Sentiment Analysis with Transfer Learning and Fine-tuning

This notebook demonstrates how to fine-tune a pre-trained model for a binary sentiment analysis task.

1. Setup and Imports

In [ ]:
```
%pip install torch transformers pandas scikit-learn datasets psutil
%pip install --upgrade ipywidgets
%pip install ydata-profiling
```

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import os
import time
import psutil
from collections import defaultdict
from datasets import load_dataset, Dataset, DatasetDict
from datasets import Features, Value
from typing import Dict, List
import warnings
from tqdm import tqdm

from ydata_profiling import ProfileReport
import webbrowser

import torch
from torch.utils.data import Dataset, DataLoader, TensorDataset
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score, f1_sco
from transformers import (
    DistilBertTokenizer, DistilBertModel, DistilBertForSequenceClassifica
    BertTokenizer, BertModel,
    RobertaTokenizer, RobertaModel
)
from transformers import logging, DataCollatorWithPadding

logging.set_verbosity_error()
warnings.filterwarnings('ignore')
```

2. System Information

In [ ]:
```
# System information
print("=== System Info ===")
print(f"PyTorch version: {torch.__version__}")
print(f"CPU cores: {psutil.cpu_count()}")
```

```
print(f"RAM: {psutil.virtual_memory().total / (1024 ** 3):.2f} GB")
print("\n=== GPU Info ===")
print(f"CUDA available: {torch.cuda.is_available()}")
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")
```

In [ ]:
```
# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

3. Load and Analyze Datasets

Getting the datasets reviews from Hugging Face:

https://huggingface.co/datasets/stanfordnlp/imdb

In [ ]:
```
# Load datasets
df_sample = pd.read_parquet('../data/sample_reviews.parquet')
print("'Sample Reviews' columns:", df_sample.columns.tolist())

imdb = load_dataset("stanfordnlp/imdb")
df_imdb = pd.DataFrame(imdb['train'])
print("\n'IMDB' columns:", df_imdb.columns.tolist())
```

In [ ]:
```
# Analyze datasets
datasets = {
    'Sample Reviews': (df_sample, 'sentence'),
    'IMDB': (df_imdb, 'text')
}

for name, (df, text_col) in datasets.items():
    print(f"\n=== '{name}' INFO ===")
    print(df.info())

    print(f"\n=== '{name}' DESCRIBE ===")
    print(df.describe())

    print(f"\n=== '{name}' HEAD ===")
    print(df.head())

    print(f"\n=== '{name}' ANALYSIS ===")
    print(f"Total samples: {len(df)}")
    print(f"Label distribution:\n{df['label'].value_counts(normalize=True

    # Text length statistics
    print(f"\n=== '{name}' STATS ===")
    lengths = df[text_col].str.len()
    word_lengths = df[text_col].str.split().str.len()
    print(f"\nText length statistics:")
    print(f"Mean   Chars: {lengths.mean():.2f}")
    print(f"Median Chars: {lengths.median():.2f}")
    print(f"Max    Chars: {lengths.max()}")
    print(f"Avg    Words: {word_lengths.mean():.2f}")
    print(f"Max    Words: {word_lengths.max()}")
    print(f"Min    Words: {word_lengths.min()}")
    print(f"Memory Usage (MB): {round(df.memory_usage(deep=True).sum() /
```

```
    # Visualize length distribution
    plt.figure(figsize=(10, 5))
    plt.hist(lengths, bins=50)
    plt.title(f'{name} - Text Length Distribution')
    plt.xlabel('Text Length')
    plt.ylabel('Count')
    plt.show()
```

In [ ]:
```
# Generate report from datasets
datasets = {
    'Sample Reviews': df_sample,
    'IMDB': df_imdb,
}
for name, df in datasets.items():
    if isinstance(df, pd.DataFrame):
        profile = ProfileReport(df, title=f"Dataset {name.upper()}")
        profile.to_notebook_iframe()
        report_path = f"../reports/ydata_report_{name.upper()}.html"
        profile.to_file(report_path)
        webbrowser.open('file://' + os.path.realpath(report_path))
    else:
        print(f"Warning: {name} is not a pandas DataFrame")
```

4. Dataset Preparation

Select dataset

In [ ]:
```
data = df_sample
# data = pd.DataFrame(df_imdb)

print(data.info())
print(data.head())
```

In [ ]:
```
# Set parameters
MAX_SAMPLES = 1000
TEST_SIZE = 0.2
VAL_SIZE = 0.1
BATCH_SIZE = 16
NUM_EPOCHS = 3
LEARNING_RATE = 0.00002
```

In [ ]:
```
# Prepare data
# Sample data if MAX_SAMPLES is set and less than dataset size
if MAX_SAMPLES and MAX_SAMPLES < len(data):
    data = data.sample(n=MAX_SAMPLES, random_state=42)
    print(f"Sampled {MAX_SAMPLES} examples from dataset")

text_column = 'sentence' if 'sentence' in data.columns else 'text'
texts = data[text_column].values
labels = data['label'].values

print(f"Final dataset size: {len(texts)}")
print(f"Label distribution:\n{pd.Series(labels).value_counts(normalize=Tr
```

In [ ]:
```
# Split data
train_texts, test_texts, train_labels, test_labels = train_test_split(
```

```
    texts, labels, test_size=TEST_SIZE, random_state=42, stratify=labels
)

train_texts, val_texts, train_labels, val_labels = train_test_split(
    train_texts, train_labels,
    test_size=VAL_SIZE/(1-TEST_SIZE),
    random_state=42,
    stratify=train_labels
)
```

In [ ]:

```
# Print split sizes and distributions
split_sizes = {
    "Dataset": ["TRAIN", "VAL", "TEST"],
    "Samples": [len(train_texts), len(val_texts), len(test_texts)],
    "      %": [
        len(train_texts) / len(texts) * 100,
        len(val_texts) / len(texts) * 100,
        len(test_texts) / len(texts) * 100
    ]
}

# Print label distribution for each split
label_distributions = {
    "Dataset": ["TRAIN", "VAL  ", "TEST "],
    "Distribution": [
        {k: f"{v:.2f}" for k, v in pd.Series(train_labels).value_counts(n
        {k: f"{v:.2f}" for k, v in pd.Series(val_labels).value_counts(nor
        {k: f"{v:.2f}" for k, v in pd.Series(test_labels).value_counts(no
    ]
}

# Print stats from text lengths
train_lengths = [len(text.split()) for text in train_texts]
val_lengths = [len(text.split()) for text in val_texts]
test_lengths = [len(text.split()) for text in test_texts]

length_stats = {
    "Dataset": ["TRAIN", "VAL", "TEST"],
    "   AVG": [
        np.mean(train_lengths),
        np.mean(val_lengths),
        np.mean(test_lengths)
    ],
    "   MAX": [
        np.max(train_lengths),
        np.max(val_lengths),
        np.max(test_lengths)
    ],
    "   MIN": [
        np.min(train_lengths),
        np.min(val_lengths),
        np.min(test_lengths)
    ]
}

split_sizes_df = pd.DataFrame(split_sizes)
length_stats_df = pd.DataFrame(length_stats)
label_distributions_df = pd.DataFrame(label_distributions)

print("\nDataset splits:")
```

```
print(split_sizes_df.to_string(index=False, float_format="{:.1f}".format)

print("\nLabel distribution in splits:")
for idx, row in label_distributions_df.iterrows():
    print(f"{row['Dataset']} set: {row['Distribution']}")

print("\nText length statistics:")
print(length_stats_df.to_string(index=False, float_format="{:.1f}".format

# Show examples from each dataset
print("\nSample texts from each dataset:")
print("\nTRAIN examples:")
for i in range(3):
    text = train_texts[i]
    print(f"{i+1}. Chars: {len(text)}, Words: {len(text.split())}")
    print(f"Text: {text[:70]}...")

print("\nVAL examples:")
for i in range(3):
    text = val_texts[i]
    print(f"{i+1}. Chars: {len(text)}, Words: {len(text.split())}")
    print(f"Text: {text[:70]}...")

print("\nTEST examples:")
for i in range(3):
    text = test_texts[i]
    print(f"{i+1}. Chars: {len(text)}, Words: {len(text.split())}")
    print(f"Text: {text[:70]}...")
```

5. Model and Platform Research

Models:

- BERT
- RoBERTa
- DistilBERT
- GPT-2
- Electra
- XLNet

DistilBERT is good balance between performance and computational efficiency. It is a lighter and faster version of BERT.

Computing platforms:

- AWS Sagemaker Studio Labs
- Google Colab

In [ ]:

```
class ModelComparator:
    def __init__(self):
        self.models: Dict = {}
        self.tokenizers: Dict = {}
        self.results: List = []

    def load_model(self, model_name: str, verbose: bool = True):
```

```
        """Loads a model and its tokenizer."""
        if verbose:
            print(f"Loading {model_name}...", end=' ')

        model_configs = {
            'distilbert': ('distilbert-base-uncased', DistilBertTokenizer
            'bert': ('bert-base-uncased', BertTokenizer, BertModel),
            'roberta': ('roberta-base', RobertaTokenizer, RobertaModel),
        }

        if model_name in model_configs:
            model_path, TokenizerClass, ModelClass = model_configs[model_
            try:
                # Load model and tokenizer
                tokenizer = TokenizerClass.from_pretrained(model_path)
                model = ModelClass.from_pretrained(model_path)

                if model_name == 'gpt2':
                    tokenizer.pad_token = tokenizer.eos_token

                self.models[model_name] = model
                self.tokenizers[model_name] = tokenizer

                if verbose:
                    print("✓")

            except Exception as e:
                if verbose:
                    print(f"❌ Error: {str(e)}")
                raise
        else:
            raise ValueError(f"Unsupported model: {model_name}")

    def show_tokenization(self, text: str):
        """Display tokenization results for each model."""
        print("\n=== Tokenization Comparison ===")
        print(f"Original text: {text}\n")

        for model_name, tokenizer in self.tokenizers.items():
            print(f"\n{model_name.upper()} Tokenization:")
            # Tokenize the text
            tokens = tokenizer.tokenize(text)
            encoded = tokenizer.encode(text)

            # Display results
            print(f"Number of tokens: {len(tokens)}")
            print("Tokens:", tokens)
            print(f"Token IDs: {encoded}")
            print("Decoded back:", tokenizer.decode(encoded))

            # Display special tokens
            print("\nSpecial tokens:")
            for token_name, token in tokenizer.special_tokens_map.items()
                print(f"{token_name}: {token}")

    def measure_performance(self, model_name: str, text: str, num_runs: i
        """Measures model performance for a given text."""
        if not self.models:
            print("No models loaded. Please load models first.")
            return
```

```
        if model_name not in self.models:
            print(f"Model {model_name} not found.")
            return

        model = self.models[model_name]
        tokenizer = self.tokenizers[model_name]

        if verbose:
            print(f"Testing {model_name}...", end=' ')

        # Performance measurements
        total_time = 0
        memory_usage = []

        # Prepare input
        inputs = tokenizer(text, return_tensors="pt", padding=True, trunc

        if model_name == 'xlnet':
            inputs['token_type_ids'] = torch.zeros_like(inputs['input_ids

        # Multiple runs for averaging
        for _ in range(num_runs):
            start_time = time.time()
            with torch.no_grad():
                outputs = model(**inputs)
            end_time = time.time()

            total_time += (end_time - start_time)
            memory_usage.append(psutil.Process().memory_info().rss / 1024

        avg_time = total_time / num_runs
        avg_memory = np.mean(memory_usage)

        self.results.append({
            'model': model_name,
            'avg_time_ms': round(avg_time * 1000, 2),
            'avg_memory_mb': round(avg_memory, 2),
            'parameters': sum(p.numel() for p in model.parameters()),
            'input_length': len(inputs['input_ids'][0])
        })

        if verbose:
            print("✓")

    def display_results(self):
        """Shows results in a DataFrame."""
        if not self.results:
            print("No results available. Please run performance tests fir
            return None

        df = pd.DataFrame(self.results)

        # Add relative speed comparison (normalized to BERT)
        if 'bert' in df['model'].values:
            bert_time = df[df['model'] == 'bert']['avg_time_ms'].values[0
            df['relative_speed'] = bert_time / df['avg_time_ms']

        return df
```

```
def main():
    print("Starting model comparison...")

    # Sample texts for analysis (English and Spanish)
    texts = [
        """This is a longer text that allows us to see how different mode
        with more extensive content. We want to analyze the differences i
        time and memory usage across various transformer architectures.""

        """Este es un texto más largo que nos permite ver cómo se comport
        modelos con contenido más extenso. Queremos analizar las diferenc
        de procesamiento y uso de memoria entre varias arquitecturas de t
    ]

    comparator = ModelComparator()
    models = ['distilbert', 'bert', 'roberta']

    # Load models
    print("\nLoading models:")
    for model in models:
        try:
            comparator.load_model(model)
        except Exception as e:
            print(f"❌ Skipped {model}: {str(e)}")

    # Show tokenization for each text
    print("\nComparing tokenization for English text:")
    comparator.show_tokenization(texts[0])

    print("\nComparing tokenization for Spanish text:")
    comparator.show_tokenization(texts[1])

    # Run performance tests for both texts
    print("\nRunning performance tests:")
    for i, text in enumerate(texts):
        print(f"\nTesting {'English' if i == 0 else 'Spanish'} text ({len
        for model in comparator.models:
            comparator.measure_performance(model, text)

    # Show results
    results = comparator.display_results()
    if results is not None:
        print("\nComparison Results:")
        print(results.to_string(index=False))

    return results

if __name__ == "__main__":
    main()
```

6. Model and Tokenizer Initialization

In [ ]:

```
# Initialize tokenizer and model
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased'
model = DistilBertForSequenceClassification.from_pretrained(
    'distilbert-base-uncased',  # "uncased" means it converts all text to
    num_labels=2  # Specifies this is a binary classification task
).to(device)
```

In [ ]:
```
# Visualize tokenization example
print("\n=== Tokenization Example ===")
sample_text = train_texts[0]
print(f"Original text: {sample_text}")
tokenized = tokenizer(sample_text, truncation=True, padding=True, return_
print(f"Words in text:    {len(sample_text.split())}")
print(f"Tokenized length: {len(tokenized['input_ids'][0])}")
print(f"Tokens: {tokenizer.convert_ids_to_tokens(tokenized['input_ids'][0
```

7. Dataset Creation

In [ ]:
```
# Tokenize all texts
train_texts_encoded = tokenizer(train_texts.tolist(), truncation=True, pa
val_texts_encoded = tokenizer(val_texts.tolist(), truncation=True, paddin
test_texts_encoded = tokenizer(test_texts.tolist(), truncation=True, padd
```

In [ ]:
```
# Create TensorDatasets
train_dataset = TensorDataset(
    torch.tensor(train_texts_encoded['input_ids']),
    torch.tensor(train_texts_encoded['attention_mask']),
    torch.tensor(train_labels)
)

val_dataset = TensorDataset(
    torch.tensor(val_texts_encoded['input_ids']),
    torch.tensor(val_texts_encoded['attention_mask']),
    torch.tensor(val_labels)
)

test_dataset = TensorDataset(
    torch.tensor(test_texts_encoded['input_ids']),
    torch.tensor(test_texts_encoded['attention_mask']),
    torch.tensor(test_labels)
)
```

In [ ]:
```
print("Datasets created successfully!")
print(f"Train dataset size:      {len(train_dataset)}")
print(f"Validation dataset size: {len(val_dataset)}")
print(f"Test dataset size:       {len(test_dataset)}")
```

In [ ]:
```
# Create dataloaders
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=T
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE)
```

In [ ]:
```
# Show batch structure
sample_batch = next(iter(train_loader))
print("\n=== Batch Structure ===")
print(f"Input IDs shape:      {sample_batch[0].shape}")
print(f"Attention mask shape: {sample_batch[1].shape}")
print(f"Labels shape:         {sample_batch[2].shape}")
```

8. Training Functions

In [ ]:
```
def evaluate_model(model, dataloader, device):
    """Evaluate the model and return loss and accuracy."""
    model.eval()
    total_loss = 0
    predictions = []
    true_labels = []

    with torch.no_grad():
        for batch in dataloader:
            input_ids, attention_mask, labels = [b.to(device) for b in ba
            outputs = model(input_ids=input_ids, attention_mask=attention

            total_loss += outputs.loss.item()
            preds = torch.argmax(outputs.logits, dim=-1)
            predictions.extend(preds.cpu().numpy())
            true_labels.extend(labels.cpu().numpy())

    accuracy = accuracy_score(true_labels, predictions) * 100
    avg_loss = total_loss / len(dataloader)

    return avg_loss, accuracy, predictions, true_labels
```

9. Training Loop

In [ ]:
```
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
history = {'train_loss': [], 'val_loss': [], 'val_acc': []}
best_val_acc = 0
```

In [ ]:
```
for epoch in range(NUM_EPOCHS):
    # Training
    model.train()
    total_loss = 0
    progress_bar = tqdm(train_loader, desc=f'Epoch {epoch+1}/{NUM_EPOCHS}

    for batch in progress_bar:
        input_ids, attention_mask, labels = [b.to(device) for b in batch]

        optimizer.zero_grad()
        outputs = model(input_ids=input_ids, attention_mask=attention_mas
        loss = outputs.loss
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        progress_bar.set_postfix({'loss': f'{loss.item():.4f}'})

    # Validation
    val_loss, val_acc, val_preds, val_true = evaluate_model(model, val_lo

    # Save metrics
    avg_train_loss = total_loss / len(train_loader)
    history['train_loss'].append(avg_train_loss)
    history['val_loss'].append(val_loss)
    history['val_acc'].append(val_acc)

    print(f"\nEpoch {epoch+1} Summary:")
    print(f"Average training loss: {avg_train_loss:.4f}")
```

```
    print(f"Validation loss:       {val_loss:.4f}")
    print(f"Validation accuracy:   {val_acc:.2f}%")
    print("\nValidation Classification Report:")
    print(classification_report(val_true, val_preds))

    # Save best model
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        torch.save(model.state_dict(), 'best_model.pt')
        print(f"New best model saved with accuracy: {val_acc:.2f}%")
```

10. Final Evaluation

In [ ]:
```
print("\n=== Final Evaluation ===")
test_loss, test_acc, test_preds, test_true = evaluate_model(model, test_l
print("\nTest Results:")
print(f"Test Loss:     {test_loss:.4f}")
print(f"Test Accuracy: {test_acc:.2f}%")
print("\nTest Classification Report:")
print(classification_report(test_true, test_preds))
```

11. Plot Training History

In [ ]:
```
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.plot(history['train_loss'], label='Training Loss', marker='o')
plt.plot(history['val_loss'], label='Validation Loss', marker='o')
plt.title('Loss vs. Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history['val_acc'], label='Validation Accuracy', marker='o')
plt.title('Validation Accuracy vs. Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

12. Save Model

In [ ]:
```
model_save_path = '../models/sentiment_model'
model.save_pretrained(model_save_path)
tokenizer.save_pretrained(model_save_path)
print(f"\nModel and tokenizer saved in {model_save_path}")
```

In [ ]:
```
if os.path.exists('best_model.pt'):
    os.remove('best_model.pt')
    print("Removed temporary checkpoint file (best_model.pt)")
```

In [ ]:
```
# Print final summary
print("\n=== Training Complete ===")
print(f"Best validation accuracy: {best_val_acc:.2f}%")
print(f"Final test accuracy:      {test_acc:.2f}%")
```