Building Neural Network Structure Using Neo4j Graph Database

Project Summary

This project aims to construct a neural network structure using the Neo4j graph database. The objective is to model a neural network using only Neo4j and Python, without relying on popular libraries like TensorFlow. The provided initial code was updated, and stored procedures containing Cypher queries were developed using Java.

Project Objectives

- Model neural network structure using Neo4j graph database
- Implement neural network training directly on the graph database
- Optimize performance using stored procedures
- Demonstrate the potential of graph databases in neural network applications

Technical Infrastructure

- **Development Environment:** IntelliJ IDEA
- **Programming Languages:** Python, Java, Cypher
- **Database:** Neo4j
- **Build Tool:** Maven
- **Main Components:**
    - Neo4jGraphAsNetwork.py (Main Python code)
    - CreateNeuron.java (Neo4j stored procedure)

Code Comparison and Improvements

CreateNeuron.java Comparison

Original Code:

```java
@Procedure(name = "nn.createNeuron",mode = Mode.WRITE)
@Description("")
public Stream createNeuron(@Name("id") String id,
                                    @Name("layer") String layer,
                                    @Name("type") String type,
                                    @Name("activation_function") String activation_fun
) {
    try (Transaction tx = db.beginTx()) {
      tx.execute("CREATE (n:Neuron {\n" +
                "id: '" + id + "',\n" +
                "layer:" + layer + ",\n" +
                "type: '" + type + "',\n" +
                "bias: 0.0,\n" +
                "output: null,\n" +
                "m_bias: 0.0,\n" +
                "v_bias: 0.0,\n" +
                "activation_function:'" + activation_function + "'\n" +
                "})");
        tx.commit();
        return Stream.of(new CreateResult("ok"));
    } catch (Exception e) {
        return Stream.of(new CreateResult("ko"));
    }
}
```

Updated Code:

```
@Procedure(name = "nn.createNeuron", mode = Mode.WRITE)
@Description("Creates a neuron with specified properties")
public Stream createNeuron(
        @Name("id") String id,
        @Name("layer") String layer,
        @Name("type") String type,
        @Name("activation_function") String activation_function
) {
    try (Transaction tx = db.beginTx()) {
        tx.execute(
                "CREATE (n:Neuron {" +
                        "id: $id, " +
                        "layer: $layer, " +
                        "type: $type, " +
                        "bias: 0.0, " +
                        "output: null, " +
                        "m_bias: 0.0, " +
                        "v_bias: 0.0, " +
                        "activation_function: $activation_function" +
                        "})",
                Map.of(
                        "id", id,
                        "layer", layer,
                        "type", type,
                        "activation_function", activation_function
                )
        );
        tx.commit();
        return Stream.of(new CreateResult("ok"));
    } catch (Exception e) {
        log.error("Error creating neuron: " + e.getMessage(), e);
        return Stream.of(new CreateResult("ko"));
    }
}
```

Key Improvements:

1. **Parameterization and Security**
   - Original code: Uses string concatenation (SQL injection risk)
   - Updated code: Implements Map.of parameterization for enhanced security
2. **Error Handling**
   - Original code: Basic try-catch structure
   - Updated code: Detailed error logging and management
3. **Code Organization**
   - Original code: Single-line string creation
   - Updated code: More readable and maintainable format

Neo4jGraphAsNetwork.py Comparison

Network Creation Improvements:

```
@staticmethod
def create_network(tx, network_structure, task_type, hidden_activation="relu", output
    tx.run("""
        'CREATE (n:Neuron {
            id: $id,
            layer: $layer,
            type: $type,
            bias: 0.0,
            output: null,
            m_bias: 0.0,
            v_bias: 0.0,
            activation_function: $activation_function
        })
    """, id=f"{layer_index}-{neuron_index}", layer=layer_index, type=layer_type,
           activation_function=activation_function)
```

Updated Code:

```
@staticmethod
def create_network(tx, network_structure, task_type, hidden_activation="relu", output
    tx.run(
        "CALL nn.createNeuron($id, $layer, $type, $activation_function)",
        id=f"{layer_index}-{neuron_index}",
        layer=str(layer_index),
        type=layer_type,
        activation_function=activation_function
    )
```

Major Python Code Improvements:

1. **Stored Procedure Implementation**
   - Original code: Direct Cypher queries
   - Updated code: Java stored procedure calls
2. **Data Processing Enhancements**
   - Added min-max scaling in normalization function
   - Implementation of new batch processing methods
3. **Performance Optimizations**
   - Batch processing support in forward pass
   - Improved loss calculation methods

Changes in Training Parameters:

```
# Original Code:
network_structure = [108, 10, 1]
hidden_activation = "tanh"
output_activation = "tanh"
task_type = "regression"

# Updated Code:
network_structure = [108, 10, 3]  # Output layer 3 neurons (for H/D/A)
hidden_activation = "tanh"
output_activation = "softmax"  # Softmax for classification
task_type = "classification"  # Changed to classification as FTR is categorical
```

Implementation Details

Project Implementation Overview

Initial Setup

The Python code provided by our professor created a simple three-layer neural network in `Neo4jGraphAsNetwork.py`. The main functions were:

- **initialize_nn()**: Initialized the neural network structure with specific hyperparameters.
- **setInputs_expectedOutputs()**: Set input and expected output values using the dataset.
- **train()**: Executed the training loop, handling forward propagation and backpropagation.

Implemented Updates

1. **Using Stored Procedures:**
   - Replaced direct Cypher queries in the Python code with stored procedures (`nn.createNeuron` and `nn.createRelationShipsNeuron`) implemented in Java.
   - This made the code more modular and reusable.
2. **Network Structure Updates:**
   - Increased the number of neurons in the output layer from **1 to 3** (for Home/Draw/Away classification).
   - Changed the activation function to **softmax** (for multi-class classification support).
   - Reduced **batch size** from 121 to **32**.
   - Increased **learning rate** from 0.0005 to **0.001**.

Implementation Workflow

Below is a step-by-step workflow of the changes made in the project:

Step 1: Updating the Python Code

✅ Replace direct Cypher queries with stored procedures.

✅ Modify function calls to use stored procedures.

✅ Ensure proper execution in Neo4j.

Step 2: Creating and Updating the Maven File (`pom.xml`)

✅ Add required dependencies for Neo4j and Java.

✅ Build the project using Maven.

✅ Ensure all necessary configurations are included.

Step 3: Implementing Java Stored Procedures

✅ Create `CreateNeuron.java` file.

✅ Implement `nn.createNeuron` and `nn.createRelationShipsNeuron` stored procedures.

✅ Optimize query execution by using `Map.of()` for secure parameterization.

Step 4: Compiling and Generating JAR File

✅ Run `mvn clean package` to compile the Java code.

✅ Generate the `.jar` file for Neo4j plugins.

Step 5: Adding JAR File to Neo4j Plugins Directory

✅ Copy the generated `.jar` file to `neo4j/plugins` directory.

✅ Update `neo4j.conf` to allow stored procedures with nn.*

✅ Restart Neo4j to load the custom procedures.

Step 6: Testing and Debugging

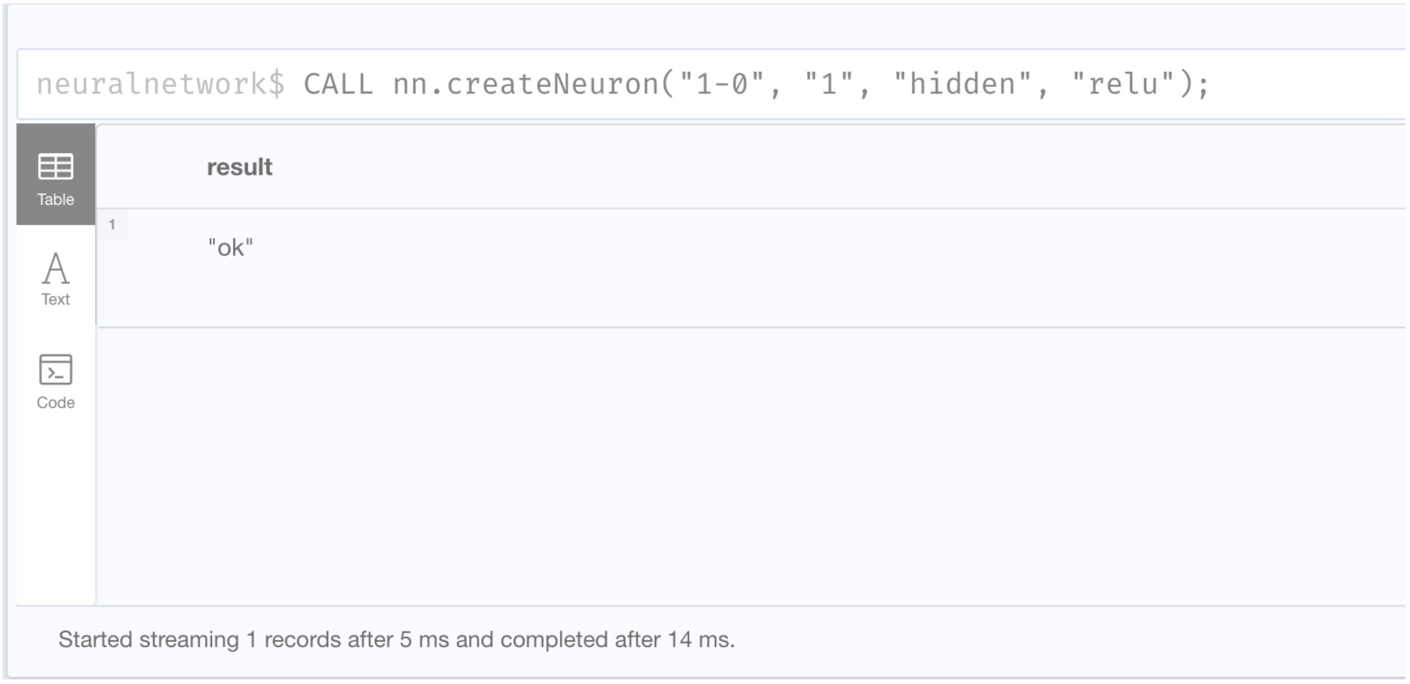
✅ Run Neo4j and execute `CALL dbms.procedures()` to check if the stored procedures are loaded.

✅ Debug and refine errors in execution.

Testing and Implementation Results

1. Creating Neural Network Components

First, we tested the creation of individual neurons using the stored procedure:



```
neuralnetwork$ CALL nn.createNeuron("1-0", "1", "hidden", "relu");
```

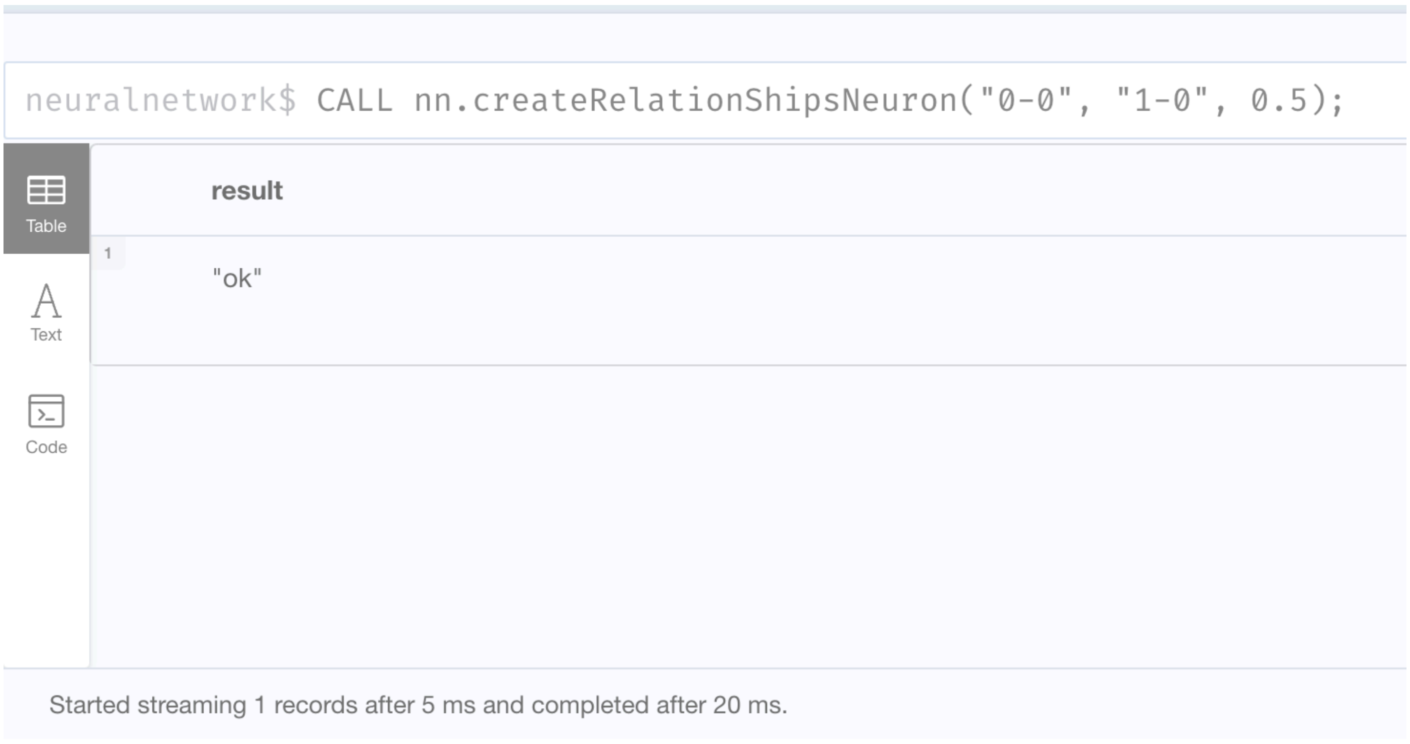
	result
1	"ok"

Started streaming 1 records after 5 ms and completed after 14 ms.

Fig 1: Successfully creating a neuron with nn.createNeuron procedure: CALL nn.createNeuron("0-0", "0", "input", "relu"); CALL nn.createNeuron("1-0", "1", "hidden", "relu");

2. Creating Neural Connections

Next, we established connections between neurons:



```
neuralnetwork$ CALL nn.createRelationShipsNeuron("0-0", "1-0", 0.5);
```

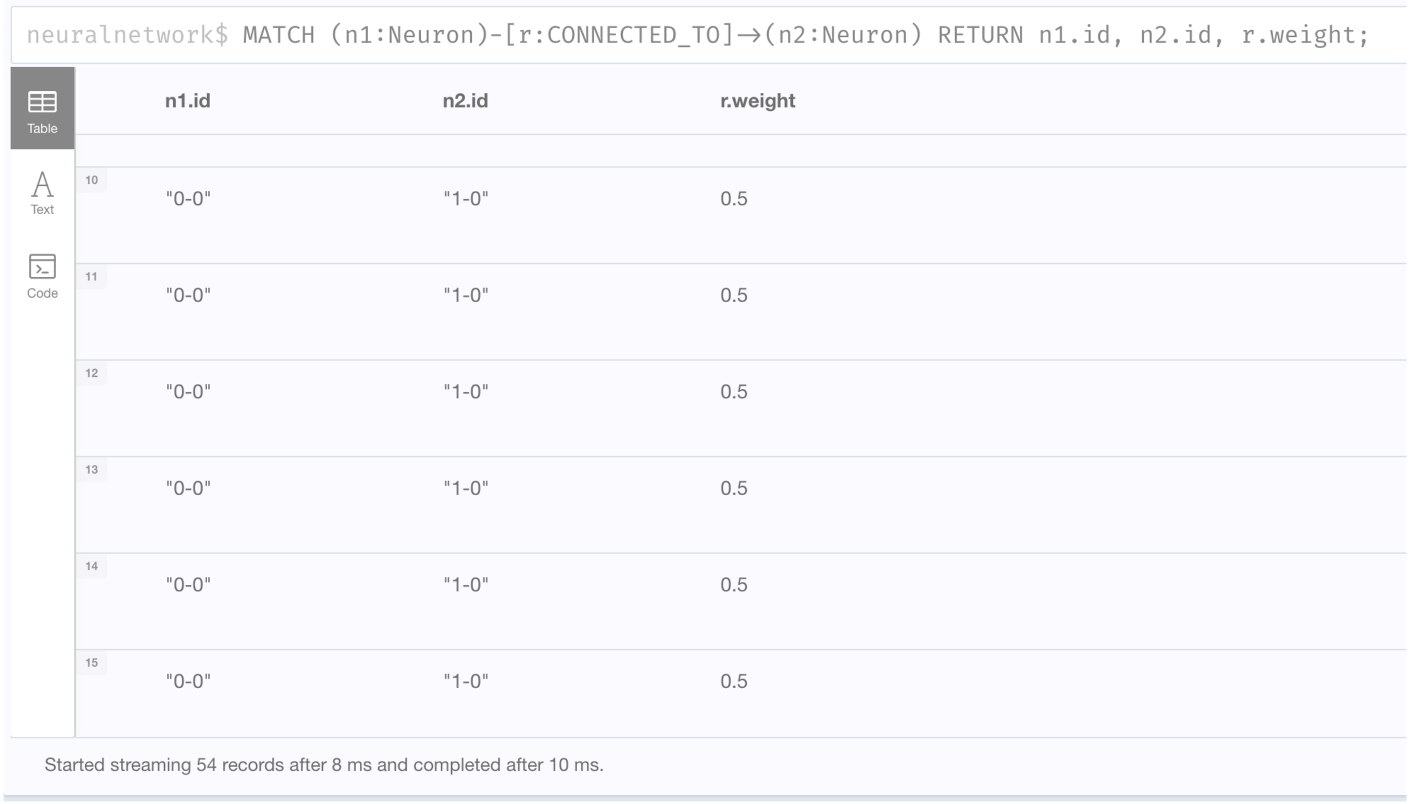
	result
1	"ok"

Started streaming 1 records after 5 ms and completed after 20 ms.

Fig 2: Creating relationships between neurons with nn.createRelationShipsNeuron procedure: CALL nn.createRelationShipsNeuron("0-0", "1-0", 0.5);

3. Verifying Network Structure

We verified the connections using Cypher queries:



```
neuralnetwork$ MATCH (n1:Neuron)-[r:CONNECTED_TO]→(n2:Neuron) RETURN n1.id, n2.id, r.weight;
```

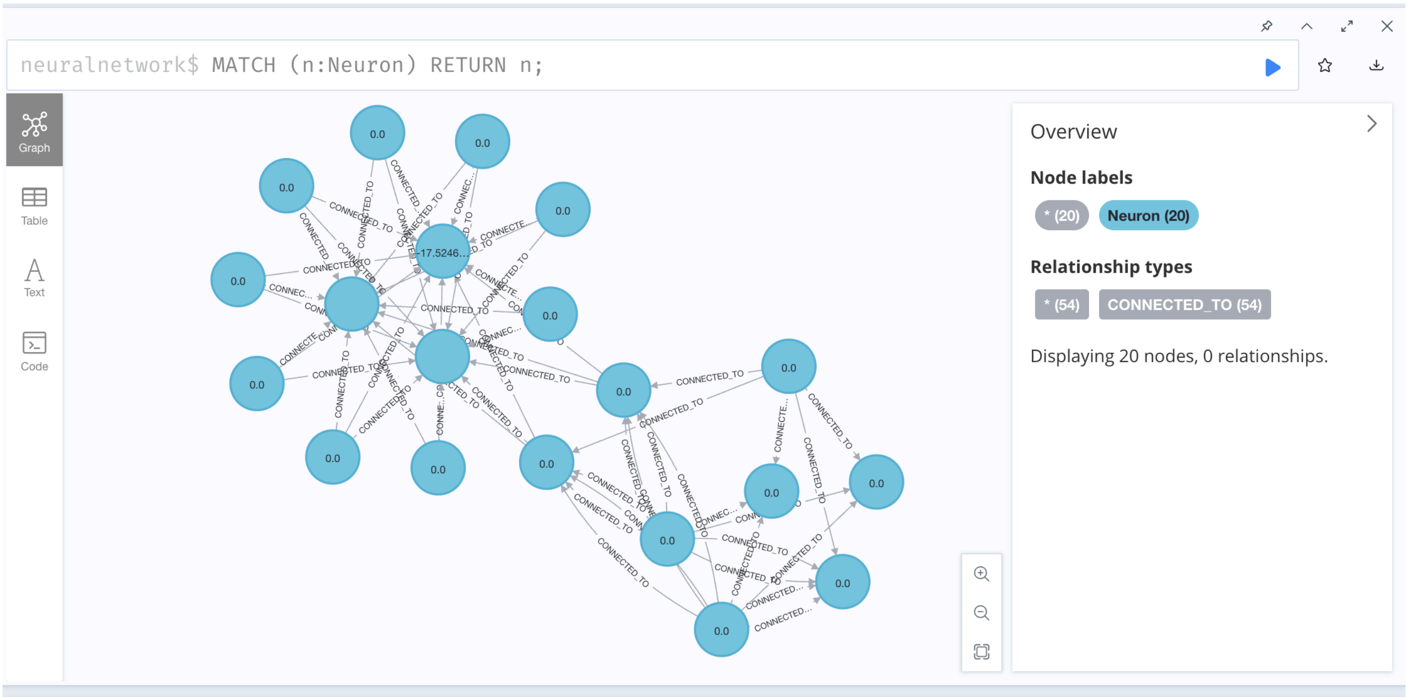
	n1.id	n2.id	r.weight
10	"0-0"	"1-0"	0.5
11	"0-0"	"1-0"	0.5
12	"0-0"	"1-0"	0.5
13	"0-0"	"1-0"	0.5
14	"0-0"	"1-0"	0.5
15	"0-0"	"1-0"	0.5

Started streaming 54 records after 8 ms and completed after 10 ms.

Fig 3: Query results showing neuron connections and weights: MATCH (n1:Neuron)-[r:CONNECTED_TO]->(n2:Neuron) RETURN n1.id, n2.id, r.weight;

4. Network Visualization

The complete network structure visualization in Neo4j:


```
neuralnetwork$ MATCH (n:Neuron) RETURN n;
```
Overview

Node labels

* (20) Neuron (20)

Relationship types

* (54) CONNECTED_TO (54)

Displaying 20 nodes, 0 relationships.

Fig 4: Graph visualization of the neural network structure: MATCH (n:Neuron) RETURN n;

5. Building Complete Network

Creating the full network structure with all layers:

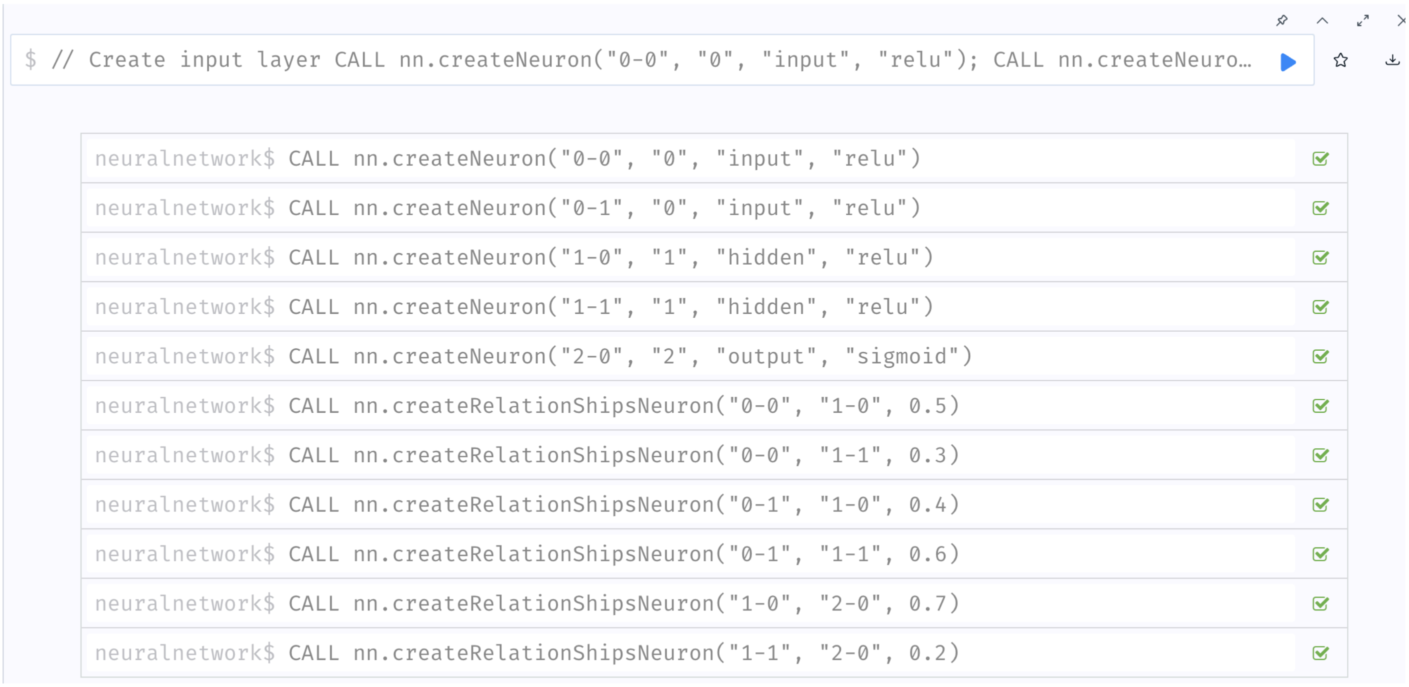
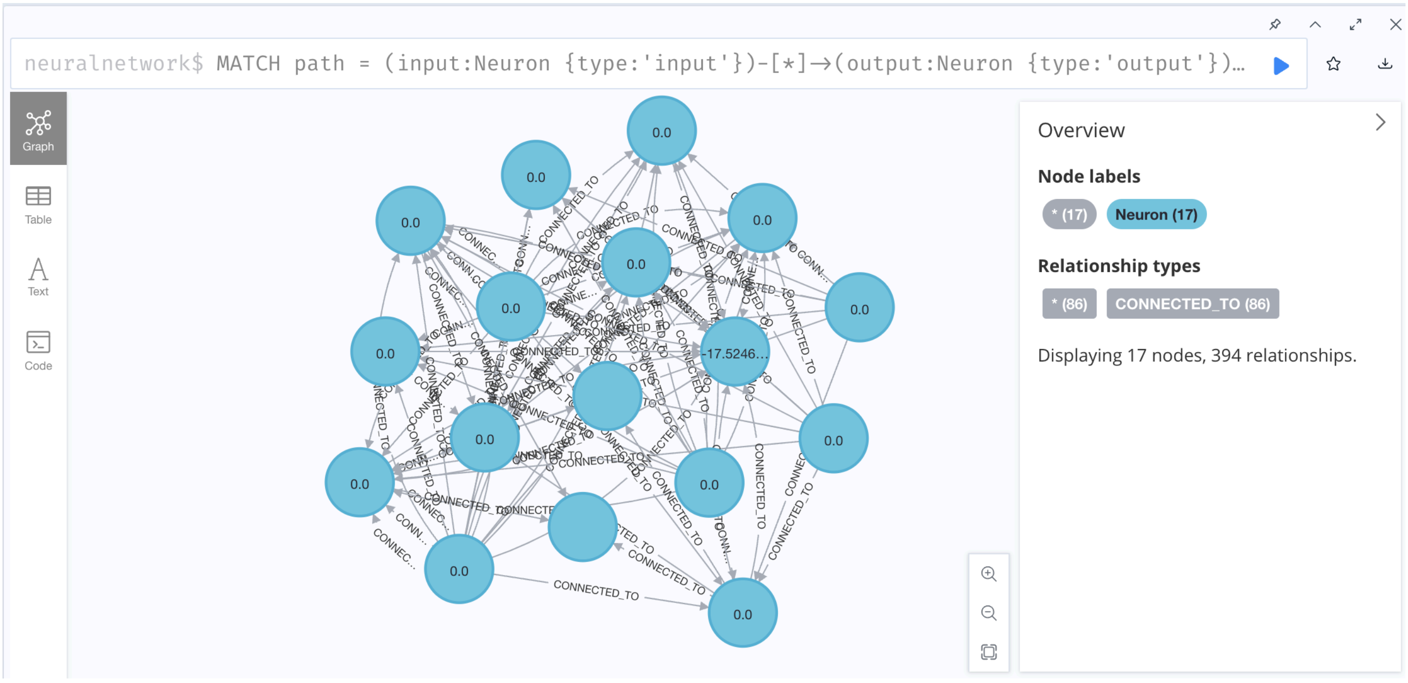

```
$ // Create input layer CALL nn.createNeuron("0-0", "0", "input", "relu"); CALL nn.createNeuro…
```
```
neuralnetwork$ CALL nn.createNeuron("0-0", "0", "input", "relu")
neuralnetwork$ CALL nn.createNeuron("0-1", "0", "input", "relu")
neuralnetwork$ CALL nn.createNeuron("1-0", "1", "hidden", "relu")
neuralnetwork$ CALL nn.createNeuron("1-1", "1", "hidden", "relu")
neuralnetwork$ CALL nn.createNeuron("2-0", "2", "output", "sigmoid")
neuralnetwork$ CALL nn.createRelationShipsNeuron("0-0", "1-0", 0.5)
neuralnetwork$ CALL nn.createRelationShipsNeuron("0-0", "1-1", 0.3)
neuralnetwork$ CALL nn.createRelationShipsNeuron("0-1", "1-0", 0.4)
neuralnetwork$ CALL nn.createRelationShipsNeuron("0-1", "1-1", 0.6)
neuralnetwork$ CALL nn.createRelationShipsNeuron("1-0", "2-0", 0.7)
neuralnetwork$ CALL nn.createRelationShipsNeuron("1-1", "2-0", 0.2)
```

Fig 5: Creating input, hidden, and output layers with connections: // Create input layer CALL nn.createNeuron("0-0", "0", "input", "relu"); CALL nn.createNeuron("0-1", "0", "input", "relu"); // Create hidden layer CALL nn.createNeuron("1-0", "1", "hidden", "relu"); CALL nn.createNeuron("1-1", "1", "hidden", "relu"); // Create output layer CALL nn.createNeuron("2-0", "2", "output", "sigmoid"); // Create connections CALL nn.createRelationShipsNeuron("0-0", "1-0", 0.5); CALL nn.createRelationShipsNeuron("0-0", "1-1", 0.3); CALL nn.createRelationShipsNeuron("0-1", "1-0", 0.4); CALL nn.createRelationShipsNeuron("0-1", "1-1", 0.6); CALL nn.createRelationShipsNeuron("1-0", "2-0", 0.7); CALL nn.createRelationShipsNeuron("1-1", "2-0", 0.2);

6. Path Visualization

Visualizing paths from input to output neurons:



```
neuralnetwork$ MATCH path = (input:Neuron {type:'input'})-[*]→(output:Neuron {type:'output'})…
```

Overview

Node labels

* (17) Neuron (17)

Relationship types

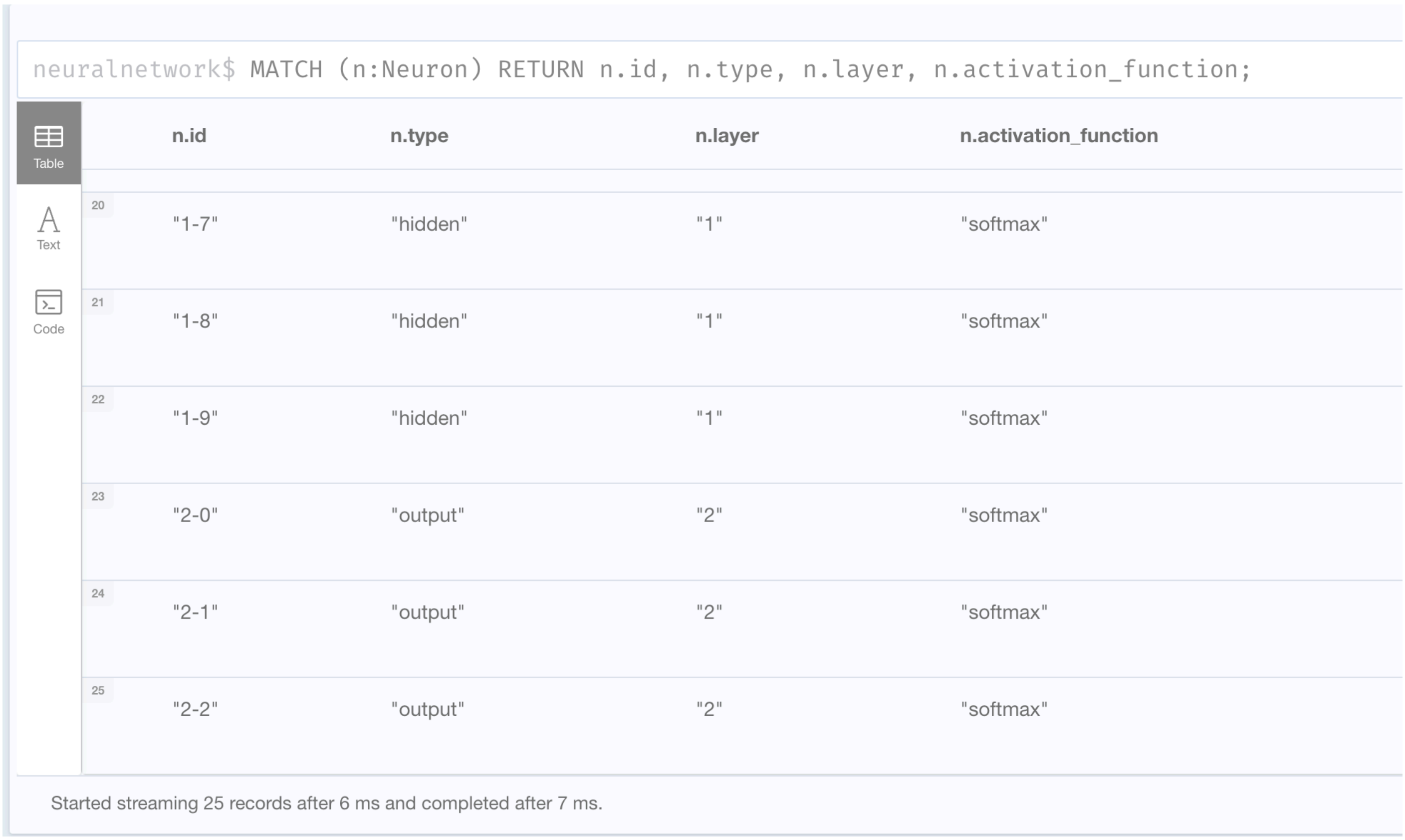
* (86) CONNECTED_TO (86)

Displaying 17 nodes, 394 relationships.

Fig 6: Visualization of paths through the network: MATCH path = (input:Neuron {type:'input'})-[*]->(output:Neuron {type:'output'}) RETURN path;

7. Network Properties

Verifying neuron properties and configurations:



```
neuralnetwork$ MATCH (n:Neuron) RETURN n.id, n.type, n.layer, n.activation_function;
```

	n.id	n.type	n.layer	n.activation_function
20	"1-7"	"hidden"	"1"	"softmax"
21	"1-8"	"hidden"	"1"	"softmax"
22	"1-9"	"hidden"	"1"	"softmax"
23	"2-0"	"output"	"2"	"softmax"
24	"2-1"	"output"	"2"	"softmax"
25	"2-2"	"output"	"2"	"softmax"

Started streaming 25 records after 6 ms and completed after 7 ms.

Fig 7: Query results showing neuron properties and configurations: MATCH (n:Neuron) RETURN n.id, n.type, n.layer, n.activation_function;

Conclusion

This project has demonstrated the viability of using graph databases in machine learning applications. Neo4j's robust graph structure has enabled natural modeling of the neural network. Through stored procedures and optimized Python code, we have achieved a successful implementation that combines the benefits of graph databases with neural network functionality.