LAPORAN TUGAS BESAR 1

“Implementasi Feed Forward Neural Network”

Laporan Ini Dibuat Untuk Memenuhi Tugas Perkuliahan

Mata Kuliah Pembelajaran Mesin (IF3270)

Disusun Oleh:

13521080	Fajar Maulana Herawan
13521109	Rizky Abdillah Rasyid
13521119	Muhammad Rizky Sya'ban
13521131	Jeremya Dharmawan Raharjo

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

DAFTAR ISI 2

BAB I
DASAR TEORI 3

BAB II
IMPLEMENTASI 4

1. ActivationFunction.py 4
2. Layer.py 6
3. OutputLayer.py 8
4. HiddenLayer.py 8
5. ANN.py 9
6. Model.py 10
7. Parser.py 11

BAB III
HASIL PENGUJIAN 12

BAB IV
PERBANDINGAN DENGAN HASIL MANUAL 19

A. Linear.json 19
B. Multilayer_softmax.json 19
C. relu.json 20
D. sigmoid.json 20

PEMBAGIAN TUGAS 22

BAB I
DASAR TEORI

Feed Forward Neural Network (FFNN) adalah tipe jaringan saraf tiruan yang sederhana yang bergerak satu arah dari node input, node tersembunyi (jika ada), dan node output yang tidak ada siklus pada jaringannya.

Pada FFNN terdiri dari tiga jenis lapisan: lapisan input(*input layer*), lapisan tersembunyi(*hidden layer*), dan lapisan output(*ouput layer*). Tiap lapisan tersusun dari unit bernama neuron yang saling berhubungan oleh bobot. Lapisan input menerima input lalu meneruskannya ke lapisan berikutnya dan jumlah neuron-nya ditentukan oleh dimensi data input. Lapisan tersembunyi tidak terkena input atau output dan berperan sebagai mesin komputasi dari Neural Network. Lapisan tersembunyi menerima input dari output lapisan sebelumnya, menerapkan fungsi aktivasi dan meneruskan hasilnya ke lapisan berikutnya. Lapisan output adalah lapisan terakhir yang menghasilkan output untuk input yang diberikan dan jumlahnya tergantung pada jumlah output. Antar neuron tiap layer saling terhubung, dan kekuatan koneksi antar neuron diwakili oleh bobot dan learning dalam neural network melibatkan pembaruan bobot berdasarkan kesalahan input.

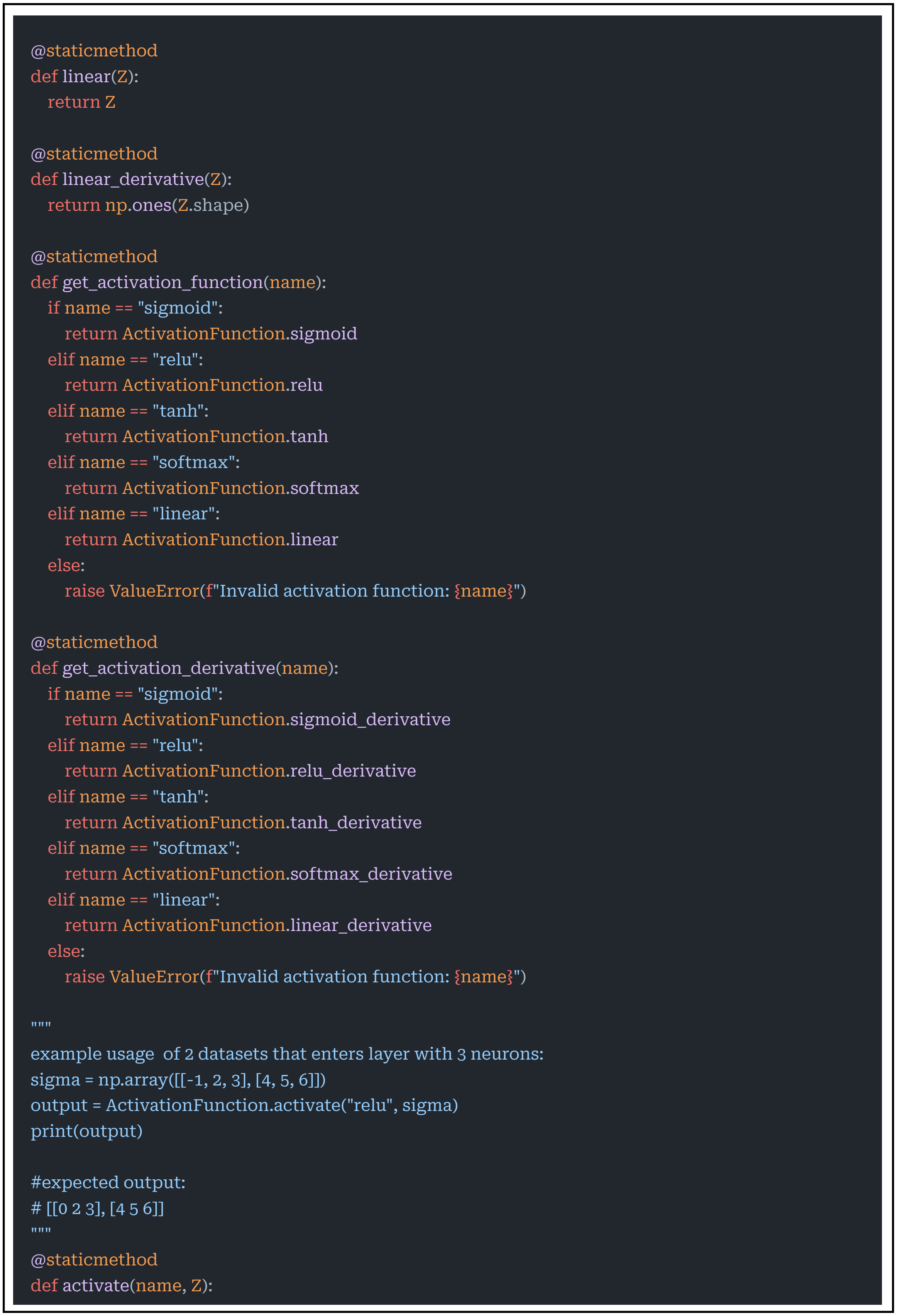
Feed Forward Neural Network bekerja dengan data input dimasukkan ke dalam jaringan dan menyebar melalui jaringan. Proses ini disebut dengan *forward propagation*. Pada setiap lapisan tersembunyi, jumlah input berbobot dihitung dan dilewatkan melalui fungsi aktivasi, yang memperkenalkan non-linearitas ke dalam model. Proses ini berlanjut sampai lapisan output tercapai, dan prediksi dibuat.

BAB II
IMPLEMENTASI

1. ActivationFunction.py

Mendefinisikan fungsi aktivasi yang dibutuhkan seperti sigmoid, ReLU, linear, softmax dan lain-lain. Serta ada beberapa fungsi pembantu untuk aktivasi.

```
import numpy as np

"""
Implement as static method
example: output = ActivationFunction("sigmoid", X)
where X = W.T @ X + b
"""
class ActivationFunction:
    @staticmethod
    def sigmoid(Z):
        return 1 / (1 + np.exp(-Z))

    @staticmethod
    def sigmoid_derivative(Z):
        return ActivationFunction.sigmoid(Z) * (1 - ActivationFunction.sigmoid(Z))

    @staticmethod
    def relu(Z):
        return np.maximum(0, Z)

    @staticmethod
    def relu_derivative(Z):
        return np.where(Z <= 0, 0, 1)

    @staticmethod
    def tanh(Z):
        return np.tanh(Z)

    @staticmethod
    def tanh_derivative(Z):
        return 1 - np.power(ActivationFunction.tanh(Z), 2)

    @staticmethod
    def softmax(Z):
        if(len(Z.shape) == 1):
            Z = Z.reshape(1, -1)
        expZ = np.exp(Z - np.max(Z))
        return expZ / expZ.sum(axis=1, keepdims=True)

    @staticmethod
    def softmax_derivative(Z):
        return ActivationFunction.softmax(Z) * (1 - ActivationFunction.softmax(Z))
```

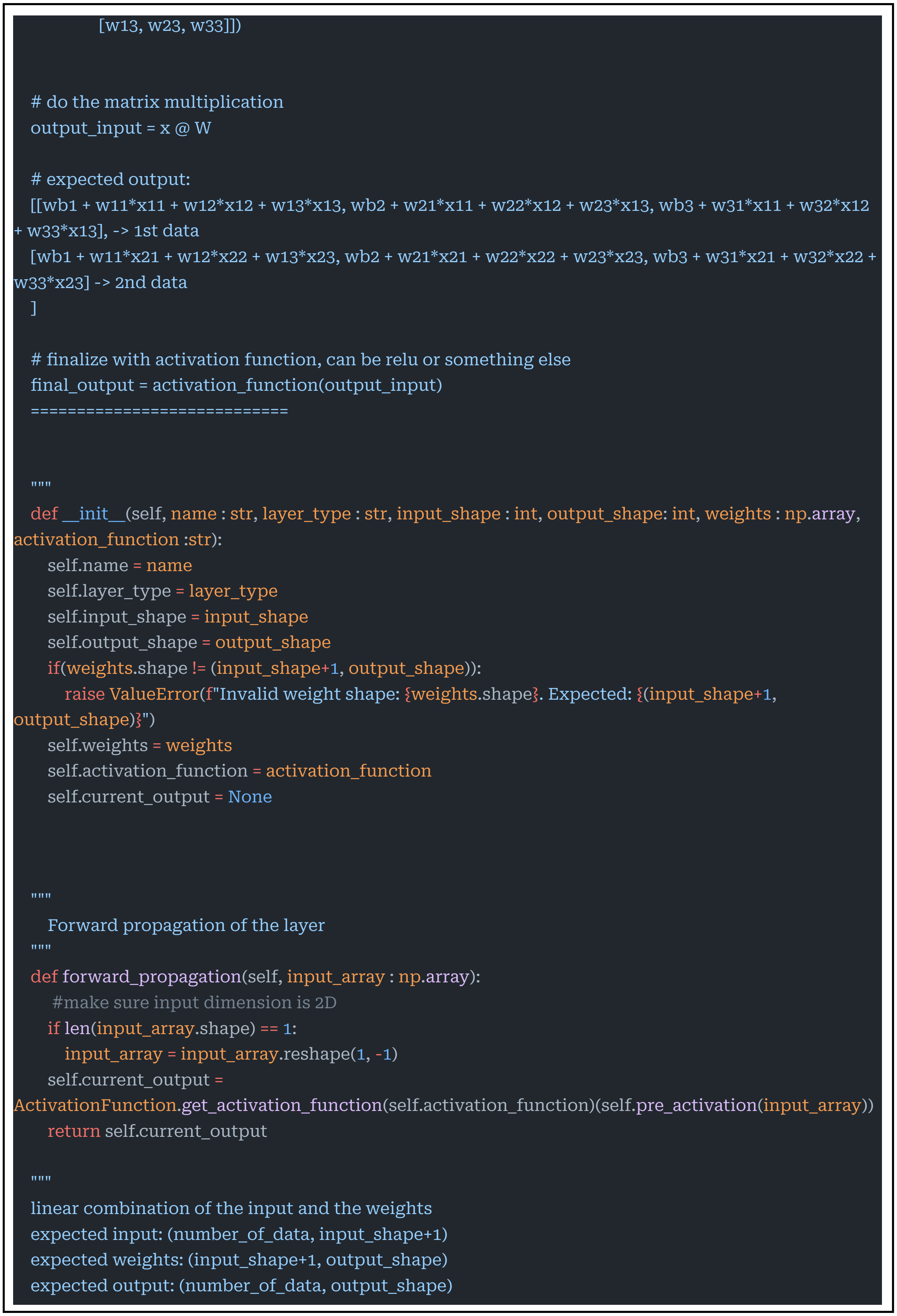


```
@staticmethod
def linear(Z):
    return Z

@staticmethod
def linear_derivative(Z):
    return np.ones(Z.shape)

@staticmethod
def get_activation_function(name):
    if name == "sigmoid":
        return ActivationFunction.sigmoid
    elif name == "relu":
        return ActivationFunction.relu
    elif name == "tanh":
        return ActivationFunction.tanh
    elif name == "softmax":
        return ActivationFunction.softmax
    elif name == "linear":
        return ActivationFunction.linear
    else:
        raise ValueError(f"Invalid activation function: {name}")

@staticmethod
def get_activation_derivative(name):
    if name == "sigmoid":
        return ActivationFunction.sigmoid_derivative
    elif name == "relu":
        return ActivationFunction.relu_derivative
    elif name == "tanh":
        return ActivationFunction.tanh_derivative
    elif name == "softmax":
        return ActivationFunction.softmax_derivative
    elif name == "linear":
        return ActivationFunction.linear_derivative
    else:
        raise ValueError(f"Invalid activation function: {name}")

"""
example usage  of 2 datasets that enters layer with 3 neurons:
sigma = np.array([[-1, 2, 3], [4, 5, 6]])
output = ActivationFunction.activate("relu", sigma)
print(output)

#expected output:
# [[0 2 3], [4 5 6]]
"""
@staticmethod
def activate(name, Z):
        return ActivationFunction.get_activation_function(name)(Z)

    @staticmethod
    def derivative(name, Z):
        return ActivationFunction.get_activation_derivative(name)(Z)
```

2. Layer.py

Menginisiasi layer dan representasi perilaku layer untuk melakukan forward propagation. Kelas Layer akan diturunkan menjadi HiddenLayer dan OutputLayer

```
import numpy as np
from lib.ActivationFunction import ActivationFunction

class Layer:
    """
    Layer class to store the layer information
    Will inherited by other layer classes (HiddenLayer and OutputLayer)

    Attributes:
    name: Name of the layer
    layer_type: Type of the layer
    input_shape: Shape of the input
    output_shape: Shape of the output
    weights: matrix of weights (input_shape x output_shape)
    biases: array of biases (output_shape, 1)

    ilustration:
    # an example of a layer with 3 inputs and 2 outputs
    =========
    x1--|
        |--> neuron_1 --> output_1
    x2--|
        |--> neuron_2 --> output_2
    x3--|
    ===========

    ======= EXAMPLE CODE =======
    # input vector: (2 x 4) -> included bias
    x = np.array([[1, x11, x12, x13], [1, x21, x22, x23]])

    # weight matrix: (4 x 3) -> first row is bias
    # output shape: number of columns of the weight matrix or W.shape[1]
    W = np.array([[wb1, wb2, wb3],
                  [w11, w21, w31],
                  [w12, w22, w32],
```



```
                    [w13, w23, w33]])


    # do the matrix multiplication
    output_input = x @ W

    # expected output:
    [[wb1 + w11*x11 + w12*x12 + w13*x13, wb2 + w21*x11 + w22*x12 + w23*x13, wb3 + w31*x11 + w32*x12 + w33*x13], -> 1st data
    [wb1 + w11*x21 + w12*x22 + w13*x23, wb2 + w21*x21 + w22*x22 + w23*x23, wb3 + w31*x21 + w32*x22 + w33*x23] -> 2nd data
    ]

    # finalize with activation function, can be relu or something else
    final_output = activation_function(output_input)
    ============================



    """
    def __init__(self, name : str, layer_type : str, input_shape : int, output_shape: int, weights : np.array, activation_function :str):
        self.name = name
        self.layer_type = layer_type
        self.input_shape = input_shape
        self.output_shape = output_shape
        if(weights.shape != (input_shape+1, output_shape)):
            raise ValueError(f"Invalid weight shape: {weights.shape}. Expected: {(input_shape+1, output_shape)}")
        self.weights = weights
        self.activation_function = activation_function
        self.current_output = None




    """
        Forward propagation of the layer
    """
    def forward_propagation(self, input_array : np.array):
         #make sure input dimension is 2D
        if len(input_array.shape) == 1:
            input_array = input_array.reshape(1, -1)
        self.current_output = ActivationFunction.get_activation_function(self.activation_function)(self.pre_activation(input_array))
        return self.current_output

    """
    linear combination of the input and the weights
    expected input: (number_of_data, input_shape+1)
    expected weights: (input_shape+1, output_shape)
    expected output: (number_of_data, output_shape)
    """
    def pre_activation(self, input_array : np.array):
        #preprocess the input
        add_bias = np.insert(input_array, 0, np.ones(input_array.shape[0]), axis=1)
        #do the matrix multiplication
        return add_bias @ self.weights

    """
        Backward propagation of the layer
        for 2nd milestone
    """
    def backward_propagation(self):
        print("backward propagation of the layer is not implemented yet")

    def debug(self):
        print(f"Layer: {self.name} | Type: {self.layer_type}", end=" ")
        print(f"| Output shape: {self.output_shape}")
        print(f"Weights:\n {self.weights}")
```

3. OutputLayer.py

Menginisiasi objek output layer, kelas merupakan turunan dari kelas Layer

```
from lib.Layer import Layer
import numpy as np

class OutputLayer(Layer):
    def __init__(self, name, input_shape, output_shape, weights, activation_function):
        super().__init__(name, "output", input_shape, output_shape, weights, activation_function)

    #override
    def forward_propagation(self, input_array : np.array):
        return super().forward_propagation(input_array=input_array)
        #make sure input dimension is 2D


    def backward_propagation(self):
        return super().backward_propagation()
```

4. HiddenLayer.py

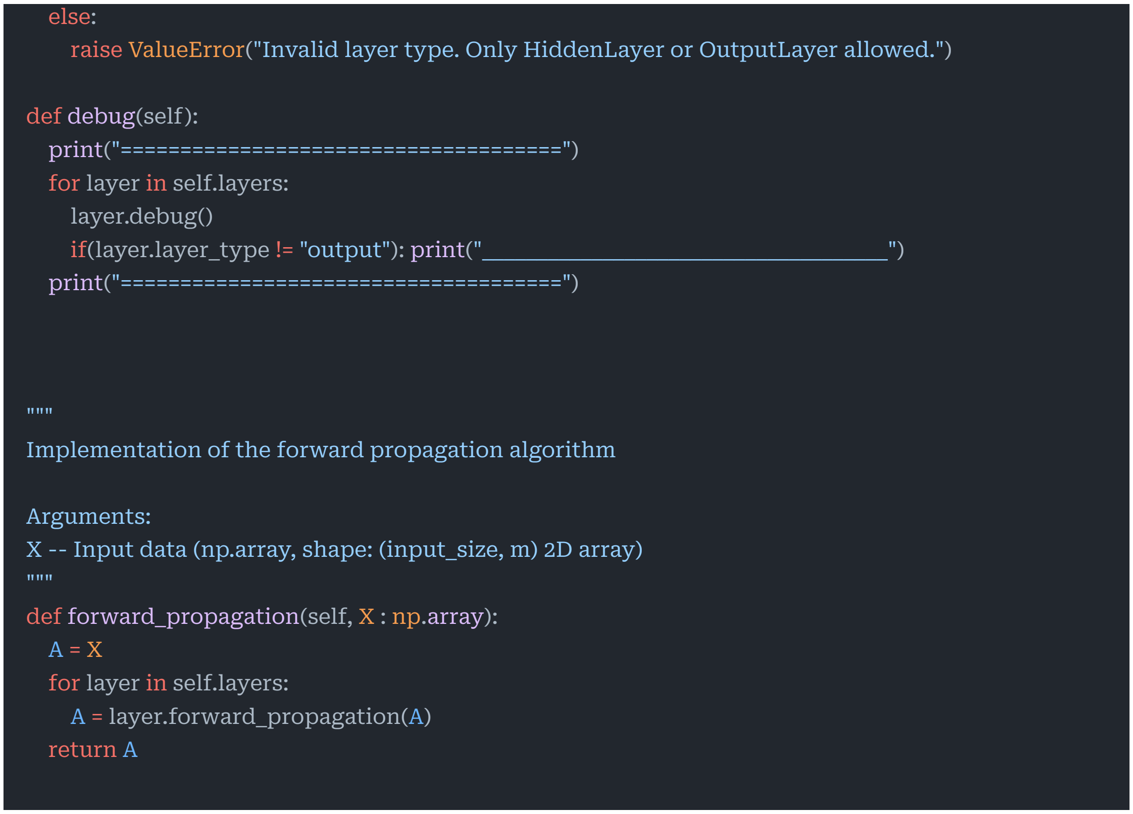
Menginisiasi objek hidden layer, kelas ini merupakan turunan dari kelas Layer

```
from lib.Layer import *

class HiddenLayer(Layer):
  def __init__(self, name, input_shape, output_shape, weights, activation_function):
    super().__init__(name, "hidden", input_shape, output_shape, weights, activation_function)

  #override
  #override
  def forward_propagation(self, input_array : np.array):
    return super().forward_propagation(input_array=input_array)
    #make sure input dimension is 2D
```

5. ANN.py

Menginisiasikan jaringan saraf buatan, terdapat fungsi untuk menambah layer pada jaringan.

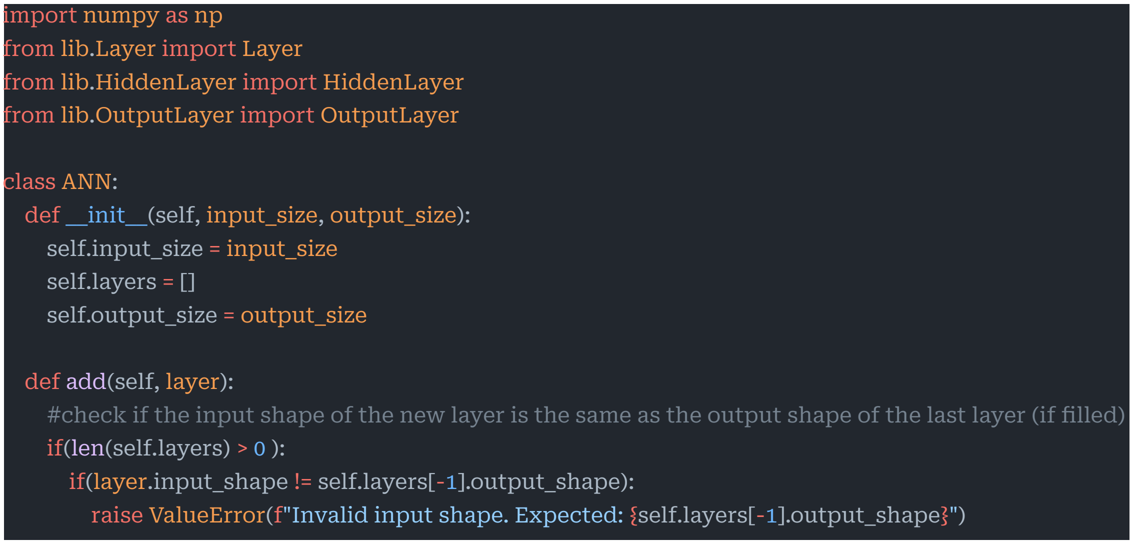
```
import numpy as np
from lib.Layer import Layer
from lib.HiddenLayer import HiddenLayer
from lib.OutputLayer import OutputLayer

class ANN:
  def __init__(self, input_size, output_size):
    self.input_size = input_size
    self.layers = []
    self.output_size = output_size

  def add(self, layer):
    #check if the input shape of the new layer is the same as the output shape of the last layer (if filled)
    if(len(self.layers) > 0 ):
      if(layer.input_shape != self.layers[-1].output_shape):
        raise ValueError(f"Invalid input shape. Expected: {self.layers[-1].output_shape}")
    else:
      if(layer.input_shape != self.input_size):
        raise ValueError(f"Invalid input shape. Expected: {self.input_size}")

    #if the last layer is output layer, then it's not valid to add another hidden layer
    if(len(self.layers) > 0 and self.layers[-1].layer_type == "output"):
        raise ValueError("OutputLayer already exist!")
    elif isinstance(layer, HiddenLayer):
      self.layers.append(layer)
    elif isinstance(layer, OutputLayer):
      self.layers.append(layer)
```



```
    else:
      raise ValueError("Invalid layer type. Only HiddenLayer or OutputLayer allowed.")

  def debug(self):
    print("====================================")
    for layer in self.layers:
      layer.debug()
      if(layer.layer_type != "output"): print("________________________________________")
    print("====================================")



  """
  Implementation of the forward propagation algorithm

  Arguments:
  X -- Input data (np.array, shape: (input_size, m) 2D array)
  """
  def forward_propagation(self, X : np.array):
    A = X
    for layer in self.layers:
      A = layer.forward_propagation(A)
    return A
```

6. Model.py

Menginisiasi Model, terdapat method untuk membuat model, menambah layer, menyimpan model dan melakukan prediksi

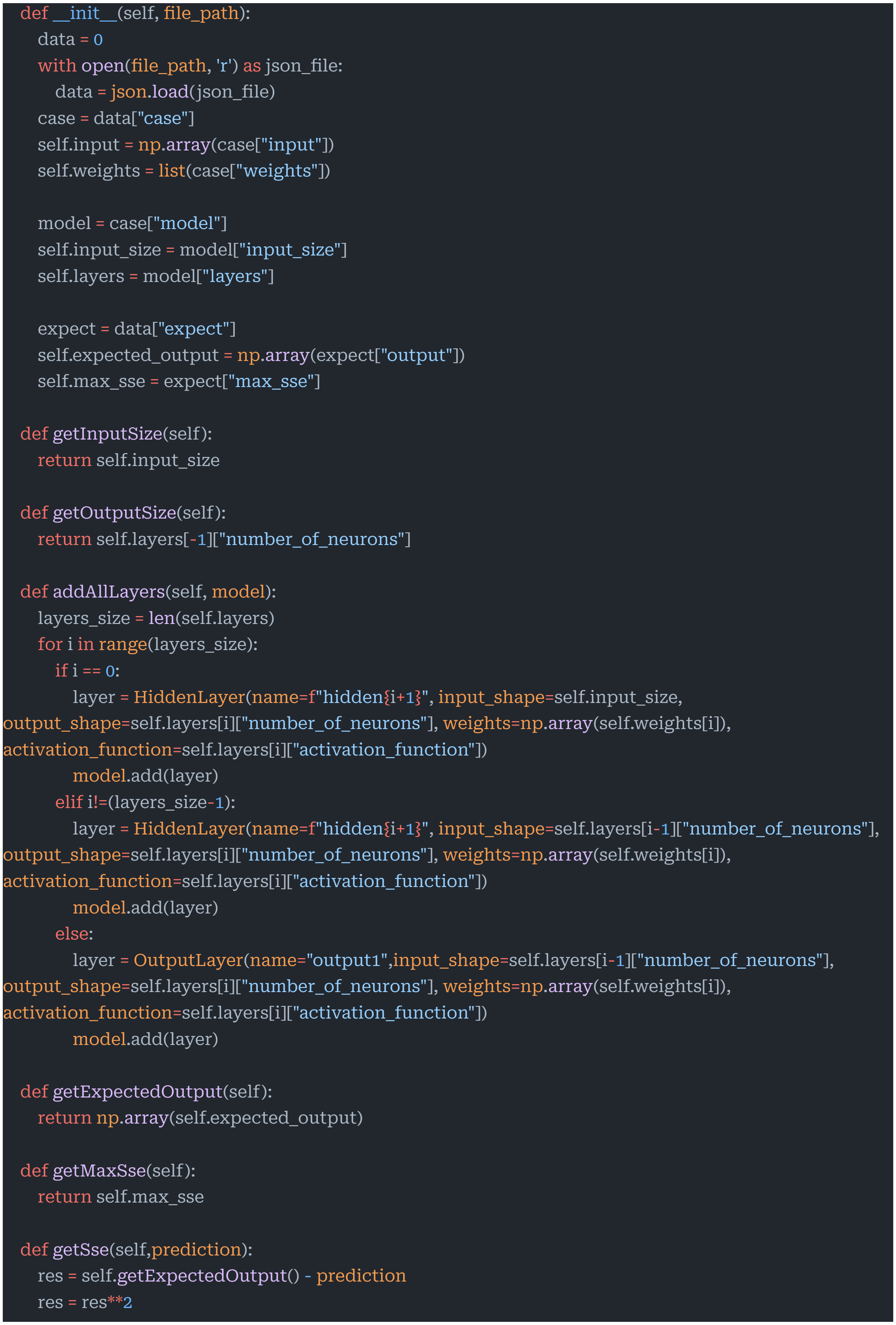


```
import numpy as np
from lib.Layer import Layer
from lib.HiddenLayer import HiddenLayer
from lib.OutputLayer import OutputLayer

class ANN:
  def __init__(self, input_size, output_size):
    self.input_size = input_size
    self.layers = []
    self.output_size = output_size

  def add(self, layer):
    #check if the input shape of the new layer is the same as the output shape of the last layer (if filled)
    if(len(self.layers) > 0 ):
      if(layer.input_shape != self.layers[-1].output_shape):
        raise ValueError(f"Invalid input shape. Expected: {self.layers[-1].output_shape}")
```

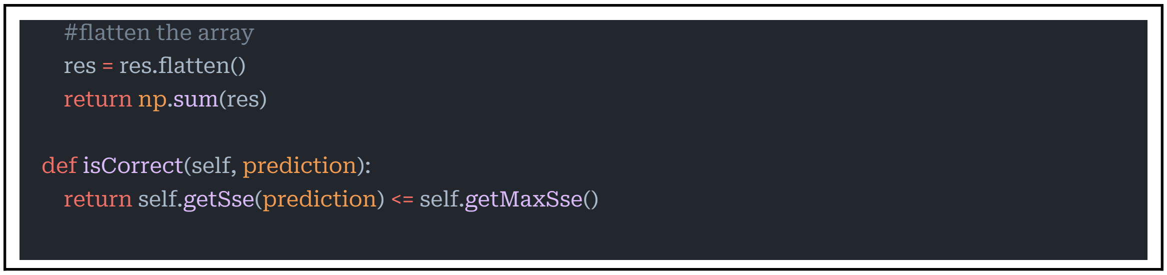
7. Parser.py

Melakukan Parsing JSON *testcase* untuk membuat model dan melakukan prediksi sesuai input yang ada

```
import json
import numpy as np
from lib.HiddenLayer import HiddenLayer
from lib.OutputLayer import OutputLayer

class Parser:
```

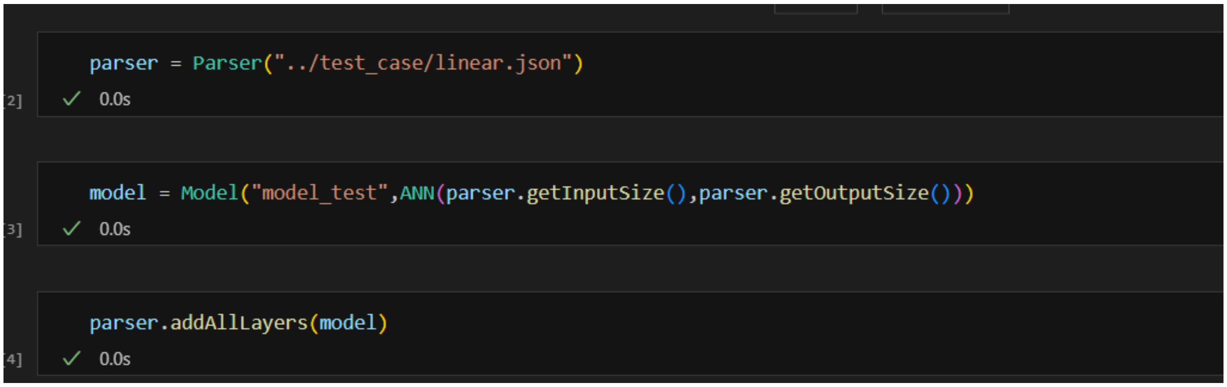


```
  def __init__(self, file_path):
    data = 0
    with open(file_path, 'r') as json_file:
      data = json.load(json_file)
    case = data["case"]
    self.input = np.array(case["input"])
    self.weights = list(case["weights"])

    model = case["model"]
    self.input_size = model["input_size"]
    self.layers = model["layers"]

    expect = data["expect"]
    self.expected_output = np.array(expect["output"])
    self.max_sse = expect["max_sse"]

  def getInputSize(self):
    return self.input_size

  def getOutputSize(self):
    return self.layers[-1]["number_of_neurons"]

  def addAllLayers(self, model):
    layers_size = len(self.layers)
    for i in range(layers_size):
      if i == 0:
        layer = HiddenLayer(name=f"hidden{i+1}", input_shape=self.input_size,
output_shape=self.layers[i]["number_of_neurons"], weights=np.array(self.weights[i]),
activation_function=self.layers[i]["activation_function"])
        model.add(layer)
      elif i!=(layers_size-1):
        layer = HiddenLayer(name=f"hidden{i+1}", input_shape=self.layers[i-1]["number_of_neurons"],
output_shape=self.layers[i]["number_of_neurons"], weights=np.array(self.weights[i]),
activation_function=self.layers[i]["activation_function"])
        model.add(layer)
      else:
        layer = OutputLayer(name="output1",input_shape=self.layers[i-1]["number_of_neurons"],
output_shape=self.layers[i]["number_of_neurons"], weights=np.array(self.weights[i]),
activation_function=self.layers[i]["activation_function"])
        model.add(layer)

  def getExpectedOutput(self):
    return np.array(self.expected_output)

  def getMaxSse(self):
    return self.max_sse

  def getSse(self,prediction):
    res = self.getExpectedOutput() - prediction
    res = res**2
```



```
    #flatten the array
    res = res.flatten()
    return np.sum(res)

def isCorrect(self, prediction):
    return self.getSse(prediction) <= self.getMaxSse()
```

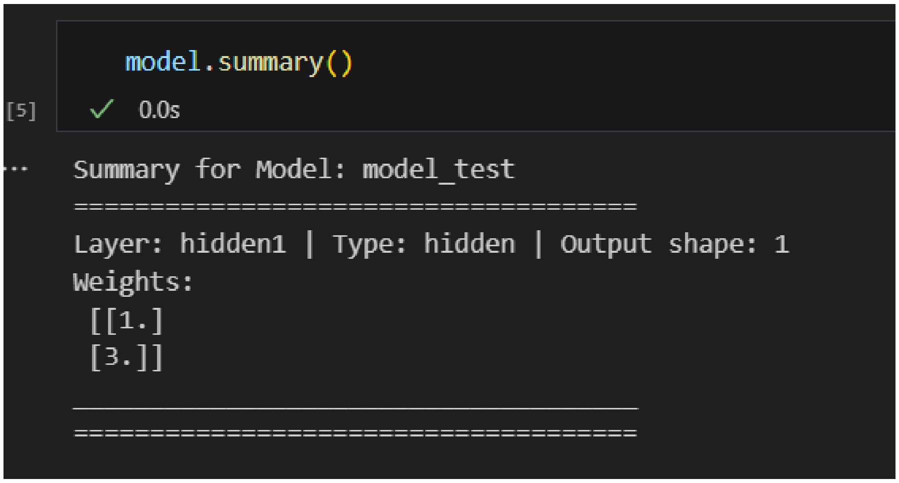
BAB III
HASIL PENGUJIAN



```
parser = Parser("../test_case/linear.json")
```

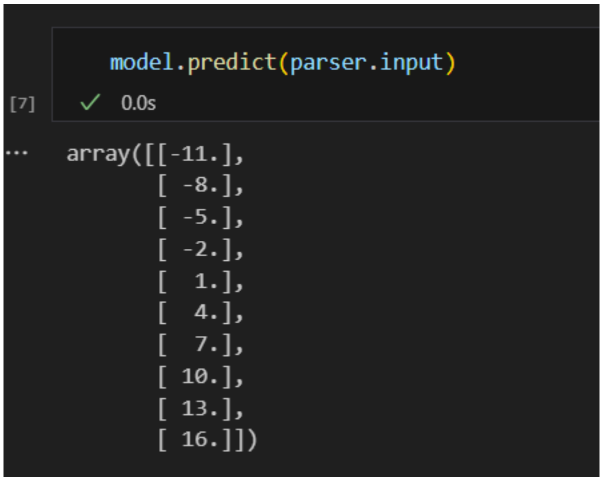
```
model = Model("model_test",ANN(parser.getInputSize(),parser.getOutputSize()))
```

```
parser.addAllLayers(model)
```



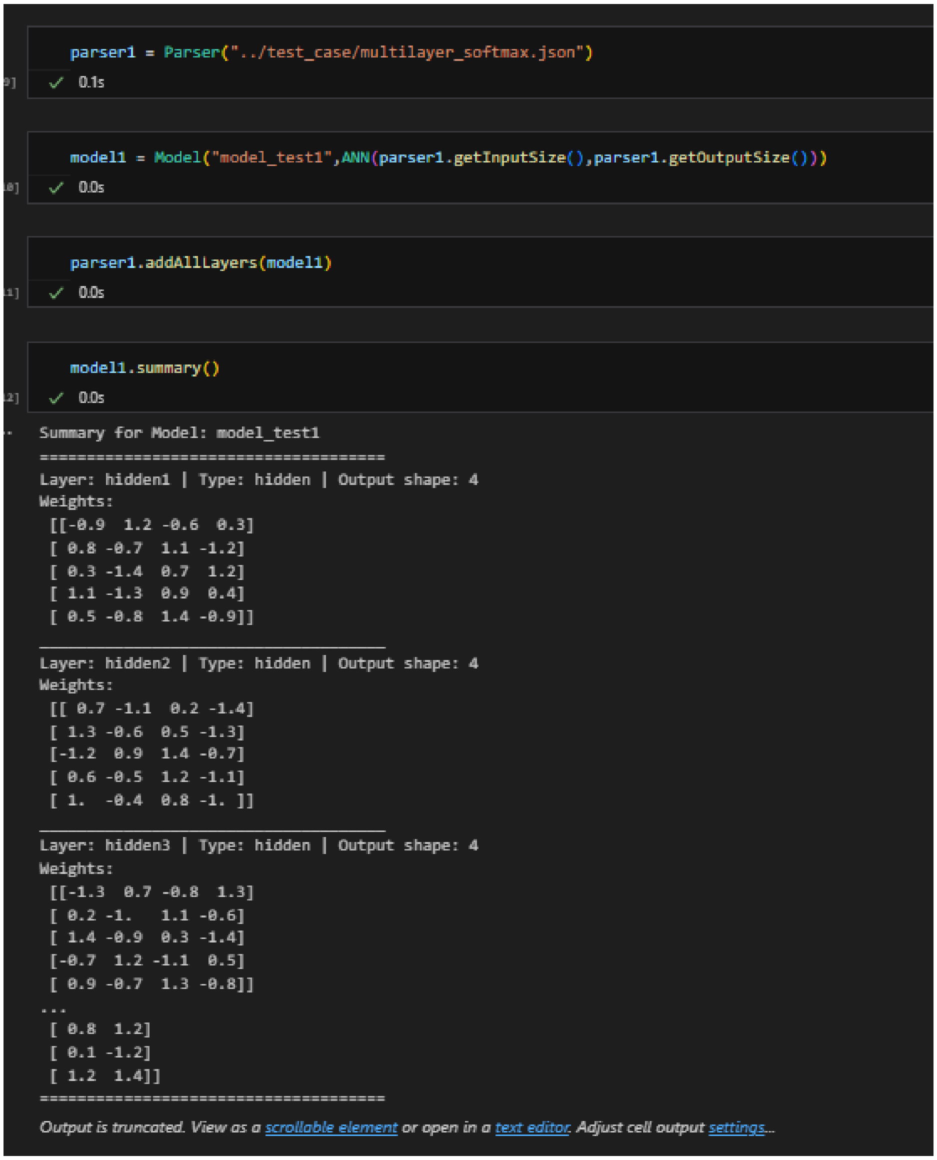
```
model.summary()
```

```
Summary for Model: model_test
========================================
Layer: hidden1 | Type: hidden | Output shape: 1
Weights:
 [[1.]
 [3.]]

________________________________________
========================================
```

File	**Linear.json**
Hasil	

```
model.predict(parser.input)
```

```
array([[-11.],
       [ -8.],
       [ -5.],
       [ -2.],
       [  1.],
       [  4.],
       [  7.],
       [ 10.],
       [ 13.],
       [ 16.]])
```



```
parser1 = Parser("../test_case/multilayer_softmax.json")
✓ 0.1s
```

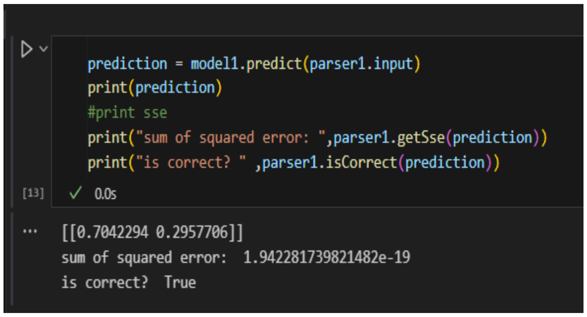
```
model1 = Model("model_test1",ANN(parser1.getInputSize(),parser1.getOutputSize()))
✓ 0.0s
```

```
parser1.addAllLayers(model1)
✓ 0.0s
```

```
model1.summary()
✓ 0.0s
```

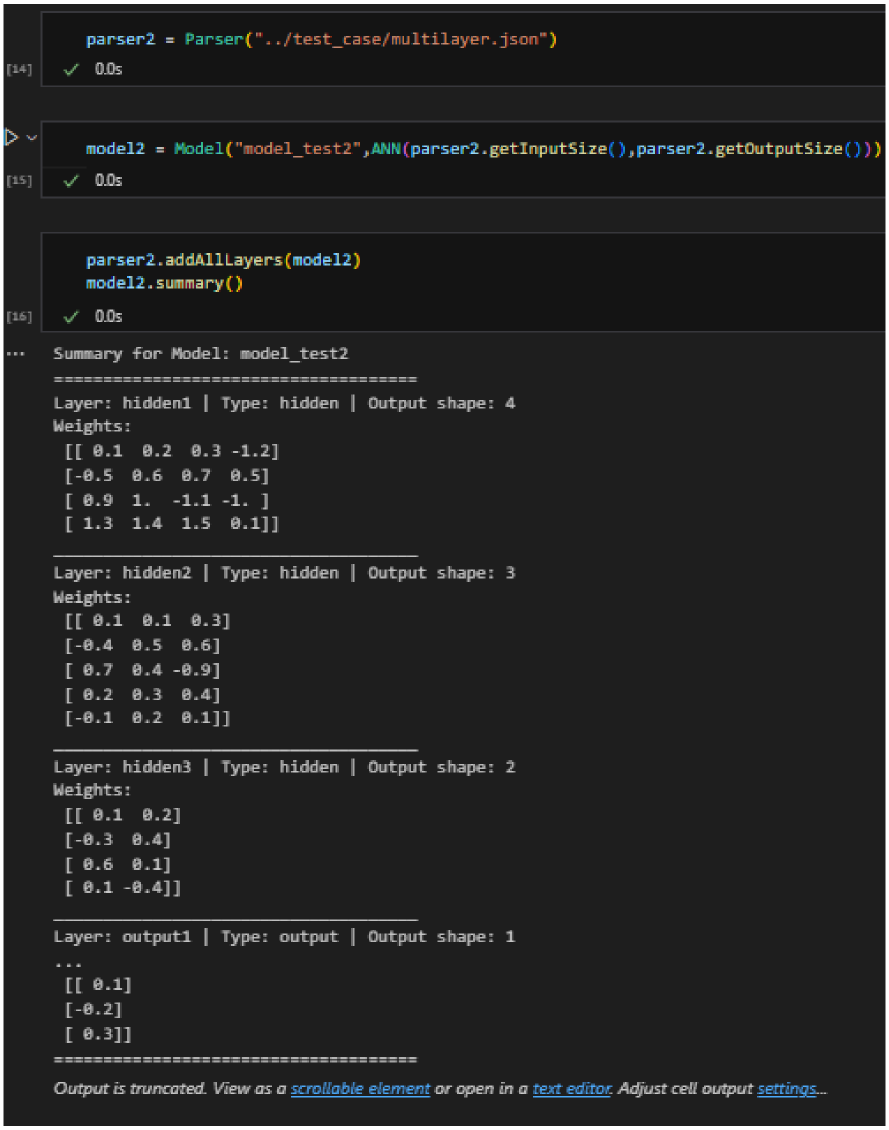
```
Summary for Model: model_test1
=====================================
Layer: hidden1 | Type: hidden | Output shape: 4
Weights:
 [[-0.9  1.2 -0.6  0.3]
 [ 0.8 -0.7  1.1 -1.2]
 [ 0.3 -1.4  0.7  1.2]
 [ 1.1 -1.3  0.9  0.4]
 [ 0.5 -0.8  1.4 -0.9]]
_____________________________________
Layer: hidden2 | Type: hidden | Output shape: 4
Weights:
 [[ 0.7 -1.1  0.2 -1.4]
 [ 1.3 -0.6  0.5 -1.3]
 [-1.2  0.9  1.4 -0.7]
 [ 0.6 -0.5  1.2 -1.1]
 [ 1.  -0.4  0.8 -1. ]]
_____________________________________
Layer: hidden3 | Type: hidden | Output shape: 4
Weights:
 [[-1.3  0.7 -0.8  1.3]
 [ 0.2 -1.   1.1 -0.6]
 [ 1.4 -0.9  0.3 -1.4]
 [-0.7  1.2 -1.1  0.5]
 [ 0.9 -0.7  1.3 -0.8]]
...
 [ 0.8  1.2]
 [ 0.1 -1.2]
 [ 1.2  1.4]]
=====================================
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Input	**multilayer_softmax.json**
Hasil	

```
prediction = model1.predict(parser1.input)
print(prediction)
#print sse
print("sum of squared error: ",parser1.getSse(prediction))
print("is correct? " ,parser1.isCorrect(prediction))
[13] ✓ 0.0s
```

```
[[0.7042294 0.2957706]]
sum of squared error:  1.942281739821482e-19
is correct?  True
```



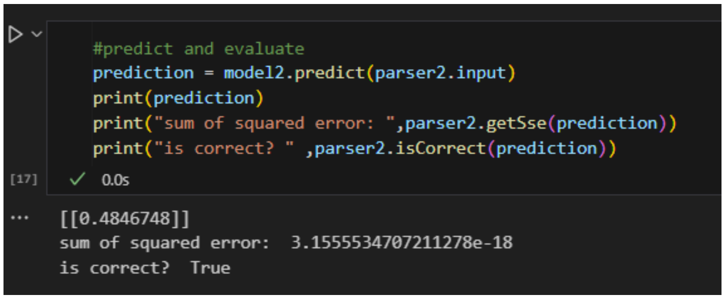
```
parser2 = Parser("../test_case/multilayer.json")
```
```
[14]  ✓ 0.0s
```

```
model2 = Model("model_test2",ANN(parser2.getInputSize(),parser2.getOutputSize()))
```
```
[15]  ✓ 0.0s
```

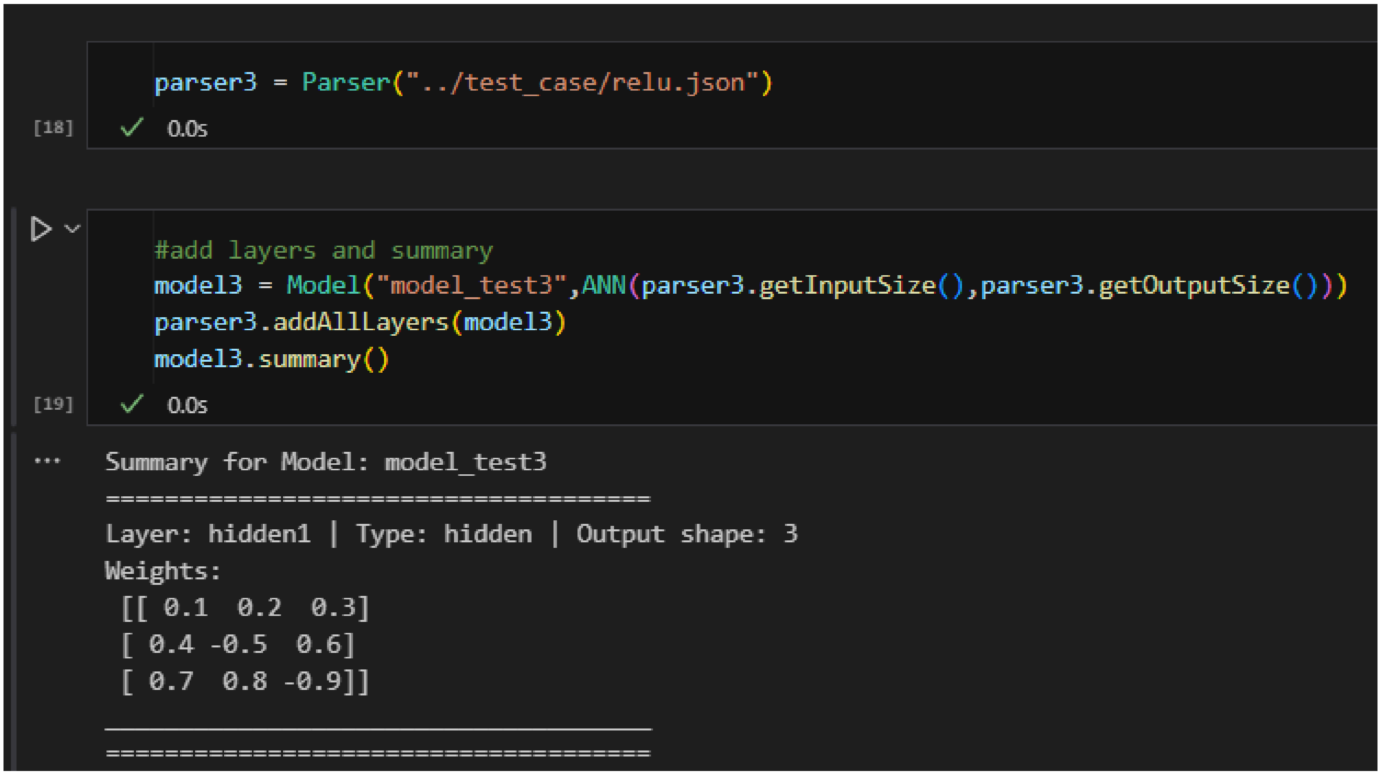
```
parser2.addAllLayers(model2)
model2.summary()
```
```
[16]  ✓ 0.0s
```

```
Summary for Model: model_test2
=====================================
Layer: hidden1 | Type: hidden | Output shape: 4
Weights:
 [[ 0.1  0.2  0.3 -1.2]
 [-0.5  0.6  0.7  0.5]
 [ 0.9  1.  -1.1 -1. ]
 [ 1.3  1.4  1.5  0.1]]
_____________________________________
Layer: hidden2 | Type: hidden | Output shape: 3
Weights:
 [[ 0.1  0.1  0.3]
 [-0.4  0.5  0.6]
 [ 0.7  0.4 -0.9]
 [ 0.2  0.3  0.4]
 [-0.1  0.2  0.1]]
_____________________________________
Layer: hidden3 | Type: hidden | Output shape: 2
Weights:
 [[ 0.1  0.2]
 [-0.3  0.4]
 [ 0.6  0.1]
 [ 0.1 -0.4]]
_____________________________________
Layer: output1 | Type: output | Output shape: 1
...
 [[ 0.1]
 [-0.2]
 [ 0.3]]
=====================================
```

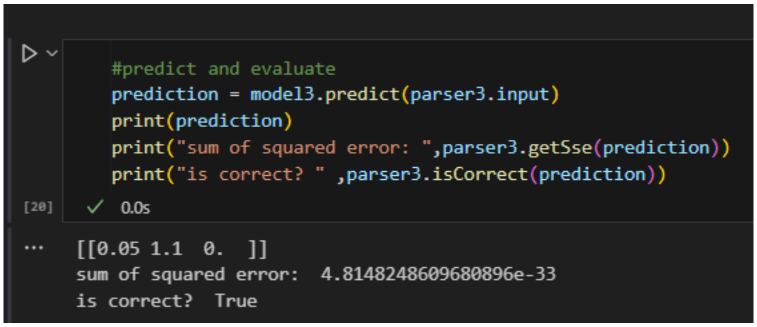
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Input	**multilayer.json**
Hasil	

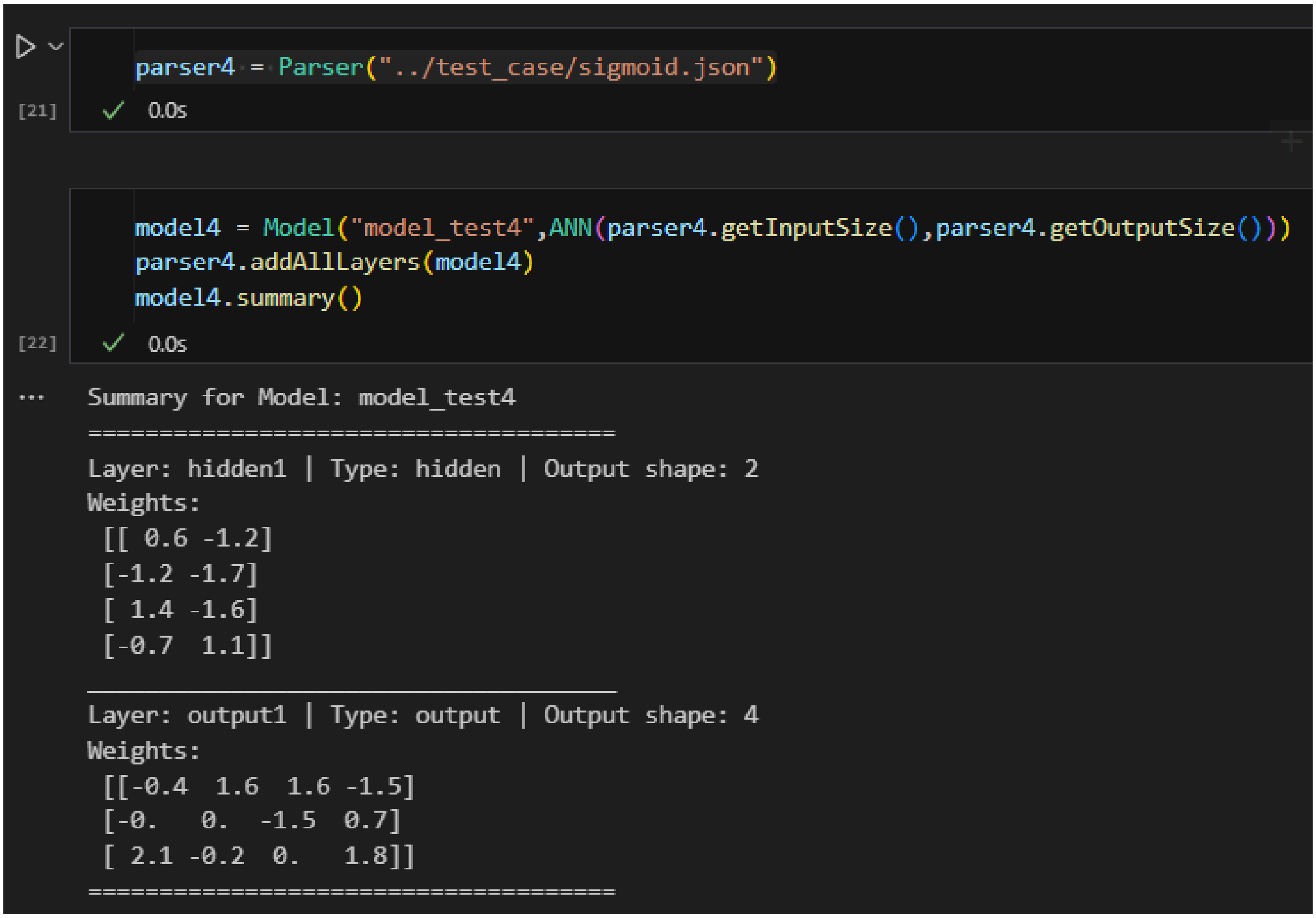
```
#predict and evaluate
prediction = model2.predict(parser2.input)
print(prediction)
print("sum of squared error: ",parser2.getSse(prediction))
print("is correct? " ,parser2.isCorrect(prediction))
```
```
[17]  ✓ 0.0s
```
```
[[0.4846748]]
sum of squared error:  3.1555534707211278e-18
is correct?  True
```



```
parser3 = Parser("../test_case/relu.json")
[18]  ✓  0.0s

#add layers and summary
model3 = Model("model_test3",ANN(parser3.getInputSize(),parser3.getOutputSize()))
parser3.addAllLayers(model3)
model3.summary()
[19]  ✓  0.0s

Summary for Model: model_test3
====================================
Layer: hidden1 | Type: hidden | Output shape: 3
Weights:
 [[ 0.1  0.2  0.3]
 [ 0.4 -0.5  0.6]
 [ 0.7  0.8 -0.9]]
____________________________________
====================================
```

Input	**relu.json**
Hasil	

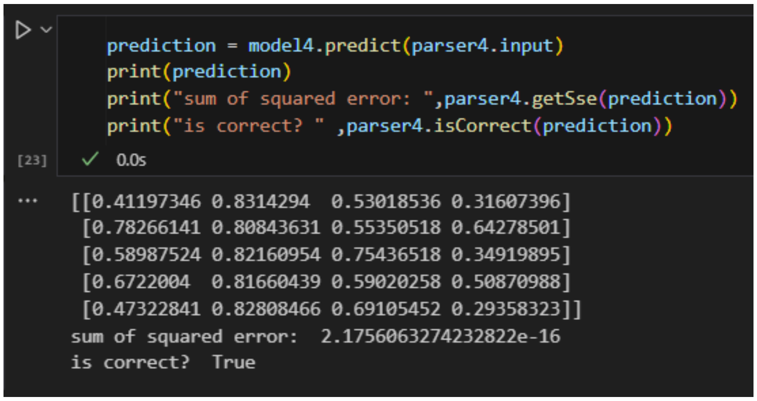
```
#predict and evaluate
prediction = model3.predict(parser3.input)
print(prediction)
print("sum of squared error: ",parser3.getSse(prediction))
print("is correct? " ,parser3.isCorrect(prediction))
[20]  ✓  0.0s

[[0.05 1.1  0.  ]]
sum of squared error:  4.8148248609680896e-33
is correct?  True
```



```
parser4 = Parser("../test_case/sigmoid.json")
[21] ✓ 0.0s
```

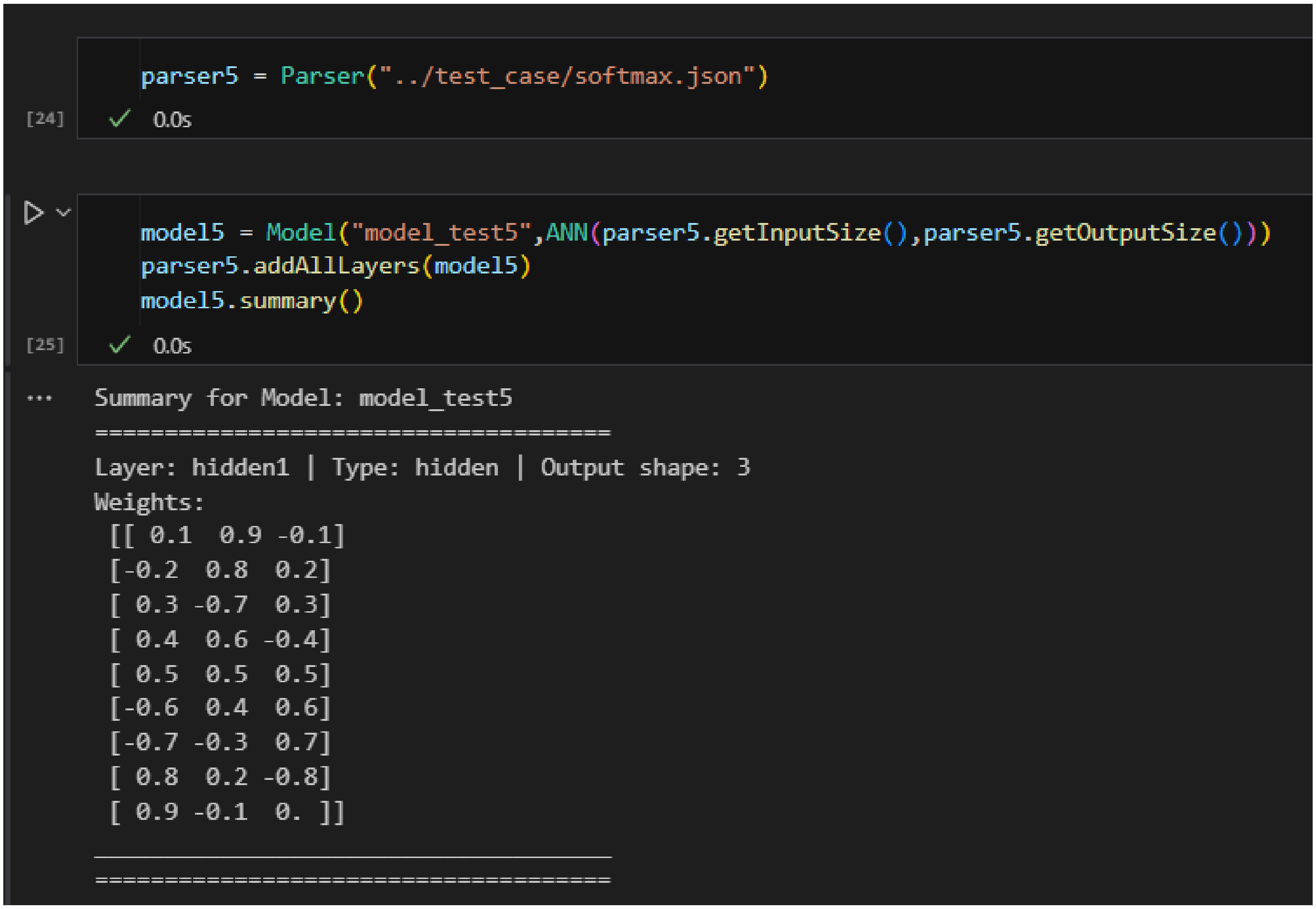
```
model4 = Model("model_test4",ANN(parser4.getInputSize(),parser4.getOutputSize()))
parser4.addAllLayers(model4)
model4.summary()
[22] ✓ 0.0s
```

```
Summary for Model: model_test4
========================================
Layer: hidden1 | Type: hidden | Output shape: 2
Weights:
 [[ 0.6 -1.2]
 [-1.2 -1.7]
 [ 1.4 -1.6]
 [-0.7  1.1]]
________________________________________
Layer: output1 | Type: output | Output shape: 4
Weights:
 [[-0.4  1.6  1.6 -1.5]
 [-0.   0.  -1.5  0.7]
 [ 2.1 -0.2  0.   1.8]]
========================================
```

Input	**sigmoid.json**
Hasil	

```
prediction = model4.predict(parser4.input)
print(prediction)
print("sum of squared error: ",parser4.getSse(prediction))
print("is correct? " ,parser4.isCorrect(prediction))
[23] ✓ 0.0s
```

```
[[0.41197346 0.8314294  0.53018536 0.31607396]
 [0.78266141 0.80843631 0.55350518 0.64278501]
 [0.58987524 0.82160954 0.75436518 0.34919895]
 [0.6722004  0.81660439 0.59020258 0.50870988]
 [0.47322841 0.82808466 0.69105452 0.29358323]]
sum of squared error:  2.1756063274232822e-16
is correct?  True
```



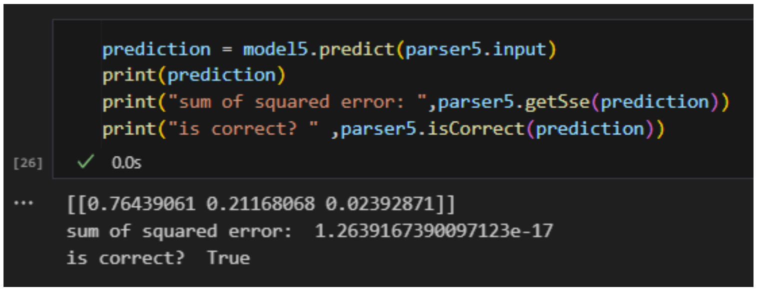
```
parser5 = Parser("../test_case/softmax.json")
```

[24] ✓ 0.0s

```
model5 = Model("model_test5",ANN(parser5.getInputSize(),parser5.getOutputSize()))
parser5.addAllLayers(model5)
model5.summary()
```

[25] ✓ 0.0s

```
Summary for Model: model_test5
=====================================
Layer: hidden1 | Type: hidden | Output shape: 3
Weights:
 [[ 0.1  0.9 -0.1]
 [-0.2  0.8  0.2]
 [ 0.3 -0.7  0.3]
 [ 0.4  0.6 -0.4]
 [ 0.5  0.5  0.5]
 [-0.6  0.4  0.6]
 [-0.7 -0.3  0.7]
 [ 0.8  0.2 -0.8]
 [ 0.9 -0.1  0. ]]
_____________________________________
=====================================
```

Input	**softmax.json**
Hasil	

```
prediction = model5.predict(parser5.input)
print(prediction)
print("sum of squared error: ",parser5.getSse(prediction))
print("is correct? " ,parser5.isCorrect(prediction))
```

[26] ✓ 0.0s

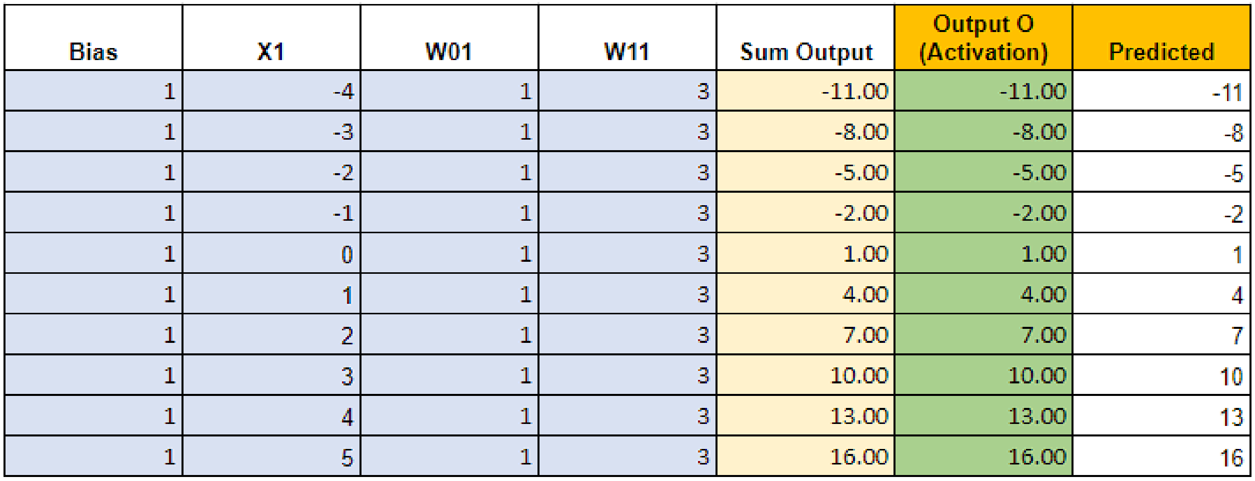
```
[[0.76439061 0.21168068 0.02392871]]
sum of squared error:  1.2639167390097123e-17
is correct?  True
```

BAB IV
PERBANDINGAN DENGAN HASIL MANUAL

Untuk lebih detailnya perhitungan manual dapat dilihat di link_sheets. Berikut adalah *screenshot* hasil perhitungan manual menggunakan *spreadsheet*

A. Linear.json

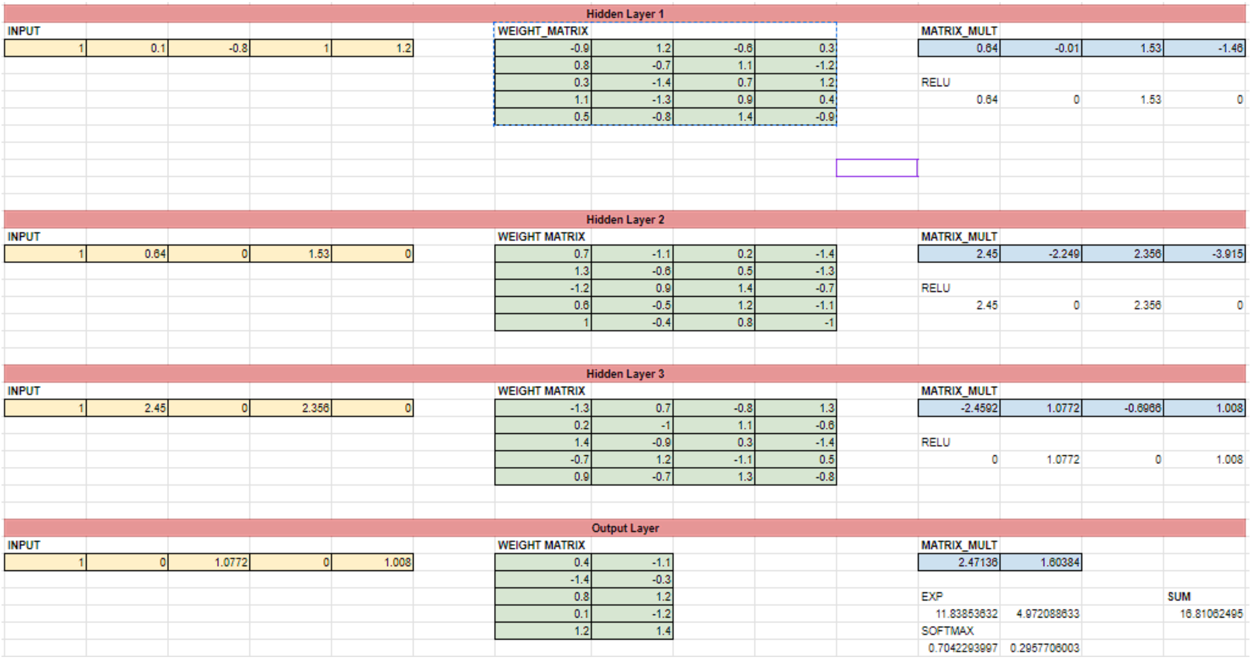
Berdasarkan hasil perhitungan menggunakan *spreadsheet*, hasil output yang diberikan sama dengan hasil menggunakan implementasi algoritma FFNN.



Bias	X1	W01	W11	Sum Output	Output O (Activation)	Predicted
1	-4	1	3	-11.00	-11.00	-11
1	-3	1	3	-8.00	-8.00	-8
1	-2	1	3	-5.00	-5.00	-5
1	-1	1	3	-2.00	-2.00	-2
1	0	1	3	1.00	1.00	1
1	1	1	3	4.00	4.00	4
1	2	1	3	7.00	7.00	7
1	3	1	3	10.00	10.00	10
1	4	1	3	13.00	13.00	13
1	5	1	3	16.00	16.00	16

B. Multilayer_softmax.json

Berdasarkan hasil perhitungan menggunakan *spreadsheet*, hasil output yang diberikan sama dengan hasil menggunakan implementasi algoritma FFNN.



Hidden Layer 1

INPUT						WEIGHT_MATRIX					MATRIX_MULT			
1	0.1	-0.8	1	1.2		-0.9	1.2	-0.6	0.3		0.64	-0.01	1.53	-1.46
						0.8	-0.7	1.1	-1.2					
						0.3	-1.4	0.7	1.2		RELU			
						1.1	-1.3	0.9	0.4		0.64	0	1.53	0
						0.5	-0.8	1.4	-0.9					

Hidden Layer 2

INPUT						WEIGHT MATRIX					MATRIX_MULT			
1	0.64	0	1.53	0		0.7	-1.1	0.2	-1.4		2.45	-2.249	2.356	-3.915
						1.3	-0.6	0.5	-1.3					
						-1.2	0.9	1.4	-0.7		RELU			
						0.6	-0.5	1.2	-1.1		2.45	0	2.356	0
						1	-0.4	0.8	-1					

Hidden Layer 3

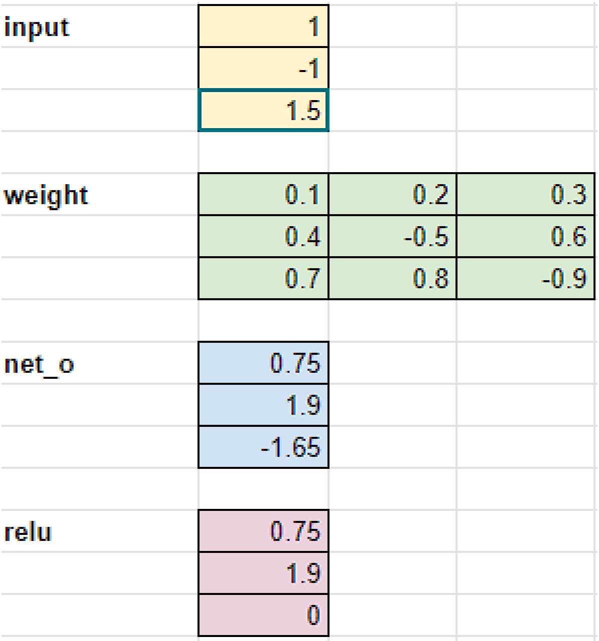
INPUT						WEIGHT MATRIX					MATRIX_MULT			
1	2.45	0	2.356	0		-1.3	0.7	-0.8	1.3		-2.4592	1.0772	-0.6966	1.008
						0.2	-1	1.1	-0.6					
						1.4	-0.9	0.3	-1.4		RELU			
						-0.7	1.2	-1.1	0.5		0	1.0772	0	1.008
						0.9	-0.7	1.3	-0.8					

Output Layer

INPUT						WEIGHT MATRIX					MATRIX_MULT			
1	0	1.0772	0	1.008		0.4	-1.1				2.47136	1.60384		
						-1.4	-0.3							
						0.8	1.2				EXP			SUM
						0.1	-1.2				11.83853632	4.972088633		16.81062495
						1.2	1.4				SOFTMAX			
											0.7042293997	0.2957706003		

C. relu.json

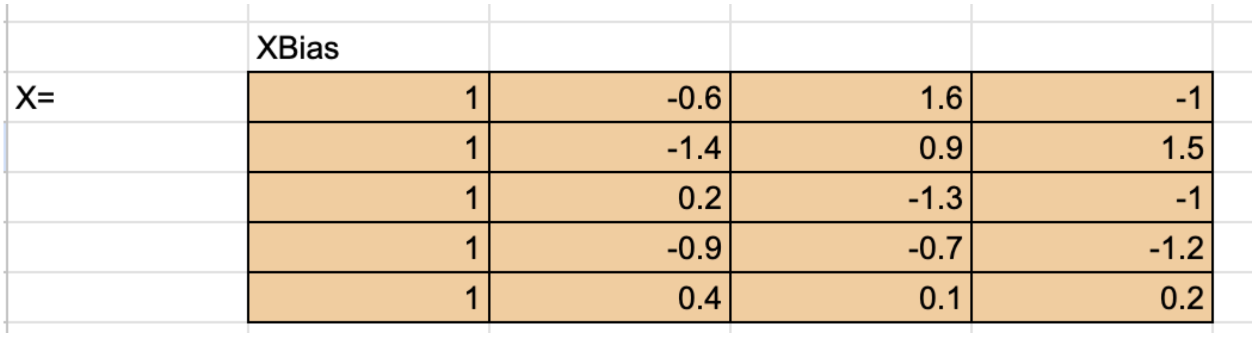
Berdasarkan hasil perhitungan menggunakan *spreadsheet*, hasil output yang diberikan sama dengan hasil menggunakan implementasi algoritma FFNN.



input	1		
	-1		
	1.5		
weight	0.1	0.2	0.3
	0.4	-0.5	0.6
	0.7	0.8	-0.9
net_o	0.75		
	1.9		
	-1.65		
relu	0.75		
	1.9		
	0		

D. sigmoid.json

Berdasarkan hasil perhitungan menggunakan *spreadsheet*, hasil output yang diberikan sama dengan hasil menggunakan implementasi algoritma FFNN.



	XBias			
X=	1	-0.6	1.6	-1
	1	-1.4	0.9	1.5
	1	0.2	-1.3	-1
	1	-0.9	-0.7	-1.2
	1	0.4	0.1	0.2

wxh=	0.6	-1.2		wheight output=	-0.4	1.6	1.6	-1.5
	-1.2	-1.7			0	0	-1.5	0.7
	1.4	-1.6			2.1	-0.2	0	1.8
	-0.7	1.1						

net_h=	4.26	-3.84
	2.49	1.39
	-0.76	-0.56
	1.54	0.13
	0.12	-1.82

h=	1	0.9860743597	0.02104134702
	1	0.9234378026	0.8005922432
	1	0.3186462662	0.3635474597
	1	0.8234647252	0.5324543064
	1	0.5299640518	0.139433873

net_o=	-0.3558131713	1.595791731	0.1208884605	-0.7718735236
	1.281243711	1.439881551	0.214843296	0.5874724995
	0.3634496654	1.527290508	1.122030601	-0.6225621862
	0.7181540434	1.493509139	0.3648029122	0.03484305915
	-0.1071888668	1.572113225	0.8050539224	-0.8780441924

PEMBAGIAN TUGAS

No.	Tugas	NIM
1	Implementasi Save Model,kode sumber jupyter notebook, Laporan	13521080
2	kode sumber Jupyter NoteBook, Laporan	13521109
3	Implementasi Parser, kode sumber Jupyter NoteBook, Laporan	13521119
4	Implementasi Struktur Folder, kelas ANN, kelas Layer-HiddenLayer-OutputLayer, kelas Model, dan kode sumber Jupyter NoteBook	13521131