Fine-tuning di BERT per NER (NERMuD 2023)

Chiara Mancuso, Virginia Ranciaro

a.a. 2024-2025

1 Introduzione

Questo progetto si propone di effettuare il fine-tuning di un modello transformer per task di **Named Entity Recognition** (NER) su documenti di lingua italiana.

In particolare, il modello è stato addestrato sul benchmark **NERMuD** (***Named-Entities Recognition on Multi-Domain Documents***), presentato nella campagna EVALITA 2023[1]. Questo task consiste nell'estrazione e classificazione delle entità nominate (nomi propri di persone, luoghi e organizzazioni) da documenti di varia tipologia testuale.

Tra i due sub-task proposti in NERMuD 2023, si è qui scelto di replicare quello di classificazione dominio-indipendente (*domain-agnostic classification*): individuare le entità nominate e classificarle in tre categorie (persone, luoghi e organizzazioni), lavorando su tre diverse tipologie di documenti (testi giornalistici, narrativa e discorsi politici) con un modello unico.

Per questo progetto si è scelto di implementare `BertForTokenClassification`, una versione di `BERT` che eredita dal modello base pre-addestrato sulla lingua italiana (`bert-base-italian-cased`).

Per valutare performance del modello sul test set si è scelto di utilizzare la metrica F_1 Macro, come visto in NERMuD 2023, così da poter confrontare i risultati ottenuti con quelli descritti nel paper. Il punteggio F_1 , essendo la media armonica degli score di richiamo (R) e precisione (P), fornisce una misura più equilibrata delle prestazioni del modello, dal momento che il dataset utilizzato si presenta sbilanciato rispetto al numero di token per classe.

2 Dataset

Il dataset utilizzato per il fine-tuning è un subset del **Kessler Italian Named-entities Dataset** (KIND), pubblicamente accessibile su GitHub[2].

[1] https://nermud.fbk.eu/

[2] https://github.com/dhfbk/KIND

Il dataset originale comprende oltre un milione di token, annotati per tre classi di entità: persone (**PER**), luoghi (**LOC**) e organizzazioni (**ORG**). La maggior parte del dataset contiene annotazioni manuali *gold* su documenti di tre tipologie: testi giornalistici, narrativa e discorsi politici.

Dal dataset è stato estratto un subset annotato appositamente per il task NERMuD 2023, con uno splitting tra training, development e test set (come in fig. 1).

In particolare, i testi utilizzati per il task NERMuD provengono da:

- Wikinews (**WN**), come fonte di testi giornalistici degli ultimi decenni;
- alcuni libri di narrativa italiana (**FIC**) di pubblico dominio, liberamente accessibili;
- scritti e discorsi del politico italiano Alcide De Gasperi (**ADG**), una raccolta di testi comprendente le opere e i discorsi di Alcide De Gasperi.

Dataset	Tokens	Train				Dev				Test			
		Total	LOC	PER	ORG	Total	LOC	PER	ORG	Total	LOC	PER	ORG
WikiNews	364,816	249,077	6,862	8,928	7,593	59,220	1,711	1,802	1,823	56,519	1,310	2,322	1,992
Fiction	219,638	170,942	733	3,439	182	21,506	463	636	284	27,190	37	443	1
De Gasperi	164,537	123,504	1,046	1,129	2,396	27,128	274	253	533	13,905	107	226	326
Total	748,991	543,523	8,641	13,496	10,171	107,854	2,448	2,691	2,640	97,614	1,454	2,991	2,319

Figura 1: Composizione del dataset

3 Script e panoramica del codice

Il codice del progetto è organizzato in tre file Python, che consentono di gestire la pipeline in maniera modulare:

1. `train.py`: file principale che gestisce l'intero flusso di addestramento, inclusi il caricamento dati, il training, l'evaluation e il salvataggio dei modelli.
2. `funzioni.py`: contiene funzioni di supporto per la lettura dei dati, la concatenazione dei domini, la tokenizzazione e l'allineamento delle label, il training di una singola epoca, la valutazione del modello e il calcolo delle metriche di valutazione.
3. `classe.py`: definisce la classe `NERDataset`, che eredita da `Dataset`, l'interfaccia standard di PyTorch che rende gli encoding tokenizzati compatibili con DataLoader, consentendo di caricare i dati in batch e passarli direttamente al modello.

3.1 Funzioni

Le funzioni implementate in `funzioni.py` coprono tutte le fasi del fine-tuning:

- `read_conll`: questa funzione legge file annotati in formato CoNLL o TSV. Prende in input il percorso del file e restituisce due liste principali: `sents`, contenente le frasi come liste di token, e `labels`, contenente le etichette corrispondenti. Il file viene letto riga per riga, e una riga vuota viene interpretata come separatore di frase. Per ogni riga non vuota, la funzione estrae il token e la relativa etichetta, li aggiunge a liste temporanee e, al termine di ciascuna frase, queste vengono aggiunte alle liste finali.

- `concat_domains`: permette di combinare i dati provenienti da più domini (WN, FIC e ADG) per creare un dataset unico. La funzione prende in input il tipo di split desiderato (`train`, `dev` o `test`), la lista dei domini e la cartella in cui si trovano i file. Per ciascun dominio, richiama `read_conll` e concatena le frasi e le etichette, ottenendo un dataset multi-dominio.

- `tokenize_and_align`: questa funzione si occupa di tokenizzare le frasi e di allineare le etichette ai token prodotti dal *tokenizer* di BERT. BERT può suddividere alcune parole in più sottotoken; solo il primo sottotoken mantiene l'etichetta originale, mentre gli altri ricevono il valore -100 per essere ignorati durante il calcolo della *loss*. Questa procedura fa in modo che l'allineamento tra token e etichette sia corretto e che il modello venga addestrato senza errori sulle etichette.

- La funzione `compute_metrics` ha lo scopo di calcolare un insieme completo di metriche per valutare le prestazioni del modello sul task di Named Entity Recognition (NER), seguendo lo stile della libreria `seqeval`. Può operare sia con etichette semplici (`PER`, `LOC`, `ORG`, `O`) sia con etichette in formato BIO.

  Si descrive di seguito il flusso operativo della funzione:

  1. Trasformazione degli ID in label testuali: le predizioni e le etichette originali sono rappresentate come identificatori numerici. La funzione li converte nei corrispondenti nomi testuali tramite il dizionario `id2label`, ignorando i token marcati con `-100`.
  2. Calcolo delle metriche globali (micro): se viene fornito un oggetto `metric` come argomento della funzione (della libreria `seqeval`), vengono calcolate precisione, richiamo e punteggio F_1 complessivi su tutte le etichette, ignorando eventuali token di padding (cioè token aggiunti per uniformare la lunghezza delle sequenze nei batch).
  3. Calcolo delle metriche per tipo di entità: per ciascuna delle tre classi principali (PER, LOC, ORG) la funzione isola le etichette corrispondenti, trasformando tutte le altre in O. Vengono quindi calcolate precisione, richiamo, punteggio F_1 e supporto (numero di occorrenze) per ciascuna classe. Così è possibile analizzare le performance indipendentemente dallo schema BIO.

4. Calcolo di metriche macro e ponderate: la funzione calcola la media aritmetica dei punteggi per ogni classe (*macro*) e la media pesata in base al numero di occorrenze di ognuna (*weighted*), ottenendo così valori aggregati che riflettono la distribuzione reale delle entità nel dataset.
5. Calcolo del punteggio F_1 per singola classe: se non viene fornita esplicitamente la lista delle classi, questa viene costruita automaticamente estraendo tutte le etichette presenti nei dati, escluse quelle O. Successivamente viene calcolato il punteggio F_1 per ogni classe.
6. Composizione del dizionario dei risultati: tutte le metriche calcolate vengono raccolte in un dizionario strutturato, comprendente: metriche micro globali (`precision_micro`, `recall_micro`, `f1_micro`); metriche macro sulle tre classi principali (`precision_macro`, `recall_macro`, `f1_macro_by_type`); metriche ponderate (`precision_weighted`, `recall_weighted`, `f1_weighted`); punteggio F_1; totale macro su tutte le classi BIO (`f1_macro_total`); punteggio F_1 per singola classe (`f1_per_class`); precisione, richiamo, F_1 e supporto per le tre entità principali (PER, LOC, ORG).
7. Inclusione della *loss* di validazione: se fornita, la funzione aggiunge al dizionario anche il valore medio della *loss* sul dataset di validazione, permettendo di correlare le metriche di accuratezza con l'errore medio.

La funzione `compute_metrics` viene utilizzata sia durante la validazione epoca per epoca sia durante il test finale; serve inoltre per implementare l'*early stopping* basato su F_1 e per salvare metriche aggregate sulle diverse run.

- `to_device`: questa funzione ha il compito di trasferire un intero batch di dati sul dispositivo specificato (CPU o GPU). Prende in input un dizionario `batch`, in cui le chiavi rappresentano i tipi di tensori (`input_ids`, `attention_mask`, `labels`, ecc.), e un oggetto `device` che indica il dispositivo target. La funzione itera su tutte le coppie chiave-valore del dizionario e applica il metodo `.to(device, non_blocking=True)` a ciascun tensore, restituendo un nuovo dizionario con i tensori disposti correttamente. In questo modo, tutti i dati di input vengono resi immediatamente compatibili con il dispositivo su cui viene eseguito il modello, riducendo i colli di bottiglia nel trasferimento CPU-GPU.

- `train_one_epoch`: questa funzione gestisce l'addestramento del modello per una singola epoca. Prende in input il modello, il `DataLoader`, l'ottimizzatore, lo scheduler, il `GradScaler` per *mixed precision*, il fattore di accumulo dei gradienti (`grad_accum`), il dispositivo (CPU/GPU) e i vari parametri di configurazione (numero di epoca, run, `use_amp=True`).

  La funzione esegue i seguenti passaggi:

1. Il modello viene posto in modalità di training e i gradienti inizializzati a zero. Per ogni batch, i dati vengono trasferiti sul dispositivo tramite `to_device`.
2. Con `torch.amp.autocast` si abilita la *Automatic Mixed Precision (AMP)*. Con AMP, PyTorch decide automaticamente quali operazioni effettuare in FP16 e quali in FP32: operazioni "sicure" (come le moltiplicazioni tra matrici) vengono svolte in FP16, rendendole molto più veloci; operazioni "sensibili" come il calcolo della *loss* invece sono eseguite in FP32, per mantenere stabilità numerica. Durante il forward pass, il modello elabora il batch e calcola la *loss*, scalata in base al fattore di accumulo dei gradienti.
3. La retropropagazione avviene tramite il `GradScaler`, che previene instabilità numeriche dovute alla precisione ridotta: esso scala dinamicamente la *loss* prima del `backward()`, rendendo i gradienti rappresentabili in FP16, e successivamente riporta alle scale originali i gradienti prima di aggiornare i pesi, garantendo stabilità durante l'ottimizzazione.
4. Il parametro `grad_accum` indica il numero di batch consecutivi sui quali accumulare i gradienti prima di aggiornare i pesi del modello. Anziché aggiornare i pesi ad ogni batch, i gradienti vengono calcolati e sommati per n batch consecutivi. Solo al termine di questi batch (o all'ultimo batch dell'epoca) viene effettuato l'aggiornamento dei pesi tramite l'ottimizzatore:
   - i gradienti vengono prima riscalati ai valori reali e normalizzati con `clip_grad_norm_` per evitare esplosioni di gradiente;
   - si eseguono `scaler.step(optimizer)` e `scaler.update()` per aggiornare i pesi in modo sicuro;
   - si azzerano i gradienti e si aggiorna lo scheduler per modificare il learning rate.
5. Infine, la barra di avanzamento (`tqdm`) mostra in tempo reale la *loss* media per batch, permettendo un monitoraggio immediato dell'andamento dell'epoca.

- La funzione `evaluate_model` calcola le prestazioni del modello sul test set, restituendo sia la *loss* media sia un insieme di metriche di valutazione.

  1. Il modello viene posto in modalità di evaluation con il comando `model.eval()`, che disattiva componenti impiegate durante l'addestramento, come il *dropout*.
  2. L'inferenza viene quindi condotta all'interno del contesto `torch.no_grad()`, che disabilita il calcolo dei gradienti e consente di ridurre il consumo di memoria. Ogni batch viene caricato sul dispositivo (CPU o GPU) e passato al modello, il quale restituisce i *logits*, ovvero le distribuzioni di probabilità grezze sulle classi, insieme

al valore di *loss*. Quest'ultima viene cumulata per consentire, al termine dell'iterazione sull'intero dataset, il calcolo della media.

3. Allo stesso tempo, dalle predizioni si ricava l'etichetta prevista per ciascun token mediante l'operazione di `argmax` sui logits. Le predizioni (`all_preds, all_labels`) vengono quindi convertite in array e salvate insieme alle etichette di riferimento, costituendo la base per il calcolo delle metriche.
4. Una volta concluso il processo di inferenza, la *loss* media viene calcolata dividendo la somma complessiva delle *loss* per il numero di batch elaborati.
5. Successivamente, le predizioni e le etichette vengono passate alla funzione `compute_metrics`, la quale consente di ottenere i valori di *precision*, *recall*, *F1-score*, oltre alla *validation loss*.
6. Infine, `evaluate_model` restituisce un dizionario contenente le metriche calcolate insieme al valore medio della *loss*, fornendo un quadro complessivo e dettagliato dell'accuratezza e della robustezza del modello nel processo di generalizzazione sui dati di validazione.

3.2 Classe

Il file `classe.py` implementa una classe `NERDataset(torch.utils.data.Dataset)`, che trasforma i dati tokenizzati e allineati (ottenuti da `tokenize_and_align`) in una forma compatibile con DataLoader, che gestisce batching, shuffle, e caricamento parallelo. Il costruttore prende in input un dizionario di encodings e lo memorizza come attributo della classe. I metodi principali sono:

- `__len__`: restituisce il numero di frasi nel dataset;
- `__getitem__`: dato un indice, restituisce un dizionario contenente i tensori corrispondenti a ciascun campo presente negli encodings (ad esempio `input_ids`, `attention_mask`, `labels`), trasformando gli array o le liste originali in tensori PyTorch. In questo modo ogni batch restituito dal `DataLoader` è già pronto per essere trasferito sul dispositivo di training.

3.3 Train

Lo script principale `train.py` gestisce il processo di fine-tuning, integrando le funzioni e la classe definite nei moduli esterni, ed è organizzato come segue:

1. In primo luogo, si definisce la configurazione di default dei parametri con `parse_args()`, che consente anche di modificarli da linea di comando al momento dell'esecuzione. I parametri qui definiti sono: la cartella in cui si trovano i dati, il modello pre-addestrato, il numero di epoche, la dimensione dei batch, il learning rate, il numero delle run, le strategie di accumulazione dei gradienti, il criterio di early stopping e la cartella di output.

2. Successivamente, vengono configurati i dispositivi di calcolo (CPU o GPU) e le impostazioni per l'uso della (*Automatic Mixed Precision*, AMP). Il `GradScaler` consente l'allenamento in AMP, scalando i gradienti per preservare la precisione numerica durante l'uso di floating point a 16 bit.

3. Vengono caricati e concatenati i dati multi-dominio (train, dev, test) tramite la funzione `concat_domains`, mentre la tokenizzazione e l'allineamento delle etichette sono gestite dalla funzione `tokenize_and_align`. I dataset tokenizzati sono quindi incapsulati nella classe `NERDataset`, che viene poi passata ai DataLoader di PyTorch.

4. I DataLoader sono configurati con `DataCollatorForTokenClassification`, supporto multi-worker (`num_workers = min(2, os.cpu_count())`) e

   `pin_memory` (`pin_memory=True` fa sì che i dati siano collocati nella memoria speciale della CPU, pronti per essere trasferiti nella GPU; ciò rende il trasferimento dei batch più veloce durante il training).

5. Inizio del ciclo della run: all'inizio di ogni run viene impostato un seed pseudo-casuale (`42 * run_id`), al fine di garantire la riproducibilità e variare leggermente l'inizializzazione dei pesi tra le diverse esecuzioni.

6. Per ciascuna run, il modello `BertForTokenClassification` viene istanziato mediante l'API `AutoModelForTokenClassification` (che seleziona automaticamente `BertForTokenClassification`) con il numero corretto di etichette e i dizionari `label2id` e `id2label`.

7. L'ottimizzatore `AdamW` aggiorna i pesi del modello e applica una regolarizzazione con `weight_decay`, che penalizza pesi troppo grandi per ridurre l'overfitting, mentre lo `scheduler` lineare aumenta gradualmente il learning rate all'inizio dell'addestramento e poi lo riduce progressivamente, favorendo una convergenza più stabile. L'ottimizzatore `AdamW` è stato scelto in quanto, tra le diverse alternative testate (come SGD o Adam standard), ha mostrato le migliori performance sul task.

8. Il training avviene iterando sulle epoche con la funzione `train_one_epoch`, mostrando alla fine di ogni epoca una valutazione intermedia sul validation set calcolate dalla funzione `evaluate_model`. Le metriche monitorate includono `validation loss`, F_1 micro e macro globali e infine, la F_1 delle tre classi (PER, LOC, ORG). Lo script implementa una strategia di *early stopping* basata sull'F_1 macro per classe: se per un numero prefissato di epoche consecutive (`patience = 2`) l'F1 non migliora, il ciclo di training si interrompe automaticamente.

9. A questo punto viene salvato su disco il modello con le migliori prestazioni per ogni run, insieme ai dizionari `label2id` e `id2label`, seed e informazioni sulla run.

10. Infine, ciascun modello ottimizzato viene valutato sul test set e i risultati delle diverse run vengono aggregati in un file CSV, che mostra anche la media e la deviazione standard su 3 run delle metriche principali.

3.4 Esecuzione della pipeline

Per avviare il fine-tuning del modello BERT su NER è sufficiente eseguire lo script principale `train.py` dalla linea di comando, specificando i parametri desiderati.

3.4.1 Scelta dei parametri di addestramento

Per questo progetto, sono state effettuate alcune scelte di iperparametri mirate a garantire sia performance elevate che replicabilità dei risultati.

Learning rate: Sono stati testati diversi valori di learning rate (5e−5, 3e−5 e 2e−5). Con `lr = 5e−5` il modello mostrava un rapido overfitting già tra la seconda e la terza epoca, con aumento della *validation loss* e oscillazioni significative delle metriche F_1. Impostando invece il learning rate a 3e−5, il modello continuava a migliorare le prestazioni per alcune classi difficili (in particolare ORG), ma mostrava una maggiore variabilità tra run indipendenti, rendendo meno stabile la replicabilità dei risultati.

Il valore di `lr` individuato come ottimale è quindi 2e−5: con questo learning rate le metriche principali risultano stabili tra le tre run, la *validation loss* rimane costante (~ 0.04) e il modello evita fenomeni di overfitting.

Batch size: La batch size è stata impostata a 16 (`batch_size = 16`): questa dimensione consente un buon compromesso tra stabilità del gradiente e utilizzo della memoria GPU disponibile. Batch più grandi non erano necessari data la dimensione relativamente contenuta del dataset, mentre batch più piccoli avrebbero potuto introdurre maggiore rumore durante l'ottimizzazione.

Numero di epoche: Sono state impostate 3 epoche di addestramento (`epochs = 3`): un numero limitato di epoche consente di ottenere un addestramento completo senza overfitting.

Numero di run: Per garantire la robustezza dei risultati e ridurre l'impatto della casualità nell'inizializzazione dei pesi, ogni esperimento è stato eseguito su 3 run indipendenti e i valori finali delle metriche sono stati mediati.

4 Estrazione metriche e salvataggio dei dati

Dopo l'addestramento e la valutazione del modello sui dataset di validazione e test, le metriche vengono elaborate e salvate in un file CSV, seguendo questi passaggi:

1. **Calcolo delle metriche**: per ogni run, le predizioni del modello vengono confrontate con le etichette di riferimento utilizzando la funzione `compute_metrics` (vedi 3.1).

2. **Aggregazione dei risultati**: le metriche delle diverse run vengono raccolte in un dizionario e successivamente aggregate calcolando la media e la deviazione standard. Ciò permette di valutare la stabilità e l'attendibilità dei risultati tra run diverse.

3. **Esportazione in CSV**: tutte le metriche di valutazione vengono salvate all'interno del file `results.csv` nella cartella `runs/bert_manual_opt`. Per ogni run (colonne) vengono riportati i punteggi di tutte le metriche calcolate (righe) insieme alla *validation loss* e al *support* (che corrisponde al numero di entità per classe). Le ultime due colonne contengono media e deviazione standard dei valori, calcolate su tutte le run effettuate.

   Nella Tabella 1 è riportato un elenco delle metriche estratte.

Metriche
val_loss
precision_micro
recall_micro
f1_micro
precision_macro
recall_macro
f1_macro_by_type
precision_weighted
recall_weighted
f1_weighted
support_PER
support_LOC
support_ORG
precision_PER
recall_PER
f1_PER
precision_LOC
recall_LOC
f1_LOC
precision_ORG
recall_ORG
f1_ORG

Tabella 1: Elenco delle metriche utilizzate per la valutazione.

5 Risultati

Le performance del modello `BertForTokenClassification` utilizzato in questo progetto sono riassunte nella Tabella 2. La deviazione standard è molto contenuta, segno di un comportamento stabile del modello.

	PER	LOC	ORG	Micro	Macro
P	92.26 ± 0.19	84.26 ± 0.85	81.34 ± 0.49	86.83 ± 0.26	85.95 ± 0.32
R	93.18 ± 0.37	87.30 ± 0.06	80.41 ± 0.21	87.59 ± 0.11	86.96 ± 0.07
$\boldsymbol{F_1}$	92.71 ± 0.09	85.75 ± 0.44	80.87 ± 0.30	87.20 ± 0.17	86.44 ± 0.19

Tabella 2: Risultati espressi come media e deviazione standard su 3 run.

La *validation loss* (`val_loss`) si mantiene costantemente su valori molto bassi (∼0.04), a conferma di un buon adattamento ai dati.
Le prestazioni per entità mostrano un alto tasso di riconoscimento delle **persone (PER)**, con un F_1 medio pari a 92.7%, buone performance sulle **località (LOC)** (85.8%) e valori più bassi per le **organizzazioni (ORG)** (80.9%). Quest'ultimo risultato è coerente con una difficoltà generale dei modelli nel riconoscere le entità organizzative, osservata nel paper NERMuD 2023.

La Tabella 3 confronta i risultati ottenuti in questo progetto (colonna BERT4MuD) con le performance di BERT riportate nel paper NERMuD 2023. La colonna Δ indica la differenza rispetto alla baseline.

Categoria	**BERT4MuD** (F_1)	**Baseline NERMuD 2023** ($\boldsymbol{F_1}$)	Δ
PER	92.71	95.0	-2.29
LOC	85.75	87.0	-1.25
ORG	80.87	86.0	-5.13
Micro	87.20	90.0	-2.80
Macro	86.44	89.0	-2.56

Tabella 3: Confronto tra i risultati del fine-tuning del progetto e la baseline ufficiale NERMuD 2023.