Natural Language Processing using Python Programming

Notebook 10.2: Fine-Tuning BERT for Text Classification

Python 3.8+ | Transformers Latest | PyTorch Latest | Datasets Latest | License MIT

Part of the comprehensive learning series: Natural Language Processing using Python Programming

Learning Objectives:

- Master the fine-tuning process for adapting pre-trained BERT models to specific classification tasks
- Learn end-to-end workflow using Hugging Face Trainer API for production-ready model training
- Understand tokenization requirements and data preparation for Transformer models
- Implement custom metrics and evaluation strategies for deep learning NLP projects
- Build practical skills for deploying state-of-the-art NLP solutions in real-world applications

---

- This notebook demonstrates the industry-standard process of **fine-tuning BERT** for custom text classification tasks, showing how to adapt powerful pre-trained models to specific domains and requirements.

- We'll explore the complete workflow from data preparation to model evaluation using the **Hugging Face Trainer API**, the professional standard for building state-of-the-art NLP solutions.

1. Setting up: Libraries and Data

- We use the **Hugging Face `datasets` library** to quickly load a tiny subset of the **AG News** multi-class classification dataset for a quick demonstration.

In [4]:
```
# Import necessary libraries
# AutoTokenizer and AutoModelForSequenceClassification are used for tokenizatior
# Trainer and TrainingArguments are used for setting up the training process
import pandas as pd
from datasets import load_dataset
from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trai
import numpy as np
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

# Load a very small sample (1000 total) for quick demo execution
# ag_news dataset has 4 classes: World, Sports, Business, Sci/Tech
```

```python
# Each class has a balanced number of samples
raw_datasets = load_dataset("ag_news", split="train[:1000]")
raw_datasets = raw_datasets.train_test_split(test_size=0.2, seed=42)

print(f"Total Training Samples: {len(raw_datasets['train'])}")
print(f"Total Testing Samples: {len(raw_datasets['test'])}")
print(f"Example News Item: {raw_datasets['train'][0]['text'][:80]}...")
```

```
Total Training Samples: 800
Total Testing Samples: 200
Example News Item: Delegates Urge Al-Sadr to Leave Shrine BAGHDAD, Iraq - Delegat
es at Iraq's Natio...
```

---

2. Tokenization and Data Preparation

- BERT models require text to be tokenized using the **exact** tokenizer the model was pre-trained with.
- The tokenizer also pads/truncates text to a uniform length (512 tokens maximum for standard BERT).

In [5]:

```python
# Load the BERT tokenizer
# The tokenizer must match the pre-trained model
model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Define a function to tokenize the dataset
# This function will be applied to each example in the dataset
def tokenize_function(examples):
    # Tokenize text and apply padding and truncation
    return tokenizer(examples["text"], padding="max_length", truncation=True)

# Apply the tokenization to the entire dataset (Map function is optimized for sp
tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)

# Rename 'label' to 'labels' as required by the Hugging Face Trainer
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")

# Set the output format to PyTorch tensors
tokenized_datasets.set_format("torch", columns=["input_ids", "attention_mask", '

train_dataset = tokenized_datasets["train"]
eval_dataset = tokenized_datasets["test"]
```

---

3. Loading the Pre-trained Model and Defining Metrics

- We load BERT's weights and define a function to compute standard metrics for evaluation.

In [6]:

```python
# Get the number of unique classes (use 'label' not 'labels' - original column r
num_labels = len(raw_datasets['train'].unique('label'))
```

```python
# Load the BERT model with a classification head initialized for our task
model = AutoModelForSequenceClassification.from_pretrained(model_name, num_label

def compute_metrics(pred):
    """Computes standard classification metrics: accuracy, precision, recall, f1
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)

    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds, av
    acc = accuracy_score(labels, preds)

    return {
        'accuracy': acc,
        'f1': f1,
        'precision': precision,
        'recall': recall
    }

print(f"Model loaded with {num_labels} classification outputs and metrics define
```

```
Some weights of BertForSequenceClassification were not initialized from the model
checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'c
lassifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it f
or predictions and inference.
Model loaded with 4 classification outputs and metrics defined.
```

---

4. Training the Model with the `Trainer` API

The Hugging Face `Trainer` handles the entire deep learning training loop, data loading, logging, and checkpoint saving.

In [12]:

```python
# Define the training hyper-parameters
training_args = TrainingArguments(
    output_dir="./results",              # Output directory for model check
    num_train_epochs=1,                  # Total number of training epochs
    per_device_train_batch_size=8,       # Batch size per device during tra
    per_device_eval_batch_size=8,        # Batch size for evaluation
    weight_decay=0.01,
    logging_steps=10,                    # Log every 10 steps
    eval_strategy="epoch",               # Evaluate at the end of each epoc
    save_strategy="epoch",               # Save checkpoint at the end of ea
    load_best_model_at_end=True,         # Load the best model found during
)

# Initialize the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)
```

```
print("Starting BERT fine-tuning (Training for 1 Epoch)... This may take some ti

# Start training!
trainer.train()

print("\nFine-Tuning Complete.")
```

```
Starting BERT fine-tuning (Training for 1 Epoch)... This may take some time.
  0%|          | 0/100 [00:00<?, ?it/s]
{'loss': 1.2557, 'grad_norm': 12.219493865966797, 'learning_rate': 4.5e-05, 'epoc
h': 0.1}
{'loss': 1.2233, 'grad_norm': 6.900300025939941, 'learning_rate': 4e-05, 'epoch':
0.2}
{'loss': 0.9965, 'grad_norm': 7.350841999053955, 'learning_rate': 3.5e-05, 'epoc
h': 0.3}
{'loss': 0.856, 'grad_norm': 8.304831504821777, 'learning_rate': 3e-05, 'epoch':
0.4}
{'loss': 0.7306, 'grad_norm': 17.605993270874023, 'learning_rate': 2.5e-05, 'epoc
h': 0.5}
{'loss': 0.6708, 'grad_norm': 8.324629783630371, 'learning_rate': 2e-05, 'epoch':
0.6}
{'loss': 0.4812, 'grad_norm': 12.787162780761719, 'learning_rate': 1.5e-05, 'epoc
h': 0.7}
{'loss': 0.4555, 'grad_norm': 7.984370231628418, 'learning_rate': 1e-05, 'epoch':
0.8}
{'loss': 0.4818, 'grad_norm': 8.38824462890625, 'learning_rate': 5e-06, 'epoch':
0.9}
{'loss': 0.5279, 'grad_norm': 6.782375335693359, 'learning_rate': 0.0, 'epoch':
1.0}
  0%|          | 0/25 [00:00<?, ?it/s]
{'eval_loss': 0.4006122946739197, 'eval_accuracy': 0.88, 'eval_f1': 0.88056426732
59235, 'eval_precision': 0.8816425745587341, 'eval_recall': 0.88, 'eval_runtime':
140.8393, 'eval_samples_per_second': 1.42, 'eval_steps_per_second': 0.178, 'epoc
h': 1.0}
{'train_runtime': 2915.8536, 'train_samples_per_second': 0.274, 'train_steps_per_
second': 0.034, 'train_loss': 0.7679438781738281, 'epoch': 1.0}

Fine-Tuning Complete.
```

---

5. Final Evaluation and Prediction

We evaluate the final model and demonstrate how to perform a prediction on a new piece of text.

In [14]:

```
# Run final evaluation on the test set
final_metrics = trainer.evaluate(eval_dataset)
print("--- Final Model Performance ---")
print(pd.Series(final_metrics).round(4))

# --- Example Prediction ---
example_text = "Google stock surged after revealing new advancements in quantum

# 1. Tokenize the input text
inputs = tokenizer(example_text, return_tensors="pt", truncation=True, padding=T

# 2. Pass through the model
```

```python
outputs = model(**inputs)
logits = outputs.logits

# 3. Get the predicted class ID (highest logit score)
predicted_class_id = logits.argmax().item()

# 4. Map the ID back to the human-readable label
label_names = raw_datasets['train'].features['label'].names
predicted_label_name = label_names[predicted_class_id]

print("\n--- Example Prediction ---")
print(f"Text: {example_text}")
print(f"Predicted Category: {predicted_label_name}")
```

```
  0%|          | 0/25 [00:00<?, ?it/s]
--- Final Model Performance ---
eval_loss                   0.4006
eval_accuracy               0.8800
eval_f1                     0.8806
eval_precision              0.8816
eval_recall                 0.8800
eval_runtime              159.5825
eval_samples_per_second     1.2530
eval_steps_per_second       0.1570
epoch                       1.0000
dtype: float64

--- Example Prediction ---
Text: Google stock surged after revealing new advancements in quantum computing a
nd AI.
Predicted Category: Sci/Tech
```

6. Summary and Next Steps

- We successfully demonstrated the complex process of **fine-tuning BERT** on a custom text classification task using the standard Hugging Face workflow.
- This is how state-of-the-art NLP solutions are built today.
- In the final practical notebook (**10.3**), we will address the challenge of taking a trained model (either Scikit-learn or this Transformer model) and making it available as a live web service via a **Flask API**.

Key Takeaways

- **Fine-Tuning Mastery:** We learned the complete fine-tuning workflow for adapting pre-trained BERT models to custom classification tasks using industry-standard practices.
- **Hugging Face Expertise:** We mastered the Trainer API, datasets library, and tokenization processes that form the backbone of modern NLP development workflows.
- **Production-Ready Skills:** We implemented proper data preparation, metrics computation, and model evaluation techniques used in real-world AI applications.

- **Deep Learning Integration:** We successfully bridged classical NLP knowledge with cutting-edge Transformer models, preparing for advanced AI development.

Next Notebook Preview

- With fine-tuned models ready, we'll learn to **deploy them as live web services**.
- Notebook 10.3 will guide you through creating **Flask API endpoints** to serve both Scikit-learn and Transformer models, making your NLP solutions accessible to real users and applications.

About This Project

This notebook is part of the **Natural Language Processing using Python Programming for Beginners** repository - a comprehensive, beginner-friendly guide for mastering NLP using Python, NLTK, and SpaCy.

Repository: `NLP`

Author

Prakash Ukhalkar
GitHub prakash-ukhalkar

Built with ❤️ for the Python community