Relazione Progetto Code Completion

Lorenzo Bartolini 7172579
Irene Scarpanti 7172580
Claudio Soricaro 7172616

Introduzione

Il progetto tratta la creazione e la modellazione di un sistema atto al completamento del codice sorgente utilizzando due diverse architetture di reti neurali: **LSTM (Long Short-Term Memory)** e **Transformer**. L'obiettivo principale è prevedere il token successivo in una sequenza di codice, basandosi su un dataset di codice sorgente tokenizzato.

Ci soffermeremo principalmente sul modello LSTM poiché il modello Transformer non ha dato risultati soddisfacenti.

Un LSTM (Long Short-Term Memory) è un tipo speciale di rete neurale ricorrente (RNN) progettato per gestire sequenze di dati ed evitare il problema della scomparsa del gradiente nelle lunghe dipendenze temporali.

L'LSTM ha una memoria a lungo termine e usa delle "gates" (porte) per controllare il flusso delle informazioni.

Implementazione

L'implementazione del modello è suddivisa in:

- Creazione del dataset;
- Tokenizzazione del dataset;
- Embedding;
- Training del modello;
- Previsione.

Il dataset utilizzato è CodeSearchNet presente su Hugging Face (viene utilizzato solo codice Python). La scelta di utilizzare questo dataset e non un altro è dovuta al fatto che altri dataset sono implementati con gli Abstract Syntax Tree, rendendoli più difficili da manipolare.

Per quanto riguarda il Tokenizer e l'Embedder ne utilizziamo due pre-trained, per semplificare il lavoro di addestramento, però dell'Embedder si effettua un fine-tuning: prendiamo un modello pre-trained e lo alleniamo nuovamente sul nostro dataset specifico. Così facendo ne miglioriamo l'accuratezza.

Per la creazione del dataset viene utilizzata la funzione **create_dataset** (alla quale vengono passate solo le prime 4000 righe di codice di CodeSearchNet) che data la stringa, effettua la tokenizzazione dei dati in sequenze di lunghezza ***input_len*** e crea l'etichetta considerando come target l'elemento successivo di ogni sequenza. Per la creazione del dataset si utilizza la tecnica sliding window: dato il codice si selezionano i primi 30 token, che diventano l'input, mentre il 31esimo diventa il target. Successivamente si scorre di uno ripetendo l'operazione e così via, per arrivare ad avere tante coppie 30-1 per allenare il modello.

Abbiamo scelto come sequenza di input 30 token: abbiamo visto che, in maniera sperimentale, è un numero sufficiente per la generazione del token successivo

Per la fase di pre-training, avendo utilizzato un embedder già addestrato, non abbiamo dovuto trovare la dimensione del dizionario a mano ma abbiamo utilizzato il dizionario già fornito.

Sono stati definiti due modelli: modello Transformer e modello LSTM.

Il modello Transformer, che prevede un'architettura formata da un modulo Transformer di PyTorch (configurato con 3 encoder e 3 decoder), un embedding layer e un layer di previsione lineare, non è risultato molto performante, motivo per il quale non ci soffermeremo ulteriormente.

Il modello LSTM, invece, prevede i seguenti strati: un embedding Layer per convertire i token in rappresentazioni vettoriali, un encoder lineare per proiettare l'embedding su uno spazio di dimensione encoding_dim così da ridurne la dimensione e concentrare, un modulo LSTM bidirezionale con più livelli e due strati lineari con dropout per la previsione del token successivo.

Tra un layer e l'altro usiamo un strato di Dropout, per migliorare l'accuratezza. I dati passano per due layer lineari per migliorare l'apprendimento, di cui l'ultimo ha la dimensione del dizionario.

Abbiamo deciso di implementare un LSTM bidirezionale: una variante del classico LSTM che processa i dati in entrambe le direzioni (avanti e indietro) per catturare meglio il contesto.

Il modello è stato addestrato dividendo i dati in train (80%) e validazione (20%), usando la CrossEntropyLoss essendo un problema di classificazione e come optimizer è stato utilizzato Adam con lr=0.002. Tale learning rate è stato scelto in quanto su più paper è stato visto che rende il modello più performante.

Il training avviene in un ciclo for per un certo numero di epoche e alla fine di ogni epoca, in base al val_loader, si effettua un ciclo di validazione. Di conseguenza ad ogni ciclo si ottiene la loss e accuracy sia per la parte di training che per la parte di validation.

Infine si ha la funzione per predire i 5 più probabili prossimi token.

La funzione prende una stringa di codice in input e la trasforma in una sequenza di token numerici usando il tokenizer. Passa l'input al modello per ottenere la previsione, una volta ottenuta ne estrae i 5 token con le probabilità più alte dalla previsione del modello e conclude decodificando i token numerici previsti in stringhe di codice usando il tokenizer e restituendo una lista contenente le 5 previsioni.

Risultati e Considerazioni

Per quanto l'architettura del modello LSTM sia relativamente semplice, raggiunge delle performance piuttosto soddisfacenti, con un'accuratezza di circa 80% (ricordando che è una top k accuracy, ovvero che la predizione si considera corretta se il token da predire si trova nei k più probabili predetti) e una Train Loss di 1.63, dimostrando la robustezza dei paradigmi di tale modello.