A

Report

On

Anti-Money Laundering in Bitcoin: Experimenting with Graph Convolutional Networks and Federated Learning on Graphs

By

CH. KARTHIK (23MCMT01)

JAGAN MADDELA (23MCMT11)

VENKATA PALLAVI NAMBURI (2OMCME12)

K. LAVANYA LAHARI (20MCME23)

1. OBJECTIVE

Blockchain provides the unique and accountable channel for financial forensics by mining its open and immutable transaction data. A recent surge has been witnessed by training machine learning models with cryptocurrency transaction data for anomaly detection, such as money laundering and other fraudulent activities. This paper presents a holistic applied data science approach to fraud detection in the Bitcoin network with the help of elliptic dataset which contains 203,769 transaction nodes, 234,355 edges , 167 features per transaction. This project is to develop a federated learning model for predicting whether Bitcoin trans- actions are illicit or licit using a Graph Convolutional Network (GCN). The federated learning approach is employed to enable decentralized training on individual clients' datasets, ensuring privacy and security while collectively improving the global model's accuracy.

2. DATASET

The Elliptic Data Set maps Bitcoin transactions to real entities belonging to licit categories versus illicit ones. The task on the dataset is to classify the illicit and licit nodes in the graph. This anonymized data set is a transaction graph collected from the Bitcoin blockchain. A node in the graph represents a transaction; an edge can be viewed as a flow of Bitcoins between one transaction and the other. The graph is made of 203,769 nodes and 234,355 edges. Two percent (4,545) of the nodes are labelled class1 (illicit). Twenty-one percent (42,019) are labelled class2 (licit). The remaining transactions are not labelled with regard to licit versus illicit. Each node has 167 features and has been labeled as being created by a "licit", "illicit" or "unknown" entity. The temporal information is encoded by a time step running from 1 to 49, a measure of the actual trans3 action time stamp. The time steps are evenly spaced with an interval of about two weeks; each one of them contains a single connected component of transactions that appeared on the blockchain within less than three hours between each other. There are no edges connecting the different time steps. Out of the 167 features, the first 94 represent local information about the transaction — including the time step described above, number of inputs/outputs, transaction fee, output volume and aggregated figures such as average BTC received (spent) by the inputs/outputs and average number of incoming (outgoing) transactions associated with the inputs/outputs. The remaining 72 features represent nonlocal (graph) information in the form of aggregated features, obtained using transaction information one-hop backward/forward from the center node — giving the maximum, minimum, standard deviation and correlation coefficients of the neighbour transactions for the same information data (number of inputs/outputs, transaction fee, etc.).

2.1 WHY GCN FOR THE ELLIPTIC DATASET

Graph Convolutional Networks (GCNs) are particularly well-suited for tasks involving graph-structured data, and they have been shown to be effective in various domains, including social network analysis, biology, and citation networks. However, whether a GCN is the "best" model for predicting elliptic datasets depends on the specific characteristics of the data and the task at hand.

Some reasons:

- Graph Structure Representation: Elliptic datasets may have inherent graph structures that capture relationships between data points. GCNs are designed to work with graph-structured data, allowing them to leverage the relationships and dependencies within the data.
- Node-Level and Graph-Level Representations: GCNs can learn both nodelevel and graph-level representations. This is beneficial for tasks where understanding the local and global structure of the data is important, such as identifying patterns in elliptic datasets.
- Spatial and Spectral Convolution: GCNs perform convolution operations in both spatial and spectral domains on the graph, capturing local and global patterns in the data. This flexibility can be advantageous for tasks where features of interest exhibit spatial dependencies.
- Handling Missing Data and Irregular Graphs:GCNs are capable of handling irregular graph structures and missing data, which can be common challenges in real-world datasets, including elliptic datasets.

2.2 WHY GCN FOR THE ELLIPTIC DATASET

While Graph Convolutional Networks (GCNs) have demonstrated success in handling graph-structured data, including various applications, it doesn’t mean that they are always the best choice for every dataset or prediction task. The choice of a model depends on the characteristics of the data, the nature of the problem, and various other factors.

Nature of the Data:

If the elliptic dataset does not exhibit clear graph structures or relationships between data points, traditional machine learning models such as support vector machines (SVMs), decision trees, or random forests might be more appropriate.

Data Size and Complexity:

For smaller datasets or simpler problems, complex models like GCNs might be unnecessary and prone to overfitting. Simpler models with fewer parameters may generalize better in such cases.

3. GCN MODEL

A Graph Convolutional Network, or GCN, is an approach for semi-supervised learning on graph-structured data. It is based on an efficient variant of convolutional neural networks which operate directly on graphs.

For this model, the key Functions are:

1. run_experiment()
2. display_learning_curves()
3. create_ffn()

The key Components for building of GCN Model are:

4. GraphConvLayer()
5. GNNNodeClassifier()

3.1 run_experiment()

The Function 'run_experiment' that takes a neural network model ('model'), training data ('x-train' and 'y-train'), and runs an experiment by compiling and training the model. The training process includes early stopping based on the validation accuracy.

Model Compilation:

- The model is compiled using the Adam optimizer with a specified learning rate.
- Sparse categorical crossentropy is used as the loss function, assuming the model's output is logits.
- The metric being monitored during training is sparse categorical accuracy.

Early Stopping Callback:

- An early stopping callback is created to monitor the validation accuracy ('val_acc').
- The patience parameter is set to 10, meaning training will stop if there is no improvement in validation accuracy for 10 consecutive epochs.

- ‘restore_best_weights=True’ ensures that the model's weights are restored to the ones that achieved the best performance on the validation set.

Model Training:

- The model is trained using the ‘fit’ method.
- Training data (‘x_train’ and ‘y_train’) are provided.
- The number of epochs, batch size, and a validation split of 15% are specified.
- The early stopping callback is included in the training process.

3.2 display_learning_curves()

The Function ‘display_learning_curves’ that takes a **history** object as input and plots the training and validation loss on one subplot and the training and validation accuracy on another subplot. This function relies on the matplotlib library for plotting.

3.3 create_ffn()

The Function ‘create_ffn’ that generates a feedforward neural network (FFN) using keras. This function takes 3 parameters:

1. **hidden_units:** A list specifying the number of units in each hidden layer.
2. **dropout_rate:** The dropout rate used in each dropout layer.
3. **name:** An optional name for the Sequential model.

The function creates a list fnn_layers containing BatchNormalization, Dropout, and Dense layers and then uses ‘keras.Sequential’ to create a feedforward neural network with the specified architecture.

i. **fnn_layers.append(layers.BatchNormalization()) :** Adds a BatchNormalization layer to the list. Batch normalization is a technique used to improve the training stability and performance of neural networks.

ii. **fnn_layers.append(layers.Dropout(dropout_rate)) :** Adds a Dropout layer to the list. Dropout is a regularization technique that helps prevent overfitting by randomly setting a fraction of input units to zero during training.

iii. **fnn_layers.append(layers.Dense(units, activation=tf.nn.gelu)) :** Adds a Dense layer to the list. The Dense layer is a fully connected layer, and ‘tf.nn.gelu’ is used as the activation function. GELU (Gaussian Error Linear Unit) is an activation function that has been shown to perform well in neural networks.

The loop iterates over the ‘hidden_units’ list, adding the specified layers for each hidden unit.

Finally, the function returns a Sequential model created from the list of layers.

3.4 GraphConvLayer()

- This layer is designed to operate on graph-structured data, where nodes are connected by edges. The layer takes node representations, edges, and edge weights as inputs and produces updated node embeddings as outputs.
- The Functionalities used in GraphConvLayer() are:

3.4.1 Initialization

The layer is initialized with parameters such as hidden_units, dropout_rate, aggregation_type, combination_type, and normalizes.

- **‘hidden_units’:** A list specifying the number of hidden units for each layer in the feedforward neural network used in the graph convolution layer.
- **‘dropout_rate’:** The dropout rate applied to the feedforward neural network.
- **‘aggregation_type’:** Specifies the type of aggregation used for combining neighboring node features. It could be "mean" or another type based on the application.
- **‘combination_type’:** Specifies how the updated node features are combined. It could be "concat" or "gated" based on the application.
- ‘**normalize’:** A boolean indicating whether to normalize the aggregated features.

Feedforward Neural Network (FFN): The ‘ffn_prepare’ attribute is initialized using a function ‘create_ffn’, which is expected to create a feedforward neural network with the specified parameters (hidden units and dropout rate).

Combination Type: If ‘combination_type’ is "gated," a GRU (Gated Recurrent Unit) layer is used as the update function (‘update_fn’). If it's not "gated," a feedforward neural network is used as the update function

3.4.2 Prepare Function

Input:

- **‘node_representation’:** This is a tensor representing node representations with shape ‘[num_edges, embedding_dim]’. Each row in this tensor likely corresponds to the representation of a node in a graph.
- **‘weights’:** An optional tensor of weights with shape ‘[num_edges]’ that can be used to modulate the messages. If provided, the messages are element-wise multiplied by these weights

Feedforward Neural Network (FFN):

- The method applies the feedforward neural network (‘ffn_prepare’) to the input ‘node_representations’. This involves passing the node representations through a series of layers defined by the ‘create_ffn’ function. The result is stored in the ‘messages’ variable

Weights Modulation:

- If ‘weights’ provided (i.e., not ‘none’), the messages are element-wise multiplied by the weights. This step allows for the modulation or adjustment of the messages based on the provided weights

Output:

- The method returns the computed messages. The shape of the output tensor is the same as the input ‘node_representation’, i.e., ‘[num_edges, embedding_dim]’.

3.4.3 Aggregation Function

Input:

- **‘node_indices’:** A tensor with shape ‘[num_edges]’ representing the indices of nodes for which the messages need to be aggregated.
- **‘neighbour_messages’:** A tensor with shape ‘[num_edges, representation_dim]’ containing the messages from neighboring nodes

Number of Nodes:

- **‘num_nodes’** is computed as the maximum value in ‘node_indices’ plus 1. This assumes that the node indices are zero-based and consecutive. This value represents the total number of nodes in the graph

Aggregation Types:

- The method supports three types of aggregation: "sum," "mean," and "max." The appropriate aggregation operation is selected based on the value of ‘self_aggregation_type’.

Aggregation Operations:

- If ‘self.aggregation_type’ is "sum," the method uses ‘tf.math.unsorted_segment_sum’ to sum the neighbor messages for each node.
- If it's "mean," it uses ‘tf.math.unsorted_segment_mean’ to compute the mean of neighbor messages for each node.
- If it's "max," it uses ‘tf.math.unsorted_segment_max’ to find the maximum neighbor message for each node.

Error Handling:

- If an invalid aggregation type is provided, a ‘ValueError’ is raised

Output:

- The method returns the aggregated messages, where each row corresponds to the aggregated message for a specific node. The shape of the output tensor is ‘[num_edges, representation_dim]’.

3.4.4 Update Function

Inputs:

- **‘node_representation’:** A tensor with shape ‘[num_edges, representation_dim]’ representing the current node representations
- **‘aggregated_messages’** A tensor with the same shape ‘[num_edges, representation_dim]’ representing the aggregated messages from neighboring nodes

Combination Types:

- The method supports three types of combinations: "gru," "concat," and "add." The appropriate combination operation is selected based on the value of ‘self.combination_type’

Combination Operations:

- If ‘self.combination_type’ is "gru," the method creates a sequence of two elements (‘node_representation’ and ‘aggregated_messages’) along the second axis and stacks them using ‘tf.stack’.
- If it's "concat," it concatenates ‘node_representation’ and ‘aggregated_messages’ along the second axis using ‘tf.concat’.
- If it's "add," it adds ‘node_representation’ and ‘aggregated_messages’.

Processing Function:

- The combined tensor ’h’ is passed through the processing function ‘self.update_fn’. If the combination type is "gru," this function is expected to be a GRU layer.
- If the combination type is "gru," the final element of the sequence is extracted using ‘tf.unstack’.

Normalization:

- If ‘self.normalize’ is ‘True’, the resulting node embeddings are L2-normalized along the last axis using ‘tf.nn.l2.normalize’

Output:

- The method returns the updated node embeddings.

3.4.5 Call Function

‘call’ method is the main processing function of the GraphConvLayer class. It takes a tuple of three elements (‘node_representations’, ‘edges’, and ‘edge_weights’) as inputs and produces the node embeddings.

Inputs:

- **‘node_representations’:** A tensor with shape ’[num_nodes, representation_dim]’ representing the current node representations.

- **‘edges’:** A tuple containing two tensors (‘node_indices’ and ‘neighbour_indices’) with shape ‘[num_edges]’ representing the indices of source and target nodes for edges in the graph.
- **‘edge_weights’:** A tensor with shape ‘[num_edges]’ representing the weights associated with the edges.

Node and Neighbour Indices:

- Extract ‘node_indices’ and ‘neighbour_indices’ from the ‘edges’ tuple.

Neighbour Representations:

- Use ‘tf.gather’ to obtain the representations of neighbouring nodes (‘neighbour_representations’) based on the ‘neighbour_indices’.

Prepare Messages:

- Call the ‘prepare’ method to prepare messages from the neighbouring representations, considering the edge weights

Aggregate Messages:

- Call the ‘aggregate’ method to aggregate the prepared neighbour messages based on the specified aggregation type

Update Node Representations:

- Call the ‘update’ method to update the node representations based on the aggregated messages

Output:

- The method returns the updated node embeddings

3.5 GNNNodeClassifier()

3.5.1 Initialization

- **‘graph_info’:** Information about the graph, including node features, edges, and edge weights.
- **‘num_classes’:** Number of classes for node classification.
- **‘hidden_units’:** Number of hidden units in the neural network layers.
- **‘aggregation_type’:** Type of aggregation used in GraphConv layers (default is "sum").
- **‘Combination_type’:** Type of aggregation used in GraphConv layers (default is "concat").
- **‘dropout_rate’ :** Dropout rate for regularization (default is 0.2).
- **‘normalize’:** Boolean indicating whether to normalize edge weights (default is True).
- **‘*args. **kwargs’ :** Additional arguments and keyword arguments.

3.5.2 Attributes

- ‘node_features’, ‘edges’ and ‘edge_weights’ are extracted from ‘group_info’.
- If ‘edge_weights’ is not provided. It is set to ones.

- The ‘edge_weights’ are then scaled to sum to 1.

3.5.3 Layers

- **‘preprocess’:** A feedforward neural network (FFN) layer created using the ‘create_ffn’ function
- **‘conv1’** and **‘conv2’:** Two instances of the GraphConvLayer class, presumably implementing graph convolutional operations.
- **‘postprocess’:** Another FFN layer for post-processing.
- **‘compute_logits’:** A dense layer to compute the final logits, with the number of units equal to ‘num_classes’.

3.5.4 ‘call’ Method

1. **Preprocess Node Features:**
    - ‘x’ is the result of preprocessing the ‘node_features’ using the ‘preprocess’ layer.
2. **First Graph Convolutional Layer(‘conv1’):**
    - ‘x1’ is the result of applying the first graph convolutional layer (‘conv1’) to the preprocessed node features.
    - A skip connection is implemented by adding ‘x’ to ‘x1’.
3. **Second Graph Convolutional Layer(‘conv2’):**
    - ‘x2’ is the result of applying the second graph convolutional layer (‘conv2’) to the output of the first layer plus the original node features.
    - A skip connection is implemented by adding ‘x’ to ‘x2’.
4. **Postprocess Node Embeddings:**
    - The result, ‘x’, is then passed through the ‘postprocess’ layer for post-processing.
5. **Node Embeddings for Input Node Indices:**
    - Node embeddings for the specified ‘input_node_indices’ are extracted using ‘tf.gather’ on the processed ‘x’.
6. **Compute Logits:**
    - The final step involves passing the node embeddings through the ‘compute_logits’ layer to compute the logits.

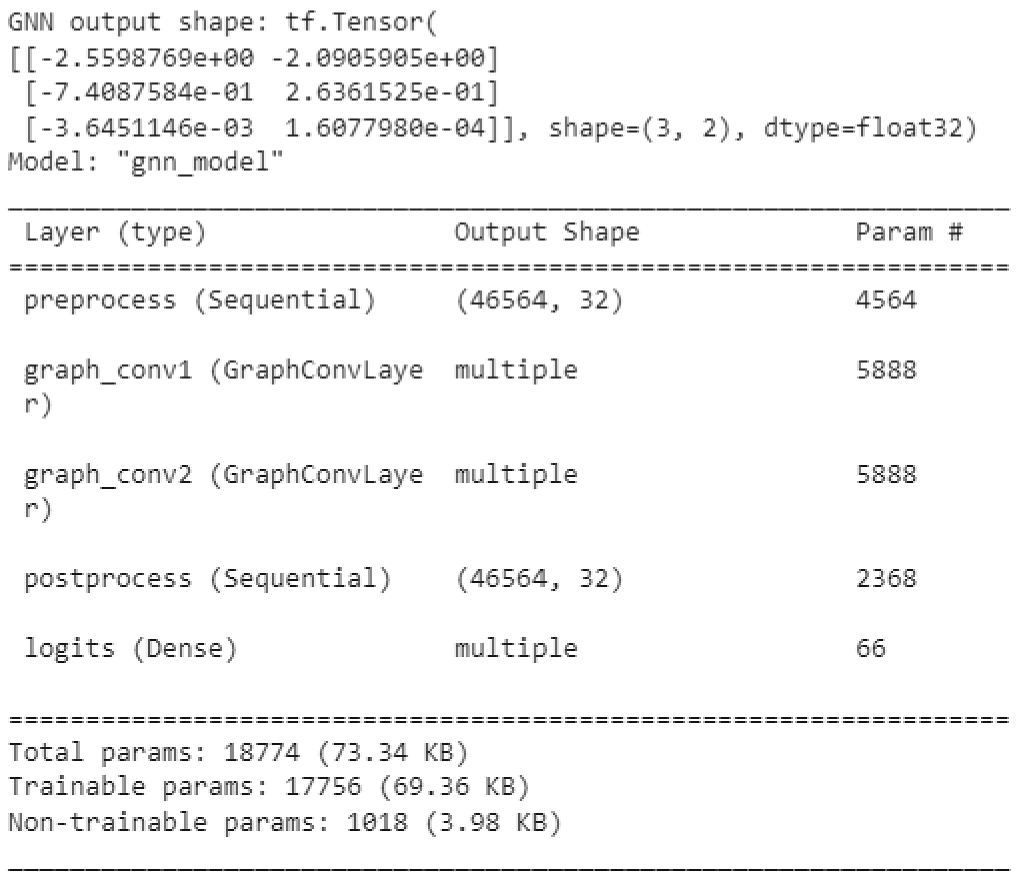
- This ‘call’ method essentially performs a series of graph convolutional operations with skip connections, followed by post-processing, and finally, computes logits for the specified nodes. It's a standard forward pass for a graph neural network for node classification. When using this model, you would typically call this method with input node indices to obtain the logits for those nodes.

3.6 HyperParameters

No of classes are classified into 2 classes.

- **‘hidden_units’:** A list specifying the number of hidden units in each hidden layer of your neural network. In your case, there are two hidden layers, each with 32 units.
- **‘learning_rate’:** The learning rate determines the size of the steps taken during the optimization process. A higher learning rate may lead to faster convergence, but it could overshoot the optimal values [0.007].
- **‘dropout_rate’:** Dropout is a regularization technique where randomly selected neurons are ignored during training. It helps prevent overfitting. Here, you've set the dropout rate to 0.6, meaning 60% of the neurons will be dropped out during training.
- **‘num_epochs’:** The number of times the entire dataset is passed forward and backward through the neural network. It's a hyperparameter that needs to be tuned based on the specific problem [30].
- **‘Batch_size’:** The number of training examples utilized in one iteration. A smaller batch size can lead to a more accurate model, but it requires more time to process each epoch [256].

4. MODEL SUMMARY



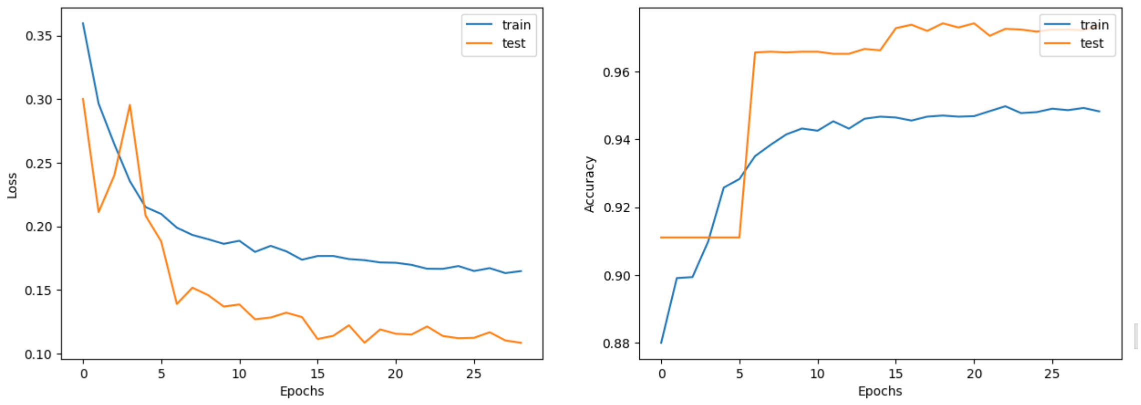
```
GNN output shape: tf.Tensor(
[[-2.5598769e+00 -2.0905905e+00]
 [-7.4087584e-01  2.6361525e-01]
 [-3.6451146e-03  1.6077980e-04]], shape=(3, 2), dtype=float32)
Model: "gnn_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 preprocess (Sequential)     (46564, 32)               4564

 graph_conv1 (GraphConvLaye  multiple                  5888
 r)

 graph_conv2 (GraphConvLaye  multiple                  5888
 r)

 postprocess (Sequential)    (46564, 32)               2368

 logits (Dense)              multiple                  66

=================================================================
Total params: 18774 (73.34 KB)
Trainable params: 17756 (69.36 KB)
Non-trainable params: 1018 (3.98 KB)
_________________________________________________________________
```

5. TRAIN THE MODEL

- **‘x’** is a DataFrame containing the features. It seems to include columns specified in ‘tx_features’ and the 'txId' column.
- **‘y’** is a Series containing the target variable ('class'). The lambda function is used to convert class labels '2' to 0 and all other labels to 1.
- **‘train_test_split’** is then used to split the data into training and testing sets. **‘test_size=0.3’** indicates that 30% of the data will be used for testing, and **‘random_state=117’** ensures reproducibility.
- **‘x_train’:** Training features; **‘x_test’:** Testing features; **‘y_train’:** Training labels; **‘y_test’:** Testing labels.

```
x_train = X_train.txId.to_numpy()
history = run_experiment(gnn_model, x_train, y_train)
```

```
[23] display_learning_curves(history)
```



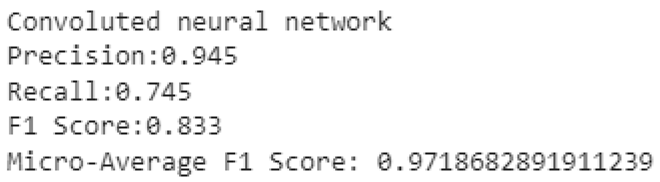
- Here the run_experiment takes the GNN model, input features (**‘x_train’**), and target labels (**‘y_train’**) to perform the training.
- The Left fig. shows the loss of train and test of model.
- The Right fig. shows accuracy of train and test of the model.

6. TEST RESULTS

- Accuracy : 97.19%

```
[24] x_test = X_test.txId.to_numpy()
     _, test_accuracy = gnn_model.evaluate(x=x_test, y=y_test, verbose=0)
     print(f"Test accuracy: {round(test_accuracy * 100, 2)}%")

     Test accuracy: 97.19%
```

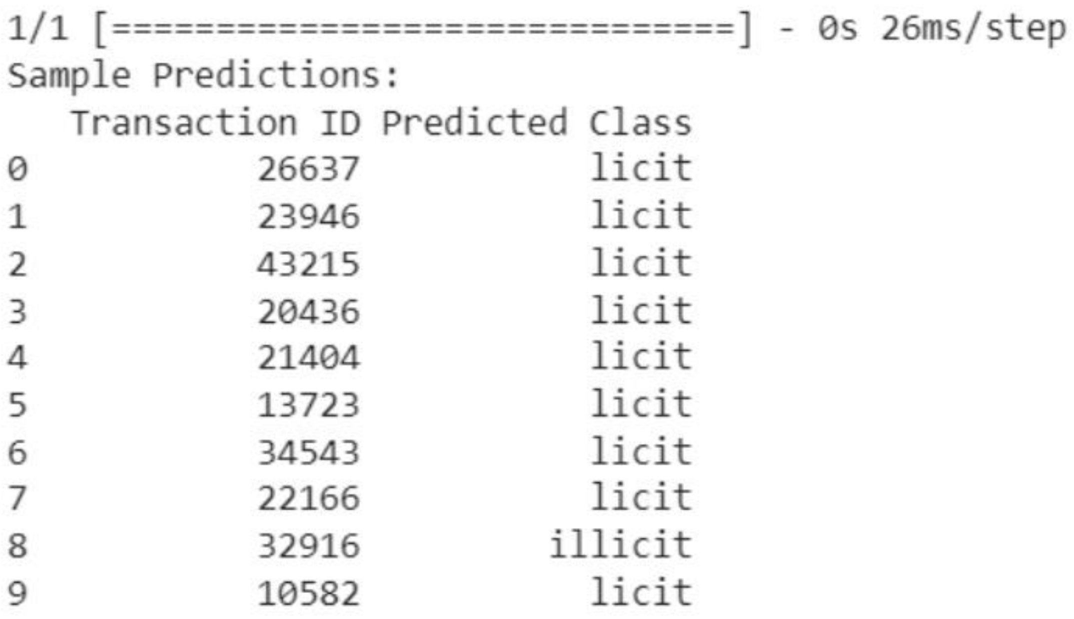
- The Precision, Recall, F1 Score and Micro-Average F1 Score values:



```
Convoluted neural network
Precision:0.945
Recall:0.745
F1 Score:0.833
Micro-Average F1 Score: 0.9718682891911239
```

7. SAVE AND LOAD OF GCN MODEL

- An issue while loading the data after saving the model. We got to know that the latest version of TensorFlow doesn't support for the predictions to do on the loaded model. The only way is to save the weights of the model and load the weights of the model for the future predictions.
- We have done some predictions using some random transactions from the dataset and output is to print weather they are licit or illicit transactions.



```
1/1 [==============================] - 0s 26ms/step
Sample Predictions:
   Transaction ID Predicted Class
0           26637           licit
1           23946           licit
2           43215           licit
3           20436           licit
4           21404           licit
5           13723           licit
6           34543           licit
7           22166           licit
8           32916         illicit
9           10582           licit
```

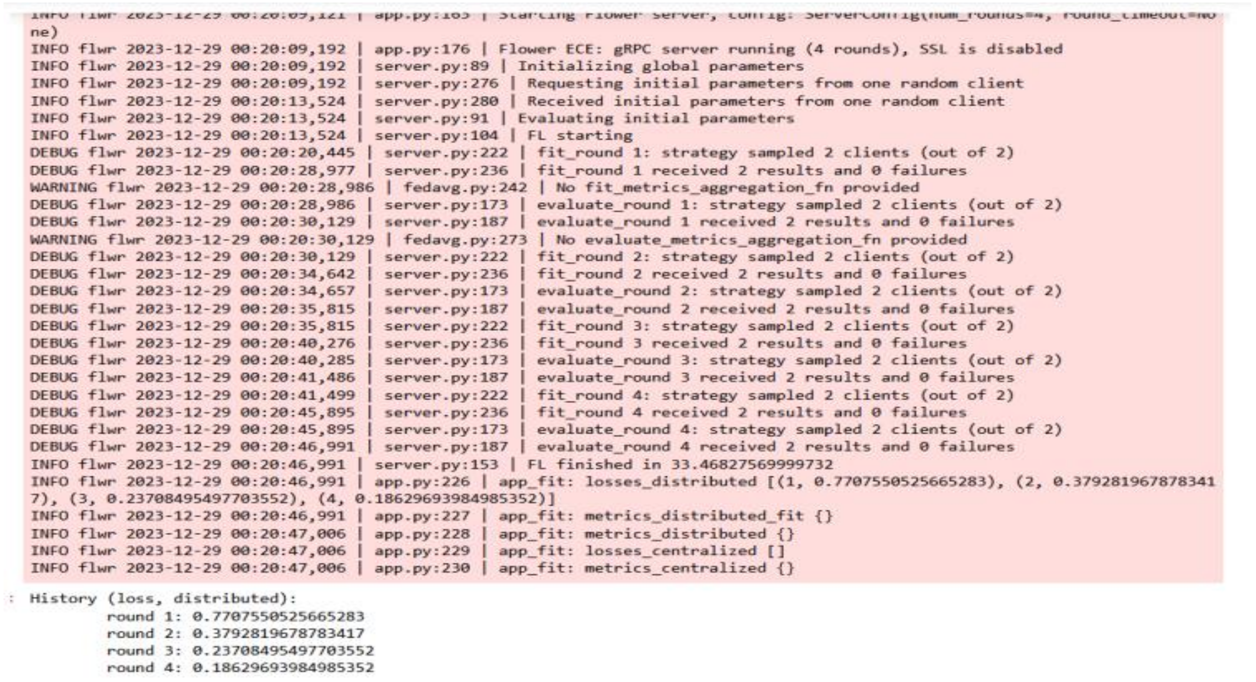
Figure 7.1: Taking input as randomly selected 10 transactions from the data to get output

8. FEDERATED LEARNING

Federated learning workflow

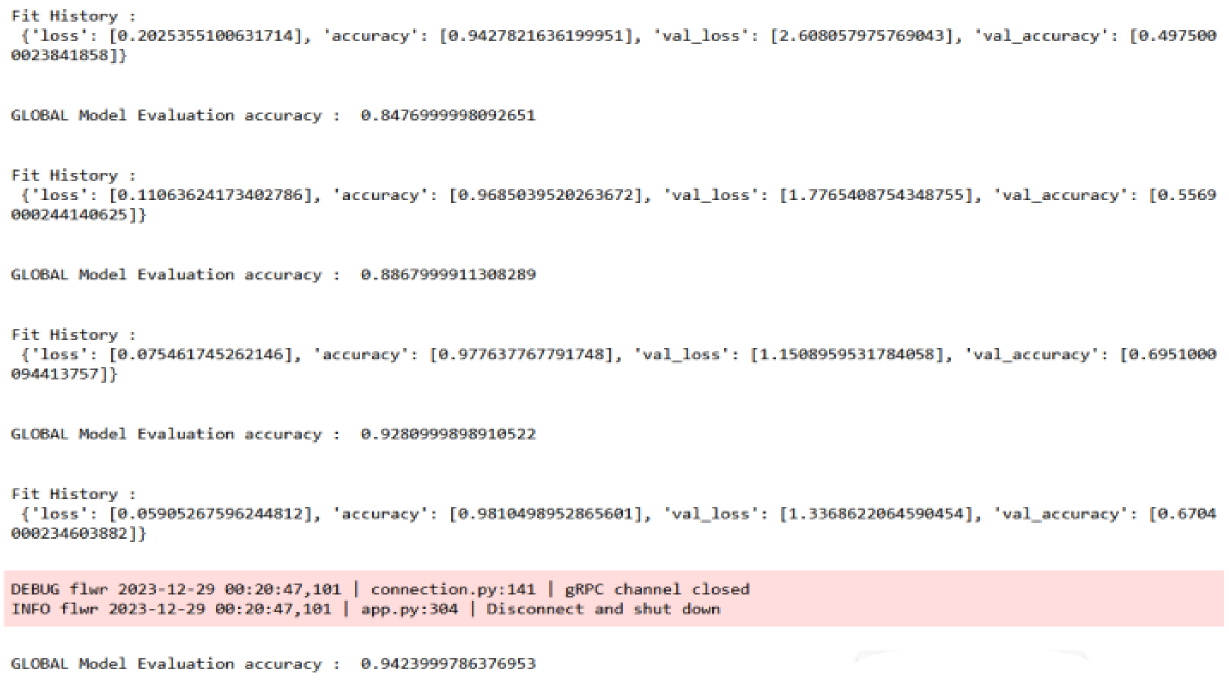
- Edge devices receive a copy of a global model from a central server.
- The model is being trained locally on the data residing on the edge devices.
- The global model weights are updated during training on each worker.
- A local copy is sent back to the central server.
- The server receives various updated model and aggregate the updates thereby improving the global model and also preserving privacy of data in which it was being trained on.

We have developed a Federated Learning environment using the FLOWER framework. I have taken a model from tensorflow keras (open source neural networks library) and implemented the same in a federated learning setup. The MNIST dataset has been used to train this model. The task selected is Image Classification. Federated system has to have minimum of 2 clients. These clients train local models on their local machines non-IID subsets of the MNIST dataset. (Non-IID data distribution implies that the clients have different subsets of the data, ensuring a more realistic and challenging Federated Learning scenario). Then send them back to the server. Upon receiving all the models from the clients, server coordinates the training process by aggregating model updates and broadcasting the global model updates.



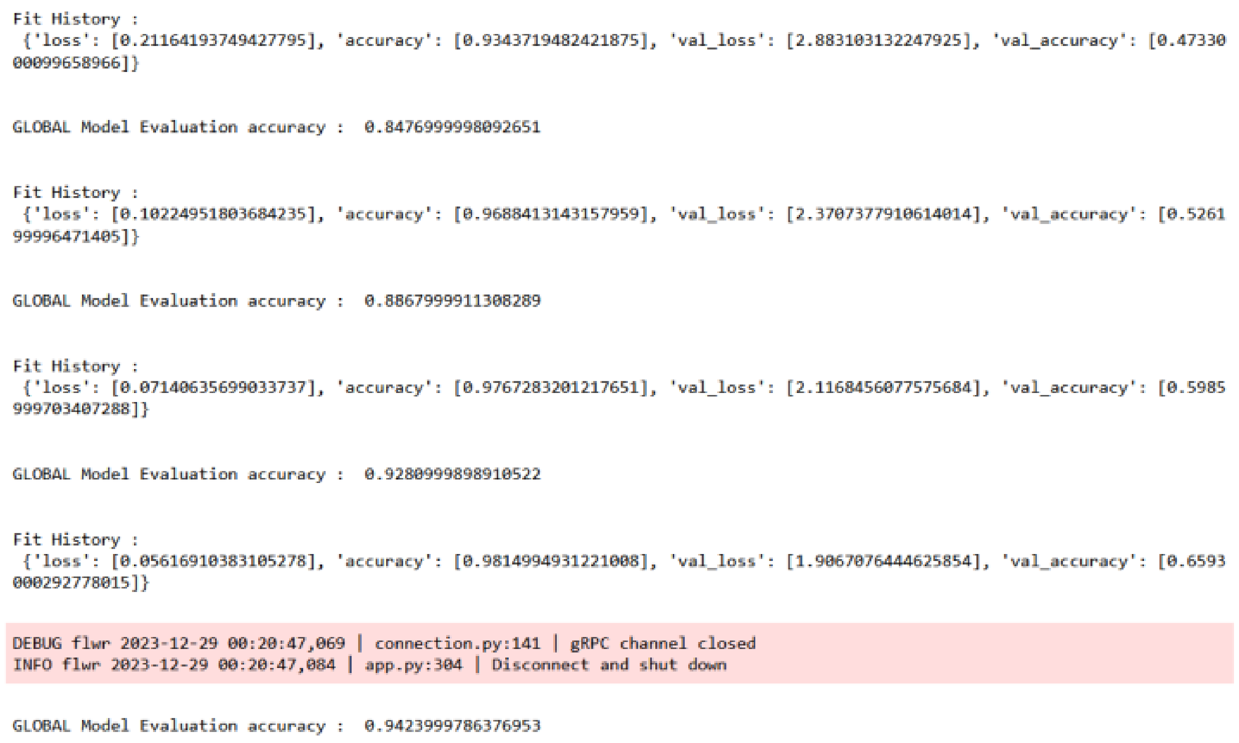
```
INFO flwr 2023-12-29 00:20:09,121 | app.py:163 | Starting Flower server, config: ServerConfig(num_rounds=4, round_timeout=No
ne)
INFO flwr 2023-12-29 00:20:09,192 | app.py:176 | Flower ECE: gRPC server running (4 rounds), SSL is disabled
INFO flwr 2023-12-29 00:20:09,192 | server.py:89 | Initializing global parameters
INFO flwr 2023-12-29 00:20:09,192 | server.py:276 | Requesting initial parameters from one random client
INFO flwr 2023-12-29 00:20:13,524 | server.py:280 | Received initial parameters from one random client
INFO flwr 2023-12-29 00:20:13,524 | server.py:91 | Evaluating initial parameters
INFO flwr 2023-12-29 00:20:13,524 | server.py:104 | FL starting
DEBUG flwr 2023-12-29 00:20:20,445 | server.py:222 | fit_round 1: strategy sampled 2 clients (out of 2)
DEBUG flwr 2023-12-29 00:20:28,977 | server.py:236 | fit_round 1 received 2 results and 0 failures
WARNING flwr 2023-12-29 00:20:28,986 | fedavg.py:242 | No fit_metrics_aggregation_fn provided
DEBUG flwr 2023-12-29 00:20:28,986 | server.py:173 | evaluate_round 1: strategy sampled 2 clients (out of 2)
DEBUG flwr 2023-12-29 00:20:30,129 | server.py:187 | evaluate_round 1 received 2 results and 0 failures
WARNING flwr 2023-12-29 00:20:30,129 | fedavg.py:273 | No evaluate_metrics_aggregation_fn provided
DEBUG flwr 2023-12-29 00:20:30,129 | server.py:222 | fit_round 2: strategy sampled 2 clients (out of 2)
DEBUG flwr 2023-12-29 00:20:34,642 | server.py:236 | fit_round 2 received 2 results and 0 failures
DEBUG flwr 2023-12-29 00:20:34,657 | server.py:173 | evaluate_round 2: strategy sampled 2 clients (out of 2)
DEBUG flwr 2023-12-29 00:20:35,815 | server.py:187 | evaluate_round 2 received 2 results and 0 failures
DEBUG flwr 2023-12-29 00:20:35,815 | server.py:222 | fit_round 3: strategy sampled 2 clients (out of 2)
DEBUG flwr 2023-12-29 00:20:40,276 | server.py:236 | fit_round 3 received 2 results and 0 failures
DEBUG flwr 2023-12-29 00:20:40,285 | server.py:173 | evaluate_round 3: strategy sampled 2 clients (out of 2)
DEBUG flwr 2023-12-29 00:20:41,486 | server.py:187 | evaluate_round 3 received 2 results and 0 failures
DEBUG flwr 2023-12-29 00:20:41,499 | server.py:222 | fit_round 4: strategy sampled 2 clients (out of 2)
DEBUG flwr 2023-12-29 00:20:45,895 | server.py:236 | fit_round 4 received 2 results and 0 failures
DEBUG flwr 2023-12-29 00:20:45,895 | server.py:173 | evaluate_round 4: strategy sampled 2 clients (out of 2)
DEBUG flwr 2023-12-29 00:20:46,991 | server.py:187 | evaluate_round 4 received 2 results and 0 failures
INFO flwr 2023-12-29 00:20:46,991 | server.py:153 | FL finished in 33.46827569999732
INFO flwr 2023-12-29 00:20:46,991 | app.py:226 | app_fit: losses_distributed [(1, 0.7707550525665283), (2, 0.379281967878341
7), (3, 0.23708495497703552), (4, 0.18629693984985352)]
INFO flwr 2023-12-29 00:20:46,991 | app.py:227 | app_fit: metrics_distributed_fit {}
INFO flwr 2023-12-29 00:20:47,006 | app.py:228 | app_fit: metrics_distributed {}
INFO flwr 2023-12-29 00:20:47,006 | app.py:229 | app_fit: losses_centralized []
INFO flwr 2023-12-29 00:20:47,006 | app.py:230 | app_fit: metrics_centralized {}
: History (loss, distributed):
          round 1: 0.7707550525665283
          round 2: 0.3792819678783417
          round 3: 0.23708495497703552
          round 4: 0.18629693984985352
```

8.1 Server Output



```
Fit History :
 {'loss': [0.2025355100631714], 'accuracy': [0.9427821636199951], 'val_loss': [2.608057975769043], 'val_accuracy': [0.497500
0023841858]}


GLOBAL Model Evaluation accuracy :  0.8476999998092651


Fit History :
 {'loss': [0.11063624173402786], 'accuracy': [0.9685039520263672], 'val_loss': [1.7765408754348755], 'val_accuracy': [0.5569
000244140625]}


GLOBAL Model Evaluation accuracy :  0.8867999911308289


Fit History :
 {'loss': [0.075461745262146], 'accuracy': [0.977637767791748], 'val_loss': [1.1508959531784058], 'val_accuracy': [0.6951000
094413757]}


GLOBAL Model Evaluation accuracy :  0.9280999898910522


Fit History :
 {'loss': [0.05905267596244812], 'accuracy': [0.9810498952865601], 'val_loss': [1.3368622064590454], 'val_accuracy': [0.6704
000234603882]}

DEBUG flwr 2023-12-29 00:20:47,101 | connection.py:141 | gRPC channel closed
INFO flwr 2023-12-29 00:20:47,101 | app.py:304 | Disconnect and shut down

GLOBAL Model Evaluation accuracy :  0.9423999786376953
```

8.2 Client1 Output



```
Fit History :
 {'loss': [0.21164193749427795], 'accuracy': [0.9343719482421875], 'val_loss': [2.883103132247925], 'val_accuracy': [0.47330
00099658966]}


GLOBAL Model Evaluation accuracy :  0.8476999998092651


Fit History :
 {'loss': [0.10224951803684235], 'accuracy': [0.9688413143157959], 'val_loss': [2.3707377910614014], 'val_accuracy': [0.5261
99996471405]}


GLOBAL Model Evaluation accuracy :  0.8867999911308289


Fit History :
 {'loss': [0.07140635699033737], 'accuracy': [0.9767283201217651], 'val_loss': [2.1168456077575684], 'val_accuracy': [0.5985
999703407288]}


GLOBAL Model Evaluation accuracy :  0.9280999898910522


Fit History :
 {'loss': [0.05616910383105278], 'accuracy': [0.9814994931221008], 'val_loss': [1.9067076444625854], 'val_accuracy': [0.6593
000292778015]}

DEBUG flwr 2023-12-29 00:20:47,069 | connection.py:141 | gRPC channel closed
INFO flwr 2023-12-29 00:20:47,084 | app.py:304 | Disconnect and shut down

GLOBAL Model Evaluation accuracy :  0.9423999786376953
```

8.3 Client2 Output

9. REFERENCES

- https://www.kaggle.com/datasets/ellipticco/elliptic-data-set/data