Relazione sulle Versioni del Codice per il Fine-Tuning di BERT

10 aprile 2025

1 Introduzione

Questo documento presenta in dettaglio l'evoluzione del codice per il fine-tuning di un modello BERT (`bert-base-uncased`) per attività di classificazione del testo, a partire dalla prima versione fino alla versione finale. Per ogni versione vengono mostrati:

- Il codice completo.
- Le modifiche introdotte rispetto alla versione precedente.
- Commenti esplicativi sulle scelte (ad esempio: aggiunta di logging, modifica degli argomenti di training, gestione di parametri deprecati, ecc.).

Il codice utilizza le librerie `transformers`, `datasets`, `evaluate` e altre utility per il preprocessing e la valutazione del modello.

2 Versione 1 — Prima Versione

Descrizione

La prima versione imposta il training su un dataset caricato da file JSON, identifica automaticamente le colonne di testo ed etichette, effettua il preprocessing delle etichette, tokenizza il dataset, crea i set di training, validazione e test e configura il `Trainer` con argomenti di training (valutazione e salvataggio ogni 100 step). Viene infine avviato il training, la valutazione e il salvataggio del modello.

Codice

```
import os
os.environ["WANDB_MODE"] = "offline"  # Logging locale

import numpy as np
from datasets import load_dataset
from transformers import AutoTokenizer, AutoModelForSequenceClassification,
    TrainingArguments, Trainer
import evaluate

# Caricamento del 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])

# Identificazione automatica delle colonne
first_example = dataset["train"][0]
text_column = None
label_column = None

# Cerca la colonna del testo basandosi sulla lunghezza della stringa
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

# Cerca la colonna delle etichette
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 si trova una colonna di etichette evidente, si sceglie una colonna non testuale
    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}'")

# Preprocessing delle etichette
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}")
```

```
68
69 # Mappatura etichette -> ID
70 label_to_id = {label: i for i, label in enumerate(sorted(unique_labels))}
71 id_to_label = {i: label for label, i in label_to_id.items()}
72
73 print(f"\nMapping etichette -> ID: {label_to_id}")
74
75 def preprocess_labels(examples):
76     result = dict(examples)
77     labels = examples[label_column]
78     processed_labels = []
79     for label in labels:
80         if isinstance(label, list):
81             processed_labels.append(label_to_id[label[0]])
82         else:
83             processed_labels.append(label_to_id[label])
84     result[label_column] = processed_labels
85     return result
86
87 processed_dataset = dataset.map(preprocess_labels, batched=True)
88
89 # Scelta del modello e tokenizer
90 model_checkpoint = "bert-base-uncased"
91 tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
92
93 def tokenize_function(examples):
94     return tokenizer(
95         examples[text_column],
96         padding="max_length",
97         truncation=True,
98         max_length=128
99     )
100
101 tokenized_datasets = processed_dataset.map(tokenize_function, batched=True)
102 tokenized_datasets = tokenized_datasets.remove_columns([col for col in
    processed_dataset["train"].column_names if col != label_column])
103 tokenized_datasets = tokenized_datasets.rename_column(label_column, "labels"
    )
104 tokenized_datasets.set_format("torch")
105
106 # Creazione dei set
107 train_testvalid = tokenized_datasets["train"].train_test_split(test_size
    =0.2, seed=42)
108 train_dataset = train_testvalid["train"]
109 validation_dataset = train_testvalid["test"]
110 test_dataset = tokenized_datasets["test"]
111
112 print(f"\nDimensione dataset training: {len(train_dataset)} esempi")
113 print(f"Dimensione dataset validazione: {len(validation_dataset)} esempi")
114 print(f"Dimensione dataset test: {len(test_dataset)} esempi")
115
116 # Impostazione delle metriche
117 metric = evaluate.load("accuracy")
118 def compute_metrics(eval_pred):
119     logits, labels = eval_pred
120     predictions = np.argmax(logits, axis=-1)
121     accuracy = metric.compute(predictions=predictions, references=labels)
122     from sklearn.metrics import precision_recall_fscore_support
123     precision, recall, f1, _ = precision_recall_fscore_support(labels,
    predictions, average='weighted')
```

```
    metrics = {
        'accuracy': accuracy['accuracy'],
        'precision': precision,
        'recall': recall,
        'f1': f1
    }
    return metrics

# Caricamento del modello
num_labels = len(label_to_id)
model = AutoModelForSequenceClassification.from_pretrained(
    model_checkpoint,
    num_labels=num_labels
)

output_directory = "my-bert-fine-tuned-model"

training_args = TrainingArguments(
    output_dir=output_directory,
    eval_steps=100,
    save_steps=100,
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    save_total_limit=2,
    report_to="none"
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=validation_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)

print("Inizio addestramento...")
trainer.train()
print("Addestramento completato!")

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

trainer.save_model(output_directory)
tokenizer.save_pretrained(output_directory)
print(f"Modello e tokenizer salvati in {output_directory}")
print("\nDizionario etichette:")
print(id_to_label)
```

Listing 1: Versione 1

3 Versione 2 — Modifica dei Parametri di Training

Descrizione

In questa versione si mantiene invariato il resto del codice, mentre si modificano i parametri relativi al training. In particolare:

- Aggiunta la voce `warmup_steps=500` per stabilizzare l'addestramento nelle prime fasi.
- Abilitazione di FP16 (`fp16=True`) per sfruttare la precisione mista su hardware compatibile.
- Abilitazione dell'accumulo dei gradienti (`gradient_accumulation_steps=2`) per simulare batch più grandi.

Codice (solo TrainingArguments modificato)

```
training_args = TrainingArguments(
    output_dir=output_directory,
    eval_steps=100,                # Valutazione ogni 100 steps
    save_steps=100,                # Salvataggio ogni 100 steps
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,            # Numero di epoche
    weight_decay=0.01,
    warmup_steps=500,              # Warmup per stabilizzare il training
    fp16=True,                     # Abilita FP16
    gradient_accumulation_steps=2,  # Batch virtualmente più grandi
    save_total_limit=2,
    report_to="none"
)
```

Listing 2: Versione 2 — TrainingArguments modificato

4 Versione 3 — Aggiunta di Logging Personalizzato

Descrizione

Questa versione introduce un **callback** personalizzato per il logging durante il training. Il callback:

- Registra le metriche (loss, learning rate, eval loss) ad ogni log.
- Alla fine dell'addestramento, stampa una tabella riassuntiva dei log, filtrando ogni 100 step.

Codice Principale con Callback

```
import pandas as pd
from transformers import TrainerCallback

class LogCallback(TrainerCallback):
    def __init__(self):
        self.logs = []  # Lista per salvare i log intermedi
```

```
    def on_log(self, args, state, control, logs=None, **kwargs):
        if logs is not None:
            self.logs.append({
                'step': state.global_step,
                'epoch': state.epoch,
                'loss': logs.get('loss', None),
                'learning_rate': logs.get('learning_rate', None),
                'eval_loss': logs.get('eval_loss', None)
            })

    def on_train_end(self, args, state, control, **kwargs):
        df = pd.DataFrame(self.logs)
        print("\n=== Riassunto Training Log ===")
        df_summary = df[df['step'] % 100 == 0]
        print(df_summary.to_string(index=False))

# Inizializzazione del Trainer con callback
log_callback = LogCallback()
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=validation_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
    callbacks=[log_callback]
)
```

Listing 3: Versione 3 — Logging Personalizzato

5 Versione 4 — Aggiunta della Stampa della Training Loss ogni 100 Step

Descrizione

Questa versione apporta una modifica minima rispetto alla versione 3:

- Viene aggiunto il parametro `logging_steps=100` in `TrainingArguments` per stampare i log (inclusa la loss) ogni 100 step.

Codice (TrainingArguments modificato)

```
training_args = TrainingArguments(
    output_dir=output_directory,
    eval_steps=100,
    save_steps=100,
    logging_steps=100,            # Stampa dei log ogni 100 step
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    warmup_steps=500,
    fp16=True,
    gradient_accumulation_steps=2,
```

```
    save_total_limit=2,
    report_to="none"
)
```

Listing 4: Versione 4 — Logging Steps

6 Versione 5 — Risoluzione del Problema del Parametro Deprecato

Descrizione

La quinta versione interviene per risolvere il problema relativo all'utilizzo di un parametro deprecato. In particolare, il parametro `tokenizer` viene sostituito con `processing_class` all'interno dell'inizializzazione del Trainer. Questa modifica garantisce la compatibilità con le versioni future della libreria Transformers.

Codice (Parte modificata nella definizione del Trainer)

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=validation_dataset,
    processing_class=tokenizer,  # Sostituzione di 'tokenizer' con '
   processing_class'
    compute_metrics=compute_metrics,
    callbacks=[log_callback]
)
```

Listing 5: Versione 5 — Uso di processing$_c lass$

7 Versione Finale — Codice Completo Finale

Descrizione

La versione finale integra tutte le modifiche precedenti ed include anche i comandi di installazione iniziali per garantire di avere le versioni più aggiornate delle librerie necessarie. In questa versione il codice risulta consolidato ed è commentato per garantire chiarezza e manutenzione futura.

Codice Completo

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

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 modalità offline
os.environ["WANDB_MODE"] = "offline"

# Directory di output
output_directory = "my-bert-fine-tuned-model"

# Caricamento del 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])

# Identificazione delle colonne di testo ed etichette
first_example = dataset["train"][0]
text_column = None
label_column = None

# Selezione della colonna testuale (basata sulla lunghezza)
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

# Selezione della colonna delle etichette
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:
    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}'")

# Preprocessing delle etichette
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)

# Scelta del modello e tokenizer
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
    )

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")

# Creazione dei 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 dataset training: {len(train_dataset)} esempi")
print(f"Dimensione dataset validazione: {len(validation_dataset)} esempi")
print(f"Dimensione dataset test: {len(test_dataset)} esempi")

# Impostazione delle metriche
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
)

# Callback per log del training
class LogCallback(TrainerCallback):
    def __init__(self):
        self.logs = []

    def on_log(self, args, state, control, logs=None, **kwargs):
        if logs is not None:
            self.logs.append({
                'step': state.global_step,
                'epoch': state.epoch,
                'loss': logs.get('loss', None),
                'learning_rate': logs.get('learning_rate', None),
                'eval_loss': logs.get('eval_loss', None)
            })

    def on_train_end(self, args, state, control, **kwargs):
        df = pd.DataFrame(self.logs)
        print("\n=== Riassunto Training Log ===")
        df_summary = df[df['step'] % 100 == 0]
        print(df_summary.to_string(index=False))

# Configurazione degli argomenti di training
training_args = TrainingArguments(
    output_dir=output_directory,
    eval_steps=100,
    save_steps=100,
    logging_steps=100,            # Stampa dei log ogni 100 step
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    warmup_steps=500,
    fp16=True,
    gradient_accumulation_steps=2,
    save_total_limit=2,
    report_to="none"
)

# Inizializzazione del Trainer utilizzando il parametro aggiornato "
   processing_class"
log_callback = LogCallback()
```

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=validation_dataset,
    processing_class=tokenizer,  \% Sostituzione di "tokenizer" con "processing_class"
    compute_metrics=compute_metrics,
    callbacks=[log_callback]
)

print("Inizio addestramento...")
trainer.train()
print("Addestramento completato!")

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

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

print("\nDizionario etichette:")
print(id_to_label)

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]')

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=== Riassunto Risultati ===")
print(df_summary.to_string(index=False))
```

Listing 6: Versione Finale — Codice Completo

8 Conclusioni

Nel corso delle varie versioni il codice ha subito le seguenti modifiche:

1. **Versione 1**: Implementazione iniziale con caricamento del dataset, preprocessing delle etichette, tokenizzazione e configurazione base del Trainer.

2. **Versione 2**: Modifica degli argomenti di training con l'introduzione di `warmup_steps`, `fp16` e `gradient_accumulation_steps` per un training più stabile e performante.

3. **Versione 3**: Aggiunta di un callback personalizzato per il logging, al fine di registrare e visualizzare le metriche intermedie.

4. **Versione 4**: Introduzione del parametro `logging_steps` per stampare i log ogni 100 step (inclusa la loss).

5. **Versione 5**: Aggiornamento finale che risolve il problema del parametro deprecato sostituendo `tokenizer` con `processing_class` nell'inizializzazione del Trainer.

6. **Versione Finale**: Consolidamento di tutte le modifiche, con aggiunta dei comandi di installazione e revisione completa del codice per garantirne la compatibilità e la chiarezza.

Questa relazione documenta in modo dettagliato l'evoluzione del codice, fornendo una base solida per eventuali aggiornamenti futuri e per la manutenzione del progetto.