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
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
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from transformers import (
    DistilBertTokenizer, DistilBertModel, DistilBertForSequenceClassifica
    BertTokenizer, BertModel,
    RobertaTokenizer, RobertaModel,
    logging
)
```

In [ ]:
```
print("=== System Info ===")
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)}")
```

2. Exploring Datasets

In [ ]:
```
# Load one dataset to check columns
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("\nIMDB columns:", df_imdb.columns.tolist())
```

In [ ]:
```
def load_and_prepare_datasets():
    df_sample = pd.read_parquet('../data/sample_reviews.parquet')
    imdb = load_dataset("stanfordnlp/imdb")
    df_imdb = pd.DataFrame(imdb['train'])

    return {
        'Sample Reviews': (df_sample, 'sentence'),
        'IMDB': (df_imdb, 'text'),
    }

def compare_datasets(datasets):
    comparison = {}

    for name, (df, text_col) in datasets.items():
        stats = {
            'Total Reviews': len(df),
            'Memory Usage (MB)': df.memory_usage(deep=True).sum() / 1024*
            'Avg Words': df[text_col].str.split().str.len().mean(),
            'Max Words': df[text_col].str.split().str.len().max(),
            'Min Words': df[text_col].str.split().str.len().min(),
            'Avg Characters': df[text_col].str.len().mean(),
            'Label Distribution': df['label'].value_counts().to_dict(),
            'Columns': ', '.join(df.columns)
        }
        comparison[name] = stats

    return pd.DataFrame(comparison).round(2)

# Load and compare
datasets = load_and_prepare_datasets()
comparison_df = compare_datasets(datasets)
print("\nDataset Comparison:")
print(comparison_df)

# Print samples
print("\nSample Reviews head:")
print(df_sample.head())

print("\nIMDB head:")
print(df_imdb.head())
```

In [ ]:
```
datasets = {
   'Sample': df_sample,
   'IMDB': df_imdb,
}

for name, df in datasets.items():
    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))
```

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datasets import load_dataset

def analyze_and_plot_datasets():
    # Load datasets
    df_sample = pd.read_parquet('../data/sample_reviews.parquet')
    imdb = load_dataset("stanfordnlp/imdb")
    df_imdb = pd.DataFrame(imdb['train'])

    # Calculate word lengths
    df_sample['word_length'] = df_sample['sentence'].str.split().str.len(
    df_imdb['word_length'] = df_imdb['text'].str.split().str.len()

    # Create figure with subplots
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

    # Box plot of word lengths
    word_lengths = {
        'Sample': df_sample['word_length'],
        'IMDB': df_imdb['word_length'],
    }
    sns.boxplot(data=word_lengths, ax=ax1)
    ax1.set_title('Review Length Distribution')
    ax1.set_ylabel('Number of Words')

    # Label distribution
    labels = {
        'Sample': df_sample['label'].value_counts(normalize=True),
        'IMDB': df_imdb['label'].value_counts(normalize=True),
    }
    pd.DataFrame(labels).plot(kind='bar', ax=ax2)
    ax2.set_title('Label Distribution')
    ax2.set_ylabel('Proportion')

    plt.tight_layout()
    return fig

# Generate plots
fig = analyze_and_plot_datasets()
plt.show()
```

3. Data Loading and Preparation

Getting the datasets reviews from Hugging Face:

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

https://huggingface.co/datasets/SetFit/amazon_reviews_multi_en

In [ ]:
```
os.environ['HF_HUB_DISABLE_SYMLINKS_WARNING'] = '1'
```

In [ ]:
```
dataset_imdb = load_dataset("stanfordnlp/imdb")

df_imdb = pd.DataFrame(dataset_imdb['train'])
print(df_imdb.head())
print(dataset_imdb)
print(df_imdb.info())
```

In [ ]:
```
dataset_sample = pd.read_parquet("../data/sample_reviews.parquet")

print(dataset_sample.head())
print(dataset_sample.info())
```

SELECT THE DATASET TO TRAIN

In [ ]:
```
# data_name = pd.DataFrame(dataset_imdb['train'])
data_name = dataset_sample

data = data_name

data.head()
```

In [ ]:
```
data.info()
```

In [ ]:
```
print(data)
```

Add column con text_length info & Remove idx column

In [ ]:
```
def add_length_column(data):
    if 'idx' in data.columns:
        data.drop('idx', axis=1, inplace=True)
    if 'text' in data.columns:
        data.rename(columns={'text': 'sentence'}, inplace=True)
    data['text_len'] = data['sentence'].str.len()
    return data

data = add_length_column(data)
print(data)
```

In [ ]:
```
data.describe()
```

Reduce the size of the dataset

In [ ]:
```
SAMPLE_SIZE = 100
MAX_LENGTH = 500
```

In [ ]:
```
# Filter by length
data = data[data['text_len'] <= MAX_LENGTH]

# Balance classes
samples_per_class = SAMPLE_SIZE // len(data['label'].unique())
data = pd.concat([
   data[data['label'] == label].sample(n=samples_per_class, random_state=
   for label in data['label'].unique()
])
```

```
print("\nClass distribution in Train:")
print(data['label'].value_counts())
```

In [ ]:
```
data.head()
```

In [ ]:
```
data.info()
```

In [ ]:
```
data.describe()
```

3. Model and Platform Research and Selection

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
- QBraid
- Google Colab

In [ ]:
```
logging.set_verbosity_error()
warnings.filterwarnings('ignore')

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 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
    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.""
    ]

    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)}")

    # Run tests
    print("\nRunning performance tests:")
    for text in texts:
        print(f"Testing with text of {len(text)} characters")
        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()
```

4. Model and Tokenizer Initialization

In [ ]:
```
class SentimentDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=512):
        self.encodings = tokenizer(texts, truncation=True, padding=True,
                                   max_length=max_length, return_tensors='p
        self.labels = torch.tensor(labels)

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        item = {key: val[idx] for key, val in self.encodings.items()}
        item['labels'] = self.labels[idx]
        return item
```

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

In [ ]:
```
# Initialize tokenizer and model
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased'
model = DistilBertForSequenceClassification.from_pretrained('distilbert-b
model.to(device)
```

In [ ]:
```
# Set hyperparameters
BATCH_SIZE = 8
LEARNING_RATE = 0.0005
NUM_EPOCHS = 3
```

In [ ]:
```
TRAIN_SIZE = 0.70
VAL_SIZE = 0.15
TEST_SIZE = 0.15
RANDOM_STATE = 42

data_texts = data['sentence'].values
data_labels = data['label'].values

temp_texts, test_texts, temp_labels, test_labels = train_test_split(
    data_texts,
    data_labels,
    test_size=TEST_SIZE,
    random_state=RANDOM_STATE
)

train_texts, val_texts, train_labels, val_labels = train_test_split(
    temp_texts,
    temp_labels,
    test_size=VAL_SIZE/(TRAIN_SIZE + VAL_SIZE),
    random_state=RANDOM_STATE
)

train_texts = train_texts.tolist() # convert the arrays to lists
val_texts = val_texts.tolist()
```

```
test_texts = test_texts.tolist()

train_dataset = SentimentDataset(train_texts, train_labels, tokenizer)
val_dataset = SentimentDataset(val_texts, val_labels, tokenizer)
test_dataset = SentimentDataset(test_texts, test_labels, tokenizer)
```

In [ ]:
```
# Create DataFrames
train_df = pd.DataFrame({'Set': 'Train', 'Label': train_labels})
val_df = pd.DataFrame({'Set': 'Val', 'Label': val_labels})
test_df = pd.DataFrame({'Set': 'Test', 'Label': test_labels})

# Combine and display counts
df_all = pd.concat([train_df, val_df, test_df])
print("\nLabel Distribution per Set:")
print(df_all.groupby(['Set', 'Label']).size().unstack())

# Batch information
print(f"\nBatch Details (BATCH_SIZE={BATCH_SIZE}):")
print(f"Train batches: {len(train_loader)}")
print(f"Val batches: {len(val_loader)}")
print(f"Test batches: {len(test_loader)}")

print("\nSequence lengths:")
print(f"Train max length: {max([len(x.split()) for x in train_texts])}")
print(f"Val max length: {max([len(x.split()) for x in val_texts])}")
print(f"Test max length: {max([len(x.split()) for x in test_texts])}")
```

In [ ]:
```
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 [ ]:
```
# Initialize optimizer
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
```

5. Data manipulation

In [ ]:
```
# Extract texts and labels from the balanced datasets
data_texts = data['sentence'].tolist()
data_labels = data['label'].tolist()

# Split training data into train and validation sets
train_texts, val_texts, train_labels, val_labels = train_test_split(
    data_texts, data_labels, test_size=0.2, random_state=42
)

# Prepare test data
test_texts = test_balanced['text'].tolist()
test_labels = test_balanced['label'].tolist()

# Now create the datasets
train_dataset = SentimentDataset(train_texts, train_labels, tokenizer)
val_dataset = SentimentDataset(val_texts, val_labels, tokenizer)
test_dataset = SentimentDataset(test_texts, test_labels, tokenizer)

# 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)

# Initialize optimizer
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
```

6. Training and Evaluation Functions

In [ ]:
```
def evaluate_model(model, dataloader, device):
    model.eval()
    total_loss = 0
    correct = 0
    total = 0

    with torch.no_grad():
        for batch in dataloader:
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)

            total_loss += outputs.loss.item()
            predictions = torch.argmax(outputs.logits, dim=-1)
            correct += (predictions == batch['labels']).sum().item()
            total += len(batch['labels'])

    return total_loss / len(dataloader), correct / total * 100
```

In [ ]:
```
best_accuracy = 0

for epoch in range(NUM_EPOCHS):
    model.train()
    total_loss = 0
    correct = 0
    total = 0

    # Training
    train_loop = tqdm(train_loader, desc=f'Epoch {epoch+1}/{NUM_EPOCHS}')
    for batch in train_loop:
        batch = {k: v.to(device) for k, v in batch.items()}

        optimizer.zero_grad()
        outputs = model(**batch)
        loss = outputs.loss

        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        predictions = torch.argmax(outputs.logits, dim=-1)
        correct += (predictions == batch['labels']).sum().item()
        total += len(batch['labels'])

        train_loop.set_postfix({'loss': loss.item(),
                                'accuracy': 100 * correct / total})

    # Validation
    val_loss, val_accuracy = evaluate_model(model, val_loader, device)

    print(f'\nEpoch {epoch+1}:')
    print(f'Training Loss: {total_loss/len(train_loader):.4f}')
```

```
    print(f'Training Accuracy: {100*correct/total:.2f}%')
    print(f'Validation Loss: {val_loss:.4f}')
    print(f'Validation Accuracy: {val_accuracy:.2f}%')

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

9. Test the Model

In [ ]:
```
# Test final model
test_loss, test_accuracy = evaluate_model(model, test_loader, device)
print(f'\nFinal Test Results:')
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Accuracy: {test_accuracy:.2f}%')
```

10. Save the Model

In [ ]:
```
model.save_pretrained('./models/sentiment_model')
tokenizer.save_pretrained('./models/sentiment_model')
print("Model saved in './modles/sentiment_model' directory")
```