

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

```
1 import os
2 os.environ["WANDB_MODE"] = "offline" # Logging locale
3
4 import numpy as np
5 from datasets import load_dataset
6 from transformers import AutoTokenizer, AutoModelForSequenceClassification,
7     TrainingArguments, Trainer
8 import evaluate
9
10 # Caricamento del dataset
11 data_files = {
12     "train": "dataset_completo.json",
```

```

12     "test": "Test2.json"
13 }
14 dataset = load_dataset('json', data_files=data_files)
15
16 print("Struttura del dataset:")
17 print(dataset)
18 print("\nColonne nel dataset train:")
19 print(dataset["train"].column_names)
20 print("\nPrimo esempio nel dataset:")
21 print(dataset["train"][0])
22
23 # Identificazione automatica delle colonne
24 first_example = dataset["train"][0]
25 text_column = None
26 label_column = None
27
28 # Cerca la colonna del testo basandosi sulla lunghezza della stringa
29 longest_text_len = 0
30 for col in first_example:
31     if isinstance(first_example[col], str) and len(first_example[col]) >
        longest_text_len:
32         longest_text_len = len(first_example[col])
33         text_column = col
34
35 # Cerca la colonna delle etichette
36 for col in first_example:
37     if 'label' in col.lower() or 'class' in col.lower() or 'category' in col
        .lower():
38         label_column = col
39         break
40
41 if text_column is None:
42     raise ValueError("Non è stata trovata una colonna di testo. Specifica
        manualmente il nome della colonna.")
43
44 if label_column is None:
45     # Se non si trova una colonna di etichette evidente, si sceglie una
        colonna non testuale
46     for col in first_example:
47         if col != text_column and not isinstance(first_example[col], str):
48             label_column = col
49             break
50
51 print(f"\nColonna di testo identificata: '{text_column}'")
52 print(f"Colonna di etichette identificata: '{label_column}'")
53
54 # Preprocessing delle etichette
55 def get_unique_labels(examples):
56     labels = examples[label_column]
57     unique_labels = set()
58     for label in labels:
59         if isinstance(label, list):
60             for l in label:
61                 unique_labels.add(l)
62         else:
63             unique_labels.add(label)
64     return list(unique_labels)
65
66 unique_labels = get_unique_labels(dataset["train"])
67 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
103     processed_dataset["train"].column_names if col != label_column])
104 tokenized_datasets = tokenized_datasets.rename_column(label_column, "labels")
105
106 tokenized_datasets.set_format("torch")
107
108 # Creazione dei set
109 train_testvalid = tokenized_datasets["train"].train_test_split(test_size
110     =0.2, seed=42)
111 train_dataset = train_testvalid["train"]
112 validation_dataset = train_testvalid["test"]
113 test_dataset = tokenized_datasets["test"]
114
115 print(f"\nDimensione dataset training: {len(train_dataset)} esempi")
116 print(f"Dimensione dataset validazione: {len(validation_dataset)} esempi")
117 print(f"Dimensione dataset test: {len(test_dataset)} esempi")
118
119 # Impostazione delle metriche
120 metric = evaluate.load("accuracy")
121 def compute_metrics(eval_pred):
122     logits, labels = eval_pred
123     predictions = np.argmax(logits, axis=-1)
124     accuracy = metric.compute(predictions=predictions, references=labels)
125     from sklearn.metrics import precision_recall_fscore_support
126     precision, recall, f1, _ = precision_recall_fscore_support(labels,
127         predictions, average='weighted')

```

```

124     metrics = {
125         'accuracy': accuracy['accuracy'],
126         'precision': precision,
127         'recall': recall,
128         'f1': f1
129     }
130     return metrics
131
132 # Caricamento del modello
133 num_labels = len(label_to_id)
134 model = AutoModelForSequenceClassification.from_pretrained(
135     model_checkpoint,
136     num_labels=num_labels
137 )
138
139 output_directory = "my-bert-fine-tuned-model"
140
141 training_args = TrainingArguments(
142     output_dir=output_directory,
143     eval_steps=100,
144     save_steps=100,
145     learning_rate=2e-5,
146     per_device_train_batch_size=16,
147     per_device_eval_batch_size=16,
148     num_train_epochs=3,
149     weight_decay=0.01,
150     save_total_limit=2,
151     report_to="none"
152 )
153
154 trainer = Trainer(
155     model=model,
156     args=training_args,
157     train_dataset=train_dataset,
158     eval_dataset=validation_dataset,
159     tokenizer=tokenizer,
160     compute_metrics=compute_metrics,
161 )
162
163 print("Inizio addestramento...")
164 trainer.train()
165 print("Addestramento completato!")
166
167 print("Valutazione sul test set...")
168 test_results = trainer.evaluate(test_dataset)
169 print("Risultati test:", test_results)
170
171 trainer.save_model(output_directory)
172 tokenizer.save_pretrained(output_directory)
173 print(f"Modello e tokenizer salvati in {output_directory}")
174 print("\nDizionario etichette:")
175 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)

```
1 training_args = TrainingArguments(  
2     output_dir=output_directory,  
3     eval_steps=100,                # Valutazione ogni 100 steps  
4     save_steps=100,                # Salvataggio ogni 100 steps  
5     learning_rate=2e-5,  
6     per_device_train_batch_size=16,  
7     per_device_eval_batch_size=16,  
8     num_train_epochs=3,            # Numero di epoche  
9     weight_decay=0.01,  
10    warmup_steps=500,              # Warmup per stabilizzare il training  
11    fp16=True,                    # Abilita FP16  
12    gradient_accumulation_steps=2, # Batch virtualmente più grandi  
13    save_total_limit=2,  
14    report_to="none"  
15 )
```

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

```
1 import pandas as pd  
2 from transformers import TrainerCallback  
3  
4 class LogCallback(TrainerCallback):  
5     def __init__(self):  
6         self.logs = [] # Lista per salvare i log intermedi
```

```

7
8     def on_log(self, args, state, control, logs=None, **kwargs):
9         if logs is not None:
10             self.logs.append({
11                 'step': state.global_step,
12                 'epoch': state.epoch,
13                 'loss': logs.get('loss', None),
14                 'learning_rate': logs.get('learning_rate', None),
15                 'eval_loss': logs.get('eval_loss', None)
16             })
17
18     def on_train_end(self, args, state, control, **kwargs):
19         df = pd.DataFrame(self.logs)
20         print("\n=== Riassunto Training Log ===")
21         df_summary = df[df['step'] % 100 == 0]
22         print(df_summary.to_string(index=False))
23
24 # Inizializzazione del Trainer con callback
25 log_callback = LogCallback()
26 trainer = Trainer(
27     model=model,
28     args=training_args,
29     train_dataset=train_dataset,
30     eval_dataset=validation_dataset,
31     tokenizer=tokenizer,
32     compute_metrics=compute_metrics,
33     callbacks=[log_callback]
34 )

```

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)

```

1 training_args = TrainingArguments(
2     output_dir=output_directory,
3     eval_steps=100,
4     save_steps=100,
5     logging_steps=100,                # Stampa dei log ogni 100 step
6     learning_rate=2e-5,
7     per_device_train_batch_size=16,
8     per_device_eval_batch_size=16,
9     num_train_epochs=3,
10    weight_decay=0.01,
11    warmup_steps=500,
12    fp16=True,
13    gradient_accumulation_steps=2,

```

```

14     save_total_limit=2,
15     report_to="none"
16 )

```

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)

```

1 trainer = Trainer(
2     model=model,
3     args=training_args,
4     train_dataset=train_dataset,
5     eval_dataset=validation_dataset,
6     processing_class=tokenizer, # Sostituzione di 'tokenizer' con '
    processing_class'
7     compute_metrics=compute_metrics,
8     callbacks=[log_callback]
9 )

```

Listing 5: Versione 5 — Uso di `processing_class`

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

```

1 ! pip install transformers datasets evaluate torch accelerate -U
2 !pip install -U transformers
3
4 import os
5 import numpy as np
6 import pandas as pd
7 from datasets import load_dataset
8 from transformers import AutoTokenizer, AutoModelForSequenceClassification,
    TrainingArguments, Trainer, TrainerCallback
9 import evaluate

```

```

10 from sklearn.metrics import precision_recall_fscore_support
11
12 # Disabilitiamo wandb in modalità offline
13 os.environ["WANDB_MODE"] = "offline"
14
15 # Directory di output
16 output_directory = "my-bert-fine-tuned-model"
17
18 # Caricamento del dataset
19 data_files = {
20     "train": "dataset_completo.json",
21     "test": "Test2.json"
22 }
23 dataset = load_dataset('json', data_files=data_files)
24
25 print("Struttura del dataset:")
26 print(dataset)
27 print("\nColonne nel dataset train:")
28 print(dataset["train"].column_names)
29 print("\nPrimo esempio nel dataset:")
30 print(dataset["train"][0])
31
32 # Identificazione delle colonne di testo ed etichette
33 first_example = dataset["train"][0]
34 text_column = None
35 label_column = None
36
37 # Selezione della colonna testuale (basata sulla lunghezza)
38 longest_text_len = 0
39 for col in first_example:
40     if isinstance(first_example[col], str) and len(first_example[col]) >
41         longest_text_len:
42         longest_text_len = len(first_example[col])
43         text_column = col
44
45 # Selezione della colonna delle etichette
46 for col in first_example:
47     if 'label' in col.lower() or 'class' in col.lower() or 'category' in col
48         .lower():
49         label_column = col
50         break
51
52 if text_column is None:
53     raise ValueError("Non è stata trovata una colonna di testo. Specifica
54     manualmente il nome della colonna.")
55 if label_column is None:
56     for col in first_example:
57         if col != text_column and not isinstance(first_example[col], str):
58             label_column = col
59             break
60
61 print(f"\nColonna di testo identificata: '{text_column}'")
62 print(f"Colonna di etichette identificata: '{label_column}'")
63
64 # Preprocessing delle etichette
65 def get_unique_labels(examples):
66     labels = examples[label_column]
67     unique_labels = set()
68     for label in labels:
69         if isinstance(label, list):

```



```

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

```

```

124     logits, labels = eval_pred
125     predictions = np.argmax(logits, axis=-1)
126     accuracy = metric.compute(predictions=predictions, references=labels)
127     precision, recall, f1, _ = precision_recall_fscore_support(labels,
128     predictions, average='weighted')
129     return {
130         'accuracy': accuracy['accuracy'],
131         'precision': precision,
132         'recall': recall,
133         'f1': f1
134     }
135 num_labels = len(label_to_id)
136 model = AutoModelForSequenceClassification.from_pretrained(
137     model_checkpoint,
138     num_labels=num_labels
139 )
140
141 # Callback per log del training
142 class LogCallback(TrainerCallback):
143     def __init__(self):
144         self.logs = []
145
146     def on_log(self, args, state, control, logs=None, **kwargs):
147         if logs is not None:
148             self.logs.append({
149                 'step': state.global_step,
150                 'epoch': state.epoch,
151                 'loss': logs.get('loss', None),
152                 'learning_rate': logs.get('learning_rate', None),
153                 'eval_loss': logs.get('eval_loss', None)
154             })
155
156     def on_train_end(self, args, state, control, **kwargs):
157         df = pd.DataFrame(self.logs)
158         print("\n=== Riassunto Training Log ===")
159         df_summary = df[df['step'] % 100 == 0]
160         print(df_summary.to_string(index=False))
161
162 # Configurazione degli argomenti di training
163 training_args = TrainingArguments(
164     output_dir=output_directory,
165     eval_steps=100,
166     save_steps=100,
167     logging_steps=100,           # Stampa dei log ogni 100 step
168     learning_rate=2e-5,
169     per_device_train_batch_size=16,
170     per_device_eval_batch_size=16,
171     num_train_epochs=3,
172     weight_decay=0.01,
173     warmup_steps=500,
174     fp16=True,
175     gradient_accumulation_steps=2,
176     save_total_limit=2,
177     report_to="none"
178 )
179
180 # Inizializzazione del Trainer utilizzando il parametro aggiornato "
181     processing_class"
182 log_callback = LogCallback()

```

```

182 trainer = Trainer(
183     model=model,
184     args=training_args,
185     train_dataset=train_dataset,
186     eval_dataset=validation_dataset,
187     processing_class=tokenizer, \ % Sostituzione di "tokenizer" con "
processing_class"
188     compute_metrics=compute_metrics,
189     callbacks=[log_callback]
190 )
191
192 print("Inizio addestramento...")
193 trainer.train()
194 print("Addestramento completato!")
195
196 print("Valutazione sul test set...")
197 test_results = trainer.evaluate(test_dataset)
198 print("Risultati test:", test_results)
199
200 trainer.save_model(output_directory)
201 tokenizer.save_pretrained(output_directory)
202 print(f"Modello e tokenizer salvati in {output_directory}")
203
204 print("\nDizionario etichette:")
205 print(id_to_label)
206
207 print("\nEsempio di utilizzo del modello:")
208 print('from transformers import AutoModelForSequenceClassification,
AutoTokenizer')
209 print(f'model = AutoModelForSequenceClassification.from_pretrained("{
output_directory}")')
210 print(f'tokenizer = AutoTokenizer.from_pretrained("{output_directory}")')
211 print('inputs = tokenizer("Esempio di testo", return_tensors="pt")')
212 print('outputs = model(**inputs)')
213 print('predictions = outputs.logits.argmax(-1).item()')
214 print('etichetta_prevista = id_to_label[predictions]')
215
216 summary_dict = {
217     "Metric": ["eval_loss", "accuracy", "precision", "recall", "f1"],
218     "Valore": [
219         test_results.get('eval_loss', 'N/A'),
220         test_results.get('eval_accuracy', 'N/A'),
221         test_results.get('eval_precision', 'N/A'),
222         test_results.get('eval_recall', 'N/A'),
223         test_results.get('eval_f1', 'N/A')
224     ]
225 }
226 df_summary = pd.DataFrame(summary_dict)
227 print("\n=== Riassunto Risultati ===")
228 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.