Tugas Besar IF3270 Pembelajaran Mesin Artificial Neural Network

Bagian A: Implementasi Forward Propagation untuk Feed Forward Neural Network

Jaringan saraf tiruan pada bagian feed forward neural network (FFNN) menggunakan bahasa python.

Kelompok 1

- 13520004 Gede Prasidha Bhawarnawa
- 13521076 Moh. Aghna Maysan Abyan
- 13521110 Yanuar Sano Nur Rasyid
- 13521152 Muhammad Naufal Nalendra

In [ ]:
```
# Import necessary libraries
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
import sys
import json
```

Fungsi aktivasi

In [ ]:
```
# Softmax Activation Function Implementation
def softmax(arr):
    arr = np.array(arr)
    arr = np.exp(arr)
    sum_arr = np.sum(arr)
    return arr/sum_arr
```

In [ ]:
```
# Sigmoid Activation Function Implementation
def sigmoid(arr):
    numerator = np.exp(arr)
    denominator = numerator + 1
    return numerator / denominator
```

In [ ]:
```
# Linear Activation Function Implementation
def linear(arr):
    return arr
```

In [ ]:
```
# Rectified Linear Unit (ReLU) Activation Function Implementation
def relu(val):
    if val < 0:
        return 0
    else:
        return val
```

In [ ]:
```
def countSSError(results, expected_results):
    results = np.array(results)
```

```
    expected_results = np.array(expected_results)
    if results.shape != expected_results.shape:
        print("Error: array shape mismatch")
        sys.exit(1)
    else:
        return np.sum(np.square(results - expected_results))
```

Implementasi

In [ ]:

```
# Node Class and Instances
class Node:
    def __init__(self, node_number, node_type, node_level, activation_function,
        self.node_number = node_number
        self.node_type = node_type
        self.node_value = node_value # For non-input nodes, this value is 0
        self.node_level = node_level
        self.activation_function = activation_function
        self.weight = weight
        self.output = 0
        self.error = 0

    def getNodeNumber(self):
        return self.node_number

    def getNodeType(self):
        return self.node_type

    def getNodeValue(self):
        return self.node_value

    def setNodeValue(self, node_value):
        self.node_value = node_value

    def getNodeLevel(self):
        return self.node_level

    def getActivationFunction(self):
        return self.activation_function

    def getActivationFunctionValue(self, value):
        if self.activation_function == "linear":
            return linear(value)
        elif self.activation_function == "sigmoid":
            return sigmoid(value)
        elif self.activation_function == "relu":
            return relu(value)
        else: # Uses softmax
            return softmax(value)

    def getWeight(self):
        return self.weight

    def getStrWeight(self):
        return str(self.weight)

    def getNextLayerNodeNumbers(self, nodes):
        next_layer_nodes = []
        for node in nodes:
            if node.getNodeLevel() == self.node_level + 1:
```

```
                next_layer_nodes.append(node.getNodeNumber())
        return next_layer_nodes
```

In [ ]:

```
class Model:
    def __init__(self, number_of_layers:int, number_of_nodes:int, nodes:list, ex
        self.number_of_layers = number_of_layers
        self.number_of_nodes = number_of_nodes
        self.nodes = nodes # We assume that bias are also nodes of value 1
        self.expected_results = expected_results
        self.max_sse = max_sse

    def getNumberOfLayers(self):
        return self.number_of_layers

    def getNumberOfNodes(self):
        return self.number_of_nodes

    def getNodeList(self):
        return self.nodes

    def getNodeByIndex(self, index):
        return self.nodes[index]

    def setNodeValueByIndex(self, index, node_value):
        self.nodes[index].setNodeValue(node_value)

    def getNodeByLevel(self, layer):
        nodes = []
        for node in self.nodes:
            if node.getNodeLevel() == layer:
                nodes.append(node)
        return nodes

    def getExpectedResults(self):
        return self.expected_results

    def getMaxSSE(self):
        return self.max_sse

    def getModelInfo(self):
        print("Number of Layers: ", self.number_of_layers)
        print("Number of Nodes: ", self.number_of_nodes) # Bias is also counted
        print("Nodes: ")
        for node in self.nodes:
            node.getNodeInfo()

    def printModelSummary(self):
        print("Model Summary:")
        print("-----------------------------------------------------")
        print("{:^10} | {:^10} | {:^10} | {:^10} | {:^10}".format("Node Number",
        for level in range(0,self.number_of_layers+2):
            nodes = self.getNodeByLevel(level)
            for node in nodes:
                print("{:^10d} | {:^10s} | {:^10d} | {:^10s} | {:^10s}".format(
                    node.getNodeNumber(), node.getNodeType(), node.getNodeLevel(
        print("-----------------------------------------------------")
        print("Expected Results: ", str(self.expected_results))
        print("Max SSE: ", self.max_sse)

    def getInputNodeNumbers(self):
```

```
        input_nodes = []
        for node in self.nodes:
            if node.getNodeLevel() == 0 and node.getNodeType() == "input":
                input_nodes.append(node.getNodeNumber())
        return input_nodes

    def getOutputNodeValues(self):
        output_nodes = []
        for node in self.nodes:
            if node.getNodeLevel() == self.number_of_layers - 1:
                output_nodes.append(node.getNodeValue())
        return output_nodes

    def printModelVisualization(self):
        edge_list = []
        layer_dict = {}
        for node in self.nodes:
            next_layer_nodes = node.getNextLayerNodeNumbers(self.nodes)
            for next_layer_node in next_layer_nodes:
                if self.getNodeByIndex(next_layer_node).getNodeType() == "bias":
                    pass
                else:
                    edge_pair = (node.getNodeNumber(), next_layer_node)
                    if edge_pair not in edge_list:
                        edge_list.append(edge_pair)
            layer_dict[node.getNodeNumber()] = node.getNodeLevel()
        G = nx.Graph(edge_list)
        nx.set_node_attributes(G, layer_dict, "layer")
        pos = nx.multipartite_layout(G, subset_key="layer", align="vertical")
        nx.draw(G, pos=pos,with_labels=True, node_size=1000, node_color='skyblue
        plt.title("Feed-forward Neural Network Model Representation")
        plt.show()

    def exportToJSON(self, pathname):
        data = {}
        data["model"] = []
        for node in self.nodes:
            node_data = {
                "node_number": node.getNodeNumber(),
                "node_type": node.getNodeType(),
                "node_value": node.getNodeValue(),
                "node_level": node.getNodeLevel(),
                "activation_function": node.getActivationFunction(),
                "weight": node.getWeight()
            }
            data["model"].append(node_data)
        data["expected_results"] = self.expected_results
        data["max_sse"] = self.max_sse
        with open(pathname, 'w') as outfile:
            json.dump(data, outfile)

    def loadJSON(self, pathname):
        with open(pathname) as json_file:
            data = json.load(json_file)
            number_of_layers = 0
            number_of_nodes = 0
            nodes = []
            for node in data["model"]:
                node_instance = Node(node["node_number"], node["node_type"], nod
                nodes.append(node_instance)
```

```
                if node["node_level"] > number_of_layers:
                    number_of_layers = node["node_level"]
                number_of_nodes += 1
            expected_results = data["expected_results"]
            max_sse = data["max_sse"]
            return Model(number_of_layers, number_of_nodes, nodes, expected_resu
```

In [ ]:

```
def readJSON(filename):
    nodes = []
    file = pd.read_json(filename)
    expect = file['expect']
    case = file['case']
    number_of_layers = len(case['model']['layers']) + 1 # Including input layers
    node_counts = [case['model']['input_size']]
    for idx in range(0, number_of_layers - 1):
        layer = case['model']['layers'][idx]
        layer_node_count = layer['number_of_neurons']
        node_counts.append(layer_node_count)
    sum_per_layer = []
    for layer_idx in range(0, number_of_layers - 1):
        sum_per_layer.append(sum(node_counts[0:layer_idx+1]) + layer_idx)
    print("Node counts for all layers in the model: ", node_counts)
    # print("Node counts for all layers in the model: ", node_counts)
    # Set node_idx to start counting the nodes from index 0
    node_idx = 0
    # Set every first node value as bias
    set_node_to_bias = True
    # Retrieve input values
    # input_vals = case['input'][0]
    # print("Input values: ", input_vals)
    # Process input nodes
    for node_count in range(case['model']['input_size'] + 1):
        node_number = node_idx
        node_type = 'bias' if set_node_to_bias else 'input'
        node_level = 0
        node_value = 1 if set_node_to_bias else 0
        activation_function = ''
        # Weights info for the input layer is located on the first array of the
        weights_info = case['weights'][0][node_count]
        weights = {}
        for next_layer_node_idx in range(node_counts[1]):
            weights_idx = -1
            if number_of_layers == 2:
                weights_idx = next_layer_node_idx + sum_per_layer[0] + 1
            elif number_of_layers > 2:
                weights_idx = next_layer_node_idx + sum_per_layer[0] + 2
            weights[weights_idx] = round(weights_info[next_layer_node_idx], 3)
        new_node = Node(node_number, node_type, node_level, activation_function,
        # Set next nodes as non-bias and add 1 to the node_idx
        set_node_to_bias = False
        node_idx += 1
        nodes.append(new_node)
    # Process hidden layer nodes
    if len(node_counts) > 2: # If there are hidden layers, more than just input
        # Reset set_node_to_bias to True for each hidden layer
        set_node_to_bias = True
        for layer_idx in range(len(node_counts) - 1):
            if layer_idx == 0:
                continue # Input Layer has been processed
            for node_count in range(node_counts[layer_idx] + 1):
```

```
                node_number = node_idx
                node_type = 'bias' if set_node_to_bias else 'hidden'
                node_level = layer_idx
                node_value = 1 if set_node_to_bias else 0
                activation_function = case['model']['layers'][layer_idx-1]['acti
                weights_info = case['weights'][layer_idx][node_count]
                weights = {}
                for next_layer_node_idx in range(node_counts[layer_idx + 1]):
                    weight_idx = -1
                    if layer_idx == number_of_layers - 2:
                        weight_idx = next_layer_node_idx + sum_per_layer[layer_i
                    else:
                        weight_idx = next_layer_node_idx + sum_per_layer[layer_i
                    weights[weight_idx] = round(weights_info[next_layer_node_idx
                new_node = Node(node_number, node_type, node_level, activation_f
                set_node_to_bias = False
                node_idx += 1
                nodes.append(new_node)
            set_node_to_bias = True
    # Process output layer nodes
    for node_count in range(node_counts[-1]):
        node_number = node_idx
        node_type = 'output'
        node_level = number_of_layers - 1
        node_value = 0
        activation_function = case['model']['layers'][-1]['activation_function']
        weights = {}
        new_node = Node(node_number, node_type, node_level, activation_function,
        node_idx += 1
        nodes.append(new_node)
    # Check nodes results:
    # for node in nodes:
    #     print("Node Info:")
    #     print("Node Number: ", node.getNodeNumber())
    #     print("Node Type: ", node.getNodeType())
    #     print("Node Level: ", node.getNodeLevel())
    #     print("Node Value: ", node.getNodeValue())
    #     print("Activation Function: ", node.getActivationFunction())
    #     print("Node weights: ", node.getWeight())
    #     print()
    expected_results = expect['output']
    max_sse = expect['max_sse']
    result = Model(number_of_layers, sum(node_counts), nodes, expected_results,
    # print("Model layers: ", number_of_layers)
    # result.getModelInfo()
    result.printModelSummary()
    result.printModelVisualization()
    return result
```

In [ ]:

```
def inferenceData(model_src_name, return_model=False):
    model = readJSON(model_src_name)
    # Fill all input nodes with values and then feed the values to the nodes of
    input_nodes = model.getInputNodeNumbers()
    file = pd.read_json(model_src_name)
    expect = file['expect']
    case = file['case']
    data = case['input']

    if np.array(data).ndim == 2:
        final_results = []
```

```
        for arr in data:
            if len(arr) != len(input_nodes):
                print("Number of input nodes and number of features are not equa
                sys.exit(1)

            for idx in range(len(arr)):
                # print("Inserting value {} to input node {}".format(arr[idx], i
                model.setNodeValueByIndex(input_nodes[idx], arr[idx])

            # Debug first layer, whether the values are inserted correctly
            # for node in model.getNodeByLevel(0):
            #     print("Node {} has value {}".format(node.getNodeNumber(), node

            for layer in range(0, model.getNumberOfLayers()+1):
                # Get all nodes in the layer and then one by one feed the values
                # At first the values will be based purely on the node's values
                # Then it will be passed to the activation function to get the f
                nodes = model.getNodeByLevel(layer)
                next_node_values = {}
                for node in nodes:
                    for next_node_key in node.getWeight().keys():
                        if next_node_key not in next_node_values.keys():
                            next_node_values[next_node_key] = node.getWeight()[n
                        else:
                            next_node_values[next_node_key] += node.getWeight()[
                isSoftmax = False
                for next_node_key in next_node_values.keys():
                    # print("Next node key: ", next_node_key)
                    next_node = model.getNodeByIndex(next_node_key)
                    next_node_value = next_node_values[next_node_key]
                    if next_node.getActivationFunction() != "softmax" and not is
                        isSoftmax = False
                        next_node_value = next_node.getActivationFunctionValue(n
                        next_node.setNodeValue(next_node_value)
                        next_node_values[next_node_key] = next_node_value
                    else:
                        next_node.setNodeValue(next_node_value)
                        next_node_values[next_node_key] = next_node_value
                        isSoftmax = True
                if isSoftmax:
                    # Process all nodes in the layer and then feed the values to
                    process_layer = layer + 1
                    nodes = model.getNodeByLevel(process_layer)
                    node_numbers_in_order = []
                    node_values_in_order = []
                    nodes_new_value_pair_key = {}
                    for node in nodes:
                        node_numbers_in_order.append(node.getNodeNumber())
                        node_values_in_order.append(node.getNodeValue())
                        nodes_new_value_pair_key[node.getNodeNumber()] = 0
                    node_values_in_order = softmax(node_values_in_order)
                    for idx in range(len(node_values_in_order)):
                        nodes_new_value_pair_key[node_numbers_in_order[idx]] = n
                    for node in nodes:
                        if node.getNodeNumber() in nodes_new_value_pair_key.keys
                            node.setNodeValue(nodes_new_value_pair_key[node.getN

                # Debug the layer, whether the values are inserted and calculate
                # for node in model.getNodeByLevel(layer+1):
                #     print("Node {} has value {}".format(node.getNodeNumber(),
```

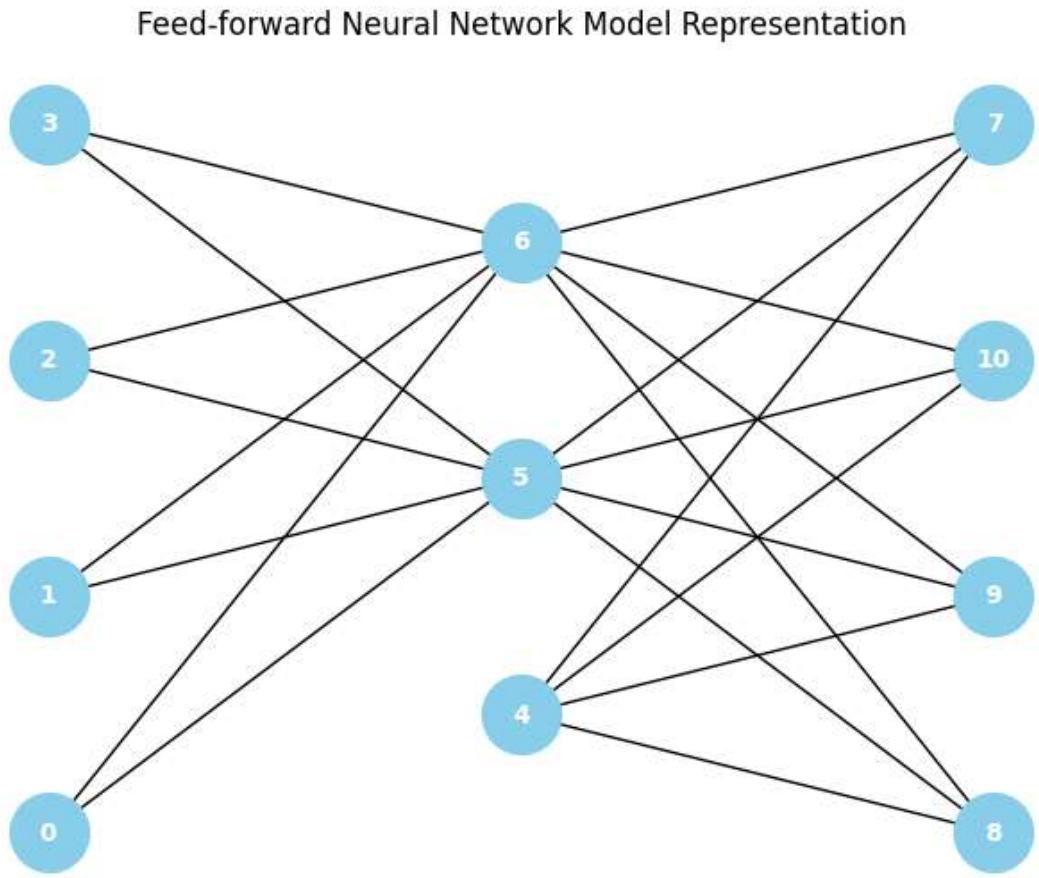
```
            # print(model.getOutputNodeValues())
            final_results.append(model.getOutputNodeValues())
        print("Raw final results: ", [x.tolist() for x in np.array(final_results
        # Compare the results with the expected results
        expected_results = model.getExpectedResults()
        print("Expected results: ", expected_results)
        print("Max SSE: ", model.getMaxSSE(), ", SSE: ", countSSError(final_resu
        if countSSError(final_results, expected_results) <= model.getMaxSSE():
            print("The model is valid as the error is less than or the same as t
        else:
            print("The model is invalid as the error is more than the maximum SS

    if return_model:
        return model
```

Test case sigmoid

In [ ]:
```
# For Sigmoid model (sigmoid.json)
src_name = "testcase/ffnn/sigmoid.json"
inferenceData(src_name, False)
```

```
Node counts for all layers in the model:  [3, 2, 4]
Model Summary:
-------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0       |    bias    |      0      |                     | {5: 0.6, 6: -1.2}
    1       |   input    |      0      |                     | {5: -1.2, 6: -1.7}
    2       |   input    |      0      |                     | {5: 1.4, 6: -1.6}
    3       |   input    |      0      |                     | {5: -0.7, 6: 1.1}
    4       |    bias    |      1      |  sigmoid   | {7: -0.4, 8: 1.6, 9: 1.6, 10:
-1.5}
    5       |   hidden   |      1      |  sigmoid   | {7: -0.0, 8: 0.0, 9: -1.5, 1
0: 0.7}
    6       |   hidden   |      1      |  sigmoid   | {7: 2.1, 8: -0.2, 9: 0.0, 10:
1.8}
    7       |   output   |      2      |  sigmoid   |     {}
    8       |   output   |      2      |  sigmoid   |     {}
    9       |   output   |      2      |  sigmoid   |     {}
    10      |   output   |      2      |  sigmoid   |     {}
-------------------------------------------------
Expected Results:  [[0.41197346, 0.8314294, 0.53018536, 0.31607396], [0.78266141,
0.80843631, 0.55350518, 0.64278501], [0.58987524, 0.82160954, 0.75436518, 0.34919
895], [0.6722003999999999, 0.8166043900000001, 0.59020258, 0.5087098800000001],
[0.47322841, 0.82808466, 0.69105452, 0.29358323000000003]]
Max SSE:  1e-06
```

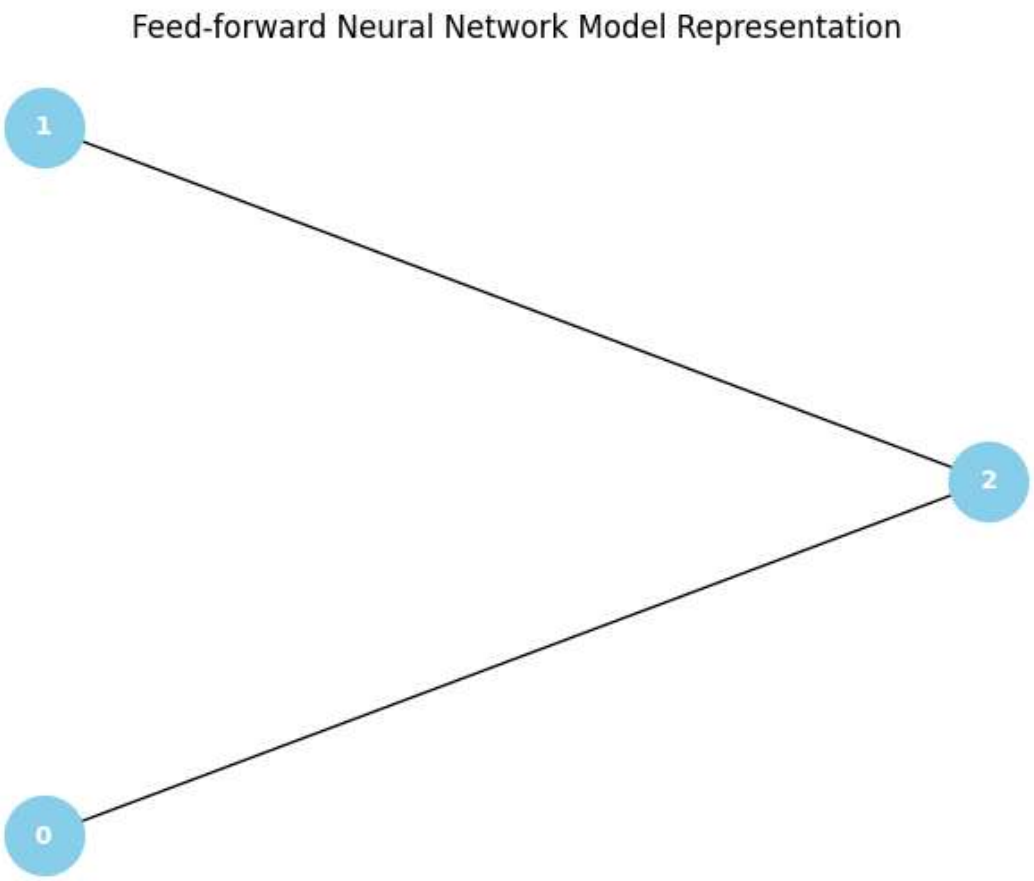



```
Raw final results:  [[0.4119734556100492, 0.8314293993694081, 0.5301853632739774,
0.31607396490909706], [0.7826614091150282, 0.808436308319498, 0.5535051760955713,
0.6427850098146376], [0.5898752435296561, 0.8216095372737501, 0.7543651777362295,
0.3491989467036674], [0.6722003953978934, 0.8166043910016146, 0.5902025844263002,
0.5087098836277426], [0.4732284109743184, 0.8280846566130694, 0.6910545248579614,
0.2935832341653264]]
Expected results:  [[0.41197346, 0.8314294, 0.53018536, 0.31607396], [0.78266141,
0.80843631, 0.55350518, 0.64278501], [0.58987524, 0.82160954, 0.75436518, 0.34919
895], [0.6722003999999999, 0.8166043900000001, 0.59020258, 0.5087098800000001],
[0.47322841, 0.82808466, 0.69105452, 0.29358323000000003]]
Max SSE:  1e-06 , SSE:  2.1756063057014992e-16
The model is valid as the error is less than or the same as the maximum SSE
```

Test case linear

In [ ]:
```
# For linear.json testcase
src_name = "testcase/ffnn/linear.json"
inferenceData(src_name, False)
```

```
Node counts for all layers in the model:  [1, 1]
Model Summary:
-------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0      |    bias    |     0       |                     |  {2: 1.0}
    1      |   input    |     0       |                     |  {2: 3.0}
    2      |   output   |     1       |   linear            |     {}
-------------------------------------------------
Expected Results:  [[-11], [-8], [-5], [-2], [1], [4], [7], [10], [13], [16]]
Max SSE:  1e-12
```

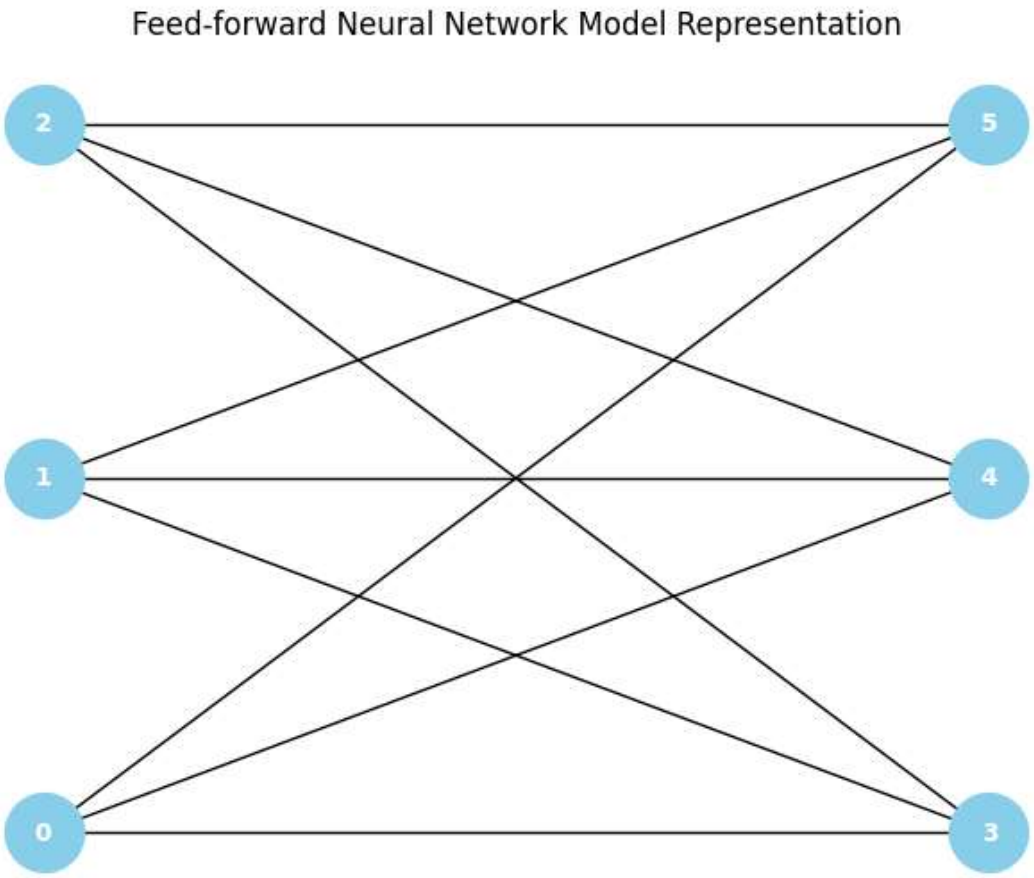



```
Raw final results:  [[-11.0], [-8.0], [-5.0], [-2.0], [1.0], [4.0], [7.0], [10.
0], [13.0], [16.0]]
Expected results:  [[-11], [-8], [-5], [-2], [1], [4], [7], [10], [13], [16]]
Max SSE:  1e-12 , SSE:  0.0
The model is valid as the error is less than or the same as the maximum SSE
```

Test case relu

In [ ]:
```
# For ReLU model (relu.json)
src_name = "testcase/ffnn/relu.json"
inferenceData(src_name, False)
```

```
Node counts for all layers in the model:  [2, 3]
Model Summary:
-------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0      |    bias    |     0       |                     | {3: 0.1, 4: 0.2, 5: 0.3}
    1      |   input    |     0       |                     | {3: 0.47, 4: -0.6, 5: 0.2}
    2      |   input    |     0       |                     | {3: 1.1, 4: -1.3, 5: 0.5}
    3      |   output   |     1       |    relu             |     {}
    4      |   output   |     1       |    relu             |     {}
    5      |   output   |     1       |    relu             |     {}
-------------------------------------------------
Expected Results:  [[0.31, 0, 0.375]]
Max SSE:  1e-06
```

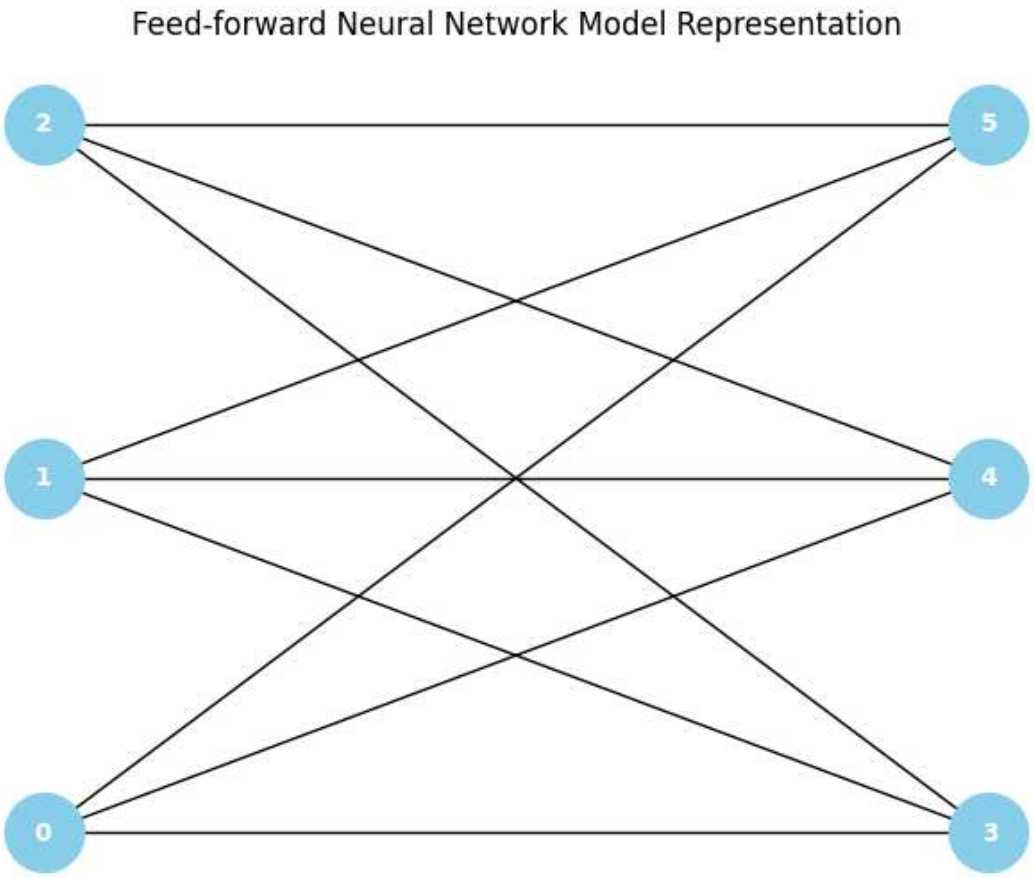



```
Raw final results:  [[0.3099999999999999, 0.0, 0.3750000000000001]]
Expected results:  [[0.31, 0, 0.375]]
Max SSE:  1e-06 , SSE:  2.465190328815662e-32
The model is valid as the error is less than or the same as the maximum SSE
```

Test case relu di docs

In [ ]:
```
# For ReLU model (reludocs.json)
src_name = "testcase/ffnn/reludocs.json"
inferenceData(src_name, False)
```

```
Node counts for all layers in the model:  [2, 3]
Model Summary:
-------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0      |    bias    |     0       |                     | {3: 0.1, 4: 0.2, 5: 0.3}
    1      |   input    |     0       |                     | {3: 0.4, 4: -0.5, 5: 0.6}
    2      |   input    |     0       |                     | {3: 0.7, 4: 0.8, 5: -0.9}
    3      |   output   |     1       |    relu             |     {}
    4      |   output   |     1       |    relu             |     {}
    5      |   output   |     1       |    relu             |     {}
-------------------------------------------------
Expected Results:  [[0.05, 1.1, 0.0]]
Max SSE:  1e-06
```

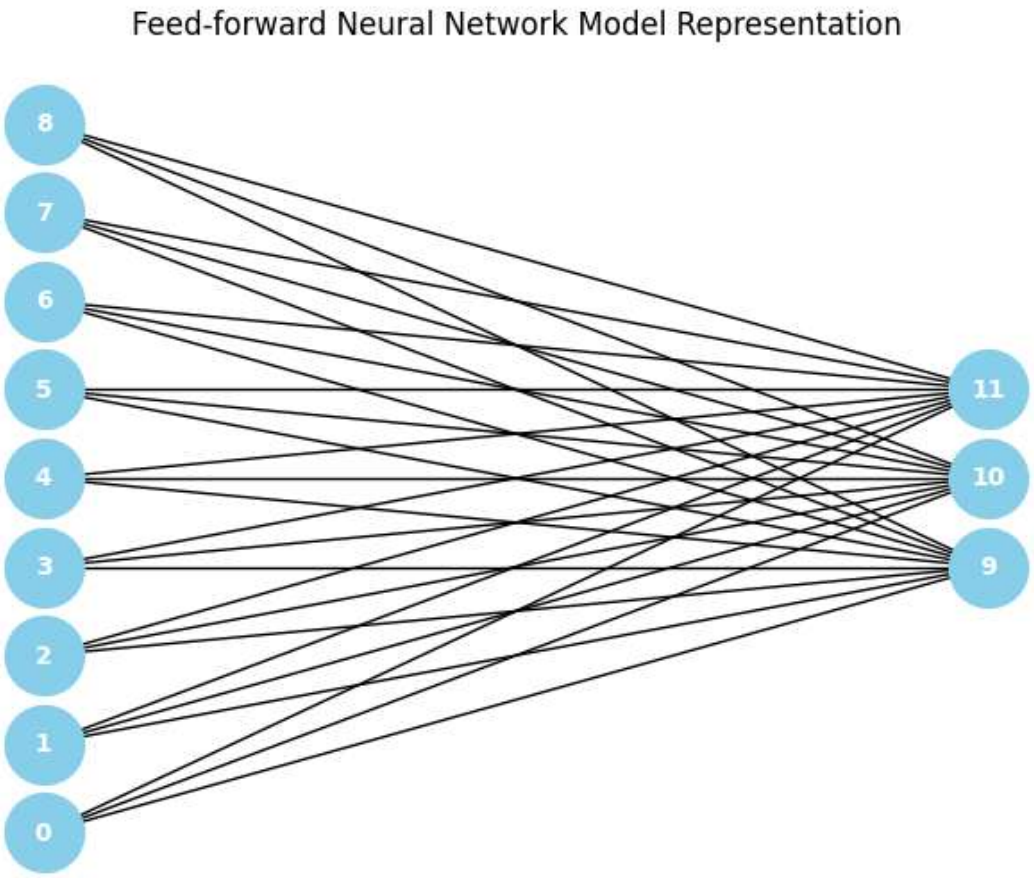



```
Raw final results:  [[0.04999999999999993, 1.1, 0.0]]
Expected results:  [[0.05, 1.1, 0.0]]
Max SSE:  1e-06 , SSE:  4.8148248609680896e-33
The model is valid as the error is less than or the same as the maximum SSE
```

Test case softmax

In [ ]:
```
# For Softmax model (softmax.json)
src_name = "testcase/ffnn/softmax.json"
inferenceData(src_name, False)
```

```
Node counts for all layers in the model:  [8, 3]
Model Summary:
-------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0       |    bias    |     0       |                     | {9: 0.1, 10: 0.9, 11: -0.1}
    1       |   input    |     0       |                     | {9: -0.2, 10: 0.8, 11: 0.2}
    2       |   input    |     0       |                     | {9: 0.3, 10: -0.7, 11: 0.3}
    3       |   input    |     0       |                     | {9: 0.4, 10: 0.6, 11: -0.4}
    4       |   input    |     0       |                     | {9: 0.5, 10: 0.5, 11: 0.5}
    5       |   input    |     0       |                     | {9: -0.6, 10: 0.4, 11: 0.6}
    6       |   input    |     0       |                     | {9: -0.7, 10: -0.3, 11: 0.7}
    7       |   input    |     0       |                     | {9: 0.8, 10: 0.2, 11: -0.8}
    8       |   input    |     0       |                     | {9: 0.9, 10: -0.1, 11: 0.0}
    9       |   output   |     1       |  softmax            |     {}
    10      |   output   |     1       |  softmax            |     {}
    11      |   output   |     1       |  softmax            |     {}
-------------------------------------------------
Expected Results:  [[0.76439061, 0.21168068, 0.02392871]]
Max SSE:  1e-05
```

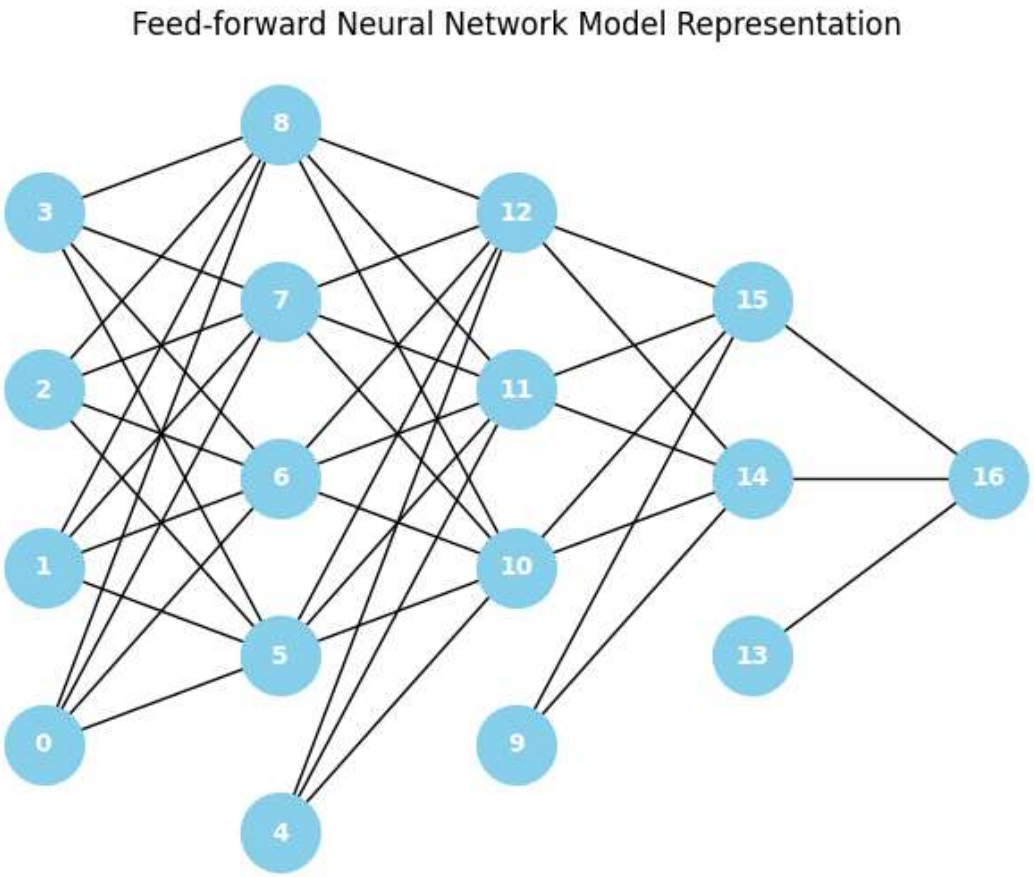



```
Raw final results:  [[0.7643906087005896, 0.21168068289764497, 0.0239287084017654
85]]
Expected results:  [[0.76439061, 0.21168068, 0.02392871]]
Max SSE:  1e-05 , SSE:  1.263916713483999e-17
The model is valid as the error is less than or the same as the maximum SSE
```

Test case multilayer

In [ ]:
```
# For Multilayer model (multilayer.json)
src_name = "testcase/ffnn/multilayer.json"
inferenceData(src_name, False)
```

```
Node counts for all layers in the model:  [3, 4, 3, 2, 1]
Model Summary:
---------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0      |    bias    |     0       |                     | {5: 0.1, 6: 0.2, 7: 0.3, 8: -
1.2}
    1      |   input    |     0       |                     | {5: -0.5, 6: 0.6, 7: 0.7, 8:
0.5}
    2      |   input    |     0       |                     | {5: 0.9, 6: 1.0, 7: -1.1, 8:
-1.0}
    3      |   input    |     0       |                     | {5: 1.3, 6: 1.4, 7: 1.5, 8:
0.1}
    4      |    bias    |     1       |    relu             | {10: 0.1, 11: 0.1, 12: 0.3}
    5      |   hidden   |     1       |    relu             | {10: -0.4, 11: 0.5, 12: 0.6}
    6      |   hidden   |     1       |    relu             | {10: 0.7, 11: 0.4, 12: -0.9}
    7      |   hidden   |     1       |    relu             | {10: 0.2, 11: 0.3, 12: 0.4}
    8      |   hidden   |     1       |    relu             | {10: -0.1, 11: 0.2, 12: 0.1}
    9      |    bias    |     2       |    relu             | {14: 0.1, 15: 0.2}
    10     |   hidden   |     2       |    relu             | {14: -0.3, 15: 0.4}
    11     |   hidden   |     2       |    relu             | {14: 0.6, 15: 0.1}
    12     |   hidden   |     2       |    relu             | {14: 0.1, 15: -0.4}
    13     |    bias    |     3       |    relu             | {16: 0.1}
    14     |   hidden   |     3       |    relu             | {16: -0.2}
    15     |   hidden   |     3       |    relu             | {16: 0.3}
    16     |   output   |     4       |  sigmoid            |     {}
---------------------------------------------------
Expected Results:  [[0.48467479999999996]]
Max SSE:  1e-06
```

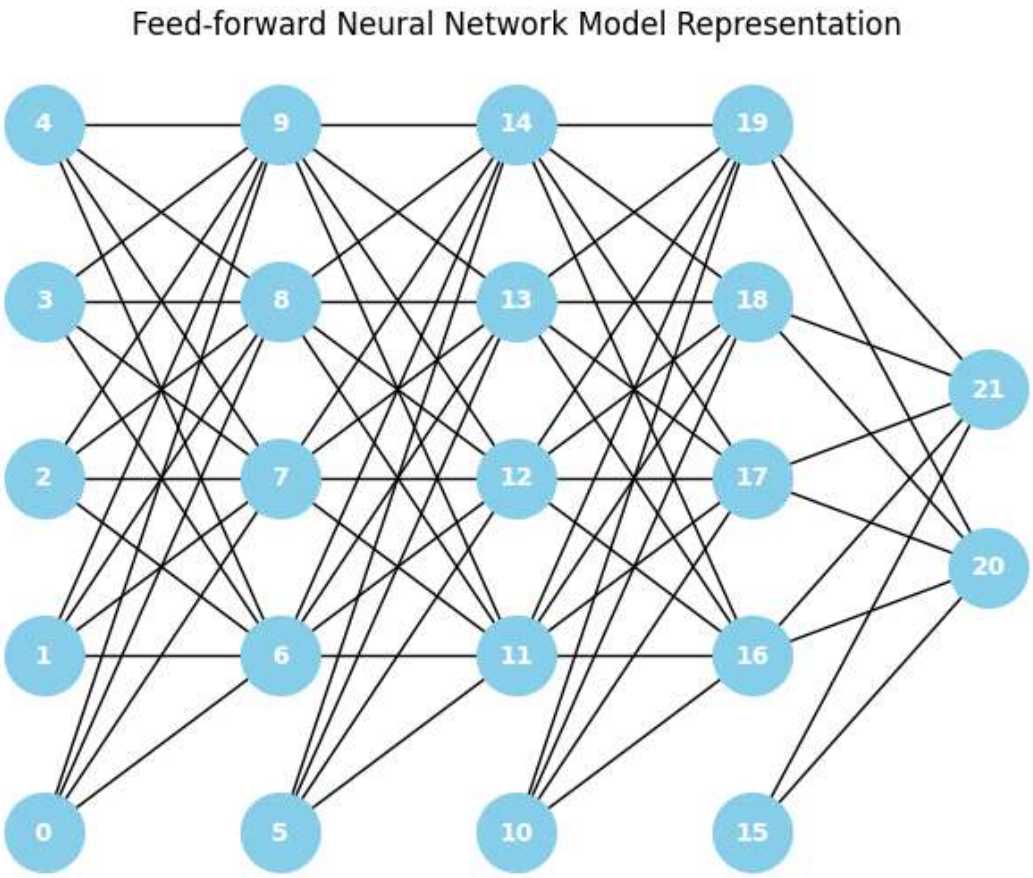



```
Raw final results:  [[0.4846748017763878]]
Expected results:  [[0.48467479999999996]]
Max SSE:  1e-06 , SSE:  3.15555366793979e-18
The model is valid as the error is less than or the same as the maximum SSE
```

Test case multilayer softmax

In [ ]:
```
# For Multilayer model (multilayer.json)
src_name = "testcase/ffnn/multilayer_softmax.json"
inferenceData(src_name, False)
```

```
Node counts for all layers in the model:  [4, 4, 4, 4, 2]
Model Summary:
-------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0      |    bias    |     0       |                     | {6: -0.9, 7: 1.2, 8: -0.6, 9:
0.3}
    1      |   input    |     0       |                     | {6: 0.8, 7: -0.7, 8: 1.1, 9:
-1.2}
    2      |   input    |     0       |                     | {6: 0.3, 7: -1.4, 8: 0.7, 9:
1.2}
    3      |   input    |     0       |                     | {6: 1.1, 7: -1.3, 8: 0.9, 9:
0.4}
    4      |   input    |     0       |                     | {6: 0.5, 7: -0.8, 8: 1.4, 9:
-0.9}
    5      |    bias    |     1       |    relu             | {11: 0.7, 12: -1.1, 13: 0.2,
14: -1.4}
    6      |   hidden   |     1       |    relu             | {11: 1.3, 12: -0.6, 13: 0.5,
14: -1.3}
    7      |   hidden   |     1       |    relu             | {11: -1.2, 12: 0.9, 13: 1.4,
14: -0.7}
    8      |   hidden   |     1       |    relu             | {11: 0.6, 12: -0.5, 13: 1.2,
14: -1.1}
    9      |   hidden   |     1       |    relu             | {11: 1.0, 12: -0.4, 13: 0.8,
14: -1.0}
    10     |    bias    |     2       |    relu             | {16: -1.3, 17: 0.7, 18: -0.8,
19: 1.3}
    11     |   hidden   |     2       |    relu             | {16: 0.2, 17: -1.0, 18: 1.1,
19: -0.6}
    12     |   hidden   |     2       |    relu             | {16: 1.4, 17: -0.9, 18: 0.3,
19: -1.4}
    13     |   hidden   |     2       |    relu             | {16: -0.7, 17: 1.2, 18: -1.1,
19: 0.5}
    14     |   hidden   |     2       |    relu             | {16: 0.9, 17: -0.7, 18: 1.3,
19: -0.8}
    15     |    bias    |     3       |    relu             | {20: 0.4, 21: -1.1}
    16     |   hidden   |     3       |    relu             | {20: -1.4, 21: 0.3}
    17     |   hidden   |     3       |    relu             | {20: 0.8, 21: 1.2}
    18     |   hidden   |     3       |    relu             | {20: 0.1, 21: -1.2}
    19     |   hidden   |     3       |    relu             | {20: 1.2, 21: 1.4}
    20     |   output   |     4       |  softmax            |     {}
    21     |   output   |     4       |  softmax            |     {}
-------------------------------------------------
Expected Results:  [[0.7042294, 0.2957706]]
Max SSE:  1e-06
```

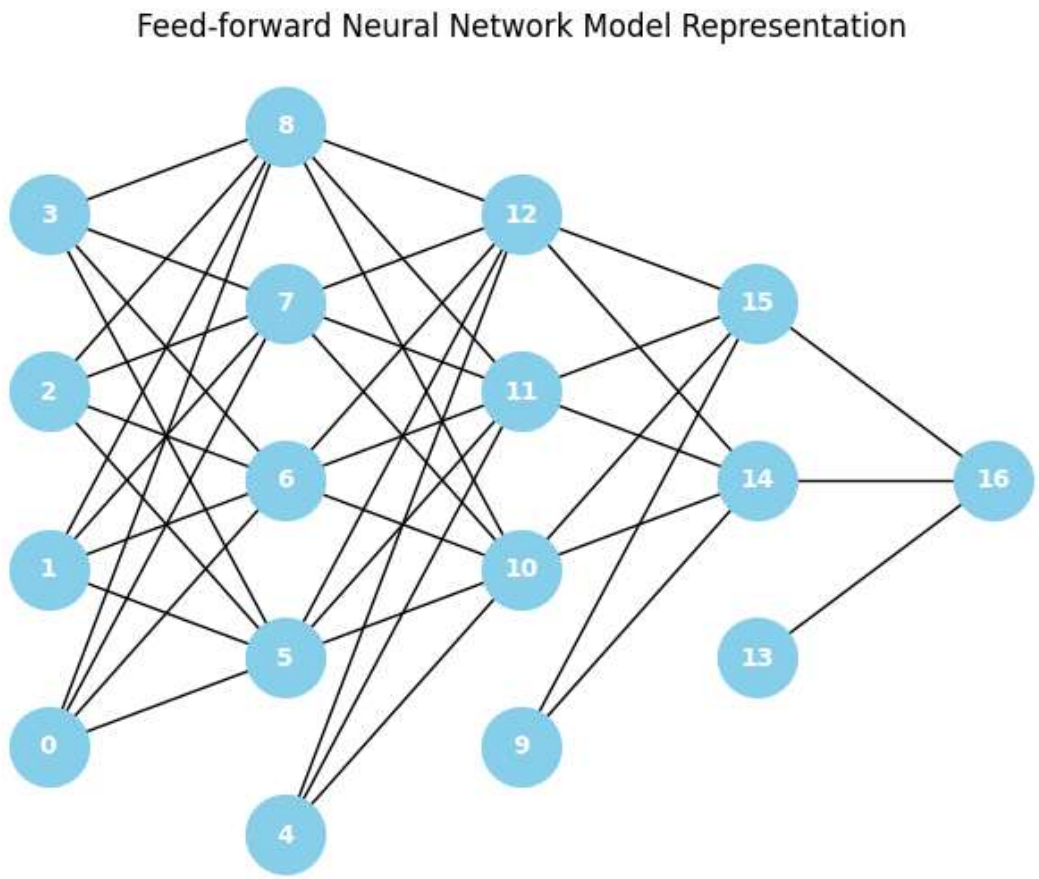



```
Raw final results:  [[0.7042293996883686, 0.2957706003116313]]
Expected results:  [[0.7042294, 0.2957706]]
Max SSE:  1e-06 , SSE:  1.942282085801731e-19
The model is valid as the error is less than or the same as the maximum SSE
```

Test case export and load from and to JSON file (multilayer model)

In [ ]:
```
multilayer_model = inferenceData("testcase/ffnn/multilayer.json", True)
multilayer_model.exportToJSON("exports/multilayer_model.json")
```

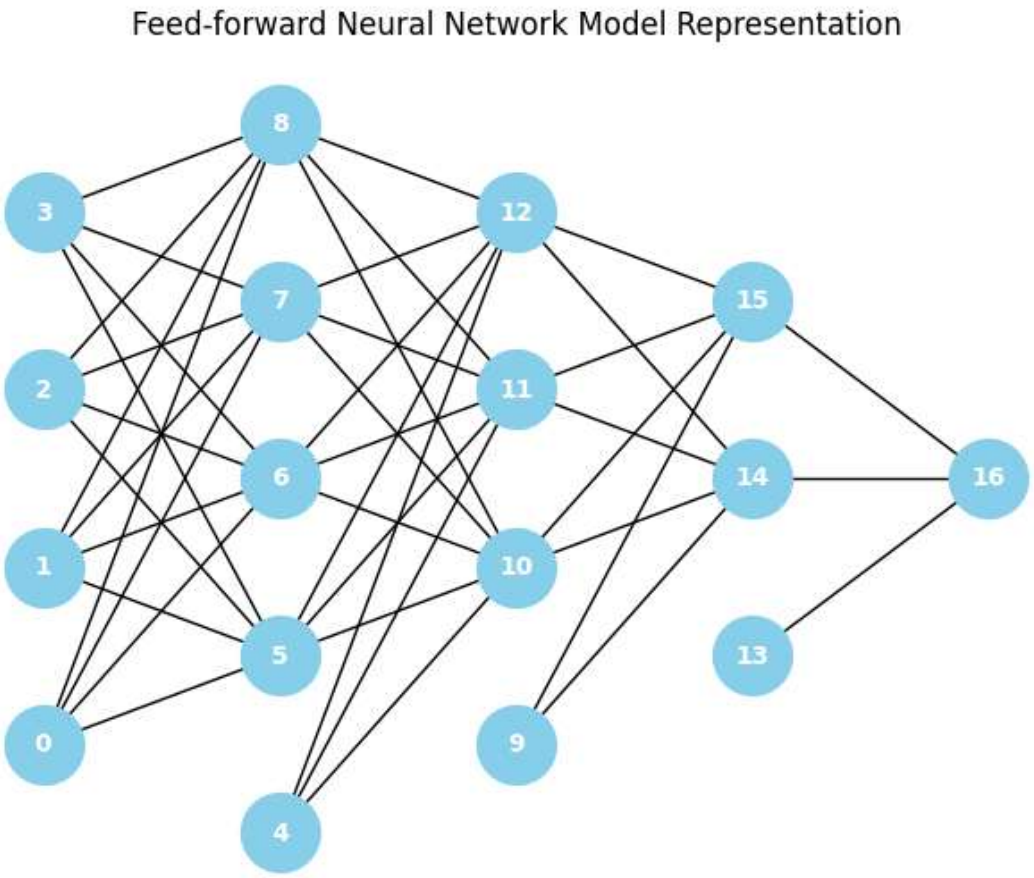
```
Node counts for all layers in the model:  [3, 4, 3, 2, 1]
Model Summary:
---------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0      |    bias    |     0       |                     | {5: 0.1, 6: 0.2, 7: 0.3, 8: -
1.2}
    1      |   input    |     0       |                     | {5: -0.5, 6: 0.6, 7: 0.7, 8:
0.5}
    2      |   input    |     0       |                     | {5: 0.9, 6: 1.0, 7: -1.1, 8:
-1.0}
    3      |   input    |     0       |                     | {5: 1.3, 6: 1.4, 7: 1.5, 8:
0.1}
    4      |    bias    |     1       |    relu             | {10: 0.1, 11: 0.1, 12: 0.3}
    5      |   hidden   |     1       |    relu             | {10: -0.4, 11: 0.5, 12: 0.6}
    6      |   hidden   |     1       |    relu             | {10: 0.7, 11: 0.4, 12: -0.9}
    7      |   hidden   |     1       |    relu             | {10: 0.2, 11: 0.3, 12: 0.4}
    8      |   hidden   |     1       |    relu             | {10: -0.1, 11: 0.2, 12: 0.1}
    9      |    bias    |     2       |    relu             | {14: 0.1, 15: 0.2}
    10     |   hidden   |     2       |    relu             | {14: -0.3, 15: 0.4}
    11     |   hidden   |     2       |    relu             | {14: 0.6, 15: 0.1}
    12     |   hidden   |     2       |    relu             | {14: 0.1, 15: -0.4}
    13     |    bias    |     3       |    relu             | {16: 0.1}
    14     |   hidden   |     3       |    relu             | {16: -0.2}
    15     |   hidden   |     3       |    relu             | {16: 0.3}
    16     |   output   |     4       |  sigmoid            |     {}
---------------------------------------------------
Expected Results:  [[0.48467479999999996]]
Max SSE:  1e-06
```




```
Raw final results:  [[0.4846748017763878]]
Expected results:  [[0.48467479999999996]]
Max SSE:  1e-06 , SSE:  3.15555366793979e-18
The model is valid as the error is less than or the same as the maximum SSE
```

In [ ]:
```
# Load a model from JSON
new_multilayer_model = Model(0, 0, [], [], 0)
new_multilayer_model = new_multilayer_model.loadJSON("exports/multilayer_model.j
new_multilayer_model.printModelSummary()
new_multilayer_model.printModelVisualization()
```

```
Model Summary:
-------------------------------------------------
Node Number | Node Type  | Layer Depth | Activation Function |   Weight
    0      |    bias    |     0       |                     | {'5': 0.1, '6': 0.2, '7': 0.
3, '8': -1.2}
    1      |   input    |     0       |                     | {'5': -0.5, '6': 0.6, '7': 0.
7, '8': 0.5}
    2      |   input    |     0       |                     | {'5': 0.9, '6': 1.0, '7': -1.
1, '8': -1.0}
    3      |   input    |     0       |                     | {'5': 1.3, '6': 1.4, '7': 1.
5, '8': 0.1}
    4      |    bias    |     1       |    relu             | {'10': 0.1, '11': 0.1, '12':
0.3}
    5      |   hidden   |     1       |    relu             | {'10': -0.4, '11': 0.5, '12':
0.6}
    6      |   hidden   |     1       |    relu             | {'10': 0.7, '11': 0.4, '12':
-0.9}
    7      |   hidden   |     1       |    relu             | {'10': 0.2, '11': 0.3, '12':
0.4}
    8      |   hidden   |     1       |    relu             | {'10': -0.1, '11': 0.2, '12':
0.1}
    9      |    bias    |     2       |    relu             | {'14': 0.1, '15': 0.2}
    10     |   hidden   |     2       |    relu             | {'14': -0.3, '15': 0.4}
    11     |   hidden   |     2       |    relu             | {'14': 0.6, '15': 0.1}
    12     |   hidden   |     2       |    relu             | {'14': 0.1, '15': -0.4}
    13     |    bias    |     3       |    relu             | {'16': 0.1}
    14     |   hidden   |     3       |    relu             | {'16': -0.2}
    15     |   hidden   |     3       |    relu             | {'16': 0.3}
    16     |   output   |     4       |  sigmoid            |     {}
-------------------------------------------------
Expected Results:  [[0.48467479999999996]]
Max SSE:  1e-06
```



Bagian B

In [ ]:
```python
from matplotlib import pyplot as plt
import numpy as np
import os

import json
import os
```

In [ ]:
```python
def parse_json_file(file_path: str):
    with open(file_path, 'r') as json_file:
        data = json.load(json_file)

    case = data['case']
    expect = data['expect']

    return case, expect

def parse_case(data: dict):
    model = data['model']
    input_data = data['input']
    initial_weights = data['initial_weights']
    target = data['target']
    learning_parameters = data['learning_parameters']

    return model, input_data, initial_weights, target, learning_parameters

def parse_model(data: dict):
    input_size = data['input_size']
    layers = data['layers']
```

```
    return input_size, layers

def parse_layers(data: dict):
    number_of_neurons = data['number_of_neurons']
    activation_function = data['activation_function']

    return number_of_neurons, activation_function

def parse_learning_parameters(data: dict):
    learning_rate = data['learning_rate']
    batch_size = data['batch_size']
    max_iteration = data['max_iteration']
    error_threshold = data['error_threshold']

    return learning_rate, batch_size, max_iteration, error_threshold

def parse_expect(data: dict):
    stopped_by = data['stopped_by']
    final_weights = data['final_weights']

    return stopped_by, final_weights

def save_json_file(file_path: str, data: dict):
    with open(file_path, 'w') as json_file:
        json.dump(data, json_file)
```

In [ ]:

```
class ActivationFunctions:
    @staticmethod
    def linear(x, derivative=False):
        if derivative:
            return 1
        return x

    @staticmethod
    def relu(x, derivative=False):
        if derivative:
            return np.where(x <= 0, 0, 1)
        return np.maximum(0, x)

    @staticmethod
    def sigmoid(x, derivative=False):
        if derivative:
            return x * (1 - x)
        return 1 / (1 + np.exp(-x))

    @staticmethod
    def softmax(x, derivative=False):
        if derivative:
            return x * (1 - x)
        return np.exp(x) / np.sum(np.exp(x), axis=0, keepdims=True)
```

In [ ]:

```
class NN:
    def __init__(self, model_config, initial_weights):
        self.input_size, self.layers = parse_model(model_config)
        self.weights = np.array(initial_weights)
        self.bias = 1
        self.learning_rate = None
        self.batch_size = None
        self.max_iteration = None
```

```
        self.error_threshold = None

    def forward_propagation(self, x):
        input_data = np.array(x)

        for layer, weight in zip(self.layers, self.weights):
            input_data = np.append(self.bias, input_data)
            activation_function = getattr(ActivationFunctions, layer['activation
            output = activation_function(np.dot(input_data, weight))

            input_data = output

        return output

    def train(self, input_data, target, learning_rate, batch_size, max_iteration
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.max_iteration = max_iteration
        self.error_threshold = error_threshold

        for epoch in range(max_iteration):
            total_error = 0.0

            for i in range(0, len(input_data), batch_size):
                batch_input = input_data[i:i+batch_size]
                batch_target = target[i:i+batch_size]

                batch_error = 0.0

                deltas = [None] * len(self.layers)

                for x, t in zip(batch_input, batch_target):
                    delta = 0
                    output = self.forward_propagation(x)

                    # Compute error
                    error = t - output
                    batch_error += np.sum(error ** 2 / 2)

                    # Backpropagation
                    delta = - error

                    for idx, layer in enumerate(self.layers[::-1]):
                        activation = getattr(ActivationFunctions, layer['activat
                        error = -1 * np.log(output) if activation == "softmax" e
                        delta_output = delta * activation(output, derivative=Tru
                        dw = np.append(self.bias, x)
                        delta_output = learning_rate * np.outer(delta_output, dw

                        if idx > 0 and deltas[-idx] is not None:
                            delta_output = np.dot(self.weights[-idx][ 1:].T, del

                        if deltas[-idx-1] is None:
                            deltas[-idx-1] = delta_output
                        else:
                            deltas[-idx-1] += delta_output


                total_error += batch_error
```

```
                # Update weights
                for idx, delta in enumerate(deltas):
                    self.weights[idx] -= delta.T

            avg_error = total_error / len(input_data)


            print(f"Epoch {epoch + 1}/{max_iteration}, Average Error: {avg_error

            if avg_error < error_threshold:
                print("Training stopped: Error threshold reached.")
                break

        print("Training complete.")
```

In [ ]:
```
file_path = os.path.join('testcase', 'linear.json')

case, expect = parse_json_file(file_path)
model, input_data, initial_weights, target, learning_parameters = parse_case(cas
stopped_by, final_weights = parse_expect(expect)


nn = NN(model, initial_weights)
learning_rate, batch_size, max_iteration, error_threshold = parse_learning_param
nn.train(input_data, target, learning_rate, batch_size, max_iteration, error_thr

print(nn.weights)
```

```
Epoch 1/1, Average Error: 0.3325
Training complete.
[[[ 0.22  0.36  0.11]
  [ 0.64  0.3  -0.89]
  [ 0.28 -0.7   0.37]]]
```

In [ ]:
```
file_path = os.path.join('testcase', 'linear_small_lr.json')

case, expect = parse_json_file(file_path)
model, input_data, initial_weights, target, learning_parameters = parse_case(cas
stopped_by, final_weights = parse_expect(expect)


nn = NN(model, initial_weights)
learning_rate, batch_size, max_iteration, error_threshold = parse_learning_param
nn.train(input_data, target, learning_rate, batch_size, max_iteration, error_thr

print(nn.weights)
```

```
Epoch 1/1, Average Error: 0.3325
Training complete.
[[[ 0.1012  0.3006  0.1991]
  [ 0.4024  0.201  -0.7019]
  [ 0.1018 -0.799   0.4987]]]
```

In [ ]:
```
file_path = os.path.join('testcase', 'linear_two_iteration.json')

case, expect = parse_json_file(file_path)
model, input_data, initial_weights, target, learning_parameters = parse_case(cas
stopped_by, final_weights = parse_expect(expect)
```

```
nn = NN(model, initial_weights)
learning_rate, batch_size, max_iteration, error_threshold = parse_learning_param
nn.train(input_data, target, learning_rate, batch_size, max_iteration, error_thr

print(nn.weights)
```

```
Epoch 1/2, Average Error: 0.3325
Epoch 2/2, Average Error: 0.09092499999999996
Training complete.
[[[ 0.166  0.338  0.153]
  [ 0.502  0.226 -0.789]
  [ 0.214 -0.718  0.427]]]
```

In [ ]:
```
file_path = os.path.join('testcase', 'relu_b.json')

case, expect = parse_json_file(file_path)
model, input_data, initial_weights, target, learning_parameters = parse_case(cas
stopped_by, final_weights = parse_expect(expect)


nn = NN(model, initial_weights)
learning_rate, batch_size, max_iteration, error_threshold = parse_learning_param
nn.train(input_data, target, learning_rate, batch_size, max_iteration, error_thr

print(nn.weights)
```

```
Epoch 1/1, Average Error: 1.3967749999999999
Training complete.
[[[-0.211   0.105   0.885 ]
  [ 0.3033  0.5285  0.3005]
  [-0.489  -0.905   0.291 ]]]
```

In [ ]:
```
file_path = os.path.join('testcase', 'sigmoid.json')

case, expect = parse_json_file(file_path)
model, input_data, initial_weights, target, learning_parameters = parse_case(cas
stopped_by, final_weights = parse_expect(expect)


nn = NN(model, initial_weights)
learning_rate, batch_size, max_iteration, error_threshold = parse_learning_param
nn.train(input_data, target, learning_rate, batch_size, max_iteration, error_thr

print(nn.weights)
```

```
Epoch 1/10, Average Error: 0.23640507030096655
Epoch 2/10, Average Error: 0.23539036744879444
Epoch 3/10, Average Error: 0.23438378905993162
Epoch 4/10, Average Error: 0.2333853294417322
Epoch 5/10, Average Error: 0.2323949777800034
Epoch 6/10, Average Error: 0.2314127183172226
Epoch 7/10, Average Error: 0.23043853053433763
Epoch 8/10, Average Error: 0.22947238933549008
Epoch 9/10, Average Error: 0.22851426523501334
Epoch 10/10, Average Error: 0.2275641245460821
Training complete.
[[[0.23291176 0.06015346]
  [0.12884088 0.64849474]
  [0.837615   0.23158199]]]
```

In [ ]:
```
file_path = os.path.join('testcase', 'mlp.json')

case, expect = parse_json_file(file_path)
model, input_data, initial_weights, target, learning_parameters = parse_case(cas
stopped_by, final_weights = parse_expect(expect)


nn = NN(model, initial_weights)
learning_rate, batch_size, max_iteration, error_threshold = parse_learning_param
nn.train(input_data, target, learning_rate, batch_size, max_iteration, error_thr

print(nn.weights)
```

```
Epoch 1/1, Average Error: 0.338476
Training complete.
[[[ 0.22144   0.1956  ]
  [-0.34752   0.58888 ]
  [ 0.350368  0.41464 ]]

 [[ 0.2748    0.188   ]
  [ 0.3252   -0.4824  ]
  [ 0.71496   0.712   ]]]
```

In [ ]:
```
file_path = os.path.join('testcase', 'softmax.json')

case, expect = parse_json_file(file_path)
model, input_data, initial_weights, target, learning_parameters = parse_case(cas
stopped_by, final_weights = parse_expect(expect)


nn = NN(model, initial_weights)
learning_rate, batch_size, max_iteration, error_threshold = parse_learning_param
nn.train(input_data, target, learning_rate, batch_size, max_iteration, error_thr

print(nn.weights)
```

```
Epoch 1/10, Average Error: 0.6626829218214885
Epoch 2/10, Average Error: 0.6497532419843801
Epoch 3/10, Average Error: 0.6363579868724346
Epoch 4/10, Average Error: 0.622602255574003
Epoch 5/10, Average Error: 0.6086185574566528
Epoch 6/10, Average Error: 0.5945610927498114
Epoch 7/10, Average Error: 0.5805961840120056
Epoch 8/10, Average Error: 0.5668899957351127
Epoch 9/10, Average Error: 0.5535956018123217
Epoch 10/10, Average Error: 0.5408418171596573
Training complete.
[[[ 0.0755546   0.9088089  -0.08597513]
  [-0.18370219  0.75633116  0.2249669 ]
  [ 0.31604839 -0.75296816  0.33318705]
  [ 0.40542907  0.58987544 -0.39566149]
  [ 0.46957724  0.49230158  0.5346057 ]
  [-0.6407751   0.43246096  0.60751439]
  [-0.70024868 -0.32084212  0.71881792]
  [ 0.74063868  0.22981958 -0.77298742]
  [ 0.85639344 -0.06579594  0.00836027]]]
```