0.908	0.906	0.893	0.899
17	f#	0.912	0.883	0.897	0.887	0.881	0.884	0.844	0.852	0.848
18	fortran	0.903	0.890	0.897	0.910	0.890	0.900	0.829	0.831	0.830
19	glsl	0.868	0.862	0.865	0.824	0.802	0.813	0.815	0.755	0.784
20	gnuplot	0.900	0.924	0.912	0.902	0.919	0.910	0.904	0.898	0.901
21	go	0.973	0.948	0.960	0.966	0.943	0.954	0.928	0.924	0.926
22	groovy	0.878	0.860	0.869	0.872	0.860	0.866	0.811	0.805	0.808
23	haskell	0.969	0.960	0.964	0.942	0.936	0.939	0.945	0.938	0.941
24	hiveql	0.860	0.807	0.833	0.832	0.812	0.822	0.781	0.738	0.759
25	html	0.885	0.971	0.926	0.918	0.957	0.937	0.890	0.924	0.907
26	java	0.838	0.933	0.883	0.836	0.888	0.861	0.835	0.879	0.856
27	javascript	0.854	0.931	0.891	0.850	0.931	0.889	0.807	0.888	0.846
28	json	0.917	1.000	0.957	0.942	1.000	0.970	0.965	0.993	0.979
29	julia	0.975	0.926	0.950	0.963	0.929	0.945	0.918	0.902	0.910
30	kotlin	0.922	0.933	0.928	0.914	0.890	0.902	0.835	0.869	0.852
31	lua	0.931	0.905	0.918	0.939	0.924	0.932	0.845	0.871	0.858
32	matlab	0.885	0.936	0.910	0.893	0.912	0.902	0.920	0.871	0.895
33	netlogo	0.993	0.969	0.981	0.983	0.960	0.971	0.983	0.962	0.972
34	objective-c	0.970	0.914	0.941	0.924	0.902	0.913	0.909	0.881	0.895
35	ocaml	0.897	0.893	0.895	0.883	0.845	0.864	0.826	0.805	0.815
36	perl	0.930	0.883	0.906	0.917	0.874	0.895	0.873	0.805	0.838

Tabella 4.3: *Valori di Precision (P), Recall (R) e F1-score (F1) per le tre reti addestrate sul dataset contenente* 61 *linguaggi. Sono rappresentati i valori per ogni singola classe, e la Macro Average delle metriche.*

		Side-Tuning			**Text CNN**			**MobileNev2**		
	Linguaggio	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
37	php	0.962	0.971	0.967	0.974	0.971	0.973	0.938	0.936	0.937
38	powershell	0.946	0.952	0.949	0.923	0.945	0.934	0.897	0.933	0.915
39	prolog	0.976	0.971	0.974	0.971	0.962	0.967	0.952	0.943	0.947
40	python	0.873	0.917	0.894	0.869	0.902	0.886	0.809	0.876	0.841
41	qml	0.952	0.943	0.947	0.933	0.931	0.932	0.914	0.888	0.901
42	r	0.969	0.983	0.976	0.946	0.967	0.956	0.950	0.945	0.947
43	raku	0.941	0.914	0.928	0.894	0.888	0.891	0.862	0.874	0.868
44	ruby	0.864	0.890	0.877	0.901	0.848	0.874	0.774	0.790	0.782
45	rust	0.993	0.988	0.990	0.976	0.979	0.977	0.980	0.957	0.969
46	sas	0.985	0.945	0.965	0.950	0.940	0.945	0.949	0.929	0.939
47	sass	0.887	0.764	0.821	0.846	0.743	0.791	0.813	0.726	0.767
48	scala	0.968	0.931	0.949	0.929	0.907	0.918	0.877	0.883	0.880
49	scheme	0.894	0.900	0.897	0.925	0.905	0.915	0.851	0.857	0.854
50	shell	0.778	0.869	0.821	0.764	0.817	0.789	0.718	0.757	0.737
51	sql	0.850	0.917	0.882	0.860	0.881	0.871	0.816	0.879	0.846
52	stata	0.976	0.955	0.965	0.969	0.955	0.962	0.914	0.917	0.916
53	swift	0.948	0.952	0.950	0.934	0.938	0.936	0.881	0.938	0.909
54	tcl	0.963	0.924	0.943	0.926	0.898	0.912	0.953	0.871	0.910
55	typescript	0.872	0.829	0.850	0.851	0.840	0.846	0.797	0.783	0.790
56	vb.net	0.925	0.821	0.870	0.894	0.824	0.857	0.828	0.755	0.790
57	vba	0.939	0.948	0.943	0.921	0.943	0.932	0.882	0.874	0.878
58	vbscript	0.912	0.869	0.890	0.871	0.855	0.863	0.802	0.821	0.812
59	vhdl	0.953	0.962	0.957	0.934	0.938	0.936	0.890	0.907	0.899
60	xquery	0.953	0.876	0.913	0.949	0.895	0.922	0.846	0.862	0.854
61	xslt	0.863	0.898	0.880	0.866	0.905	0.885	0.869	0.869	0.869
	macro avg	0.917	0.915	**0.915**	0.905	0.904	**0.904**	0.870	0.869	**0.869**

Tabella 4.4: *Valori di Precision (P), Recall (R) e F1-score (F1) per le tre reti addestrate sul dataset contenente* 69 *linguaggi. Sono rappresentati i valori per ogni singola classe, e la Macro Average delle metriche.*

		Side-Tuning			Text CNN			MobileNev2		
	Linguaggio	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
1	abap	0.970	0.940	0.955	0.949	0.946	0.948	0.918	0.927	0.923
2	actionscript	0.947	0.902	0.924	0.928	0.895	0.911	0.889	0.810	0.847
3	ada	0.950	0.902	0.925	0.917	0.876	0.896	0.894	0.854	0.873
4	antlr	0.941	0.905	0.922	0.846	0.889	0.867	0.878	0.822	0.849
5	applescript	0.970	0.927	0.948	0.980	0.917	0.948	0.946	0.892	0.918
6	assembly	0.840	0.851	0.845	0.842	0.794	0.817	0.761	0.740	0.750
7	autohotkey	0.984	0.962	0.973	0.962	0.956	0.959	0.951	0.930	0.941
8	awk	0.790	0.790	0.790	0.835	0.756	0.793	0.772	0.743	0.757
9	c	0.727	0.863	0.790	0.728	0.806	0.765	0.734	0.771	0.752
10	c#	0.807	0.943	0.870	0.796	0.892	0.841	0.791	0.867	0.827
11	c++	0.798	0.803	0.801	0.763	0.810	0.786	0.690	0.727	0.708
12	clojure	0.967	0.924	0.945	0.935	0.921	0.928	0.897	0.886	0.891
13	common-lisp	0.945	0.921	0.932	0.910	0.927	0.918	0.895	0.870	0.882
14	coq	0.987	0.959	0.973	0.955	0.946	0.951	0.934	0.937	0.935
15	css	0.884	0.971	0.926	0.894	0.962	0.927	0.865	0.933	0.898
16	d	0.876	0.851	0.863	0.869	0.841	0.855	0.801	0.743	0.771
17	dart	0.909	0.914	0.911	0.917	0.914	0.916	0.791	0.854	0.821
18	elixir	0.973	0.930	0.951	0.941	0.905	0.922	0.866	0.844	0.855
19	erlang	0.936	0.930	0.933	0.931	0.898	0.914	0.858	0.886	0.872
20	f#	0.903	0.921	0.912	0.906	0.883	0.894	0.878	0.844	0.861
21	fortran	0.893	0.898	0.896	0.852	0.898	0.875	0.846	0.838	0.842
22	glsl	0.812	0.822	0.817	0.799	0.784	0.792	0.780	0.743	0.761
23	gnuplot	0.888	0.905	0.896	0.876	0.876	0.876	0.855	0.863	0.859
24	go	0.978	0.978	0.978	0.987	0.965	0.976	0.935	0.962	0.948
25	groovy	0.905	0.876	0.890	0.901	0.867	0.883	0.812	0.797	0.804
26	haskell	0.971	0.962	0.967	0.929	0.956	0.942	0.957	0.917	0.937
27	hiveql	0.798	0.803	0.801	0.799	0.810	0.804	0.707	0.727	0.717
28	html	0.868	0.962	0.913	0.857	0.952	0.902	0.846	0.943	0.892
29	inno-setup	0.939	0.883	0.910	0.904	0.867	0.885	0.823	0.854	0.838
30	java	0.825	0.898	0.860	0.781	0.905	0.838	0.726	0.816	0.768
31	javascript	0.820	0.898	0.858	0.847	0.895	0.870	0.781	0.848	0.813
32	json	0.915	0.997	0.954	0.935	0.997	0.965	0.960	0.987	0.973
33	julia	0.930	0.930	0.930	0.950	0.914	0.932	0.911	0.873	0.891
34	kotlin	0.963	0.911	0.936	0.922	0.898	0.910	0.882	0.857	0.870
35	liquid	0.924	0.854	0.888	0.917	0.841	0.877	0.894	0.832	0.862
36	lua	0.907	0.927	0.917	0.932	0.921	0.927	0.848	0.867	0.857

Tabella 4.4: *Valori di Precision (P), Recall (R) e F1-score (F1) per le tre reti addestrate sul dataset contenente* 69 *linguaggi. Sono rappresentati i valori per ogni singola classe, e la Macro Average delle metriche.*

		Side-Tuning			Text CNN			MobileNev2		
	Linguaggio	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
37	matlab	0.932	0.921	0.927	0.929	0.914	0.922	0.874	0.879	0.877
38	netlogo	0.987	0.971	0.979	0.957	0.978	0.967	0.941	0.959	0.950
39	objective-c	0.930	0.886	0.907	0.908	0.883	0.895	0.927	0.844	0.884
40	ocaml	0.892	0.892	0.892	0.862	0.892	0.877	0.850	0.867	0.858
41	pascal	0.861	0.883	0.871	0.819	0.860	0.839	0.765	0.816	0.790
42	perl	0.896	0.902	0.899	0.903	0.889	0.896	0.856	0.848	0.852
43	php	0.962	0.959	0.960	0.961	0.946	0.954	0.922	0.933	0.927
44	powershell	0.955	0.937	0.946	0.936	0.924	0.930	0.897	0.886	0.891
45	processing	0.801	0.756	0.778	0.755	0.775	0.765	0.672	0.651	0.661
46	prolog	0.977	0.956	0.966	0.952	0.949	0.951	0.952	0.940	0.946
47	python	0.863	0.921	0.891	0.873	0.940	0.905	0.830	0.886	0.857
48	qml	0.957	0.917	0.937	0.916	0.895	0.905	0.904	0.835	0.868
49	r	0.981	0.962	0.971	0.980	0.949	0.965	0.931	0.949	0.940
50	raku	0.916	0.905	0.911	0.929	0.917	0.923	0.898	0.841	0.869
51	ruby	0.855	0.879	0.867	0.823	0.857	0.840	0.730	0.816	0.771
52	rust	0.990	0.987	0.989	0.987	0.965	0.976	0.971	0.959	0.965
53	sas	0.986	0.927	0.956	0.964	0.927	0.945	0.973	0.917	0.944
54	sass	0.848	0.759	0.801	0.851	0.778	0.813	0.766	0.717	0.741
55	scala	0.943	0.940	0.941	0.938	0.905	0.921	0.853	0.886	0.869
56	scheme	0.894	0.911	0.903	0.906	0.892	0.899	0.865	0.876	0.871
57	shell	0.716	0.841	0.774	0.721	0.778	0.748	0.656	0.749	0.699
58	smalltalk	0.930	0.933	0.932	0.942	0.924	0.933	0.878	0.914	0.896
59	sql	0.823	0.870	0.846	0.840	0.867	0.853	0.815	0.797	0.806
60	stata	0.962	0.959	0.960	0.958	0.937	0.947	0.924	0.930	0.927
61	swift	0.936	0.971	0.953	0.913	0.968	0.940	0.886	0.933	0.909
62	tcl	0.951	0.917	0.934	0.962	0.889	0.924	0.911	0.848	0.878
63	typescript	0.869	0.819	0.843	0.855	0.806	0.830	0.789	0.711	0.748
64	vb.net	0.890	0.819	0.853	0.885	0.803	0.842	0.788	0.768	0.778
65	vba	0.926	0.908	0.917	0.893	0.927	0.910	0.850	0.860	0.855
66	vbscript	0.876	0.854	0.865	0.824	0.835	0.830	0.770	0.743	0.756
67	vhdl	0.956	0.956	0.956	0.946	0.937	0.941	0.904	0.895	0.900
68	xquery	0.964	0.841	0.898	0.933	0.886	0.909	0.906	0.829	0.866
69	xslt	0.879	0.921	0.899	0.883	0.908	0.895	0.847	0.876	0.861
	macro avg	0.906	0.904	**0.905**	0.894	0.892	**0.893**	0.853	0.852	**0.852**

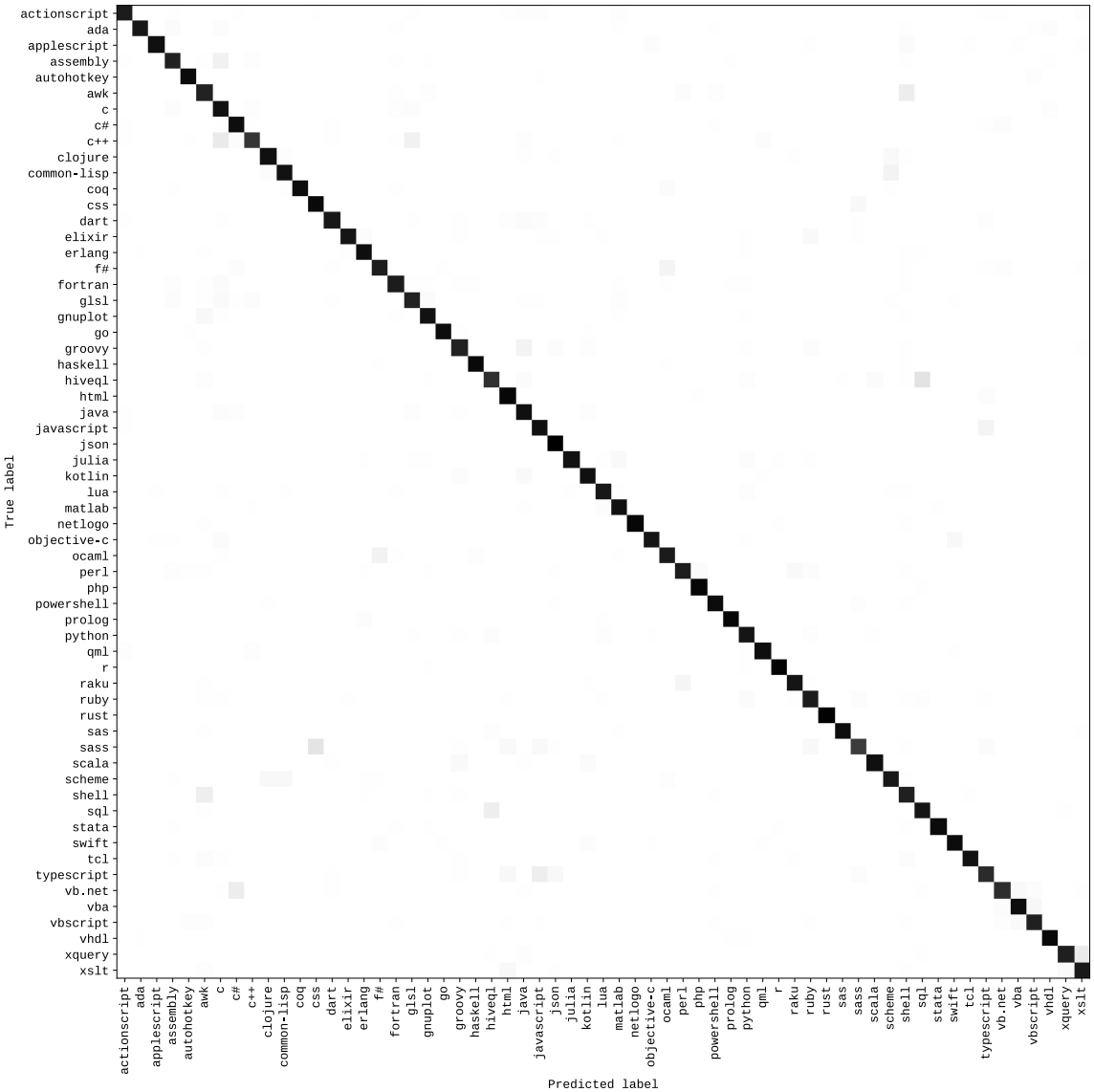


Figura 4.7: *Matrice di confusione sui dati di test del dataset con* 61 *linguaggi per la rete side-tuning con* $\alpha_0 = 0.2,\ \alpha_1 = 0.3,\ \alpha_2 = 0.5$.

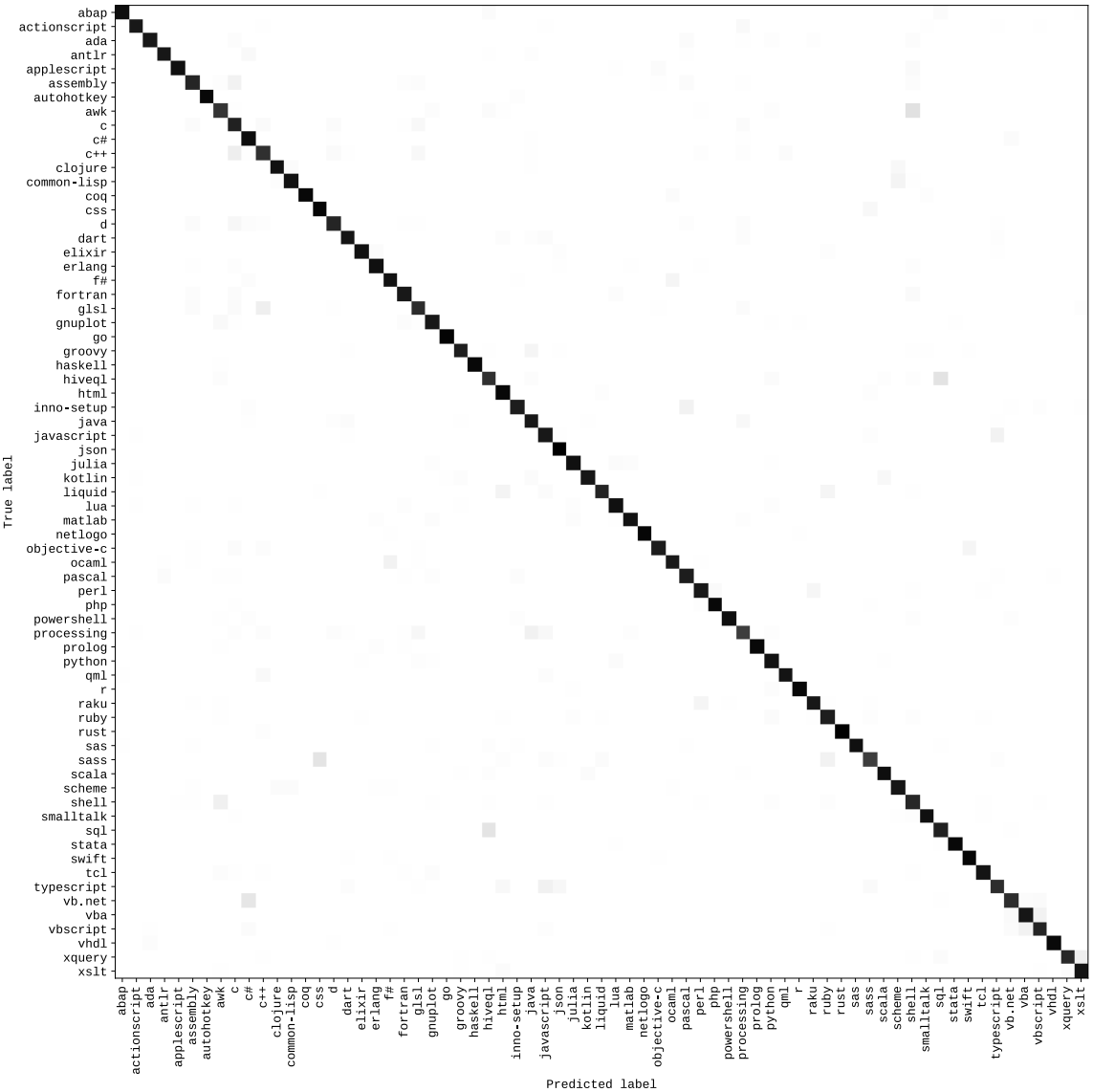


Figura 4.8: *Matrice di confusione sui dati di test del dataset con* 69 *linguaggi per la rete side-tuning con* $\alpha_0 = 0.2,\ \alpha_1 = 0.3,\ \alpha_2 = 0.5$.

Conclusioni

Nel lavoro di tesi svolto è stato affrontato il problema del riconoscimento del linguaggio di programmazione per *snippet* di codice sorgente proponendo un approccio innovativo di *side-tuning multimodale*. Tale approccio permette l'adattamento di una rete base pre-addestrata per la risoluzione di un nuovo *task* attraverso l'affiancamento di una o più reti di dimensioni inferiori. Si è ricorso all'uso della *MobileNetv2* come rete base a cui è stata affianca una sua replica (su cui viene effettuato l'aggiornamento dei parametri sul nuovo *task*) e una *Convolutional Neural Network* (CNN) composta di pochi *layer* e appresa interamente sui *dataset* in esame. L'input gestito dalle reti del *framework* implementato è eterogeneo, composto di rappresentazioni testuali degli *snippet* e di immagini renderizzate dal testo. Ogni rete si occupa dunque di estrarre *features* da immagini e dai *word embedding* ottenuti con un modello *word2vector*. La combinazione delle reti avviene attraverso l'operazione di *alpha-blending*, dove i valori degli α_i determinano l'importanza data alle *features* estratte dalle diverse reti.

Prima di procedere alla sperimentazione del modello proposto, si è reso necessario mettere in pratica un metodo efficace per l'estrazione di *snippet* di codice sorgente da *StackOverflow* (SO). Analizzando quanto fatto in letteratura, i pochi *dataset* di *snippet* presenti considerano un numero esiguo di linguaggi e sono stati estratti per finalità differenti dal *task* in esame. Per l'estrazione del *dataset* si è dunque esaminato il contenuto di SO, determinando delle euristiche decisive per l'estrazione di un corpus di *snippet* allineati in modo sufficientemente adeguato con il rispettivo linguaggio di

programmazione. I corpus estratti comprendono 61 linguaggi e 2000 *snippet* per linguaggio, 69 linguaggi e 1500 *snippet* per linguaggio.

Il modello di *side-tuning multimodale* implementato ha ottenuto prestazioni interessanti, se comparate con lo stato dell'arte, sui corpus estratti. Occorre però precisare che i risultati migliori sono stati ottenuti dando maggiore importanza alle *features* estratte dagli input testuali, dimostrando sia la validità del modello proposto, che la validità dei *word embeddings* appresi. Nel dettaglio, è stata raggiunta un'accuratezza media del 91.5% su 61 linguaggi, e del 90.4% su 69 linguaggi.

Al fine di comprendere gli effettivi benefici apportati dall'uso del *side-tuning*, sono state addestrate e testate le architetture di base sugli stessi dati. Il *fine-tuning* della *MobileNetv2* ha permesso il raggiungimento di un'accuratezza media dell'86.9% su 61 linguaggi e dell'85.2% su 69 linguaggi. La CNN testuale ha invece ottenuto un'accuratezza del 90.4% su 61 linguaggi e dell'89.2% su 69 linguaggi.

I risultati ottenuti mettono in risalto la difficoltà intrinseca del *task* affrontato e la necessità di esplorare ulteriormente tecniche che possano permetterne la risoluzione in modo efficace. Dal lavoro di tesi svolto è emersa anche la necessità di disporre di corpus di *snippet* etichettati in modo corretto per un numero di linguaggio più ampio. Infatti, una delle sfide affrontate, è stata determinare il modo più adeguato per l'etichettatura dei dati. La disponibilità di un corpus qualitativamente corretto permetterebbe inoltre l'esplorazione di ulteriori *task* sugli *snippet* estratti da piattaforme come SO. Tra questi vi sono il *task* di *code search* attraverso *query* in linguaggio naturale; conversione di codice sorgente da un linguaggio ad un altro; conversione da linguaggio naturale a codice sorgente o viceversa, e così via.

Un'idea potrebbe essere quella di sottoporre alle *community* di sviluppatori corpus di *snippet* di codice per la loro etichettatura con il linguaggio adeguato e l'*intent* espresso in linguaggio naturale. *Microsoft*, ad esempio,

ha recentemente rilasciato un *Benchmark dataset* (*CodeXGLUE*) [28] che comprende 14 differenti *dataset* per la risoluzione di 10 differenti task che operano sul codice sorgente [1].

Riprendendo in esame il *task* affrontato, visti i risultati ottenuti, si potrebbe esplorare l'uso del *side-tuning* usando altri tipi di architetture base. Inoltre, data la validità degli *embedding* ottenuti, si potrebbero considerare altri modelli per la loro costruzione. Ad esempio, un gruppo di ricercatori presso *Microsoft*, attiva negli ultimi anni sulle tematiche relative all'analisi e gestione del codice sorgente, hanno introdotto di recente *CodeBert* [13]. Si tratta di un adattamento di *Bert* per la costruzione di *embedding* bi-modali, basati sul linguaggio naturale e su codice sorgente. Si potrebbe allora esplorare la costruzione di *embedding* con sistemi più sofisticati quali *CodeBert* al fine di avere rappresentazioni vettoriali più accurate di *snippet* di codice sorgente.

Bibliografia

[1] Allamanis, M., Barr, E. T., Devanbu, P. T., and Sutton, C. A survey of machine learning for big code and naturalness. *CoRR abs/1709.06182* (2017).

[2] Alrashedy, K., Dharmaretnam, D., Germán, D. M., Srinivasan, V., and Gulliver, T. A. SCC: automatic classification of code snippets. *CoRR abs/1809.07945* (2018).

[3] Alrashedy, K., Dharmaretnam, D., Germán, D. M., Srinivasan, V., and Gulliver, T. A. SCC++: predicting the programming language of questions and snippets of stack overflow. *J. Syst. Softw. 162* (2020).

[4] Audebert, N., Herold, C., Slimani, K., and Vidal, C. Multimodal deep networks for text and image-based document classification. *CoRR abs/1907.06370* (2019).

[5] Baltes, S., Dumani, L., Treude, C., and Diehl, S. Sotorrent: Reconstructing and analyzing the evolution of stack overflow posts. *CoRR abs/1803.07311* (2018).

[6] Baltes, S., and Treude, C. Code duplication on stack overflow. In *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020* (2020), G. Rothermel and D. Bae, Eds., ACM, pp. 13–16.

[7] Baquero, J. F., Camargo, J. E., Restrepo-Calle, F., Aponte, J. H., and González, F. A. Predicting the programming language: Extracting knowledge from stack overflow posts. In *Communications in Computer and Information Science*. Springer International Publishing, 2017, pp. 199–210.

[8] Barchi, F., Parisi, E., Urgese, G., Ficarra, E., and Acquaviva, A. Exploration of convolutional neural network models for source code classification. *Eng. Appl. Artif. Intell. 97* (2021), 104075.

[9] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. A neural probabilistic language model. *J. Mach. Learn. Res. 3* (2003), 1137–1155.

[10] Bonifro, F. D., Gabbrielli, M., and Zacchiroli, S. Content-based textual file type detection at scale. *CoRR abs/2101.08508* (2021).

[11] Chirkova, N., and Troshin, S. A simple approach for handling out-of-vocabulary identifiers in deep learning for source code. *CoRR abs/2010.12663* (2020).

[12] Deng, J., Dong, W., Socher, R., Li, L., Li, K., and Li, F. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA* (2009), IEEE Computer Society, pp. 248–255.

[13] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages. *CoRR abs/2002.08155* (2020).

[14] Ganesan, K., and Foti, R. C# or java? typescript or javascript? machine learning based classification of programming languages, 2019. Data di consultazione: 11-01-2021.

[15] Gilda, S. Source code classification using neural networks. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)* (07 2017), pp. 1–6.

[16] Gülçehre, Ç., Ahn, S., Nallapati, R., Zhou, B., and Bengio, Y. Pointing the unknown words. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers* (2016), The Association for Computer Linguistics.

[17] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[18] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR abs/1207.0580* (2012).

[19] Hinton, G. E., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *CoRR abs/1503.02531* (2015).

[20] Hong, J., Mizuno, O., and Kondo, M. An empirical study of source code detection using image classification. In *10th International Workshop on Empirical Software Engineering in Practice, IWESEP 2019, Tokyo, Japan, December 13-14, 2019* (2019), IEEE, pp. 1–6.

[21] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR abs/1704.04861* (2017).

[22] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR abs/1609.04836* (2016).

[23] Khasnabish, J. N., Sodhi, M., Deshmukh, J., and Srinivasaraghavan, G. Detecting programming language from source code using

bayesian learning techniques. In *Machine Learning and Data Mining in Pattern Recognition - 10th International Conference, MLDM 2014, St. Petersburg, Russia, July 21-24, 2014. Proceedings* (2014), P. Perner, Ed., vol. 8556 of *Lecture Notes in Computer Science*, Springer, pp. 513–522.

[24] Kim, Y. Convolutional neural networks for sentence classification. *CoRR abs/1408.5882* (2014).

[25] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[26] Kiyak, E. O., Cengiz, A. B., Birant, K. U., and Birant, D. Comparison of image-based and text-based source code classification using deep learning. *SN Computer Science 1*, 5 (Aug. 2020).

[27] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States* (2012), P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., pp. 1106–1114.

[28] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR abs/2102.04664* (2021).

[29] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. In *1st International*

Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings (2013), Y. Bengio and Y. LeCun, Eds.

[30] Mikolov, T., Yih, W., and Zweig, G. Linguistic regularities in continuous space word representations. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA* (2013), L. Vanderwende, H. D. III, and K. Kirchhoff, Eds., The Association for Computational Linguistics, pp. 746–751.

[31] Ohashi, H., and Watanobe, Y. Convolutional neural network for classification of source codes. In *13th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC 2019, Singapore, Singapore, October 1-4, 2019* (2019), IEEE, pp. 194–200.

[32] Ott, J., Atchison, A., Harnack, P., Best, N., Anderson, H., Firmani, C., and Linstead, E. Learning lexical features of programming languages from imagery using convolutional neural networks. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018* (2018), F. Khomh, C. K. Roy, and J. Siegmund, Eds., ACM, pp. 336–339.

[33] Ponzanelli, L., Mocci, A., Bacchelli, A., and Lanza, M. Understanding and classifying the quality of technical forum questions. In *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014* (2014), IEEE, pp. 343–352.

[34] Razavian, A. S., Azizpour, H., Sullivan, J., and Carlsson, S. CNN features off-the-shelf: an astounding baseline for recognition. *CoRR abs/1403.6382* (2014).

[35] Reyes, J., Ramirez, D., and Paciello, J. Automatic classification of source code archives by programming language: A deep learning

approach. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)* (Dec. 2016), IEEE.

[36] Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018* (2018), IEEE Computer Society, pp. 4510–4520.

[37] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[38] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[39] Smith, L. N., and Topin, N. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR abs/1708.07120* (2017).

[40] Tóth, L., Nagy, B., Janthó, D., Vidács, L., and Gyimóthy, T. Towards an accurate prediction of the question quality on stack overflow using a deep-learning-based NLP approach. In *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019* (2019), M. van Sinderen and L. A. Maciaszek, Eds., SciTePress, pp. 631–639.

[41] Yang, D., Hussain, A., and Lopes, C. V. From query to usable code: An analysis of stack overflow code snippets. *CoRR abs/1605.04464* (2016).

[42] Yao, Z., Weld, D. S., Chen, W., and Sun, H. Staqc: A systematically mined question-code dataset from stack overflow. *CoRR abs/1803.09371* (2018).

[43] Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018* (2018), A. Zaidman, Y. Kamei, and E. Hill, Eds., ACM, pp. 476–486.

[44] Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada* (2014), Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., pp. 3320–3328.

[45] Zevin, S., and Holzem, C. Machine learning based source code classification using syntax oriented features. *CoRR abs/1703.07638* (2017).

[46] Zhang, J. O., Sax, A., Zamir, A. R., Guibas, L. J., and Malik, J. Side-tuning: A baseline for network adaptation via additive side networks. In *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part III* (2020), A. Vedaldi, H. Bischof, T. Brox, and J. Frahm, Eds., vol. 12348 of *Lecture Notes in Computer Science*, Springer, pp. 698–714.

[47] Zingaro, S. P., Lisanti, G., and Gabbrielli, M. Multimodal side-tuning for document classification. In *25th International Conference on Pattern Recognition (ICPR), Jan 10-15, 2021, Milan, Italy* (2020), pp. 5206–5213.