Federated Learning Project Report

Katta Sathwik
20mcme16

December 29, 2023

Contents

Abstract

This report presents a detailed analysis of a Log Anomaly Detection using federated learning project focusing on sequence classification using a fine-tuned BERT model. The project involves client-server communication using Flower framework for federated learning. This report outlines the key concepts and tasks involved in the implementation.

1 Introduction

Federated learning is a distributed machine learning approach that allows training models across decentralized devices or servers. In this project, I have used the Flower framework for federated learning to implement the solution for sequence classification. The model used is a fine-tuned BERT (Bidirectional Encoder Representations from Transformers) model, a state-of-the-art transformer-based architecture.

2 Objective

The main objective of the project is to implement a federated learning system for sequence classification. The client-side code defines a Flower client using NumPyClient. The client utilizes a custom BERT model for sequence classification and a tokenizer for text encoding. The model is fine-tuned on a dataset and saved for later use.

3 Explanation of Model

The implementation of federated learning using the Flower framework is explained. Here are the key points:

3.1 Starting the Flower Server

1. `fl.server.start_server` is called to start the Flower server. It listens on a specific address and port (localhost and the port specified as a command-line argument).

2. `ServerConfig` is used to configure the server with the number of federated learning rounds (`num_rounds`) and the maximum message length.

3.2 Flower Client Implementation

1. Initially, import the necessary libraries for the implementation:

    - `flwr` for federated learning.
    - `tensorflow` and `keras` for machine learning and neural network models.

- `sys` to access command-line arguments.
- `numpy` for numerical operations.

2. Define auxiliary methods:

   - `getDist`: Meant for visualizing data class distribution.
   - `getData`: Customize the data distribution.

3. Load and compile the Keras model:

   - Define a neural network model using the Keras Sequential API. It's a simple feedforward model with three layers: input, hidden, and output layers.
   - Compile the model with an optimizer (`"adam"`), a loss function (`"sparse_categorical_crossentropy"`), and a metric to track (`"accuracy"`).

4. Load the dataset:

   - Load the Fashion MNIST dataset using TensorFlow's built-in dataset loader.
   - Normalize the images to values between 0 and 1.
   - Specify a data distribution (`dist`) for Fashion MNIST. This distribution controls how many samples are included from each class (0-9).

5. Define the Flower client:

   - Define a custom Flower client class, `CustomClient`, which extends `fl.client.NumPyClient`.
   - Implement three main methods:
     (a) `get_parameters`: Returns the current model's weights.
     (b) `fit`: Used for model training. Receives model parameters, updates the model, and fits the model to the training data for one epoch. Returns the updated model parameters and the number of training samples used.
     (c) `evaluate`: Evaluates the model on the test data and returns the loss and accuracy.

6. Start the Flower client:

   - `fl.client.start_numpy_client` is used to start the Flower client.
   - It connects to the Flower server at the specified address.
   - Use the custom client class (`CustomClient`), and set the maximum message length to ensure compatibility with the server.

3.3 Fashion MNIST Dataset

Fashion MNIST is a dataset commonly used in machine learning and computer vision tasks, often serving as a drop-in replacement for the original MNIST dataset. Instead of handwritten digits, Fashion MNIST contains grayscale images of various clothing items and accessories. Here's an explanation of the Fashion MNIST dataset:

3.3.1 Images

Fashion MNIST consists of a collection of 28×28-pixel grayscale images. Each image represents one of ten different fashion categories.

3.3.2 Classes

There are ten classes, each corresponding to a specific type of clothing item or accessory. These classes are as follows:

1. T-shirt/top
2. Trouser
3. Pullover
4. Dress
5. Coat
6. Sandal
7. Shirt
8. Sneaker
9. Bag
10. Ankle boot

3.3.3 Size

The dataset is relatively small compared to some other image datasets. It includes 60,000 training images and 10,000 test images.

4 Basic Implementing a Federated Learning Setup using Terminals as Clients with FashionMNIST Dataset

4.1 Starting the Flower Server

To start the Flower server, use the following command in the terminal:

```
fl.server.start_server --server_address="127.0.0.1:5012" --
    num_rounds=5 --max_msg_len=1000000
```

Listing 1: Starting the Flower Server

4.2 Starting Flower Clients

Open multiple terminals and run the Flower clients using the following command:

```
python fashion_mnist_client.py --server_address="
    127.0.0.1:5012"
```

Listing 2: Starting Flower Clients

Ensure that Flower clients are running on different terminals to simulate a federated learning environment.

5 Federated client for aggregation

5.1 Importing the necessary libraries

These lines import necessary libraries and modules for working with PyTorch, Flower (Federated Learning framework), Hugging Face Transformers (for BERT), and other utilities.

```
from dataclasses import dataclass
from typing import Any, Dict, List, Union
import torch
import flwr as fl
from transformers import BertForSequenceClassification,
    BertTokenizer, Trainer, TrainingArguments
from transformers import AdamW,
    get_linear_schedule_with_warmup
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import zipfile
import pandas as pd
import shutil
import os
from sklearn.model_selection import train_test_split
```

Listing 3: Hdfs Client for aggregation

5.2 Custom Dataset Class:

This class defines a custom dataset for PyTorch. It takes encodings (tokenized input sequences) and labels and provides methods for getting items and the length of the dataset.

```python
class CustomDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor([self.labels[idx], 1 - self.labels[idx]]).float()
        return item

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

5.3 Initialization and Data Preprocessing:

This section initializes the BERT model and tokenizer and loads a labeled dataset from a CSV file. It selects a subset of the dataset (rows 1 to 200) and preprocesses the data.

```python
model_saved = BertForSequenceClassification.from_pretrained("bert-base-uncased")
tokenizer_saved = BertTokenizer.from_pretrained("bert-base-uncased")
model_folder = "fine_tuned_bert_model_for_HDFS"
pickle_file = "pickle_fine_tuned_bert_model_for_HDFS.pickle"
df = pd.read_csv('Hdfs_labelled_sequence.csv', sep=',', quotechar='"', names=["text", "label"])
df = df[1:200]
X = list(df['text'])
y = pd.get_dummies(df['label'], drop_first=True)['Normal'].values
y = y.astype(int)
```

5.4 Model Saving and Loading:

This part saves the initial BERT model's weights and then loads the saved weights from a ZIP file. It ensures that the model and tokenizer are initialized with the same state as before saving.

```python
torch.save(model_saved.state_dict(), os.path.join(model_folder, "model_weights.pth"))
shutil.make_archive(model_folder, 'zip', model_folder)
with zipfile.ZipFile(f"{model_folder}.zip", "r") as zip_ref:
    with zip_ref.open("model_weights.pth", "r") as file:
        model_weights = torch.load(file)
model_saved = BertForSequenceClassification.from_pretrained("bert-base-uncased")
```

```
model_saved.load_state_dict(model_weights)
tokenizer_saved = BertTokenizer.from_pretrained("bert-base-
    uncased")
```

5.5 Training Arguments and Trainer Initialization:

Here, training arguments for the Trainer are defined, specifying various parameters like batch size, evaluation steps, etc. The Trainer is then initialized with the BERT model, training arguments, and training and evaluation datasets.

```
training_args = TrainingArguments(
        output_dir="./bert_base_model",
        evaluation_strategy="steps",
        eval_steps=100,
        per_device_train_batch_size=2,
        per_device_eval_batch_size=2,
        save_steps=1000,
        save_total_limit=2,
        num_train_epochs=10,
        logging_dir="./logs",
    )
trainer = Trainer(
    model=model_saved,
    args=training_args,
    train_dataset=dataset_train,
    eval_dataset=test_dataset,
    data_collator=None
)
```

5.6 FlowerClient Class:

This class is a custom client for federated learning using the Flower framework. It has methods for getting parameters, fitting the model, and evaluating the model, among others.

```
class FlowerClient(fl.client.NumPyClient):
    ...
```

5.7 Fitting the Model (Training):

loop runs the training process for a specified number of epochs. It iterates through batches of the local dataset, computes the loss, performs backpropagation, and updates the model parameters.

```
for epoch in range(3):
    for batch in DataLoader(test_dataset, batch_size=2,
    shuffle=True):
        ...
```

5.8 Evaluating the Model:

This loop runs the evaluation process on the local dataset. It computes the loss and evaluates the model's performance.

```
for batch in DataLoader(test_dataset, batch_size=2, shuffle=False):
    ...
```

5.9 Printing Fit History and Evaluation Metrics:

These print statements provide information on the fit history (training loss) and evaluation metrics (evaluation loss and global evaluation accuracy).

```
print("Fit history: {'loss':", loss.item(), "}")
print("Evaluation metrics: {'eval_loss':", average_loss, "}")
print("Now the global Eval accuracy:", 1.0 - average_loss)
```

5.10 Flower Client Initialization and Execution:

The script starts the Flower client, connecting it to a server address and using the custom FlowerClient for federated learning.

```
if __name__ == '__main__':
    fl.client.start_numpy_client(
        server_address="127.0.0.1:5020",
        client=FlowerClient(),
    )
```

6 HDFS Trained Model Aggregation With Federated Learning

Initially, I have worked on the HDfs trained model in order to aggregate the model using the federated learning, which automatically works for the aggregating three trained models but that was taking too long and more ram even if take a small amount of data for preprocessing.

7 Conclusion

This report demonstrates federated learning with BERT using the Flower framework. It includes data preprocessing, model saving/loading, training, and evaluation in a federated learning setup. The Flower client facilitates communication between the server and clients for federated learning. The provided print statements offer insights into the training and evaluation processes.

8 Reference

https://github.com/sushanthk-262/Log anomaly detection

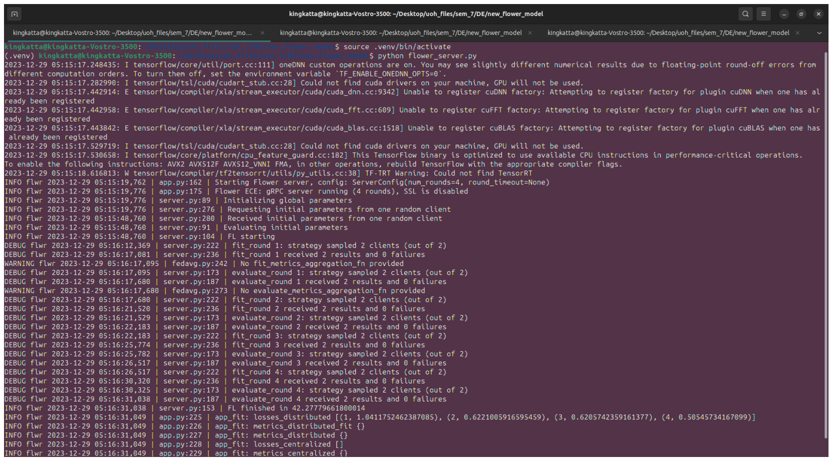

Figure 1: Output of the server

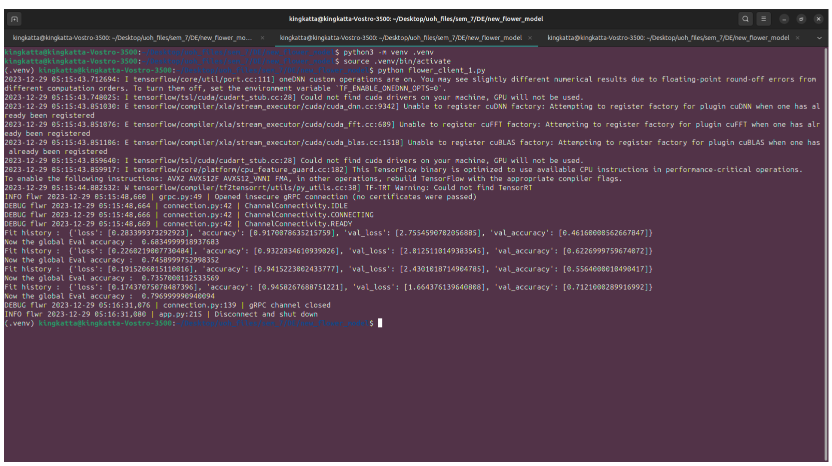

Figure 2: Output of client 1

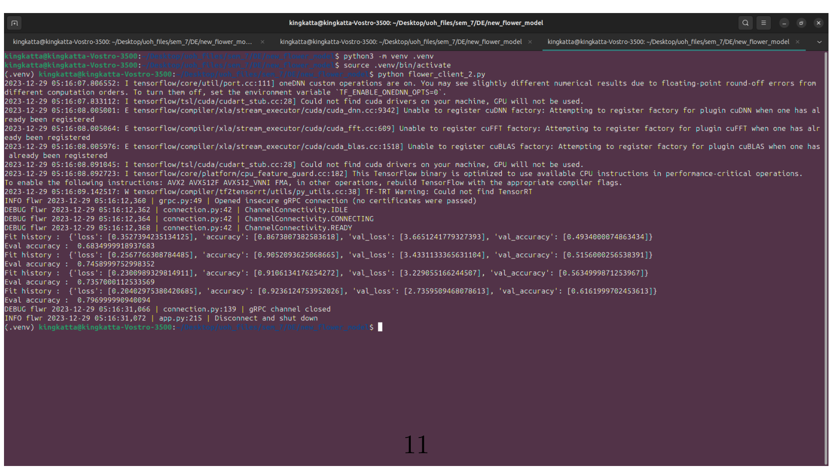

Figure 3: Output of client 2

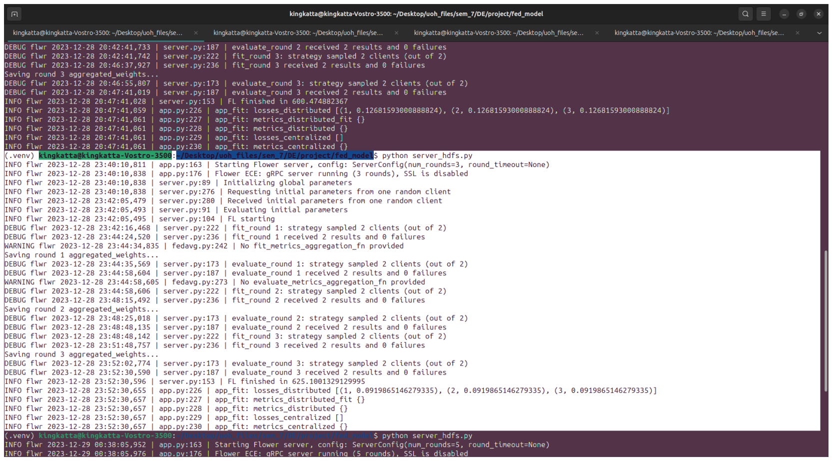

Figure 4: Hdfs Server

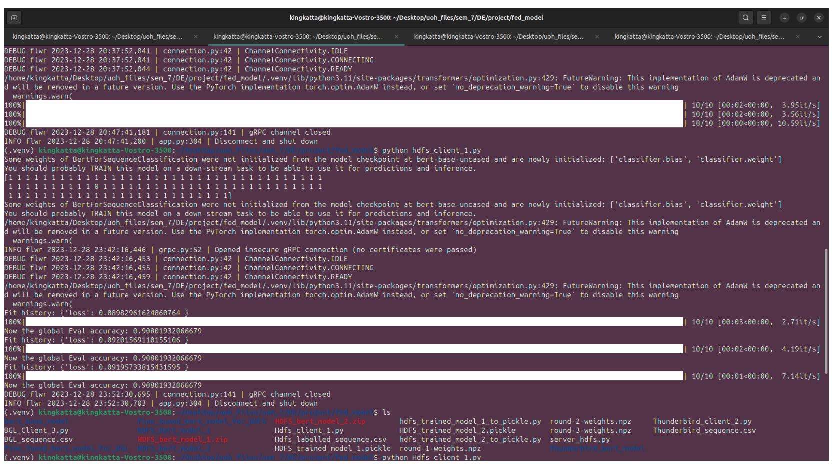

Figure 5: Hdfs Client 1

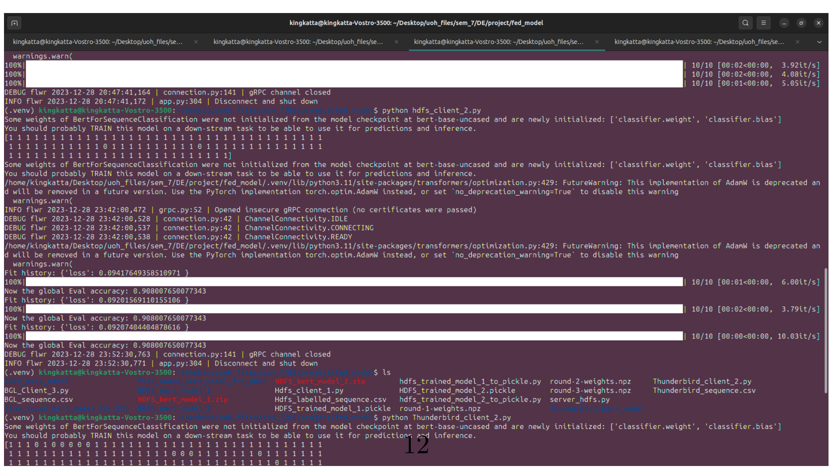

Figure 6: Hdfs Client 1

Figure 7: Ram usage while Training