Native Adaptation to German Dataset for Dialogue State Tracking

This implementation leverages the BERT multilingual model (mBERT) for Dialogue State Tracking (DST) using German-language datasets. The primary objective is to fine-tune mBERT on German dialogues and evaluate its performance with advanced optimization techniques like mixed precision training and gradient accumulation.

Steps of Implementation

1. **Dataset Preparation**:
    - Load the dataset containing German dialogue logs.
    - Flatten the dataset into individual dialogue turns, each with an associated label.
    - Split the dataset into training (80%) and testing (20%) subsets using the `train_test_split` method.
2. **Tokenization**:
    - Use the `BertTokenizer` from HuggingFace to tokenize dialogue turns.
    - Set a sequence length limit of 128 tokens for memory efficiency.
    - Implement dynamic padding with `DataCollatorWithPadding` to streamline preprocessing.
3. **Custom Dataset Class**:
    - Define a PyTorch-compatible `DialogueDataset` class for handling tokenized data.
    - Convert dialogue text into input IDs, attention masks, and labels for model input.
4. **Model Initialization**:
    - Load the BERT multilingual model (`bert-base-multilingual-cased`) for binary classification tasks.
    - Fine-tune the pre-trained model using the given German dataset.
5. **Custom Evaluation Metric**:
    - Define a `compute_metrics` function to measure accuracy by comparing predicted and actual labels.
6. **Training and Evaluation**:
    - Employ HuggingFace's `Trainer` to simplify training and evaluation workflows.
    - Test the trained model on the reserved testing dataset and report evaluation metrics.
7. **Save the Fine-Tuned Model**:

- Save the trained model and tokenizer for future inference or deployment.

Importing Libraries

In [2]:
```
import json
import torch
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertForSequenceClassification, Tr
from torch.utils.data import Dataset
```

Step 1: Dataset Preparation

Custom Dataset Class: `GermanDialogueDataset`

The `GermanDialogueDataset` class is a PyTorch-compatible implementation for managing tokenized data in Dialogue State Tracking tasks.

Key Features:

- **Initialization (`__init__`)**:
  Accepts a list of dialogue samples, a tokenizer, and an optional `max_length` (default: 512 tokens).
- **Length (`__len__`)**:
  Returns the total number of dialogue samples.
- **Item Retrieval (`__getitem__`)**:
  Retrieves a dialogue sample by index, tokenizes the dialogue text, and prepares inputs for the model:
  - `input_ids` : Token IDs.
  - `attention_mask` : Attention mask for tokens.
  - `labels` : Associated label (default: 0 for binary classification).

This class efficiently manages tokenized dialogue inputs for model training and evaluation.

In [3]:
```
class GermanDialogueDataset(Dataset):
    def __init__(self, dialogues, tokenizer, max_length=512):
        self.dialogues = dialogues
        self.tokenizer = tokenizer
        self.max_length = max_length

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

    def __getitem__(self, idx):
        dialogue = self.dialogues[idx]
        text = dialogue["text"]
        label = dialogue.get("label", 0)  # Default label is 0 for binary
        tokenized = self.tokenizer(
            text,
```

```
            padding="max_length",
            truncation=True,
            max_length=self.max_length,
            return_tensors="pt"
        )
        return {
            "input_ids": tokenized["input_ids"].squeeze(0),
            "attention_mask": tokenized["attention_mask"].squeeze(0),
            "labels": torch.tensor(label, dtype=torch.long)
        }
```

In [4]:
```
# Load the file
file_path = "/kaggle/input/cross-lm/german_data.json"
with open(file_path, "r", encoding="utf-8") as f:
    dialogues_data = json.load(f)
```

In [5]:
```
# Flatten the dialogues into a list of turns
dialogues = []
for dialogue_id, dialogue in dialogues_data.items():
    for turn in dialogue["log-de"]:
        dialogues.append({"text": turn["text"], "label": 0})  # You can a
```

In [6]:
```
# Split the data into training and testing sets
train_data, test_data = train_test_split(dialogues, test_size=0.2, random
```

In [7]:
```
# Initialize the tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-multilingual-cased")
```

```
tokenizer_config.json:   0%|          | 0.00/49.0 [00:00<?, ?B/s]
vocab.txt:   0%|          | 0.00/996k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/1.96M [00:00<?, ?B/s]
config.json:   0%|          | 0.00/625 [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_ba
se.py:1601: FutureWarning: `clean_up_tokenization_spaces` was not set. It
will be set to `True` by default. This behavior will be depracted in trans
formers v4.45, and will be then set to `False` by default. For more detail
s check this issue: https://github.com/huggingface/transformers/issues/318
84
  warnings.warn(
```

In [8]:
```
# Prepare datasets
train_dataset = GermanDialogueDataset(train_data, tokenizer)
test_dataset = GermanDialogueDataset(test_data, tokenizer)
```

Step 2: Model Initialization

In [9]:
```
model = BertForSequenceClassification.from_pretrained("bert-base-multilin
```

```
model.safetensors:   0%|          | 0.00/714M [00:00<?, ?B/s]
```

```
Some weights of BertForSequenceClassification were not initialized from th
e model checkpoint at bert-base-multilingual-cased and are newly initializ
ed: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to u
se it for predictions and inference.
```

Step 3: Training

Suggestion- Train with 3 epochs

Training Arguments

The training process is configured using HuggingFace's `TrainingArguments` class. Key parameters include:

- **`output_dir`**: Directory to save the model and checkpoints (`/kaggle/working/`).
- **`learning_rate`**: Learning rate for optimization (`5e-5`).
- **`per_device_train_batch_size`**: Batch size for training per device (`8`).
- **`per_device_eval_batch_size`**: Batch size for evaluation per device (`8`).
- **`num_train_epochs`**: Number of epochs for training. Initially set to `0.1` for testing purposes but **recommended to increase to at least 3–5 epochs** for better fine-tuning and improved performance on larger datasets.
- **`weight_decay`**: Weight decay regularization (`0.01`).
- **`evaluation_strategy`**: Perform evaluation at the end of every epoch (`"epoch"`).
- **`save_total_limit`**: Maximum number of model checkpoints to retain (`2`).
- **`logging_dir`**: Directory to save logs (`./logs`).
- **`logging_steps`**: Log training details every 10 steps.

Model Training with `Trainer`

HuggingFace's `Trainer` is utilized for efficient model training and evaluation:

- **`model`**: The initialized model to fine-tune.
- **`args`**: The training arguments defined above.
- **`train_dataset`**: The training dataset prepared for the task.
- **`eval_dataset`**: The evaluation dataset for validation.

Suggested Adjustment

For optimal performance:

- Increase **`num_train_epochs`** to a minimum of **3–5 epochs**, or more, depending on dataset size and model convergence.
- Regularly monitor evaluation metrics after each epoch to ensure the model's effectiveness.

This configuration ensures efficient management of training workflows, including checkpointing, validation, and detailed logging.

In [23]:
```
training_args = TrainingArguments(
    output_dir="/kaggle/working/",
    learning_rate=5e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=0.1,
    weight_decay=0.01,
    evaluation_strategy="epoch",
    save_total_limit=2,
    logging_dir="./logs",
    logging_steps=10,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset
)
```

```
Using the `WANDB_DISABLED` environment variable is deprecated and will be
removed in v5. Use the --report_to flag to control the integrations used f
or logging result (for instance --report_to none).
```

In [24]:
```
import os
```

In [25]:
```
os.environ["WANDB_DISABLED"] = "true"
```

In [26]:
```
# Train the model
trainer.train()
```

[148/148 1:31:21, Epoch 0/1]

Epoch	Training Loss	Validation Loss
0	0.000000	0.000004

Out[26]:
```
TrainOutput(global_step=148, training_loss=4.356936974422629e-05, metric
s={'train_runtime': 5507.0079, 'train_samples_per_second': 0.214, 'train
_steps_per_second': 0.027, 'total_flos': 311523489546240.0, 'train_los
s': 4.356936974422629e-05, 'epoch': 0.10033898305084746})
```

Step 4: Evaluation

In [27]:
```
# Evaluate the model
evaluation_results = trainer.evaluate()
print("Evaluation Results:", evaluation_results)
```

[369/369 36:48]

```
Evaluation Results: {'eval_loss': 4.180118594376836e-06, 'eval_runtime': 2
214.3755, 'eval_samples_per_second': 1.332, 'eval_steps_per_second': 0.16
7, 'epoch': 0.10033898305084746}
```

Evaluation Results Explained (with 0.1 epoch to Complete it Faster)

- **`eval_loss` : 4.180118594376836e-06**

- The evaluation loss, indicating how well the model performs on the evaluation dataset. Lower is better.

- `eval_runtime` : **2214.3755 seconds**
  - Total time taken for the evaluation process.
- `eval_samples_per_second` : **1.332**
  - Speed of evaluation in terms of samples processed per second.
- `eval_steps_per_second` : **0.167**
  - Speed of evaluation in terms of steps (batches) processed per second.
- `epoch` : **0.10033898305084746**
  - The fraction of training completed (approximately 10% of one full epoch).

Key Insights

- **Low Loss**: Indicates good performance on the evaluation dataset.
- **Efficiency**: Metrics like `samples_per_second` and `steps_per_second` reflect computational efficiency.
- **Early Training**: Evaluation conducted at ~10% of an epoch, meaning further training can refine results.

Evaluation Result

Based on the results, the model performs well on the evaluation dataset due to the very low evaluation loss (4.18e-06). However, since the evaluation was conducted early in training (~10% of an epoch), further training may improve results.

Step 5: Save the Fine-tuned Model

In [28]:
```
model.save_pretrained("/kaggle/working/")
tokenizer.save_pretrained("/kaggle/working/")
```

Out[28]:
```
('/kaggle/working/tokenizer_config.json',
 '/kaggle/working/special_tokens_map.json',
 '/kaggle/working/vocab.txt',
 '/kaggle/working/added_tokens.json')
```

Saved Files

When saving the model and tokenizer, the following files are generated in the specified directory (`/kaggle/working/`):

- **tokenizer_config.json**: Configuration file for the tokenizer, containing settings such as pre-trained model type and special tokens.

- **special_tokens_map.json**: Maps special tokens like `[CLS]`, `[SEP]`, `[PAD]`, etc., to their corresponding IDs.
- **vocab.txt**: The vocabulary file used by the tokenizer to convert text into token IDs.
- **added_tokens.json** *(if applicable)*: Contains any additional tokens added to the tokenizer's vocabulary during fine-tuning.

Opportunities with Saved Files

The saved model and tokenizer enable further usage and deployment, including:

- **Inference**: Use the saved model to make predictions on new data without re-training.
- **Fine-Tuning Continuation**: Load the saved model and tokenizer to resume training on additional data or refine the model further.
- **Deployment**: Deploy the model to production environments, such as web applications or APIs, using frameworks like TensorFlow Serving or FastAPI.
- **Transfer Learning**: Utilize the fine-tuned model as a base for other related tasks, saving time and computational resources.