Federated Learning Framework Documentation

Overview

The Federated Learning Framework is a modular and extensible framework designed to facilitate federated learning in various applications, including but not limited to NLP, self-driving cars, Unmanned Aerial Vehicles (UAVs), Robotics, and other AI domains. The framework includes support for homomorphic encryption to ensure the privacy and security of model weights during federated training.

Features

- **Modular Design**: Easily customizable for different machine learning and deep learning tasks.
- **Federated Learning**: Supports decentralized model training across multiple clients.
- **Homomorphic Encryption**: Ensures the privacy and security of model weights during transmission and aggregation.
- **Flexible Communication**: Supports various connection methods including socket programming.
- **Active Learning**: Incorporates active learning strategies to improve model performance.

Potential Applications

Healthcare

Federated learning can be used to train models on patient data from multiple hospitals without sharing sensitive information. This approach can improve medical diagnostics and treatment recommendations while preserving patient privacy.

Autonomous Vehicles

By collecting and learning from data across multiple autonomous vehicles, the framework can help improve the safety and performance of self-driving cars without exposing individual vehicle data.

Drones

Drones can use federated learning to share and learn from data collected during their operations, enhancing their navigation, object detection, and other capabilities while ensuring data security.

Natural Language Processing (NLP)

Federated learning can be applied to train NLP models on data from multiple sources, such as user devices, to improve language understanding and generation without compromising user privacy.

Finance

Financial institutions can use federated learning to develop fraud detection and risk management models by leveraging data from multiple sources while keeping customer data secure.

Smart Homes and IoT Devices

IoT devices in smart homes can collaboratively learn from user interactions to optimize performance and provide better services without sharing raw data.

Package Structure

```
federated_learning_framework/
├── README.md
├── setup.py
├── requirements.txt
├── federated_learning_framework/
│   ├── __init__.py
│   ├── central_server.py
│   ├── client_device.py
│   ├── encryption.py
│   ├── active_learning.py
│   ├── connection.py
│   ├── decorators.py
│   └── utils.py
└── tests/
    ├── __init__.py
    ├── test_central_server.py
    ├── test_client_device.py
    ├── test_encryption.py
    ├── test_active_learning.py
    └── test_utils.py
```

Detailed Component Description

Central Server

File: `central_server.py`

The central server orchestrates the federated learning process by coordinating the communication and aggregation of model weights from various client devices.

Key Functions:

- `run_server`: Starts the server to handle client connections.
- `handle_client`: Manages incoming messages from clients.
- `transmit_weights`: Broadcasts the aggregated weights to clients.
- `send_data_to_client`: Sends specific data to a client.
- `get_data_from_client`: Requests and receives data from a client.
- `query_active_learning`: Implements active learning strategies to select data for labeling.

Client Device

File: `client_device.py`

Client devices perform local training on their datasets and communicate with the central server.

Key Functions:

- `connect_to_server`: Connects to the central server.
- `federated_learning`: Coordinates local training and communication with the server.
- `receive_weights`: Receives model weights from the central server.
- `send_weights`: Sends model weights to the central server.
- `receive_data`: Receives data from the central server.

Encryption

File: `encryption.py`

Provides functions for creating encryption contexts and encrypting/decrypting model weights.

Key Functions:

- `create_context`: Sets up the encryption context using TenSEAL.
- `encrypt_weights`: Encrypts model weights.
- `decrypt_weights`: Decrypts encrypted model weights.

Active Learning

File: `active_learning.py`

Implements active learning strategies to enhance the training process by selectively querying informative data points.

Key Functions:

- `select_informative_samples`: Selects samples for labeling based on uncertainty.

Connection

File: `connection.py`

Manages the connection types and protocols (e.g., WebSocket) for communication between the central server and client devices.

Key Functions:

- `run_server`: Starts a WebSocket server.
- `connect_to_server`: Establishes a WebSocket connection to the server.

Decorators

File: `decorators.py`

Provides decorators for adding federated learning and encryption functionalities to functions.

Key Functions:

- `federated_learning_decorator`: Wraps a function to enable federated learning.
- `encryption_decorator`: Wraps a function to enable homomorphic encryption.

Utilities

File: `utils.py`

Includes utility functions used throughout the framework.

Installation

1. Clone the repository:

   ```
   git clone
   https://github.com/mehrdaddjavadi/federated_learning_framework.git
   ```

2. Navigate to the directory:

   ```
   cd federated_learning_framework
   ```

3. Install the dependencies:

   ```
   pip install -r requirements.txt
   ```

Usage

Setting Up the Central Server

```
import asyncio
from federated_learning_framework.central_server import CentralServer

async def main():
    server = CentralServer()
    await server.run_server()

asyncio.run(main())
```

Setting Up a Client Device

```
import asyncio
import tensorflow as tf
from federated_learning_framework.client_device import ClientDevice
from federated_learning_framework.encryption import create_context

# Define your model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation='relu', input_shape=(3072,)),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Create context for encryption
context = create_context()

# Initialize the client device
client = ClientDevice(client_id=1, model=model, context=context)

async def main():
    uri = "ws://localhost:8089"
    await client.connect_to_central_server(uri)
    x_train, y_train = ...  # Load your training data
    await client.federated_learning(uri, x_train, y_train)
    # Optionally receive data from central server
    data = await client.receive_data()
    print(f"Received data: {data}")

asyncio.run(main())
```

Using Decorators

```
python
import asyncio
import tensorflow as tf
from federated_learning_framework.decorators import
federated_learning_decorator, encryption_decorator
from federated_learning_framework.client_device import ClientDevice
from federated_learning_framework.encryption import create_context

# Create context for encryption
context = create_context()
```

```
# Define your model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation='relu', input_shape=(3072,)),
    tf.keras.layers.Dense(10, activation='softmax')
])

@federated_learning_decorator(uri="ws://localhost:8089")
@encryption_decorator(context=context)
async def main():
    client = ClientDevice(client_id=1, model=model, context=context)
    await client.connect_to_central_server('ws://localhost:8089')
    x_train, y_train = ...  # Load your training data
    await client.federated_learning('ws://localhost:8089', x_train, y_train)

asyncio.run(main())
```

Running Tests

To run the tests, execute the following command in the root directory:

```
python -m unittest discover -s tests
```

License

Contact: mehrdaddjavadi@gmail.com

Contributing

Feel free to contribute by submitting a pull request or opening an issue.