Analisi del Fine Tuning del Modello BERT per l'Analisi di Risposte LLM, Versione 1.8

10 aprile 2025

Sommario

Questo documento presenta un'analisi dettagliata del processo di fine tuning effettuato su un modello BERT (bert-base-uncased) per l'analisi di risposte generate da modelli LLM. Vengono descritti il dataset utilizzato, la metodologia di training, l'andamento della loss, ed i risultati quantitativi ottenuti in fase di valutazione. In coda al documento è incluso il codice completo di training.

1 Introduzione

Il fine tuning di modelli pre-addestrati come BERT permette di adattare la rappresentazione linguistica a task specifici, come in questo caso l'analisi delle risposte generate da modelli LLM. L'approccio adottato prevede l'uso di un dataset suddiviso in set di train, validazione e test. Il modello è stato inizializzato con i pesi di `bert-base-uncased` e successivamente sottoposto a ulteriori allenamenti sul dataset target, aggiornando anche il classificatore (con parametri `classifier.weight` e `classifier.bias`) che non erano presenti nel checkpoint di partenza.

2 Dataset e Pre-Processing

Il dataset utilizzato è organizzato in un `DatasetDict` con le seguenti caratteristiche:

- **Training:** 12.000 esempi (poi suddivisi in 9.600 per l'addestramento e 2.400 per la validazione)
- **Test:** 1.773 esempi

Le colonne principali del dataset sono:

- `response`: Contiene le risposte generate dai modelli LLM.
- `label`: Indicatore della classe (con etichette uniche: `"0"` e `"1"`).

Il pre-processing ha incluso la mappatura delle etichette sui rispettivi ID, la tokenizzazione dei testi con un tokenizer pre-addestrato e la rimozione di colonne non utili all'addestramento.

3 Dettagli del Training e Risultati

3.1 Impostazioni e Log del Training

Il training è avvenuto per 3 epoche con i seguenti parametri chiave:

- **Learning rate:** 2×10^{-5}
- **Batch size:** 16 per dispositivo (sia per train che per eval)
- **Warmup steps:** 500
- **Weight decay:** 0.01
- **FP16:** Abilitato per l'ottimizzazione
- **Gradient Accumulation:** 2 steps

Il training log mostra un andamento della **loss** decrescente. Alcuni step rappresentativi sono:

- **Step 100:** Loss ≈ 0.6206
- **Step 300:** Loss ≈ 0.2167
- **Step 700:** Loss ≈ 0.0881
- **Step 900:** Loss ≈ 0.0752

Questi valori indicano una convergenza adeguata del modello nel corso del training, con una progressiva riduzione dell'errore.

3.2 Valutazione sul Test Set

I risultati finali sul test set evidenziano un'alta capacità predittiva del modello:

- **Loss:** 0.501045
- **Accuracy:** 90.02%
- **Precision:** 90.34%
- **Recall:** 90.02%
- **F1-score:** 89.92%

Questi risultati suggeriscono che il modello, dopo il fine tuning, sia in grado di catturare efficacemente le caratteristiche del problema di classificazione. L'accuratezza elevata e i valori bilanciati di precision e recall indicano una buona generalizzazione, sebbene una loss non trascurabile possa ancora essere oggetto di ulteriori ottimizzazioni, per esempio tramite tecniche di regolarizzazione o un ulteriore tuning degli iperparametri.

3.3 Considerazioni Tecniche

- Il messaggio `"Some weights of BertForSequenceClassification were not initialized..."` indica che il classificatore è stato inizializzato casualmente e quindi ha dovuto apprendere da zero la parte finale della rete.
- L'approccio di suddividere il dataset in training, validazione e test consente una stima affidabile della generalizzazione del modello.
- L'uso di metriche multiple (accuracy, precision, recall e F1-score) offre una visione completa delle performance, importante in applicazioni di NLP dove la distribuzione delle classi può essere sbilanciata.

4 Conclusioni

Il fine tuning effettuato sul modello BERT ha prodotto un sistema capace di classificare con una buona accuratezza le risposte generate dai modelli LLM. I risultati quantitativi confermano l'efficacia dell'approccio adottato e la corretta impostazione dei parametri di training. Per futuri miglioramenti si potrebbero esplorare:

- Ulteriore ottimizzazione degli iperparametri;
- Tecniche di data augmentation per incrementare la robustezza del modello;
- Strategie di regolarizzazione per ridurre ulteriormente la loss.

Codice di Training

Di seguito viene riportato il codice utilizzato per il fine tuning del modello:

```
! pip install transformers datasets evaluate torch accelerate -U
!pip install -U transformers

# 'accelerate' è raccomandato per Trainer per ottimizzare l'uso della GPU/TPU
import os
```

```
import numpy as np
import pandas as pd
from datasets import load_dataset
from transformers import AutoTokenizer,
    AutoModelForSequenceClassification, TrainingArguments,
    Trainer, TrainerCallback
import evaluate
from sklearn.metrics import precision_recall_fscore_support

# Disabilitiamo wandb in modalita offline (salva i log
    localmente)
os.environ["WANDB_MODE"] = "offline"

# Definiamo la directory di output dove salvare il modello e il
     tokenizer
output_directory = "my-bert-fine-tuned-model"

# Carichiamo il dataset
data_files = {
    "train": "dataset_completo.json",
    "test": "Test2.json"
}
dataset = load_dataset('json', data_files=data_files)

print("Struttura del dataset:")
print(dataset)
print("\nColonne nel dataset train:")
print(dataset["train"].column_names)
print("\nPrimo esempio nel dataset:")
print(dataset["train"][0])

# Determiniamo le colonne di testo e etichette
first_example = dataset["train"][0]
text_column = None
label_column = None

# Trova la colonna del testo (quella più lunga)
longest_text_len = 0
for col in first_example:
    if isinstance(first_example[col], str) and len(
        first_example[col]) > longest_text_len:
        longest_text_len = len(first_example[col])
        text_column = col

# Trova la colonna delle etichette (cerca 'label', 'class' o '
    category')
for col in first_example:
    if 'label' in col.lower() or 'class' in col.lower() or '
        category' in col.lower():
        label_column = col
        break

if text_column is None:
    raise ValueError("Non è stata trovata una colonna di testo.
```

```
        Specifica manualmente il nome della colonna.")
if label_column is None:
    # Se non troviamo una colonna di etichette evidente, utilizziamo una colonna non di testo
    for col in first_example:
        if col != text_column and not isinstance(first_example[col], str):
            label_column = col
            break

print(f"\nColonna di testo identificata: '{text_column}'")
print(f"Colonna di etichette identificata: '{label_column}'")

# Pre-processiamo le etichette e creiamo il mapping label -> ID
def get_unique_labels(examples):
    labels = examples[label_column]
    unique_labels = set()
    for label in labels:
        if isinstance(label, list):
            for l in label:
                unique_labels.add(l)
        else:
            unique_labels.add(label)
    return list(unique_labels)

unique_labels = get_unique_labels(dataset["train"])
print(f"\nEtichette uniche trovate: {unique_labels}")

label_to_id = {label: i for i, label in enumerate(sorted(unique_labels))}
id_to_label = {i: label for label, i in label_to_id.items()}
print(f"\nMapping etichette -> ID: {label_to_id}")

def preprocess_labels(examples):
    result = dict(examples)
    labels = examples[label_column]
    processed_labels = []
    for label in labels:
        if isinstance(label, list):
            processed_labels.append(label_to_id[label[0]] if label else 0)
        else:
            processed_labels.append(label_to_id[label])
    result[label_column] = processed_labels
    return result

processed_dataset = dataset.map(preprocess_labels, batched=True)

# Scegliamo il modello e il tokenizer (ad es. "bert-base-uncased")
model_checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

```

```
def tokenize_function(examples):
    return tokenizer(
        examples[text_column],
        padding="max_length",
        truncation=True,
        max_length=128
    )

# Tokenizziamo il dataset
tokenized_datasets = processed_dataset.map(tokenize_function, batched=True)
tokenized_datasets = tokenized_datasets.remove_columns([col for col in processed_dataset["train"].column_names if col != label_column])
tokenized_datasets = tokenized_datasets.rename_column(label_column, "labels")
tokenized_datasets.set_format("torch")

# Creiamo i set di training, validazione e test
train_testvalid = tokenized_datasets["train"].train_test_split(test_size=0.2, seed=42)
train_dataset = train_testvalid["train"]
validation_dataset = train_testvalid["test"]
test_dataset = tokenized_datasets["test"]

print(f"\nDimensione del dataset di training completo: {len(train_dataset)} esempi")
print(f"Dimensione del dataset di validazione: {len(validation_dataset)} esempi")
print(f"Dimensione del dataset di test: {len(test_dataset)} esempi")

# Impostiamo la metrica di accuracy
metric = evaluate.load("accuracy")

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    accuracy = metric.compute(predictions=predictions, references=labels)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, predictions, average='weighted')
    return {
        'accuracy': accuracy['accuracy'],
        'precision': precision,
        'recall': recall,
        'f1': f1
    }

num_labels = len(label_to_id)
model = AutoModelForSequenceClassification.from_pretrained(
    model_checkpoint,
    num_labels=num_labels
)
```

```
144
145 # Definiamo un callback personalizzato per salvare e stampare i
        log di training
146 class LogCallback(TrainerCallback):
147     def __init__(self):
148         self.logs = []  # Lista per salvare i log intermedi
149
150     def on_log(self, args, state, control, logs=None, **kwargs)
            :
151         if logs is not None:
152             # Salviamo solo i log rilevanti (es. loss, lr, step
                , epoch)
153             self.logs.append({
154                 'step': state.global_step,
155                 'epoch': state.epoch,
156                 'loss': logs.get('loss', None),
157                 'learning_rate': logs.get('learning_rate', None
                    ),
158                 'eval_loss': logs.get('eval_loss', None)
159             })
160
161     def on_train_end(self, args, state, control, **kwargs):
162         # Alla fine dell'addestramento stampiamo una tabella
                riassuntiva
163         df = pd.DataFrame(self.logs)
164         print("\n===␣Riassunto␣Training␣Log␣===")
165         # Stampiamo log ogni 100 step
166         df_summary = df[df['step'] % 100 == 0]
167         print(df_summary.to_string(index=False))
168
169 # Configuriamo gli argomenti di addestramento
170 training_args = TrainingArguments(
171     output_dir=output_directory,
172     eval_steps=100,                 # Valutazione ogni 100 step
173     save_steps=100,                 # Salvataggio ogni 100 step
174     logging_steps=100,              # Stampa dei log ogni 100
            step
175     learning_rate=2e-5,
176     per_device_train_batch_size=16,
177     per_device_eval_batch_size=16,
178     num_train_epochs=3,
179     weight_decay=0.01,
180     warmup_steps=500,
181     fp16=True,
182     gradient_accumulation_steps=2,
183     save_total_limit=2,
184     report_to="none"
185 )
186
187 # Inizializziamo il trainer passando il parametro aggiornato "
        processing_class" invece di "tokenizer"
188 log_callback = LogCallback()
189 trainer = Trainer(
190     model=model,
```

```
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=validation_dataset,
    processing_class=tokenizer,  # Utilizziamo il nuovo parametro in sostituzione di 'tokenizer'
    compute_metrics=compute_metrics,
    callbacks=[log_callback]
)

print("Inizio addestramento sull'intero dataset...")
trainer.train()
print("Addestramento completato!")

# Valutazione sul test set
print("Valutazione sul test set completo...")
test_results = trainer.evaluate(test_dataset)
print("Risultati test:", test_results)

# Salva il modello e il tokenizer
trainer.save_model(output_directory)
tokenizer.save_pretrained(output_directory)
print(f"Modello e tokenizer salvati in {output_directory}")

# Stampa il dizionario delle etichette per uso futuro
print("\nDizionario delle etichette (utile per interpretare le previsioni):")
print(id_to_label)

# Esempio di utilizzo del modello:
print("\nEsempio di utilizzo del modello:")
print('from transformers import AutoModelForSequenceClassification, AutoTokenizer')
print(f'model = AutoModelForSequenceClassification.from_pretrained("{output_directory}")')
print(f'tokenizer = AutoTokenizer.from_pretrained("{output_directory}")')
print('inputs = tokenizer("Esempio di testo", return_tensors="pt")')
print('outputs = model(**inputs)')
print('predictions = outputs.logits.argmax(-1).item()')
print('etichetta_prevista = id_to_label[predictions]  # Converti l\'ID nell\'etichetta originale')

# Stampiamo una tabella finale con i risultati complessivi
summary_dict = {
    "Metric": ["eval_loss", "accuracy", "precision", "recall", "f1"],
    "Valore": [
        test_results.get('eval_loss', 'N/A'),
        test_results.get('eval_accuracy', 'N/A'),
        test_results.get('eval_precision', 'N/A'),
        test_results.get('eval_recall', 'N/A'),
        test_results.get('eval_f1', 'N/A')
    ]
```

```
}
df_summary = pd.DataFrame(summary_dict)
print("\n=== Tabella Riassuntiva dei Risultati dell'
    Adestramento ===")
print(df_summary.to_string(index=False))
```

Listing 1: Codice di Training per il Fine Tuning di BERT