Fraud Email using BERT Classification

Notebook Overview

This notebook is designed for the following steps:

- **Loading and Cleaning Data:**
  Import email data (for example, a CSV file) containing emails and corresponding fraud labels, then clean the text using regular expressions.
- **Tokenization:**
  Use a BERT tokenizer to convert raw email text into token IDs and attention masks that BERT requires.
- **Dataset and DataLoader Construction:**
  Create a PyTorch dataset class that feeds tokenized inputs to the model and a DataLoader for batching.
- **Model Definition and Initialization:**
  Use a Hugging Face pre-trained BERT model (typically `BertForSequenceClassification`) with a classification head. This model will be fine-tuned for binary classification (fraudulent vs. non-fraudulent).
- **Training Loop:**
  Define the optimizer, learning rate scheduler, and loss function, and then run the training loop over several epochs. In each epoch, the notebook performs both training and evaluation.
- **Inference and Model Saving:**
  Save the model after training and include a helper function for making predictions on new email text.

Detailed Explanation (Cell by Cell)

Cell 1: Importing Libraries

python
Copy code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
import re
from transformers import BertTokenizer, BertForSequenceClassification,
get_linear_schedule_with_warmup
```

- **Torch and submodules:**
  Provide tensor operations, neural network modules, and optimization routines.
- **Dataset/DataLoader:**
  Used to construct custom datasets and to batch data during training.
- **Pandas and NumPy:**
  For handling structured data (CSV files) and general numeric operations.
- **Matplotlib:**
  For visualization (e.g., plotting loss curves).
- **Regular Expressions (re):**
  Helps with cleaning and preprocessing text.
- **Transformers Modules:**
  The Hugging Face library offers the BERT tokenizer and a pre-built model (`BertForSequenceClassification`), as well as a scheduler for managing the learning rate during training.

Cell 2: Data Loading and Preprocessing

python
Copy code

```
# Load the dataset
df = pd.read_csv('fraud_emails.csv')

# Display the first few rows to verify the data
print(df.head())

# Define a function for cleaning email text
def clean_text(text):
    text = text.lower()               # Convert text to lowercase for
consistency
    text = re.sub(r'\s+', ' ', text)   # Replace multiple spaces with
a single space
    text = re.sub(r'\W', ' ', text)    # Remove any non-word
```

```
characters (punctuation, etc.)
    return text.strip()               # Strip leading/trailing
whitespace

# Clean the email text column
df['cleaned_text'] = df['email_text'].apply(clean_text)
```

- **Loading Data:**
  The CSV file (here assumed to be named `fraud_emails.csv`) is read into a Pandas DataFrame.
- **Previewing:**
  A quick print of the first few rows confirms that the file was loaded correctly.
- **Cleaning:**
  The function `clean_text` standardizes the email text by lowering the case, removing extra spaces, and eliminating non-alphanumeric characters. This step is crucial to reduce noise in the input text.
- **Applying Cleaning:**
  The cleaned text is stored in a new column (`cleaned_text`) in the DataFrame.

Cell 3: Tokenization and Creating the Dataset

python
Copy code

```
# Initialize the BERT tokenizer (using a model like 'bert-base-
uncased')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Create a custom dataset class for our emails
class FraudEmailDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

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

    def __getitem__(self, idx):
```

```
        text = self.texts[idx]
        label = self.labels[idx]

        # Tokenize the text and encode it to a fixed max length
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,      # Adds CLS and SEP tokens as
needed
            max_length=self.max_length,
            truncation=True,              # Truncates sentences longer
than max_length
            padding='max_length',         # Pads sentences shorter
than max_length
            return_attention_mask=True,
            return_tensors='pt'           # Return as PyTorch tensors
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'label': torch.tensor(label, dtype=torch.long)
        }

# Split the data into training and validation sets (80/20 split)
from sklearn.model_selection import train_test_split
train_texts, val_texts, train_labels, val_labels = train_test_split(
    df['cleaned_text'].tolist(),
    df['label'].tolist(),
    test_size=0.2,
    random_state=42
)

# Create instances of the dataset
train_dataset = FraudEmailDataset(train_texts, train_labels,
tokenizer)
val_dataset   = FraudEmailDataset(val_texts, val_labels, tokenizer)

# Create DataLoader objects
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
```

```
val_loader = DataLoader(val_dataset, batch_size=16)
```

- **Tokenizer Initialization:**
  The `BertTokenizer` converts raw text to token IDs; the chosen model is typically `bert-base-uncased` for English texts.
- **Custom Dataset Definition:**
  The `FraudEmailDataset` class stores the email text, labels, and the tokenizer.
  - The `__getitem__` method tokenizes each text and ensures the sequence is of a fixed length (e.g., 128 tokens) by truncating or padding.
  - The returned dictionary contains flattened `input_ids`, `attention_mask`, and a label.
- **Train/Validation Split:**
  The data is divided into training and validation sets using scikit-learn's `train_test_split` function.
- **DataLoader Creation:**
  DataLoader objects handle batching and shuffling during training and evaluation.

Cell 4: Model Initialization

python
Copy code

```
# Instantiate a pre-built BERT classifier
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
```

- **Model Loading:**
  `BertForSequenceClassification` is loaded with a pre-trained BERT model and automatically creates a classification head.
  - `num_labels=2` indicates binary classification (fraudulent vs. non-fraudulent).
- **Device Setup:**
  The model is transferred to a GPU if available; otherwise, it runs on the CPU.

Cell 5: Optimizer, Scheduler, and Loss Function

python
Copy code

```
optimizer = optim.AdamW(model.parameters(), lr=2e-5,
correct_bias=False)

epochs = 3
total_steps = len(train_loader) * epochs

scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0,
    num_training_steps=total_steps
)

loss_fn = nn.CrossEntropyLoss()
```

- **Optimizer:**
  AdamW is chosen because it works well with transformer models. The learning rate (2e-5) is common for fine-tuning BERT.
- **Learning Rate Scheduler:**
  A linear scheduler gradually decreases the learning rate during training, computed over the total number of steps.
- **Loss Function:**
  CrossEntropyLoss is used because it is standard for classification tasks.

Cell 6: Training and Evaluation Loop

python
Copy code

```
for epoch in range(epochs):
    model.train()  # Activate training mode (enables dropout, etc.)
    total_train_loss = 0

    # Training loop over batches
    for batch in train_loader:
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['label'].to(device)
```

```
        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels
        )
        loss = outputs.loss  # Model output includes the loss when
labels are provided
        total_train_loss += loss.item()

        loss.backward()
        optimizer.step()
        scheduler.step()

    avg_train_loss = total_train_loss / len(train_loader)
    print(f"Epoch {epoch+1}: Training Loss = {avg_train_loss}")

    # Evaluate on validation data
    model.eval()
    total_val_loss = 0
    correct_predictions = 0

    with torch.no_grad():
        for batch in val_loader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['label'].to(device)

            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask,
                labels=labels
            )
            loss = outputs.loss
            total_val_loss += loss.item()
            logits = outputs.logits
            preds = torch.argmax(logits, dim=1)
            correct_predictions += torch.sum(preds == labels)

    avg_val_loss = total_val_loss / len(val_loader)
```

```
    val_accuracy = correct_predictions.double() / len(val_dataset)
    print(f"Epoch {epoch+1}: Validation Loss = {avg_val_loss},
Accuracy = {val_accuracy}")
```

- **Epoch Loop:**
  For each epoch, the model is set into training mode.
- **Batch Training:**
  For each batch:
  - Gradients are reset with `optimizer.zero_grad()`.
  - Input data (input IDs and attention masks) and labels are moved to the device.
  - The model is called with inputs and labels; since labels are provided, the model returns a loss value.
  - Backpropagation is performed and optimizer as well as scheduler are stepped.
- **Logging:**
  After processing all batches, the average training loss is calculated and printed.
- **Validation Loop:**
  The model is switched to evaluation mode (disables dropout). For each batch in the validation set:
  - The loss is calculated and predictions (argmax of logits) are compared to labels.
  - Overall accuracy is computed and printed.

Cell 7: Saving the Model and Inference

python
Copy code

```
# Save the trained model in a format that can be reloaded later
model.save_pretrained('fraud_email_bert_model')

# Define a function for predicting the fraud probability for a new
email
def predict_email(email_text):
    model.eval()
    # Clean the email text using the same function as during
preprocessing
    text_clean = clean_text(email_text)
    encoding = tokenizer.encode_plus(
        text_clean,
        add_special_tokens=True,
        max_length=128,
```

```
        truncation=True,
        padding='max_length',
        return_attention_mask=True,
        return_tensors='pt'
    )
    input_ids = encoding['input_ids'].to(device)
    attention_mask = encoding['attention_mask'].to(device)
    outputs = model(input_ids=input_ids,
attention_mask=attention_mask)
    logits = outputs.logits
    probabilities = torch.softmax(logits, dim=1)
    return probabilities.detach().cpu().numpy()

# Example inference on a sample email
sample_email = "Congratulations! You have won a lottery. Please send
your bank details."
print(predict_email(sample_email))
```

- **Saving the Model:**
  The model weights and configuration are saved using the `save_pretrained` method so that you can later reload with `BertForSequenceClassification.from_pretrained(...)`.
- **Inference Function:**
  A function `predict_email` is defined to:
  - Put the model into evaluation mode.
  - Clean and tokenize the input email.
  - Produce the model's output logits and convert them to probabilities.
  - Return the probabilities as a NumPy array.
- **Running an Inference Example:**
  A sample email string is passed to the function and the probabilities are printed, showing the model's output (e.g., the likelihood the email is fraudulent).