

**IF3270 Pembelajaran Mesin
Feedforward Neural Network**

Tugas Besar 1

Disusun untuk memenuhi tugas mata kuliah Pembelajaran Mesin
pada Semester 2 (dua) Tahun Akademik 2024/2025

Tugas Besar IF3270 Pembelajaran Mesin



Oleh

Shabrina Maharani **13522134**

Muhammad Davis Adhipramana **13522157**

Valentino Chryslie Triadi **13522164**

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2025**

DAFTAR ISI

DAFTAR ISI	2
BAB 1 DESKRIPSI persoalan	3
BAB 2 PEMBAHASAN	4
2.1 Penjelasan Implementasi	4
2.1.1 Deskripsi Kelas, Atribut, dan Method	4
2.1.2 Forward Propagation	34
2.1.3 Backward Propagation dan Weight Update	36
2.2 Hasil Pengujian	40
2.2.1 Pengaruh depth dan width	40
2.2.2 Pengaruh fungsi aktivasi	59
2.2.3 Pengaruh learning rate	70
2.2.4 Pengaruh inisialisasi bobot	77
2.2.5 Pengaruh regularisasi	85
2.2.5 Perbandingan dengan library sklearn	92
BAB 3 KESIMPULAN DAN SARAN	93
3.1 Kesimpulan	93
3.1 Saran	93
Pembagian Tugas	95
Referensi	96

BAB 1 DESKRIPSI PERSOALAN

Feedforward Neural Network (FFNN) merupakan salah satu arsitektur dasar dalam artificial neural network yang terdiri dari beberapa lapisan neuron yang terhubung (*fully connected*) dari input menuju output tanpa adanya *loop*. FFNN digunakan dalam berbagai aplikasi *machine learning*, seperti klasifikasi, regresi, dan pemrosesan sinyal. Pada tugas besar ini, penulis diminta untuk mengimplementasikan modul FFNN *from scratch* tanpa menggunakan *library deep learning* seperti TensorFlow atau PyTorch. Implementasi harus mencakup berbagai fitur yang memungkinkan pengguna untuk menyesuaikan struktur jaringan, memilih fungsi aktivasi, serta melatih model menggunakan metode optimisasi berbasis *gradient descent*.

Modul FFNN yang dikembangkan harus mampu menerima jumlah neuron di tiap layer, termasuk *input* dan *output* layer. Fungsi aktivasi yang didukung mencakup fungsi Linear, ReLU, Sigmoid, Hyperbolic Tangent (*tanh*), dan Softmax. Selain itu, model harus dapat mendukung berbagai fungsi loss seperti *Mean Squared Error* (MSE), *Binary Cross-Entropy*, dan *Categorical Cross-Entropy* dengan logaritma natural sebagai basisnya. Inisialisasi bobot dapat dilakukan dengan beberapa metode seperti *zero initialization*, distribusi uniform dengan batas atas dan bawah, serta distribusi normal dengan parameter *mean* dan *variance*. Model yang dibuat juga harus dapat ditampilkan dalam bentuk struktur jaringan beserta bobot dan gradiennya dalam bentuk graf, serta dapat menyimpan (*save*) dan memuat (*load*) model untuk keperluan penggunaan yang berulang.

Implementasi pastinya harus mencakup *forward propagation* yang mendukung *input* dalam bentuk *batch* dan *backward propagation* yang menggunakan konsep *chain rule* untuk menghitung gradien bobot terhadap loss function. Model juga harus mampu memperbarui bobot menggunakan *gradient descent* dan mendukung parameter *training* seperti batch size, learning rate, jumlah epoch, serta opsi verbose untuk menampilkan progress dalam *training*. Selama *training*, model juga harus mencatat *training loss* dan *validation loss* pada setiap epoch.

Untuk evaluasi mandiri, model akan diuji menggunakan dataset MNIST 784 yang dimuat melalui `fetch_openml` dari `sklearn`. Pengujian akan melibatkan analisis terhadap pengaruh jumlah layer dan neuron per layer, perbedaan hasil dengan berbagai fungsi aktivasi, variasi learning rate, serta metode inisialisasi bobot yang digunakan. Selain itu, hasil FFNN yang diimplementasikan akan dibandingkan dengan model yang dibuat menggunakan `MLPClassifier` dari `sklearn` untuk melihat kinerja model yang dikembangkan oleh penulis dibandingkan dengan implementasi yang sudah tersedia dalam *library*.

BAB 2 PEMBAHASAN

2.1 Penjelasan Implementasi

2.1.1 Deskripsi Kelas, Atribut, dan Method

Berikut adalah penjelasan dari kelas, atribut dari masing-masing kelas, dan method yang ada dalam masing-masing kelas.

Class FFNN2 (ffnn2.py)

Kelas FFNN2 bertanggung jawab untuk mengimplementasikan modul FFNN dengan feedforward yang bisa menerima berbagai fungsi aktivasi, inisialisasi bobot, dan fungsi loss, serta dilengkapi dengan mekanisme training menggunakan backpropagation dan optimasi berbasis mini-batch dengan multithreading untuk meningkatkan efisiensi komputasi. Kelas ini memungkinkan pengguna untuk menentukan arsitektur jaringan yang ingin digunakan dalam melakukan training menggunakan algoritma forward dan backward propagation, memperbarui bobot berdasarkan gradien yang dihitung, serta menyimpan dan memuat model menggunakan library pickle untuk keperluan pemuatan (save dan load). Selain itu, kelas ini juga menyediakan visualisasi distribusi bobot dan gradien untuk membantu analisis performa jaringan selama proses pelatihan.

Atribut

```
 1 def __init__(  
 2     self,  
 3     jumlah_neuron: list[int],  
 4     fungsi_aktivasi: list[Literal["Linear", "Sigmoid", "ReLU", "Tanh", "Softmax"]],  
 5     fungsi_loss: Literal["MSE", "BinaryCrossEntropy", "CategoricalCrossEntropy"],  
 6     inisialisasi_bobot: Literal[  
 7         "zero", "uniform", "normal", "xavier-uniform", "xavier-normal"  
 8     ],  
 9     lower_bound: float = -1.0,  
10     upper_bound: float = 1.0,  
11     mean: float = 0.0,  
12     std: float = 1.0,  
13     seed: int = 0,  
14     verbose: int = 0,  
15     l1_lambda: float = 0.0,  
16     l2_lambda: float = 0.0,  
17 ):  
18     self.jumlah_neuron = jumlah_neuron  
19     self.jumlah_layer = len(jumlah_neuron) - 1  
20  
21     # Inisialisasi fungsi aktivasi  
22     self.fungsi_aktivasi_str = fungsi_aktivasi  
23     self.fungsi_aktivasi = list(  
24         ActivationFunction(fungsi_aktivasi).get_batch_activation_function()  
25     )  
26  
27     # Inisialisasi fungsi loss  
28     self.fungsi_loss_str = fungsi_loss  
29     self.fungsi_loss_class = LossFunction(self.fungsi_loss_str)  
30     self.fungsi_loss = self.fungsi_loss_class.get_lost_function()  
31  
32     # Inisialisasi bobot  
33     # ? Bias dimasukkan ke dalam bobot indeks terakhir  
34     self.inisialisasi_bobot_str = inisialisasi_bobot  
35     self.inisialisasi_bobot = WeightInitiation(  
36         inisialisasi_bobot, self.jumlah_layer, jumlah_neuron  
37     )  
38  
39     # Inisialisasi Lambda  
40     self.l1_lambda = l1_lambda  
41     self.l2_lambda = l2_lambda  
42  
43     self.lower_bound = lower_bound  
44     self.upper_bound = upper_bound  
45     self.mean = mean  
46     self.std = std  
47     self.seed = seed  
48     self.init_bobot()  
49  
50     self.X = None  
51     self.y = None  
52     self.hasil = None  
53     self.loss = None  
54     self.verbose = verbose  
55     self.weight_history = []  
56     self.lr = None  
57  
58     # Validasi input  
59     try:  
60         self.validate_input()  
61     except ValueError as e:  
62         print(f"Error: {e}")  
63         raise
```

Nama Atribut	Penjelasan
self.verbose	Menentukan apakah akan menampilkan informasi pelatihan atau tidak
self.jumlah_neuron	List yang berisi jumlah neuron pada setiap layer
self.jumlah_layer	Jumlah layer dalam jaringan, dihitung dari len(jumlah_neuron) - 1
self.fungsi_aktivasi	List fungsi aktivasi yang digunakan di setiap layer
self.fungsi_loss_class	Objek dari kelas LossFunction, digunakan untuk menghitung loss dan turunannya
self.fungsi_loss	Fungsi loss yang digunakan tiap neuron
self.fungsi_loss_str	String nama fungsi loss yang dipilih
self.inisialisasi_bobot_str	String metode inisialisasi bobot
self.inisialisasi_bobot	Objek dari kelas WeightInitiation yang digunakan untuk inisialisasi bobot
self.fungsi_aktivasi_str	List string nama fungsi aktivasi
self.bobot	List matriks bobot untuk setiap layer
self.gradients	Gradien bobot, diperbarui setelah backward pass (hanya ada setelah training dijalankan)
self.lr	Learning rate, tetapi hanya disimpan jika sudah di-train (fit() dipanggil)
self.hasil	Menyimpan hasil output dari forward pass
self.loss	Menyimpan nilai loss per epoch
self.weight_history	Menyimpan nilai bobot setiap neuron dan setiap iterasi
self.l1_lambda	Menyimpan koefisien (<i>hyperparameter</i>) untuk regularisasi L1, yang digunakan untuk mengontrol seberapa kuat penalti L1 diterapkan pada bobot model

self.l2_lambda	Menyimpan koefisien (<i>hyperparameter</i>) untuk regularisasi L2, yang digunakan untuk mengontrol seberapa kuat penalti L2 diterapkan pada bobot model (biasanya berupa penjumlahan kuadrat bobot).
Konstruktor	
Fungsi <code>__init__</code> di atas juga sekaligus konstruktor yang menginisialisasi jumlah neuron, fungsi aktivasi, fungsi loss, metode inisialisasi bobot, serta mengatur atribut penting seperti bobot jaringan, jumlah layer, dan metode aktivasi, sehingga model siap digunakan untuk training dan prediksi.	
Fungsi/Prosedur	Penjelasan
<code>def init_bobot(self) :</code>	Metode <code>init_bobot</code> bertanggung jawab untuk menginisialisasi bobot tiap neuron menggunakan metode inisialisasi yang telah ditentukan sebelumnya, memastikan bahwa setiap layer memiliki bobot awal sebelum proses training dimulai.
<code>def forward(self, x) :</code>	Metode <code>forward</code> bertanggung jawab untuk melakukan forward propagation dengan menerapkan bobot dan fungsi aktivasi pada setiap layer, sehingga menghasilkan output akhir yang dapat digunakan untuk prediksi atau perhitungan error.
<code>def backward(self, hasil, y) :</code>	Metode <code>backward</code> bertanggung jawab untuk menghitung gradien error terhadap bobot

	menggunakan algoritma backpropagation, dimulai dari perhitungan error pada layer output hingga propagasi mundur ke layer sebelumnya dengan mempertimbangkan turunan dari fungsi aktivasi dan loss.
<pre>def update(self, gradients, lr):</pre>	Metode update bertanggung jawab untuk memperbarui bobot dari tiap neuron berdasarkan gradien yang telah dihitung selama proses backpropagation, dengan menyesuaikan nilai bobot menggunakan learning rate agar model dapat belajar dari data yang diberikan.
<pre>def fit(self, X, y, batch, lr, epochs):</pre>	Metode fit bertanggung jawab untuk melatih model dengan membagi data menjadi batch kecil, menerapkan forward dan backward propagation pada setiap batch, serta memperbarui bobotnya secara paralel menggunakan ThreadPoolExecutor, sambil menampilkan perkembangan training loss selama training berlangsung.
<pre>def predict(self, X):</pre>	Metode predict bertanggung jawab untuk melakukan prediksi terhadap data input dengan menerapkan forward propagation, kemudian mengambil indeks dengan

	probabilitas tertinggi dari hasil akhir untuk menentukan kelas prediksi.
<pre>def save_model_pickle(self, filename: str):</pre>	Metode save_model_pickle bertanggung jawab untuk menyimpan model ke dalam file menggunakan format pickle, termasuk parameter utama seperti jumlah neuron, bobot, fungsi aktivasi, dan loss, sehingga model dapat digunakan kembali tanpa perlu dilatih ulang.
<pre>def load_model_pickle(self, filename: str):</pre>	Metode load_model_pickle bertanggung jawab untuk memuat kembali model dari file pickle yang telah disimpan, dengan membaca parameter yang ada di dalam file dan mengembalikan model ke keadaan yang siap digunakan untuk prediksi atau training lanjutan.
<pre>def tampilkan_distribusi_bob bot(self, layer: list[int]):</pre>	Metode tampilkan_distribusi_bobot bertanggung jawab untuk menampilkan histogram distribusi bobot dari layer tertentu dalam model, bisa digunakan untuk analisis terhadap distribusi bobot untuk memahami bagaimana bobot berkembang selama training.

```

def
    tampilkan_distribusi_gr
    adient_bobot(self,
layer: list[int]):

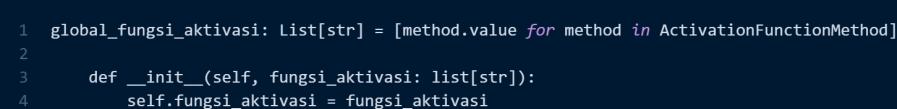
```

Metode backward bertanggung jawab untuk menghitung gradien error terhadap bobot menggunakan algoritma backpropagation, dimulai dari perhitungan error pada layer output hingga propagasi mundur ke layer sebelumnya dengan mempertimbangkan turunan dari fungsi aktivasi dan loss.

Class ActivationFunctionMethod (activationFunction.py)

Kelas ActivationFunction bertanggung jawab untuk menyediakan berbagai fungsi aktivasi yang digunakan dalam FFNN, termasuk metode untuk menghitung nilai aktivasi serta turunannya yang diperlukan dalam proses backpropagation.

Atribut



```

● ● ●
1 global_fungsi_aktivasi: List[str] = [method.value for method in ActivationFunctionMethod]
2
3 def __init__(self, fungsi_aktivasi: list[str]):
4     self.fungsi_aktivasi = fungsi_aktivasi

```

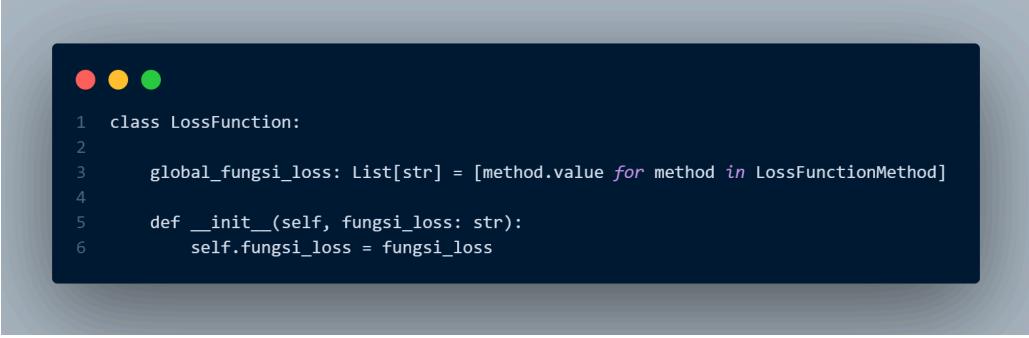
Nama Atribut	Penjelasan
global_fungsi_aktivasi	Daftar semua metode aktivasi yang tersedia dalam bentuk string
fungsi_aktivasi	Daftar fungsi aktivasi yang digunakan dalam suatu instance.

Konstruktor

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi fungsi aktivasi yang digunakan dalam proses *training*.

Fungsi/Prosedur	Penjelasan
<pre data-bbox="393 494 780 616"><code>def validate_input(self, fungsi_aktivasi):</code></pre>	Metode <code>validate_input</code> bertanggung jawab untuk memverifikasi apakah fungsi aktivasi yang diberikan valid dengan membandingkannya dengan daftar fungsi aktivasi yang tersedia.
<pre data-bbox="393 726 731 762"><code>def linear(self, x):</code></pre>	Metode <code>linear</code> bertanggung jawab untuk mengimplementasikan fungsi aktivasi linear yang hanya mengembalikan input sebagaimana adanya.
<pre data-bbox="393 863 780 986"><code>def linear_derivative(self, x):</code></pre>	Metode <code>linear_derivative</code> bertanggung jawab untuk menghitung turunan dari fungsi aktivasi linear, yang selalu bernilai 1.
<pre data-bbox="393 1096 698 1132"><code>def relu(self, x):</code></pre>	Metode <code>relu</code> bertanggung jawab untuk mengimplementasikan fungsi aktivasi ReLU (Rectified Linear Unit), yang mengembalikan nilai input jika positif dan nol jika negatif.
<pre data-bbox="393 1265 747 1387"><code>def relu_derivative(self, x):</code></pre>	Metode <code>relu_derivative</code> bertanggung jawab untuk menghitung turunan dari fungsi aktivasi ReLU, yang bernilai 1 untuk input positif dan 0 untuk input negatif.
<pre data-bbox="393 1497 747 1533"><code>def sigmoid(self, x):</code></pre>	Metode <code>sigmoid</code> bertanggung jawab untuk mengimplementasikan fungsi aktivasi sigmoid yang memetakan input ke rentang (0,1), berguna untuk model klasifikasi biner.
<pre data-bbox="393 1655 784 1778"><code>def sigmoid_derivative(self, x):</code></pre>	Metode <code>sigmoid_derivative</code> bertanggung jawab untuk menghitung turunan dari fungsi aktivasi sigmoid, yang menggunakan rumus $\text{sigmoid}(x) * (1 - \text{sigmoid}(x))$.

<pre>def tanh(self, x):</pre>	Metode tanh bertanggung jawab untuk mengimplementasikan fungsi aktivasi tangens hiperbolik (tanh), yang memetakan input ke rentang (-1,1).
<pre>def tanh_derivative(self, x):</pre>	Metode tanh_derivative bertanggung jawab untuk menghitung turunan dari fungsi aktivasi tanh dengan rumus $(2 / (\exp(x) - \exp(-x)))^2$.
<pre>def softmax(self, x):</pre>	Metode softmax bertanggung jawab untuk mengimplementasikan fungsi aktivasi softmax yang mengonversi input menjadi distribusi probabilitas, sering digunakan pada lapisan output dalam klasifikasi multi-kelas.
<pre>def softmax_derivative(self , x):</pre>	Metode softmax_derivative bertanggung jawab untuk menghitung turunan dari fungsi aktivasi softmax, namun dalam implementasi ini hanya menghitung elemen diagonal dari matriks Jacobian.
<pre>def get_activation_function (self, fungsi_aktivasi: ActivationFunctionMetho d):</pre>	Metode get_activation_function bertanggung jawab untuk mengambil referensi fungsi aktivasi yang sesuai berdasarkan string metode aktivasi yang diberikan.
<pre>def get_batch_activation_fu nction(self):</pre>	Metode get_batch_activation_function bertanggung jawab untuk menghasilkan fungsi aktivasi dalam bentuk generator, yang memungkinkan iterasi melalui fungsi aktivasi dalam batch.
<pre>def get_activation_derivative(self, fungsi_aktivasi: str):</pre>	Metode get_activation_derivative bertanggung jawab untuk mengambil referensi turunan dari fungsi aktivasi yang sesuai berdasarkan metode aktivasi yang diberikan.

<pre>def get_batch_activation_derivative(self):</pre>	Metode get_batch_activation_derivative bertanggung jawab untuk menghasilkan turunan fungsi aktivasi dalam bentuk generator, memungkinkan iterasi melalui turunan fungsi aktivasi dalam batch.
Class LossFunctionMethod (lossFunction.py)	
Kelas LossFunction bertanggung jawab untuk mengelola berbagai fungsi loss (kerugian) yang digunakan dalam pelatihan jaringan saraf tiruan, termasuk Mean Squared Error (MSE), Binary Cross-Entropy, dan Categorical Cross-Entropy, serta menghitung turunannya untuk keperluan backpropagation.	
Atribut	
 <pre>1 class LossFunction: 2 3 global_fungsi_loss: List[str] = [method.value for method in LossFunctionMethod] 4 5 def __init__(self, fungsi_loss: str): 6 self.fungsi_loss = fungsi_loss</pre>	
Nama Atribut	Penjelasan
global_fungsi_loss	Daftar semua metode perhitungan loss yang tersedia dalam bentuk string
fungsi_loss	Daftar fungsi loss yang digunakan dalam suatu instance.
Konstruktor	
Fungsi __init__ di atas juga sekaligus konstruktor yang menginisialisasi fungsi loss yang digunakan dalam proses <i>training</i> .	
Fungsi/Prosedur	Penjelasan
<pre>def inject_class_num(self,</pre>	Metode inject_class_num bertanggung jawab untuk menyimpan jumlah kelas dalam atribut class_num, yang berguna pada categorical loss

<pre>class_num) :</pre>	
<pre>def get_lost_function(self) :</pre>	Metode get_lost_function bertanggung jawab untuk mengembalikan fungsi loss yang sesuai berdasarkan metode loss yang telah dipilih dalam instance.
<pre>def get_loss_derivative(sel f) :</pre>	Metode get_loss_derivative bertanggung jawab untuk mengembalikan fungsi turunan dari loss yang dipilih agar dapat digunakan dalam perhitungan backpropagation.
<pre>def mse(self, y_pred, y_true) :</pre>	Metode mse bertanggung jawab untuk menghitung nilai Mean Squared Error (MSE), yaitu rata-rata dari jumlah kuadrat selisih antara prediksi dan target.
<pre>def binarycrossentropy(self , y_pred, y_true) :</pre>	Metode binarycrossentropy bertanggung jawab untuk menghitung nilai Binary Cross-Entropy, yang digunakan dalam klasifikasi biner dengan memastikan nilai prediksi tidak mencapai nol atau satu agar menghindari error pada logaritma.
<pre>def categoricalcrossentropy (self, y_pred, y_true) :</pre>	Metode categoricalcrossentropy bertanggung jawab untuk menghitung nilai Categorical Cross-Entropy, yang digunakan dalam klasifikasi multi-kelas dengan cara mengalikan target one-hot dengan logaritma dari probabilitas prediksi.
<pre>def mse_derivative(self, y_pred, y_true) :</pre>	Metode mse_derivative bertanggung jawab untuk menghitung turunan dari fungsi loss MSE dengan rumus $2 * (y_{pred} - y_{true})$.
<pre>def binary_cross_entropy_de rivative(self, y_pred, y_true) :</pre>	Metode binary_cross_entropy_derivative bertanggung jawab untuk menghitung turunan dari Binary Cross-Entropy dengan menghindari pembagian oleh nol menggunakan nilai epsilon kecil.

```

def
categorical_cross_entropy_derivative(self,
y_pred, y_true):

```

Metode categorical_cross_entropy_derivative bertanggung jawab untuk menghitung turunan dari Categorical Cross-Entropy dengan mengembalikan selisih antara prediksi dan target.

GraphNode (node.py)

Kelas GraphNode bertanggung jawab untuk merepresentasikan satu neuron dalam *artificial neural network* (FFNN) menggunakan model *linked list*. Kelas ini memiliki beberapa atribut utama yang digunakan untuk menyimpan informasi tentang node dalam jaringan.

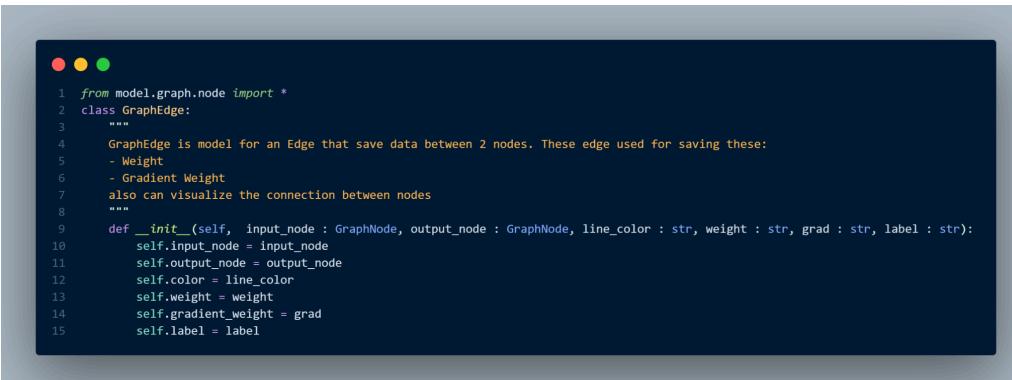
Atribut

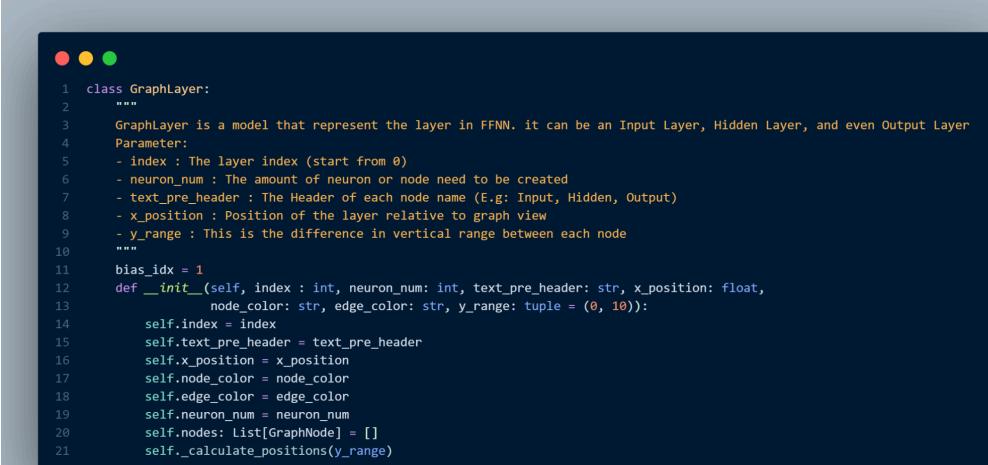
```

● ● ●
1 import numpy as np
2 class GraphNode:
3     """
4         GraphNode is a linked list model representing one neuron in FFNN Neural network
5         Parameter:
6             - text : the text written in the node
7             - pos : position of the node in the viewport
8             - color : the color of the node
9     """
10    def __init__(self, text : str, pos: tuple, color: str):
11        self.text = text
12        self.pos = np.array(pos, dtype=float)
13        self.color = color

```

Nama Atribut	Penjelasan
self.text = text	Atribut text bertanggung jawab untuk menyimpan teks yang ditampilkan dalam node, yang biasanya merepresentasikan identitas atau label dari neuron tersebut.
self.pos = np.array(pos, dtype=float)	Atribut pos bertanggung jawab untuk menyimpan posisi node dalam viewport sebagai array NumPy dengan tipe data float. Posisi ini memungkinkan manipulasi yang lebih fleksibel saat memvisualisasikan atau mengubah posisi node dalam graf.

self.color = color	Atribut color bertanggung jawab untuk menentukan warna node, yang bisa digunakan untuk membedakan berbagai jenis node dalam jaringan.
Konstruktor	
Metode <code>__init__</code> bertanggung jawab untuk menginisialisasi objek GraphNode dengan nilai teks, posisi, dan warna yang diberikan saat pembuatan instance.	
GraphEdge (edge.py)	
Kelas GraphEdge bertanggung jawab merepresentasikan sebuah edge atau hubungan antara dua node dalam <i>neural network</i> . Edge ini berfungsi sebagai penghubung antara dua neuron dan menyimpan informasi penting mengenai bobot dan gradien bobot dari koneksi tersebut.	
Atribut	
 <pre> 1 from model.graph.node import * 2 class GraphEdge: 3 """ 4 GraphEdge is model for an Edge that save data between 2 nodes. These edge used for saving these: 5 - Weight 6 - Gradient Weight 7 also can visualize the connection between nodes 8 """ 9 def __init__(self, input_node : GraphNode, output_node : GraphNode, line_color : str, weight : str, grad : str, label : str): 10 self.input_node = input_node 11 self.output_node = output_node 12 self.color = line_color 13 self.weight = weight 14 self.gradient_weight = grad 15 self.label = label </pre>	
Nama Atribut	Penjelasan
self.input_node dan self.output_node	Atribut ini bertanggung jawab untuk menyimpan referensi ke node asal dan node tujuan dalam koneksi memungkinkan jaringan untuk memahami bagaimana informasi mengalir dari satu neuron ke neuron lainnya.
self.color	Atribut ini bertanggung jawab untuk menyimpan warna garis yang digunakan untuk merepresentasikan koneksi antar node dalam visualisasi graf.
self.weight	Atribut ini digunakan untuk menyimpan bobot koneksi, yang menentukan seberapa kuat pengaruh input node terhadap output node dalam jaringan saraf tiruan.

self.gradient_weight	Atribut gradient_weight bertanggung jawab untuk menyimpan gradien dari bobot koneksi. Informasi ini digunakan dalam proses pembelajaran untuk memperbarui bobot berdasarkan error yang diperoleh.
self.label	Atribut label digunakan untuk memberikan identitas atau deskripsi pada edge, yang dapat digunakan dalam visualisasi atau debugging jaringan saraf tiruan.
Konstruktor	
Metode <code>__init__</code> bertanggung jawab untuk menginisialisasi objek GraphEdge dengan node input, node output, warna garis, bobot, gradien bobot, dan label yang diberikan saat pembuatan instance.	
GraphLayer (layer.py)	
Kelas GraphLayer bertanggung jawab untuk merepresentasikan satu <i>layer</i> dalam <i>neural network feedforward</i> (FFNN). Lapisan ini dapat berupa Input Layer, Hidden Layer, maupun Output Layer, yang masing-masing berisi sejumlah neuron yang bertugas memproses data.	
Atribut	
 <pre> 1 class GraphLayer: 2 """ 3 GraphLayer is a model that represent the layer in FFNN. it can be an Input Layer, Hidden Layer, and even Output Layer 4 Parameter: 5 - index : The layer index (start from 0) 6 - neuron_num : The amount of neuron or node need to be created 7 - text_pre_header : The Header of each node name (E.g: Input, Hidden, Output) 8 - x_position : Position of the layer relative to graph view 9 - y_range : This is the difference in vertical range between each node 10 """ 11 bias_idx = 1 12 def __init__(self, index: int, neuron_num: int, text_pre_header: str, x_position: float, 13 node_color: str, edge_color: str, y_range: tuple = (0, 10)): 14 self.index = index 15 self.text_header = text_pre_header 16 self.x_position = x_position 17 self.node_color = node_color 18 self.edge_color = edge_color 19 self.neuron_num = neuron_num 20 self.nodes: List[GraphNode] = [] 21 self._calculate_positions(y_range) </pre>	
Nama Atribut	Penjelasan
self.index	Atribut index digunakan untuk menyimpan indeks lapisan dalam jaringan, di mana lapisan pertama dimulai dari indeks 0.

self.neuron_num	Atribut neuron_num menentukan jumlah neuron yang akan dibuat dalam <i>layer</i> tersebut. Setiap neuron direpresentasikan sebagai objek dari kelas GraphNode.
self.text_pre_header	Atribut text_pre_header berfungsi sebagai awalan dari nama setiap node dalam <i>layer</i> ini, misalnya "Input", "Hidden", atau "Output", untuk membedakan jenis <i>layer</i> .
self.x_position	Atribut x_position menentukan posisi horizontal dari <i>layer</i> ini dalam tampilan graf.
self.node_color	Atribut node_color digunakan untuk menentukan warna yang digunakan dalam visualisasi neuron dalam <i>layer</i> ini, sedangkan edge_color menentukan warna garis yang menghubungkan node dalam jaringan.
self.nodes	Atribut nodes adalah sebuah list yang menyimpan semua neuron yang ada dalam lapisan ini. Neuron-neuron ini dibuat dengan mempertimbangkan distribusi vertikalnya dalam y_range, yang merupakan rentang posisi vertikal antar neuron.
Konstruktor	
Fungsi/Prosedur	Penjelasan
<pre>def _calculate_positions(se lf, y_range: tuple):</pre>	Metode _calculate_positions bertanggung jawab untuk menghitung posisi setiap neuron dalam <i>layer</i> berdasarkan rentang y_range, di mana posisi neuron dihitung menggunakan np.linspace, yang membagi rentang secara merata berdasarkan jumlah neuron yang ada. Dalam metode ini, terdapat perlakuan khusus untuk bias neuron, yang ditempatkan pada setiap lapisan kecuali Output Layer. Bias neuron memiliki label "Bias" dan diberi indeks khusus yang terpisah dari neuron lainnya. Setiap neuron yang dibuat akan memiliki nama unik yang terdiri dari awalan (misalnya "Hidden"), indeks lapisan, dan nomor urut neuron. Misalnya, sebuah neuron dalam hidden layer pertama bisa diberi nama "Hidden0-2".

GraphModel (model.py)

GraphModel adalah sebuah model yang menyimpan hampir keseluruhan data yang akan dibutuhkan untuk seluruh View nantinya.

Atribut

```
 1 class GraphModel:
 2     """
 3     GraphModel is the main model which are the connector between the ML model and the views
 4     Parameter:
 5     - neuron_list : List of integer that specify the amount of neuron each layers have
 6     - weight_neuron_data : List of NpArray that save the weight of each edge
 7     - weight_grads_data : List of NpArray that save the gradient weight of each edge
 8     """
 9     def __init__(self, neuron_list : list[int], weight_neuron_data : list[list[list[str]]], weight_grads_data : list[list[list[str]]]):
10         self.nodes : List[GraphNode] = []
11         self.edges : List[GraphEdge]= []
12         self.layers : List[GraphLayer] = []
13         self.weights: List[np.ndarray] = []
14         self.gradien_weight : List[np.ndarray] = []
15         self.x_spacing = GraphConfig.LAYER_SPACING
16         self.y_range = GraphConfig.LAYER_Y_RANGE
17
18         self.layer_colors = ColorHelper.generate_colors(len(neuron_list), 'dark')
19
20         self._initiate_layers(neuron_list)
21         self._save_weight_gradient_weight(weight_neuron_data, weight_grads_data)
22         self._create_fully_connected_edges()
```

Nama Atribut	Penjelasan
nodes	Seluruh neuron yang diinisiasi
edges	Seluruh edge yang menghubungkan antar neuron yang diinisiasi
layers	Seluruh layer baik input, hidden, maupun output yang diinisiasi
weights	Menyimpan weight yang terdapat pada tiap edge. Disimpan secara terpisah untuk pengambilan data yang lebih mudah
gradien_weight	Menyimpan Gradien Weight yang terdapat pada tiap edge juga. Disimpan secara terpisah untuk pengambilan data yang lebih mudah
x_spacing	Menyimpan data jarak horizontal antar layer
y_range	Menyimpan data tuple (y1, y2) yaitu jarak vertikal antar node pada suatu layer yang berkisar antara y1 atau y2
layers_color	Menyimpan data color atau warna seluruh layers

Konstruktor	
Fungsi __init__ di atas juga sekaligus konstruktor yang menginisialisasi model dimana pada awalnya seluruh layers dibuat terlebih dahulu. Dilanjut dengan pembuatan seluruh edges	
Fungsi/Prosedur	Penjelasan
<pre>def _save_weight_gradient_weight(self, weight_neuron_data : list[list[list[str]]], weight_grads_data : list[list[list[str]]])</pre>	Menyimpan data weight neuron dan weight gradient data untuk akses yang lebih mudah
<pre>def _initiate_layers(self, neuron_list : list[int]):</pre>	Membuat seluruh layers termasuk neuron didalamnya
<pre>def _add_layer(self, layer: GraphLayer):</pre>	Helper function untuk menambahkan layer baru pada self.layers sekaligus menambahkan node pada layer tersebut pada self.nodes
<pre>def _add_node(self, node : GraphNode):</pre>	Helper function untuk menambahkan node baru pada self.nodes
<pre>def _create_fully_connected_edges(self):</pre>	Setelah seluruh layers dan nodes selesai dibuat, menginisiasi-kan seluruh edge antar node

<pre>def getAllEdgesFromNode(self, node) -> list[GraphEdge] :</pre>	Getter function untuk menambahkan edge yang berasal dari node tertentu
<pre>def getNode(self, node_index : int):</pre>	Getter function untuk mendapatkan node pada index tertentu
PaginatedTableWidget (paginatedTable.py)	
Sebuah View class yang menampilkan table weight dan gradient dari suatu neuron. Table ini juga memiliki pagination sehingga seluruh data weight sebuah neuron dapat ditampilkan dengan baik	
Table ini akan dimunculkan ketika melakukan inspect suatu neuron	
Atribut	
 <pre>1 class PaginatedTableWidget(QWidget): 2 closeRequested = pyqtSignal() 3 def __init__(self, data, rows_per_page=10): 4 """ 5 data: list of tuples, each tuple should be (weight_label, weight_value, gradient_value) 6 rows_per_page: number of rows to show per page 7 """ 8 super().__init__() 9 self.data = data 10 self.rows_per_page = rows_per_page 11 self.current_page = 1 12 self.total_pages = max(1, (len(self.data) + rows_per_page - 1) // rows_per_page) 13 self.node_name = "" 14 15 self._initUI()</pre>	
Nama Atribut	Penjelasan
closeRequested	Sebuah signal yang digunakan untuk close button slot
data	Data adalah list of tuple yang berisikan weight_label (e.g: W[I1][H1-10] yaitu Weight dari

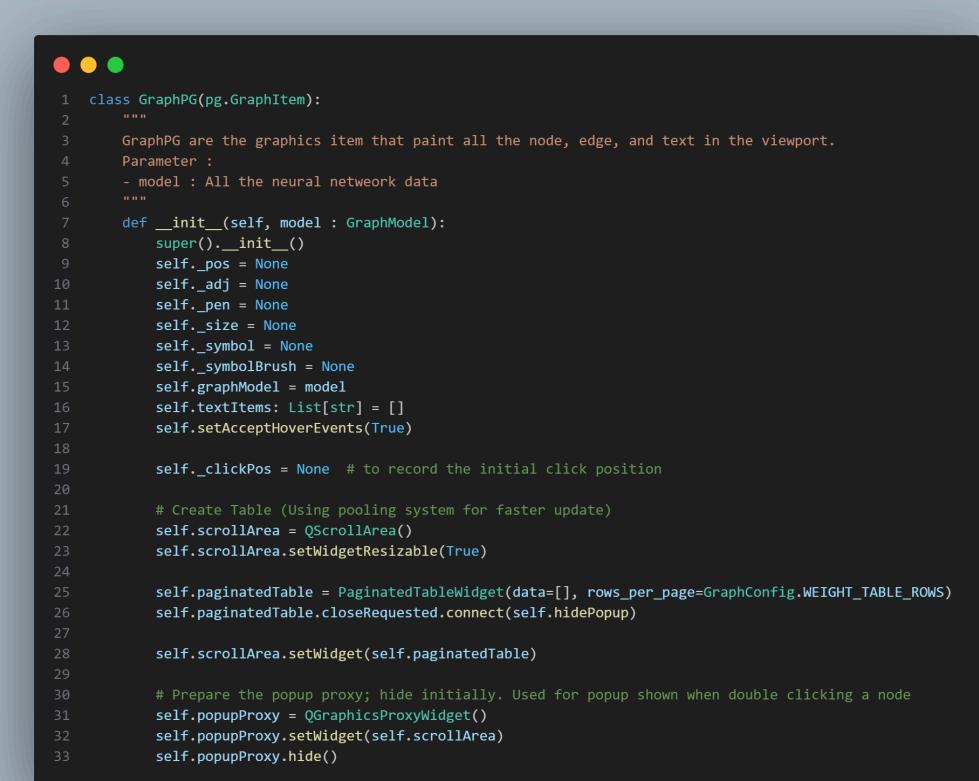
	input 1 ke Hidden layer 1 ke 10), weight_value, dan gradient_value
rows_per_page	Jumlah baris pada sebuah page (Digunakan untuk pagination)
current_page	State yang menyimpan page saat ini
total_page	Jumlah keseluruhan page
node_name	Nama dari neuron yang sedang di inspect
Konstruktor	
Fungsi __init__ di atas juga sekaligus konstruktor yang membangun seluruh table. Construtor disini hanya akan membuat table dengan data yang sifatnya placeholder. Data dari table ini akan selalu berubah ubah pada tiap inspeksi suatu neuron	
Fungsi/Prosedur	Penjelasan
<pre>def __initUI(self):</pre>	Membuat Widget UI yang berisikan data awal pada konstruktor. UI disini berisikan seperti TextWidget untuk node_name, TableWidget, dan Buttons untuk pagination, serta QTextEdit untuk search page tertentu
<pre>def updateTitle(self):</pre>	Melakukan update tiap kali inspeksi sebuah neuron dilakukan. Update ini melakukan penggantian pada TextWidget dengan node_name terbaru
<pre>def _loadPage(self, page):</pre>	Load data dari page tertentu dan menggantikan table dengan list of data pada page tersebut
<pre>def _goToPreviousPage(self):</pre>	Fungsi untuk kembali ke page sebelumnya
<pre>def _goToNextPage(self):</pre>	Fungsi untuk pergi ke page selanjutnya

<pre>def _goToPage(self):</pre>	Fungsi yang digunakan spesial untuk QTextEdit yang akan otomatis berpindah ke page tertentu setelah dilakukan input
<pre>def updateData(self, new_data, color, node_name):</pre>	Fungsi yang melakukan update Data. Fungsi ini akan selalu dipanggil ketika akan melakukan inspeksi neuron baru. Fungsi ini akan melakukan reset seluruh data dan attribute yang ada.

GraphPG(graphPG.py)

Class utama yang menggambarkan Model Neuron pada canvas. Class ini yang berfungsi menggambarkan seluruh graph dan Table untuk weight nya

Atribut



```

1  class GraphPG(pg.GraphItem):
2      """
3          GraphPG are the graphics item that paint all the node, edge, and text in the viewport.
4          Parameter :
5              - model : All the neural netwework data
6      """
7      def __init__(self, model : GraphModel):
8          super().__init__()
9          self._pos = None
10         self._adj = None
11         self._pen = None
12         self._size = None
13         self._symbol = None
14         self._symbolBrush = None
15         self.graphModel = model
16         self.textItems: List[str] = []
17         self.setAcceptHoverEvents(True)
18
19         self._clickPos = None # to record the initial click position
20
21         # Create Table (Using pooling system for faster update)
22         self.scrollArea = QScrollArea()
23         self.scrollArea.setWidgetResizable(True)
24
25         self.paginatedTable = PaginatedTableWidget(data=[], rows_per_page=GraphConfig.WEIGHT_TABLE_ROWS)
26         self.paginatedTable.closeRequested.connect(self.hidePopup)
27
28         self.scrollArea.setWidget(self.paginatedTable)
29
30         # Prepare the popup proxy; hide initially. Used for popup shown when double clicking a node
31         self.popupProxy = QGraphicsProxyWidget()
32         self.popupProxy.setWidget(self.scrollArea)
33         self.popupProxy.hide()

```

Nama Atribut	Penjelasan
pos	Menyimpan list of position dari tiap neuron

adj	List of list of adjacency dari seluruh neuron (Tiap neuron punya tetangga, variable ini menyimpan list dari list of tetangga tersebut)
pen	Menyimpan list dari warna tiap garis
symbol	Bentuk dari Neuron
size	Ukuran dari tiap neuron
symbolBrush	List of warna tiap neuron
graphModel	Model dari neuron network
textItems	Menyimpan texts yaitu label dari seluruh neuron
_clickPos	State yang menyimpan posisi click awal
scrollArea	Scroll area widget yang digunakan untuk scroll pada PaginatedTableWidget.
popupProxy	Popup widget yang muncul yang berisikan PaginatedTableWidgets yang dapat di scroll.
kd_tree	Sebuah cKDTree yang akan digunakan untuk mencari jarak antara posisi klik dengan object tertentu
Konstruktor	
Fungsi __init__ di atas juga sekaligus konstruktor yang menginisiasi seluruh view dari neuron network. Sekaligus menginisiasi PaginatedTableWidgets. Setelah itu table widgets akan disembunyikan hal ini untuk menghindari pembuatan table widgets setiap saat akan dipanggil. Sehingga cukup dengan mengaktifkan kembali saja table widgets tersebut.	
Fungsi/Prosedur	Penjelasan
<code>def hidePopup(self)</code>	Menyembunyikan Popup yang berisikan PaginatedTableWidgets.
<code>def setGraphData(self, pos: np.ndarray, adj: np.ndarray, pen, size: float, symbol: str,</code>	Menggambar ulang seluruh Graphic yang ada kecuali untuk popup paginatedTable

	<code>symbolBrush) :</code>	
	<code>def initTextItems(self, pos, textItems):</code>	Menginisiasi pembuatan textItems yaitu label untuk seluruh neuron
	<code>def updateTextItems(self, textItems):</code>	Mengupdate textItems
	<code>def mouseDoubleClickEvent(self, ev):</code>	Mendeteksi double click event. Jika Double click dilakukan didekat suatu node, akan dilakukan inspeksi node yaitu menampilkan Popup berupa PaginatedTableWidgets yang berisikan seluruh data weight dan gradien suatu neuron.
	<code>def showScrollablePanel(self, node_index: int):</code>	Menampilkan popup dan mengupdate data sesuai dengan node_index. Node_index adalah neuron atau node yang terdeteksi ketika double click dilakukan
GraphWidget (graph.py)		
Sebuah class yang menyimpan GraphPG dan membungkusnya dalam sebuah widget agar dapat digunakan pada layout tertentu. Class ini juga sekaligus menginisialisasikan bentuk awal dari Neural Network Nya		
Atribut		
<pre> 1 class GraphWidget(QWidget): 2 """ 3 GraphWidget are the widget storing GraphPG graphical items. In this class, all the required data like node and edge position, texts, colors that will be used in GraphPG are initialized 4 Parameter: 5 - graph_model : All the neural network Data 6 - parent: The widget parent 7 """ 8 def __init__(self, graph_model: GraphModel, parent=None): 9 super().__init__(parent) 10 self.graph_model = graph_model 11 self.initUI() </pre>		
Nama Atribut	Penjelasan	
graph_model	Model dari keseluruhan Graph	

graph_item	Graphic yaitu GraphPg yang menampilkan keseluruhan canvas
Konstruktor	
Fungsi __init__ di atas juga sekaligus konstruktor yang menginisiasikan layout, widget, dan data dari graphic yang diperlukan pada GraphPG	
Fungsi/Prosedur	Penjelasan
<pre>def initUI(self):</pre>	Menginisiasikan GraphPG dan data-data yang diperlukannya. Dan melingkupinnya dengan Layout sehingga dapat digunakan diluar class
SinglePlotDistribution (layersDistribution.py)	
Kelas SinglePlotDistribution bertanggung jawab untuk memvisualisasikan distribusi bobot atau gradien dalam <i>neural network</i> (FFNN). Distribusi ini dapat divisualisasikan dalam bentuk histogram, scatter plot, atau kurva Gaussian. Kelas ini menggunakan PyQt dan pyqtgraph untuk membuat tampilan grafik.	
Atribut	
 <pre> 1 def __init__(self, layer_data: list[tuple[int, np.ndarray]], distribution_mode: str = 'gaussian', parent=None): 2 """ 3 This class is just combining Multiple SinglePlotDistribution that each layer get it's own Graph 4 Parameters: 5 - layer_data: List of tuples (layer_index, data) 6 - distribution_mode: Distribution mode for each plot ('gaussian', 'histogram', or 'scatter') 7 """ 8 super().__init__(parent) 9 self.setWindowTitle("Multi-Layer Distribution") 10 self.distribution_mode = distribution_mode.lower() 11 layout = QVBoxLayout(self) 12 13 self.scrollArea = QScrollArea(self) 14 self.scrollArea.setWidgetResizable(True) 15 layout.addWidget(self.scrollArea) 16 container = QWidget() 17 self.containerLayout = QVBoxLayout(container) 18 self.scrollArea.setWidget(container) 19 20 21 for layer_idx, data in layer_data: 22 23 widget = SinglePlotDistribution(layer_data=[(layer_idx, data)], distribution_mode=self.distribution_mode) 24 self.containerLayout.addWidget(widget) </pre>	
Nama Atribut	Penjelasan

self.layer_data	Menyimpan data bobot atau gradien setiap lapisan dalam bentuk tuple (layer_index, data).
self.distribution_mode	Menentukan mode distribusi yang akan digunakan ('histogram', 'scatter', atau 'gaussian').
self.plotWidget	Widget untuk menampilkan grafik distribusi
self.legend	Menampilkan legenda untuk tiap lapisan grafik
self.timer	Timer untuk memperbarui grafik tiap 1 detik
Konstruktor	
Metode <code>__init__</code> digunakan untuk inisialisasi objek SinglePlotDistribution dengan data bobot/gradien dan mode distribusi tertentu.	
Fungsi/Prosedur	Penjelasan
<code>def setupPlot(self) :</code>	Metode <code>setupPlot()</code> digunakan untuk membersihkan plot sebelum menggambar ulang dan menambahkan legenda baru ke dalamnya.
<code>def updatePlot(self) :</code>	<code>updatePlot()</code> akan memanggil <code>setupPlot()</code> untuk menghapus grafik sebelumnya dan memilih metode visualisasi yang sesuai berdasarkan nilai <code>distribution_mode</code> .
<code>def createHistogramDistribution(self, bin_count=50)</code>	Metode <code>createHistogramDistribution(bin_count=50)</code> bertanggung jawab untuk membuat histogram distribusi bobot atau gradien.
<code>def createScatterDistribution(self, bin_count=50) :</code>	<code>createScatterDistribution(bin_count=50)</code> membuat scatter plot dari distribusi bobot atau gradien dengan cara memproses data menggunakan <code>np.histogram()</code> untuk menentukan frekuensi, lalu menambahkan titik-titik ke dalam plot menggunakan <code>pg.ScatterPlotItem</code> .
<code>def createGaussianCurveDistribution(self) :</code>	Metode <code>createGaussianCurveDistribution()</code> akan menghitung rentang nilai global semua bobot dalam <i>network</i> , menggunakan mean dan standard deviation dari bobot tiap lapisan, serta menghitung kurva Gaussian berdasarkan distribusi probabilitas.

<pre><code>def WeightDistribution(cls, graph_model : GraphModel, layer_index_list : list[int],distribution_ mode : str = 'gaussian', parent = None) :</code></pre>	<p>Digunakan untuk membuat objek SinglePlotDistribution berdasarkan bobot lapisan tertentu dalam model graf.</p>
<pre><code>def GradientWeightDistribution(cls, graph_model : GraphModel, layer_index_list : list[int],distribution_ mode : str = 'gaussian', parent = None) :</code></pre>	<p>GradientWeightDistribution(graph_model, layer_index_list, distribution_mode='gaussian', parent=None) berfungsi hampir sama, tetapi mengambil data gradien bobot dari graph_model.gradien_weight.</p>
<h3>MultiPlotDistribution (layersDistribution.py)</h3>	
<p>Kelas MultiPlotDistribution bertanggung jawab untuk menampilkan distribusi bobot atau gradien setiap network layer dalam bentuk beberapa grafik terpisah. Tidak seperti SinglePlotDistribution, yang menggabungkan semua network layer dalam satu grafik, MultiPlotDistribution membuat satu grafik untuk setiap network layer dan menampilkannya dalam satu jendela menggunakan scroll area, sehingga pengguna dapat melihat distribusi dari masing-masing network layer dengan lebih jelas.</p>	
<p>Atribut</p>	

```

1 def __init__(self, layer_data: list[tuple[int, np.ndarray]], distribution_mode: str = 'gaussian', parent=None):
2     """
3         This class is just combining Multiple SinglePlotDistribution that each layer get it's own Graph
4     Parameters:
5         - layer_data: List of tuples (layer_index, data)
6             = distribution_mode: Distribution mode for each plot ('gaussian', 'histogram', or 'scatter')
7     """
8     super().__init__(parent)
9     self.setWindowTitle("Multi-Layer Distribution")
10    self.distribution_mode = distribution_mode.lower()
11    layout = QVBoxLayout(self)
12
13    scrollArea = QScrollArea(self)
14    scrollArea.setWidgetResizable(True)
15    layout.addWidget(scrollArea)
16    container = QWidget()
17    self.containerLayout = QVBoxLayout(container)
18    scrollArea.setWidget(container)
19
20    for layer_idx, data in layer_data:
21
22        widget = SinglePlotDistribution(layer_data=[(layer_idx, data)], distribution_mode=self.distribution_mode)
23        self.containerLayout.addWidget(widget)

```

Nama Atribut	Penjelasan
self.distribution_mode	Menentukan metode visualisasi distribusi ('gaussian', 'histogram', atau 'scatter'), serta scrollArea, yang digunakan untuk menampung beberapa grafik dalam satu tampilan dengan kemampuan scrolling agar tidak terlalu penuh di satu layar. Semua grafik individual ditempatkan dalam containerLayout, yang diletakkan di dalam QScrollArea.
Konstruktor	
	<p>Konstruktor <code>__init__</code> menginisialisasi objek dengan daftar data network layer dan mode distribusi tertentu. Setelah memanggil konstruktor induk QWidget, ia mengatur tampilan jendela dengan judul "Multi-Layer Distribution", menyimpan mode distribusi dalam atribut <code>distribution_mode</code>, dan membuat tata letak vertikal QVBoxLayout untuk mengatur elemen-elemen UI. Scroll area (QScrollArea) dibuat agar tampilan bisa bergulir ketika jumlah grafik bertambah. Untuk setiap network layer dalam <code>layer_data</code>, konstruktor membuat objek SinglePlotDistribution dengan data network layer tersebut dan menambahkannya ke dalam layout container (containerLayout), sehingga setiap network layer memiliki grafik distribusi sendiri.</p>
Fungsi/Prosedur	Penjelasan
def WeightDistribution(cls,	Digunakan untuk membuat objek MultiPlotDistribution berdasarkan bobot dari network layer tertentu dalam model graf. Metode ini mengambil daftar indeks network

```

graph_model :
GraphModel,
layer_index_list :
list[int],distribution_
mode : str =
'gaussian', parent =
None) :

```

layer (layer_index_list), mengekstrak bobot dari graph_model.weights, lalu membuat objek MultiPlotDistribution dengan daftar bobot tersebut.

```

def
GradientWeightDistribut
ion(cls, graph_model :
GraphModel,
layer_index_list :
list[int],distribution_
mode : str =
'gaussian', parent =
None) :

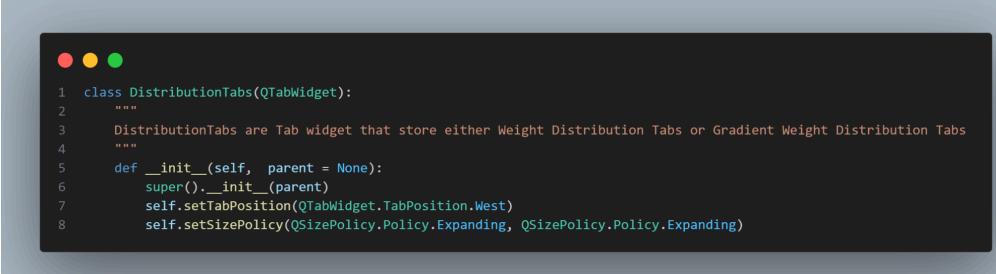
```

Bertanggung jawab untuk membuat objek MultiPlotDistribution berdasarkan distribusi gradien bobot dalam neural network. Metode ini menerima model graf graph_model, daftar indeks network layer yang akan divisualisasikan layer_index_list, mode distribusi distribution_mode, serta parameter opsional parent.

DistributionTabs (distributionTabs.py)

Sebuah TabWidget yang digunakan untuk menyimpan plot dari data distribusi baik Weight maupun Gradien. Terdapat beberapa plot yang dapat dipilih dan dapat diubah ubah mengikuti tab yang ada.

Atribut



```

● ○ ●
1  class DistributionTabs(QTabWidget):
2      """
3          DistributionTabs are Tab widget that store either Weight Distribution Tabs or Gradient Weight Distribution Tabs
4      """
5      def __init__(self, parent = None):
6          super().__init__(parent)
7          self.setTabPosition(QTabWidget.TabPosition.West)
8          self.setSizePolicy(QSizePolicy.Policy.Expanding, QSizePolicy.Policy.Expanding)

```

Konstruktor

Def __init__ tersebut pada dasarnya merupakan konstruksi kosong dan hanya melakukan setup policy saja.

Fungsi/Prosedur	Penjelasan
<pre data-bbox="393 397 796 798">@classmethod def CreateWeightDistribution(cls, graph_model : GraphModel, layer_selected :list[int], parent = None) :</pre>	<p>Sebuah class method yang membuat class DistributionTabs baru yang menambahkan seluruh singlePlotDistribution class dan MultiPlotDistribution class sebagai tab baru.</p> <p>Namun, class method ini berfokus pada distribusi weight.</p>
<pre data-bbox="393 897 796 1326">@classmethod def CreateGradientWeightDistribution(cls, graph_model : GraphModel, layer_selected :list[int], parent = None) :</pre>	<p>Sebuah class method yang membuat class DistributionTabs baru yang menambahkan seluruh singlePlotDistribution class dan MultiPlotDistribution class sebagai tab baru.</p> <p>Namun, class method ini berfokus pada distribusi gradient weight.</p>
ColorHelper (colorHelper.py)	
<p>Kelas ColorHelper bertanggung jawab untuk mengelola warna dalam representasi graf, termasuk mengubah kecerahan warna, mengonversi format warna, serta menghasilkan warna secara acak berdasarkan mode tertentu. Kelas ini menggunakan format warna hexadecimal (#RRGGBB) serta format RGBAW (Red, Green, Blue, Alpha, Width) untuk kebutuhan visualisasi graf.</p>	
Konstruktor	
<p>Konstruktor <code>__init__</code> tidak didefinisikan dalam kelas ini karena seluruh metode yang ada bersifat stateless dan digunakan secara langsung tanpa memerlukan inisialisasi objek ColorHelper.</p>	

Fungsi/Prosedur	Penjelasan
<pre>def adjust_color(hex_color: str, factor: float) -> str:</pre>	Digunakan untuk menyesuaikan tingkat kecerahan suatu warna dengan mengalikan setiap komponen RGB dengan faktor tertentu, lalu mengembalikan warna dalam format hexadecimal yang baru.
<pre>def hex_to_rgba_tuple(hex_color: str, width: float = = 1.0):</pre>	Mengonversi warna hexadecimal menjadi tuple RGBAW, yang mencakup nilai merah, hijau, biru, alpha (jika tersedia), dan lebar garis. Jika format warna tidak valid, metode ini akan mengembalikan ValueError.
<pre>def create_lines_array(colors: list[str], default_width: float = 1.0):</pre>	Mengubah daftar warna dalam format hexadecimal menjadi numpy array dengan format data ('red', 'green', 'blue', 'alpha', 'width'). Hal ini digunakan untuk menerjemahkan warna garis dalam graf agar dapat diolah lebih efisien dalam visualisasi.
<pre>def generate_colors(num: int, mode: str = 'light') -> list[str]:</pre>	Bertanggung jawab untuk menghasilkan daftar warna secara acak dengan jumlah tertentu (num).
GraphConfig (graphConfig.py)	
Kelas GraphConfig bertanggung jawab untuk menyimpan konfigurasi global yang digunakan dalam visualisasi graf, termasuk ukuran teks, warna, ukuran node, jarak antar-layer, dan berbagai parameter lain yang mengatur tampilan graf secara keseluruhan.	
Atribut	

```

1  class GraphConfig:
2      TEXT_SIZE = 12
3      TEXT_COLOR = "#ffffff"
4      BACKGROUND_COLOR = "#000000"
5      NODE_SIZE = 100
6      LAYER_SPACING = 5
7      LAYER_Y_RANGE = (0, 120)
8      LINE_SIZE = 2
9      WEIGHT_TABLE_ROWS = 10
10     MARGIN_RIGHT = 200
11
12     class Colors:
13         class Light:
14             red = "#ff4d4d"
15             green = "#4dff88"
16             blue = "#00ffaa"
17             yellow = "#ffea00"
18             cyan = "#00e0e0"
19             magenta = "#ff00ff"
20         class Dark:
21             red = "#cc0000"
22             green = "#009933"
23             blue = "#007f7f"
24             yellow = "#cc9900"
25             cyan = "#007777"
26             magenta = "#cc00cc"

```

Nama Atribut	Penjelasan
TEXT_SIZE	Menentukan ukuran teks yang digunakan dalam elemen graf
TEXT_COLOR	Menentukan warna teks dalam format hexadecimal (#ffffff untuk putih)
BACKGROUND_COLOR	Mengatur warna latar belakang graf, yang dalam konfigurasi ini diatur menjadi hitam (#000000).
NODE_SIZE	Menentukan ukuran node dalam visualisasi, sementara LAYER_SPACING mengontrol jarak antar network layer dalam graf.
LAYER_Y_RANGE	LAYER_Y_RANGE adalah tuple yang mendefinisikan rentang posisi vertikal node dalam setiap network layer.
LINE_SIZE	Menentukan ketebalan garis yang menghubungkan node
WEIGHT_TABLE_ROWS	Mengontrol jumlah baris dalam tabel bobot jika ada fitur untuk menampilkan bobot dalam bentuk tabel.
MARGIN_RIGHT	Mengatur margin kanan dalam tampilan graf untuk memberi ruang tambahan jika diperlukan.
Kelas di dalam kelas	

Di dalam kelas GraphConfig, terdapat kelas Colors yang menyimpan kumpulan warna dalam dua mode utama, yaitu Light, yang berisi warna-warna cerah untuk tampilan graf dengan latar terang seperti merah terang, hijau terang, biru toska, kuning, cyan, dan magenta, serta Dark, yang menyediakan warna lebih redup untuk tampilan dengan latar gelap seperti merah gelap, hijau tua, biru gelap, kuning gelap, cyan tua, dan magenta gelap, sehingga konfigurasi ini memungkinkan penyesuaian tampilan graf secara fleksibel tanpa perlu mengubah kode utama, menjadikannya lebih modular dan mudah dikelola.

2.1.2 Forward Propagation

Forward propagation adalah tahap pertama dalam proses perhitungan *feed forward neural network*, di mana data *input* diproses melalui setiap *layer* sampai menghasilkan output akhir berupa *output layer*. Proses ini sangat penting karena menentukan bagaimana informasi berupa bobot dan bias mengalir melalui *network* dan berpengaruh terhadap performa model dalam melakukan prediksi. Dalam *forward propagation*, *input* yang diberikan akan dikalikan dengan bobot masing-masing neuron, ditambahkan bias, dan kemudian dilanjutkan ke *layer* berikutnya melalui fungsi aktivasi untuk mendapatkan nilai output dari neuron tersebut. Hasil perhitungan ini kemudian diteruskan ke *layer* berikutnya hingga mencapai *layer output*. Berikut adalah hasil implementasi dari proses *forward propagation*.



```

1  def __init__(self, jumlah_neuron, fungsi_aktivasi, fungsi_loss, inisialisasi_bobot, verbose=0, lower_bound=-1.0, upper_bound=1.0, mean=0.0, std=1.0, seed=0):
2:
3      self.verbose = verbose
4      self.jumlah_neuron = jumlah_neuron
5      self.jumlah_layer = len(jumlah_neuron) - 1
6      self.fungsi_aktivasi = [
7          ActivationFunction([method]).get_activation_function(method)
8          for method in fungsi_aktivasi
9      ]
10
11
12      self.fungsi_loss_class = LossFunction(fungsi_loss)
13      self.fungsi_loss = self.fungsi_loss_class.get_lost_function()
14      self.fungsi_loss_str = fungsi_loss
15
16      self.inisialisasi_bobot_str = inisialisasi_bobot
17      self.inisialisasi_bobot = WeightInitiation(
18          inisialisasi_bobot, self.jumlah_layer, jumlah_neuron
19      )
20
21
22      self.init_bobot()
23      self.fungsi_aktivasi_str = fungsi_aktivasi
24
25
26  def init_bobot(self):
27      self.bobot = self.inisialisasi_bobot.init_weights()
28      # for i in range(self.jumlah_layer):
29      #     print(f"Shape bobot layer {i}: {self.bobot[i].shape}")
30
31
32  def forward(self, X):
33      hasil = [X]
34      for i in range(self.jumlah_layer):
35          X_with_bias = np.hstack([np.ones((hasil[-1].shape[0], 1)), hasil[-1]])
36          Z = np.dot(X_with_bias, self.bobot[i])
37          hasil.append(self.fungsi_aktivasi[i](Z))
38
39
40  return hasil

```

Dalam kode tersebut, sebelum melakukan perhitungan untuk *forward propagation*, bobot diinisialisasi terlebih dahulu melalui metode `init_bobot()`, yang memanfaatkan objek `WeightInitiation` untuk menghasilkan bobot awal berdasarkan metode inisialisasi yang dipilih. Selain itu, atribut-atribut penting seperti jumlah neuron di setiap *layer*, fungsi aktivasi, dan fungsi loss juga diatur saat objek dibuat supaya dapat digunakan dalam tahapan selanjutnya.

Setelah semua komponen diinisialisasi, proses *forward propagation* dimulai. Langkah pertama dalam *forward propagation* adalah menyimpan *input* awal dalam sebuah *list* yang disimpan dalam variabel bernama `hasil`. *List* ini nantinya yang akan digunakan untuk men-track nilai *output* yang dihasilkan di setiap *layer*. Dalam kode di atas direpresentasikan dengan:

`hasil = [X]`

Kemudian, perhitungan dilakukan secara iteratif untuk setiap *layer* sebanyak jumlah layer yang sudah ditentukan dalam *network* menggunakan *for loop*. Pada setiap langkah, dari

layer sebelumnya akan ditambahkan dengan angka bias, yaitu angka tambahan yang membantu model belajar lebih baik. Bias ini memungkinkan hasil di setiap *layer* menghasilkan output yang lebih fleksibel dan tidak selalu harus bergantung sepenuhnya pada *input*, sehingga bisa lebih mudah mengenali pola dalam data.

Dalam kode di atas, bias ditambahkan dengan cara menggabungkan input dan bias menggunakan np.hstack(), seperti berikut:

```
X_with_bias = np.hstack([np.ones((hasil[-1].shape[0], 1)), hasil[-1]])
```

Setelah bias ditambahkan, input yang sudah ditambahkan biasnya dikalikan dengan bobot yang sesuai untuk *layer* saat ini menggunakan operasi perkalian matriks. Perhitungan ini didasarkan pada rumus mencari nilai *output* dalam *neural network*:

$$Z = W \cdot X + b$$

Dalam kode di atas, cara tersebut direpresentasikan dengan menggunakan operasi np.dot sebagai berikut:

```
Z = np.dot(X_with_bias, self.bobot[i])
```

Nilai hasil perhitungan yang disimpan dalam variabel Z tersebut, yang dinamakan sebagai *pre-activation value*, nilai ini nantinya akan dimasukkan ke dalam fungsi aktivasi yang telah ditentukan untuk *layer* tersebut. Fungsi aktivasi ini berperan untuk membuat *neural network* dapat mengenali pola dari data yang lebih rumit, bukan hanya sekedar hubungan linier sederhana. Setelah fungsi aktivasi diterapkan, hasilnya disimpan kembali ke dalam list hasil untuk digunakan sebagai *input* pada *layer* berikutnya.

```
hasil.append(self.fungsi_aktivasi[i](z))
```

Proses ini berlanjut sampai semua *layer* telah diproses dan nilai akhirnya didapatkan sebagai *layer output* yang dikembalikan oleh fungsi ini dalam bentuk “hasil”. Hasil dari *forward propagation* ini nantinya akan digunakan dalam tahap *backpropagation* untuk memperbaiki atau memperbarui bobot berdasarkan perbedaan antara *output* yang dihasilkan dan nilai target yang diharapkan.

2.1.3 Backward Propagation dan Weight Update

a. Backward Propagation

Backward propagation adalah tahap di mana *neural network* menghitung error dan mengirimkan *error* ke lapisan sebelumnya dalam *network* untuk menyesuaikan bobot dengan memperbarui nilai bobot. Proses ini memungkinkan model untuk belajar dari kesalahannya dan meningkatkan akurasinya. Inti proses dari *backward propagation* adalah menghitung gradien *error* berdasarkan perbedaan antara *output* yang dihasilkan dan nilai target, lalu mengirimkan secara mundur dari *output layer* ke *hidden layer* hingga ke *input layer*. Berikut adalah hasil implementasi proses *backward propagation*.



```

1 def backward(self, hasil: np.ndarray, y: np.ndarray):
2     """
3         Melakukan backpropagation pada model.
4     """
5     deltas = [None] * self.jumlah_layer
6     loss_derivative = self.fungsi_loss_class.get_loss_derivative()
7     activation_derivative = ActivationFunction(
8         self.fungsi_aktivasi_str
9     ).get_activation_derivative(self.fungsi_aktivasi_str[-1])
10
11    if self.fungsi_aktivasi_str[-1] == "Softmax":
12        loss_grad = loss_derivative(hasil[-1], y)
13        deltas[-1] = activation_derivative(hasil[-1], loss_grad)
14    else:
15        deltas[-1] = loss_derivative(hasil[-1], y) * activation_derivative(
16            hasil[-1]
17        )
18
19    for i in range(self.jumlah_layer - 2, -1, -1):
20        activation_derivative_i = ActivationFunction(
21            self.fungsi_aktivasi_str
22        ).get_activation_derivative(self.fungsi_aktivasi_str[i])
23
24        upstream_gradient = deltas[i + 1] @ self.bobot[i + 1][1:].T
25
26        if self.fungsi_aktivasi_str[i] == "Softmax":
27            # Compute delta = (upstream_gradient) @ Jacobian
28            deltas[i] = activation_derivative_i(
29                hasil[i + 1], grad_loss=upstream_gradient
30            )
31        else:
32            deltas[i] = upstream_gradient * activation_derivative_i(hasil[i + 1])
33
34    gradients = []
35    for i in range(self.jumlah_layer):
36        X_with_bias = np.hstack([np.ones((hasil[i].shape[0], 1)), hasil[i]])
37
38        weight_gradient = (X_with_bias.T @ deltas[i]) / hasil[i].shape[0]
39
40        # L1
41        if self.l1_lambda > 0:
42            weight_gradient += self.l1_lambda * np.sign(self.bobot[i])
43        # L2
44        if self.l2_lambda > 0:
45            weight_gradient += self.l2_lambda * self.bobot[i]
46
47        # Update gradient
48        gradients.append(weight_gradient)
49
50    return gradients

```

Backward propagation dimulai dengan membuat *list* untuk menyimpan gradien *error* di setiap *layer*. Dalam kode di atas, *list* ini disimpan dalam variabel bernama *deltas* seperti berikut:

```
deltas = [None] * self.jumlah_layer
```

Kemudian, proses dilanjutkan dengan menghitung *error* pada *output layer* terlebih dahulu. Error pada *layer output* dihitung terlebih dahulu dengan mengalikan turunan fungsi loss dengan turunan fungsi aktivasi pada lapisan output seperti sebagai berikut.

```
deltas[-1] = loss_derivative(hasil[-1], y) *
activation_derivative(hasil[-1])
```

Selanjutnya, *error* ini dikirimkan secara mundur ke *layer-layer* sebelumnya (*hidden layer* dan seterusnya) menggunakan loop yang berjalan secara terbalik. Pada setiap langkah (*layer*), error dihitung untuk setiap neuron dengan cara mengalikan error dari *layer* setelahnya(*deltas[i+1]*) dengan bobot yang menghubungkannya dengan *layer* saat ini, kemudian dikalikan dengan turunan fungsi aktivasi dari *layer* tersebut.

```
for i in range(self.jumlah_layer - 2, -1, -1):
    activation_derivative_i = ActivationFunction(
        self.fungsi_aktivasi_str
    ).get_activation_derivative(self.fungsi_aktivasi_str[i])
    deltas[i] = (
        deltas[i + 1] @ self.bobot[i + 1][1:].T
    ) * activation_derivative_i(hasil[i + 1])
```

deltas[i] pada kode tersebut akan menyimpan *error* dari seluruh neuron pada tiap *layer*-nya.

Setelah semua *error* dihitung, gradien yang diperlukan untuk memperbarui bobot akan dikumpulkan dalam sebuah list yang disimpan dalam variabel bernama *gradients*.

```
gradients = []
```

```

for i in range(self.jumlah_layer):
    X_with_bias = np.hstack([np.ones((hasil[i].shape[0], 1)), hasil[i]])
    weight_gradient = (X_with_bias.T @ deltas[i]) / hasil[i].shape[0]
    # L1
    if self.l1_lambda > 0:
        weight_gradient += self.l1_lambda * np.sign(self.bobot[i])
    # L2
    if self.l2_lambda > 0:
        weight_gradient += self.l2_lambda * self.bobot[i]
    # Update gradien
    gradients.append(weight_gradient)

```

Gradien bobot dihitung sebagai rata-rata dari perubahan bobot yang diperlukan untuk mengurangi error. Perhitungan dasar dilakukan dengan mengalikan input (yang telah ditambahkan bias) dengan error dari lapisan tersebut (`deltas[i]`) menggunakan rumus:

$X_{\text{with_bias}}.T @ \text{deltas}[i]$

Hasil ini kemudian dibagi dengan jumlah sampel untuk mendapatkan rata-rata gradien.

Setelah gradien dasar diperoleh, regularisasi ditambahkan jika diaktifkan:

- Jika `l1_lambda > 0`, maka gradien bobot ditambahkan dengan penalti L1 yang dihitung menggunakan `np.sign(self.bobot[i])`. Penalti L1 ini mendorong bobot untuk menjadi lebih spars (mendekati nol), yang efektif dalam seleksi fitur. Hasil penalti kemudian dibatasi (clipped) agar tidak terlalu besar menggunakan `np.clip` dalam rentang -0.1 hingga 0.1, sebelum ditambahkan ke gradien bobot.
- Jika `l2_lambda > 0`, maka gradien bobot ditambahkan dengan penalti L2 yang dihitung langsung berdasarkan nilai `self.bobot[i]`. Penalti L2 ini membantu menjaga bobot tetap kecil dan stabil. Sama seperti L1, hasil penalti juga dibatasi menggunakan `np.clip` agar berada dalam rentang -0.1 hingga 0.1, sebelum ditambahkan ke gradien bobot.

Gradien akhir yang sudah mencakup regularisasi disimpan dalam `gradients`, dan akan digunakan dalam proses pembaruan bobot agar model dapat belajar secara efektif dan menghindari overfitting.

b. Weight Update

Setelah *backward propagation* menghitung gradien bobot, langkah berikutnya adalah memperbarui bobot menggunakan metode optimasi, seperti *Gradient Descent*. Tujuan dari tahap ini adalah menyesuaikan bobot agar output prediksi semakin mendekati nilai target dengan meminimalkan error. Berikut adalah hasil implementasi untuk *update* bobot.



Dalam kode ini, bobot diperbarui dengan cara mengurangkan bobot lama dengan hasil perkalian antara gradien dan learning rate seperti rumus di bawah ini.

$$W_{new} = W_{old} - \alpha \left(\frac{\partial \mathcal{L}}{\partial W_{old}} \right)$$

Learning rate adalah faktor skala yang menentukan seberapa besar perubahan bobot di setiap iterasi pelatihan. Jika learning rate terlalu kecil, model akan belajar dengan lambat karena *step size* yang terlalu kecil, sedangkan jika terlalu besar, model bisa melompat-lompat dan tidak konvergen dengan baik atau bisa dikatakan sebagai *overshoot*.

2.2 Hasil Pengujian

2.2.1 Pengaruh depth dan width

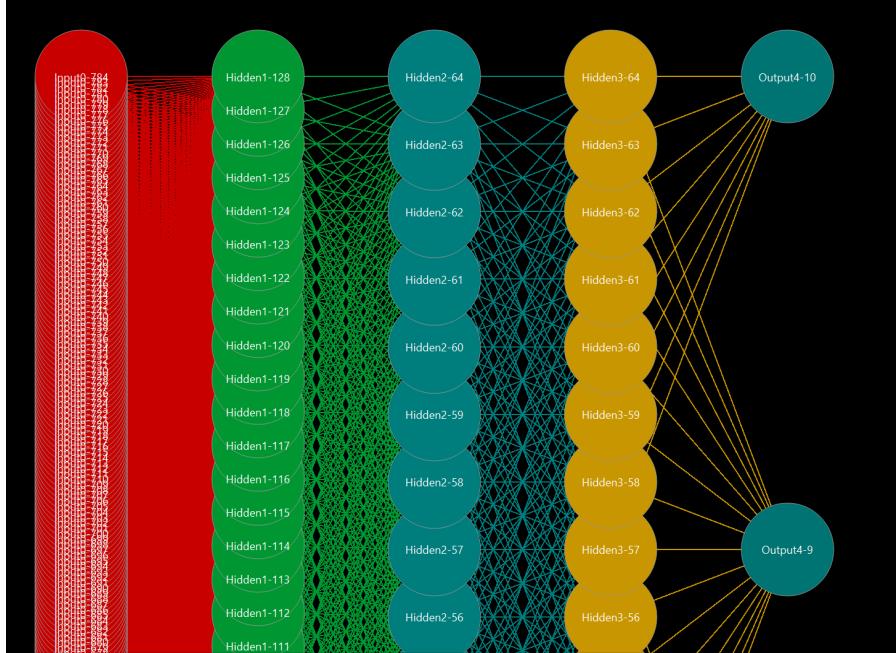
2.2.1.1 Depth

- a. Depth = 3 hidden layer dengan width = 128, 64, 64
 - Prediction Score

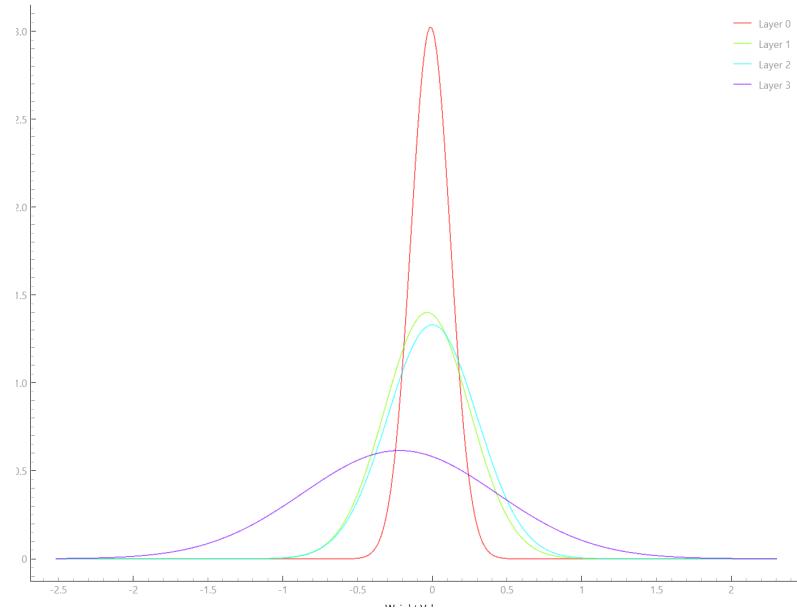
```
[██████████] 100%
Loss: 0.353082
Training time: 97.23 seconds
Model berhasil disimpan ke ffnn_model.pkl
```

Model F1-Score: 0.8803

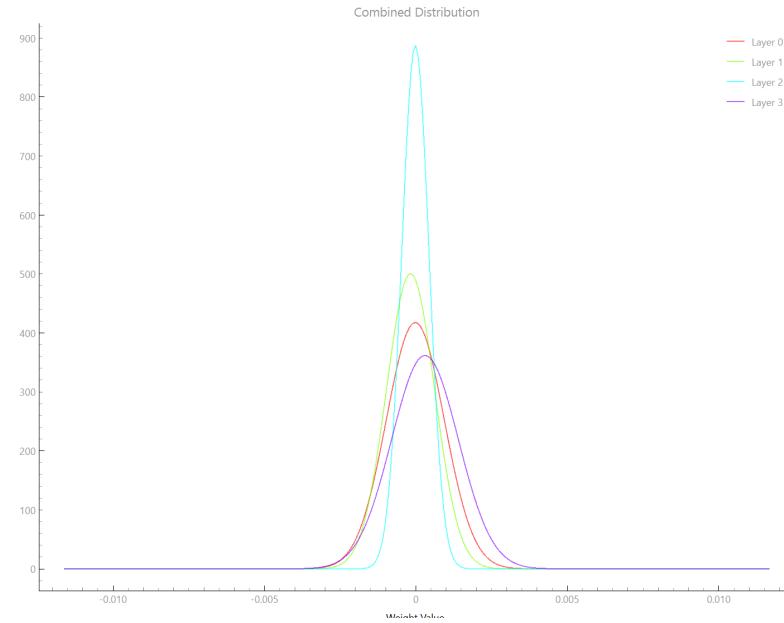
- Neuron Model



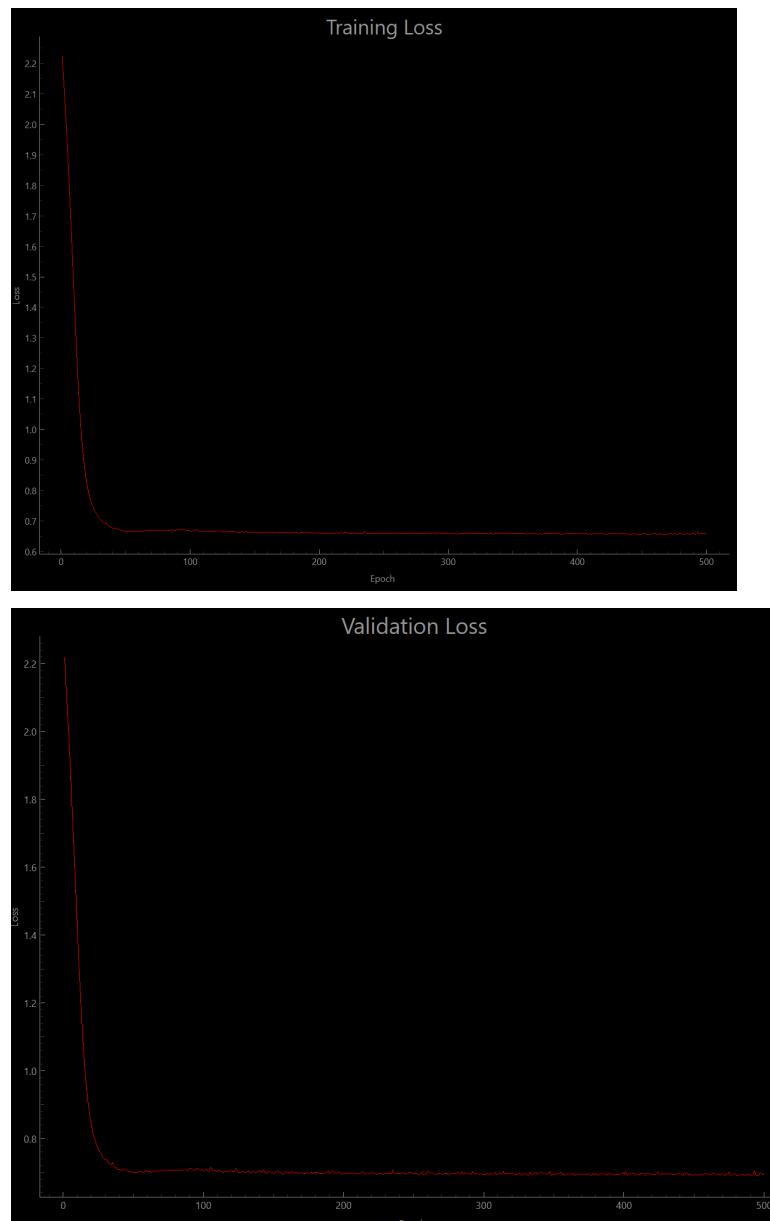
● Weight Distribution



● Gradient Weight Distribution



● Perbandingan Training dengan Validation



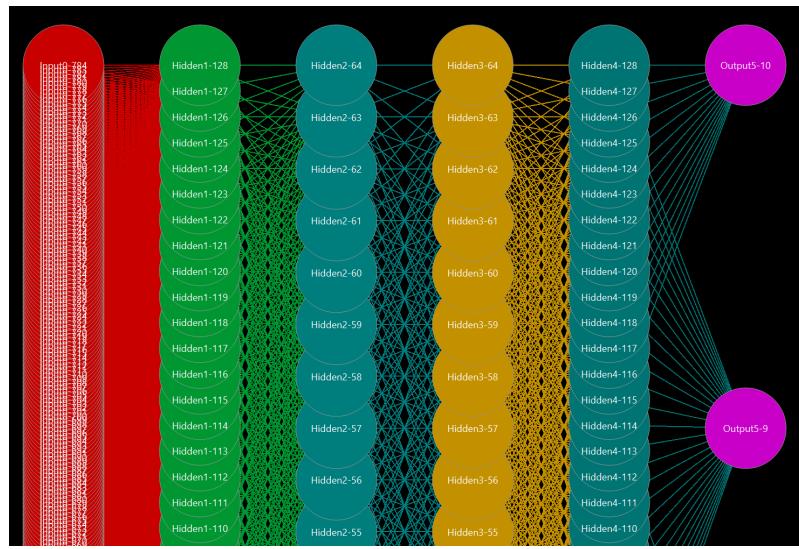
b. Depth = 4 layer dengan width = 128, 64, 64, 128

- **Prediction Score**

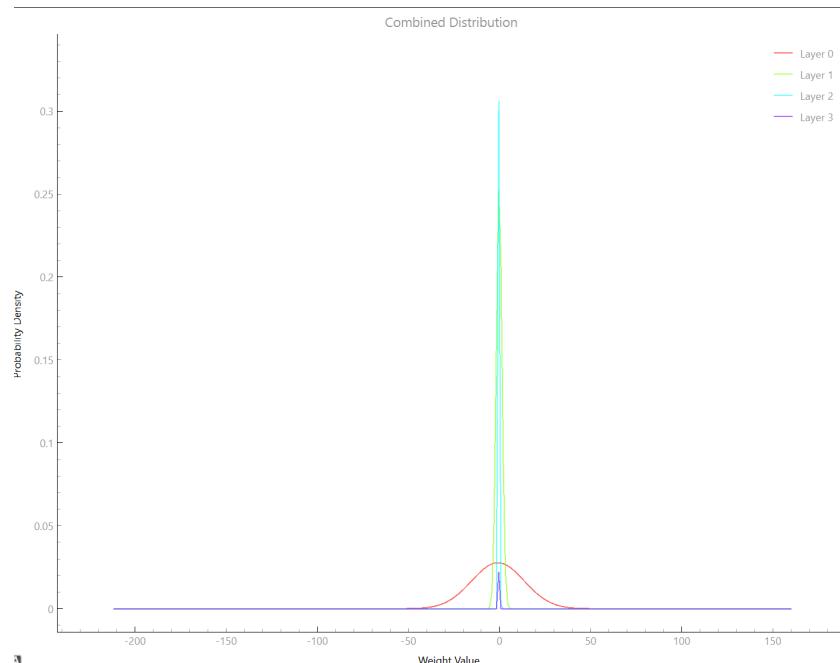
```
[██████████] - 500/500 - 100.0% - 247.67
Loss: 1.689697
Training time: 247.67 seconds
```

Model F1-Score: 0.2436

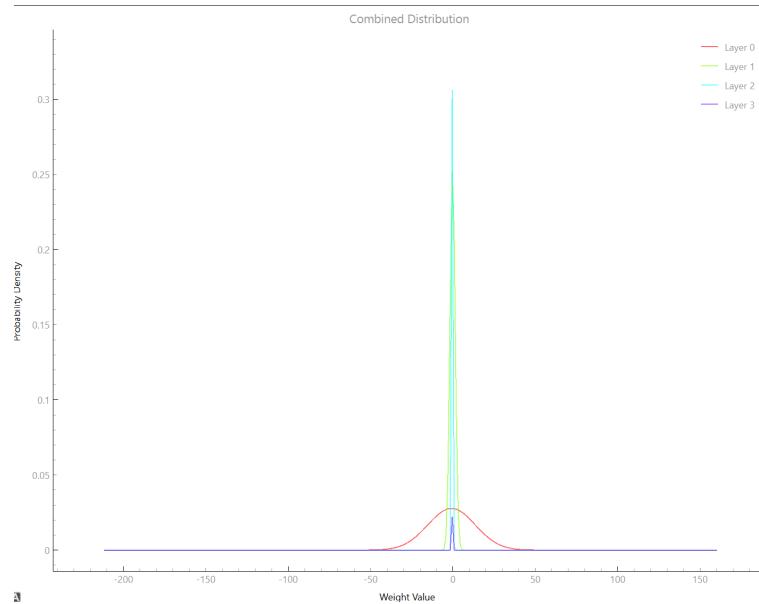
- **Neuron Model**



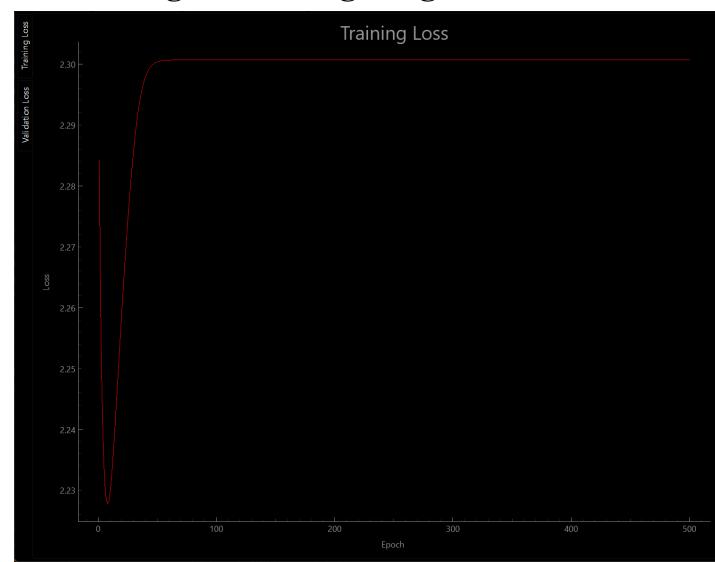
- **Weight Distribution**

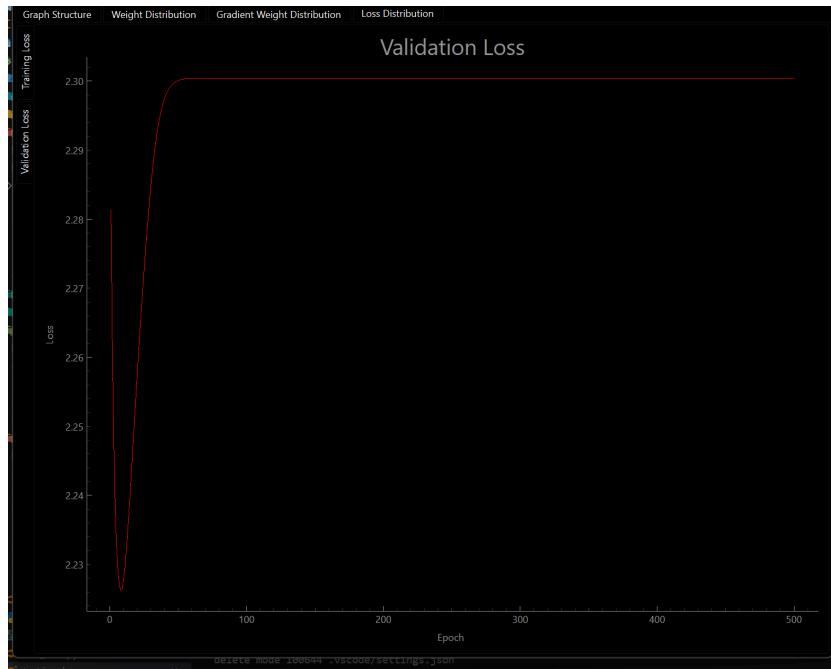


- **Gradient Weight Distribution**



- **Perbandingan Training dengan Validation**





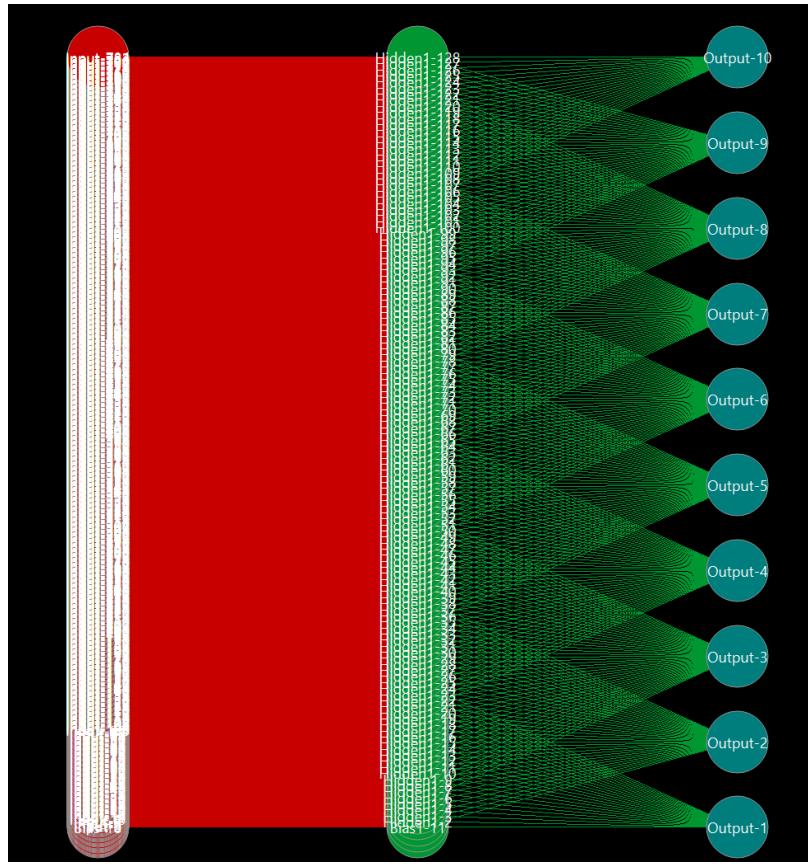
c. Depth = 1 layer dengan width = 128

- Prediction Score

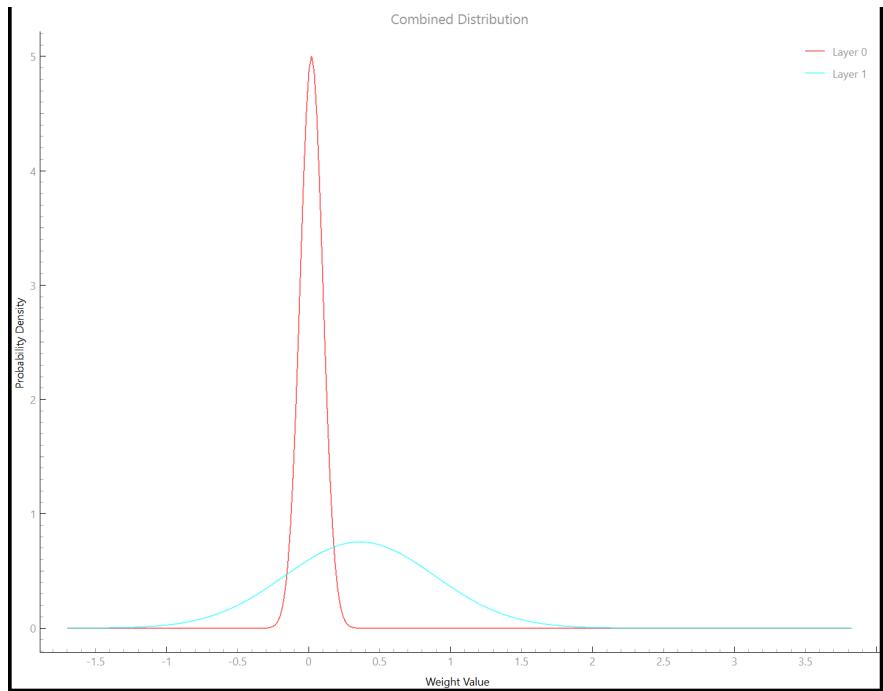
```
[██████████] - 500/500
Loss: 0.066663
Training time: 62.90 seconds
```

Model F1-Score: 0.9193

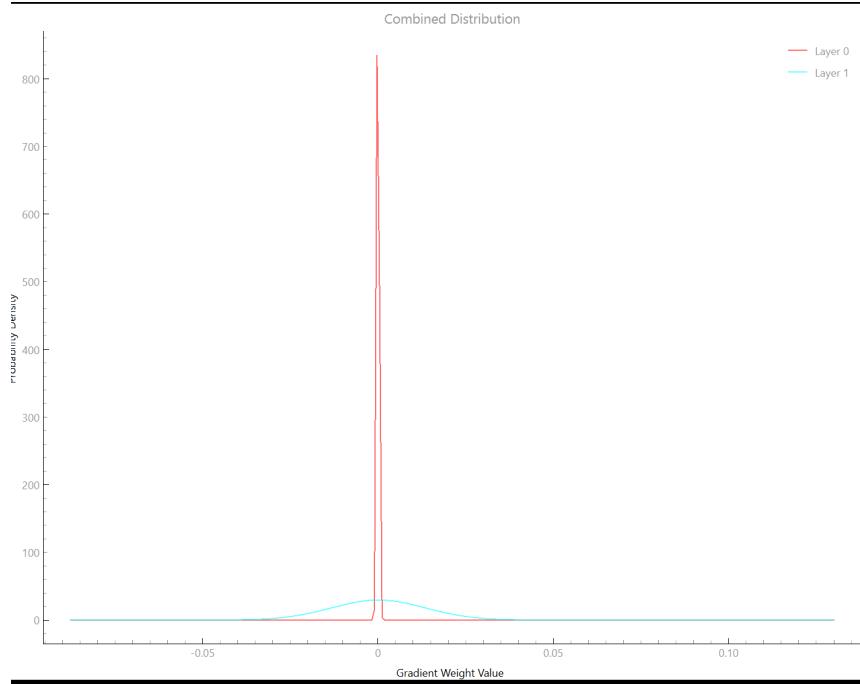
- Neuron Model



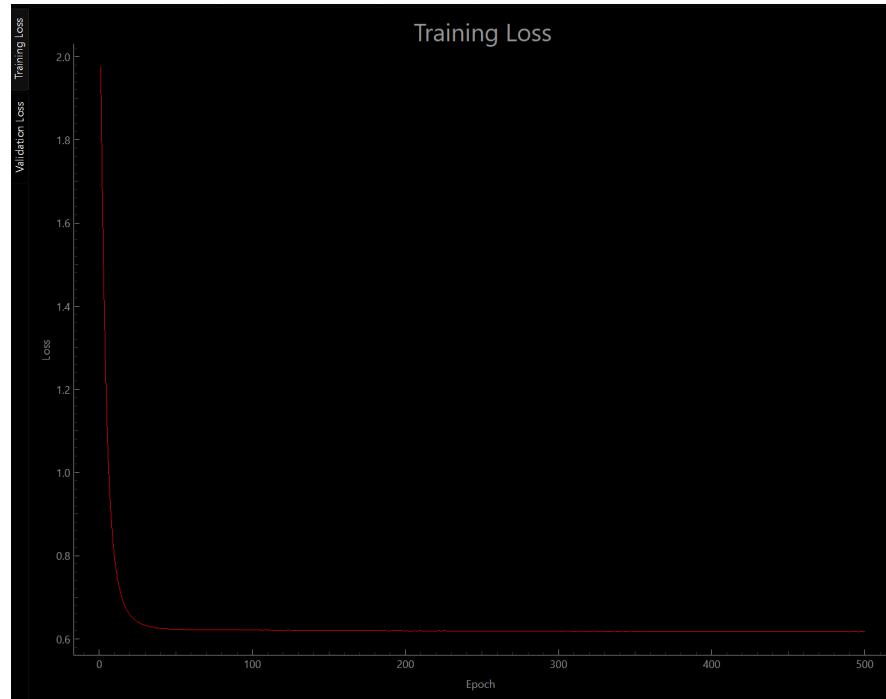
- **Weight Distribution**



- **Gradient Weight Distribution**



- **Perbandingan Training dengan Validation**





2.2.1.2 Width

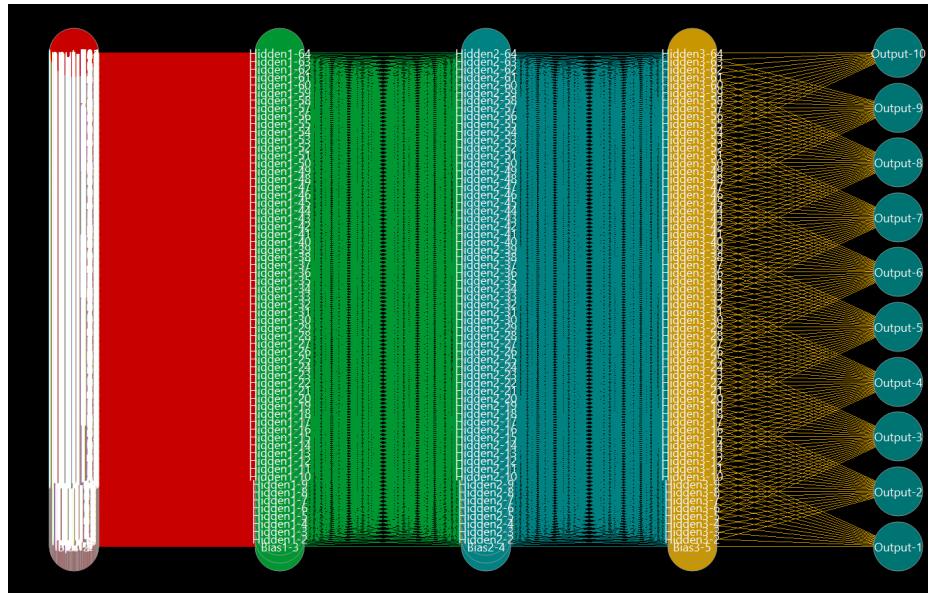
- a. Width = 64, 64, 64 neuron dengan depth = 3

- Prediction Score

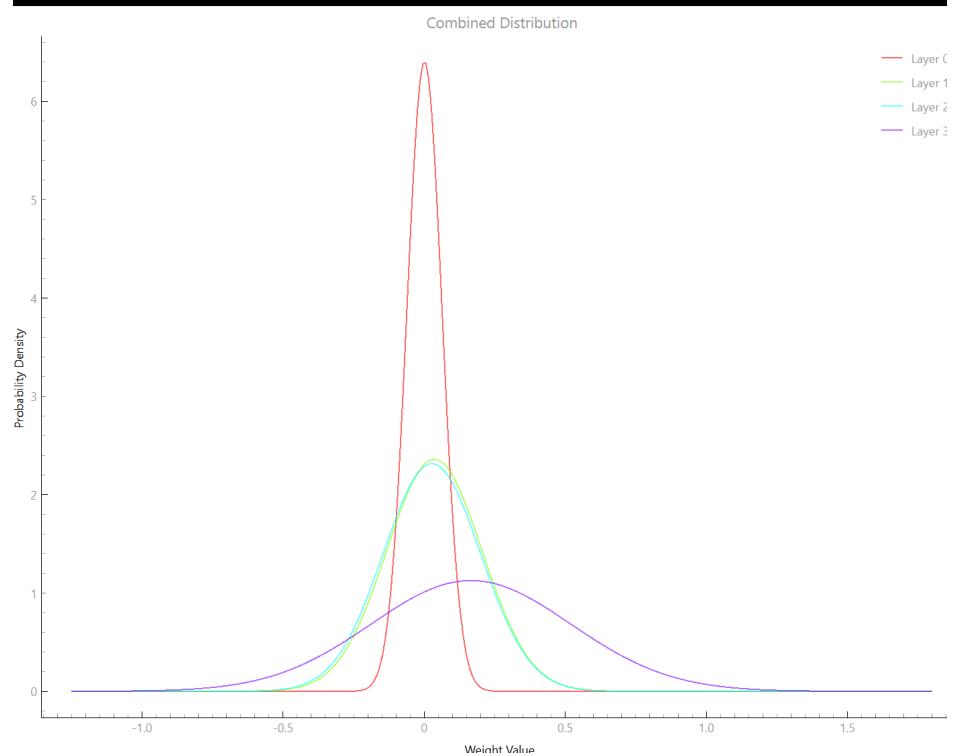
```
[  
Loss: 0.000506  
Training time: 63.46 seconds]
```

Model F1-Score: 0.9309

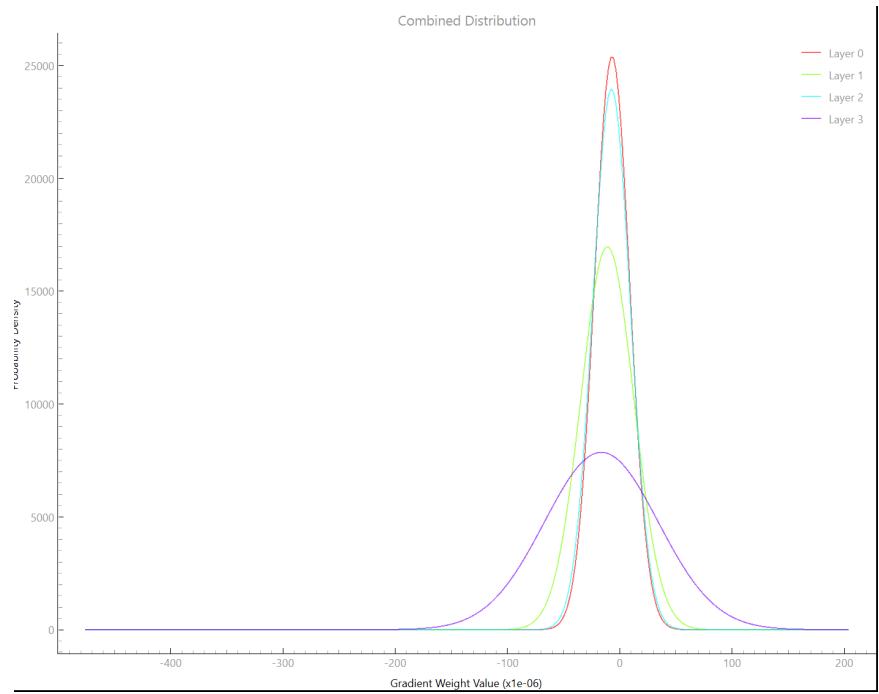
- Neuron Model



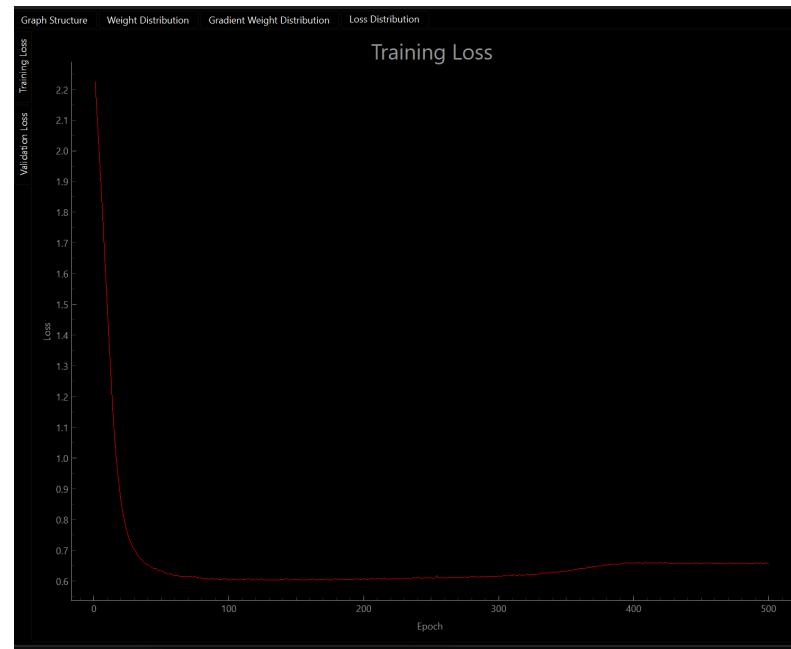
● Weight Distribution

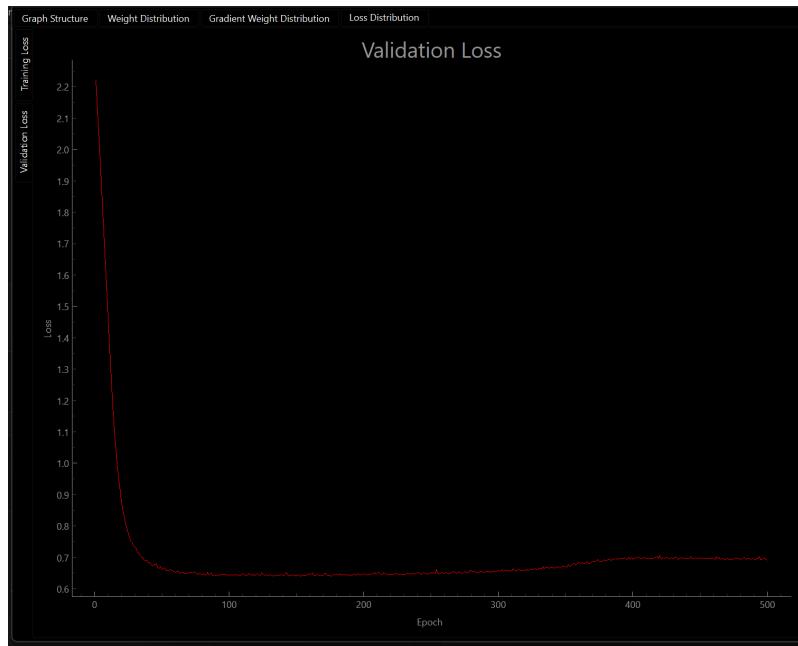


● Gradient Weight Distribution



● Perbandingan Training dengan Validation





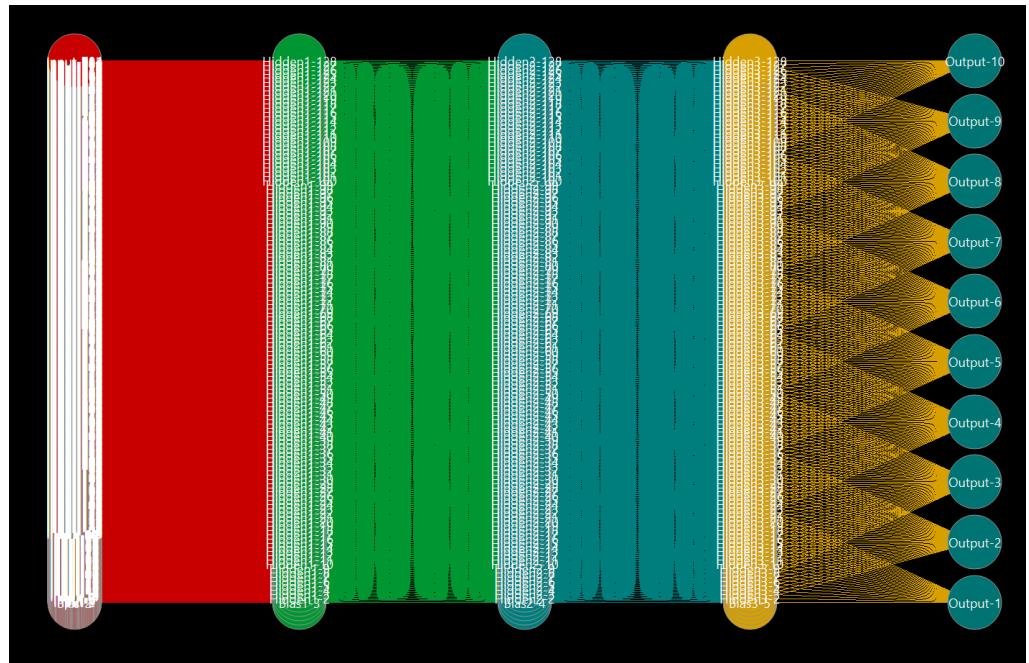
b. Width = 128, 128, 128 neuron dengan depth = 3

- Prediction Score

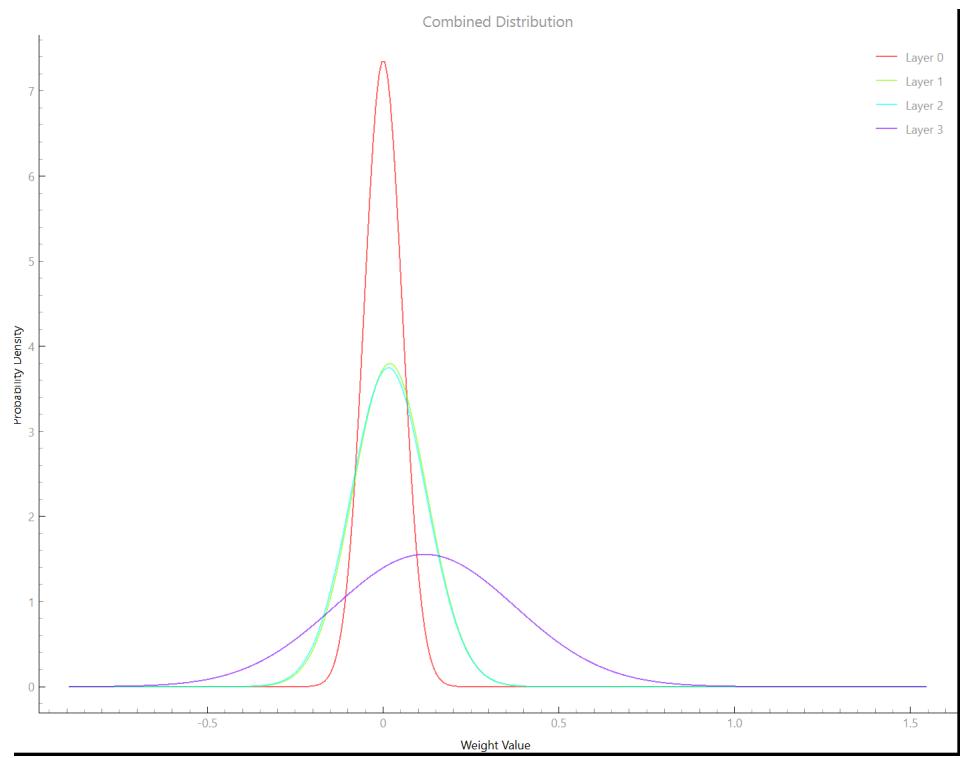
```
[██████████] -  
Loss: 0.000928  
Training time: 133.74 seconds
```

Model F1-Score: 0.9319

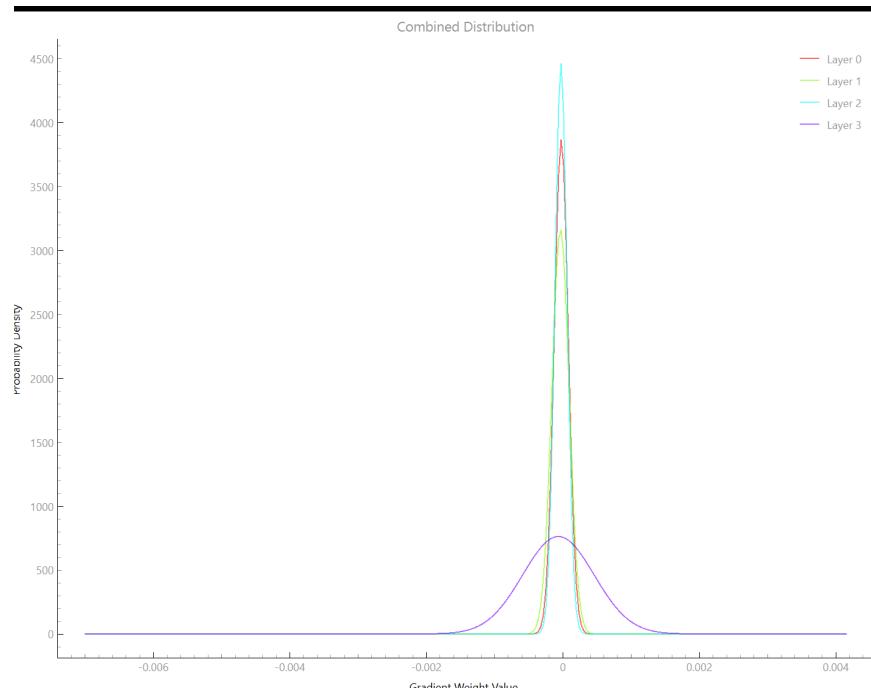
- Neuron Model



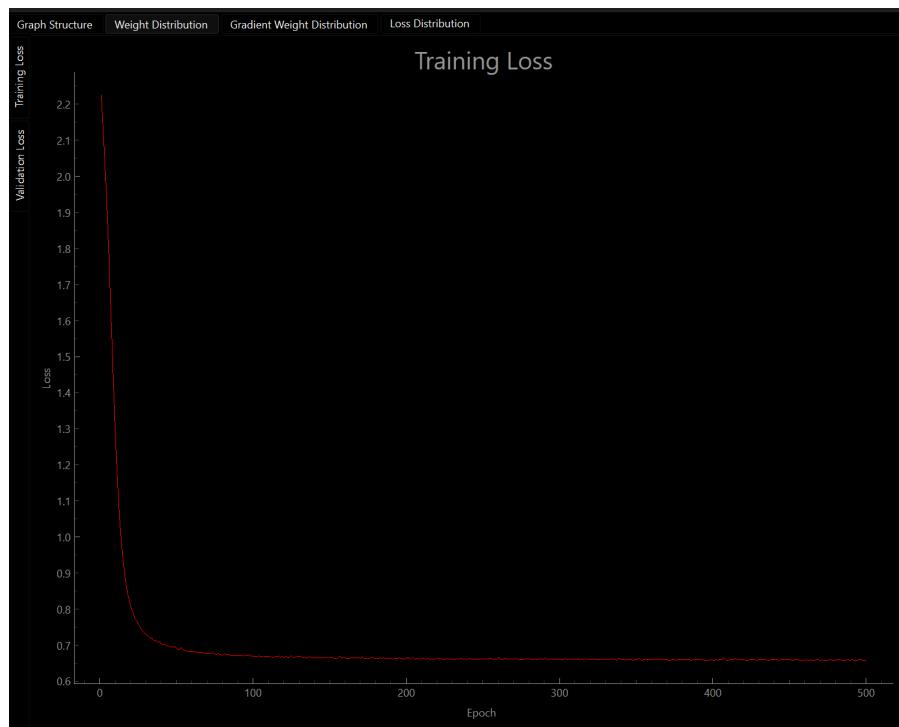
- **Weight Distribution**

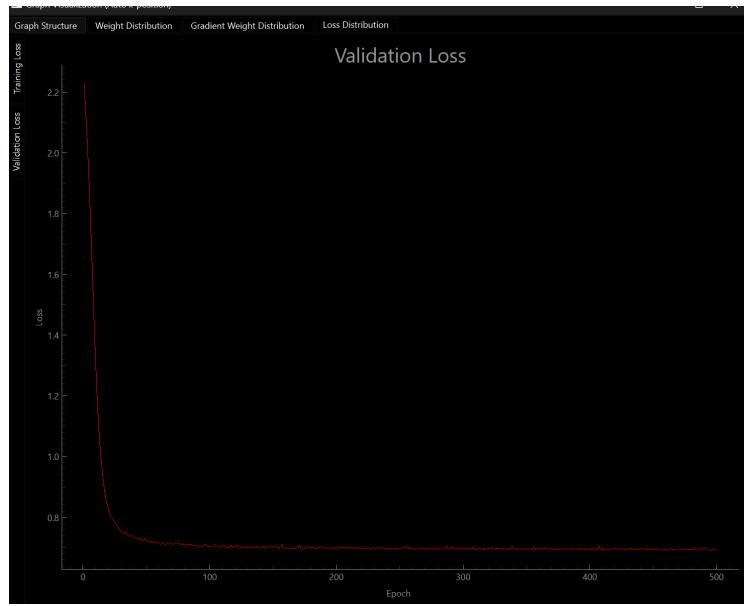


- **Gradient Weight Distribution**



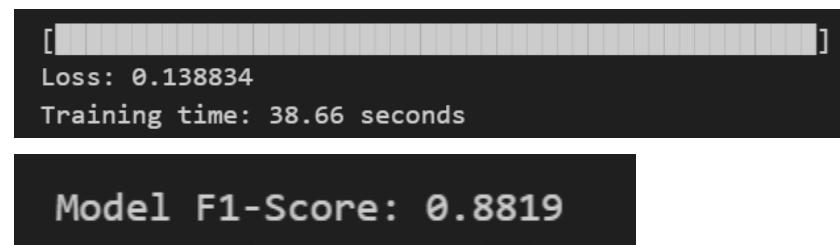
- **Perbandingan Training dengan Validation**



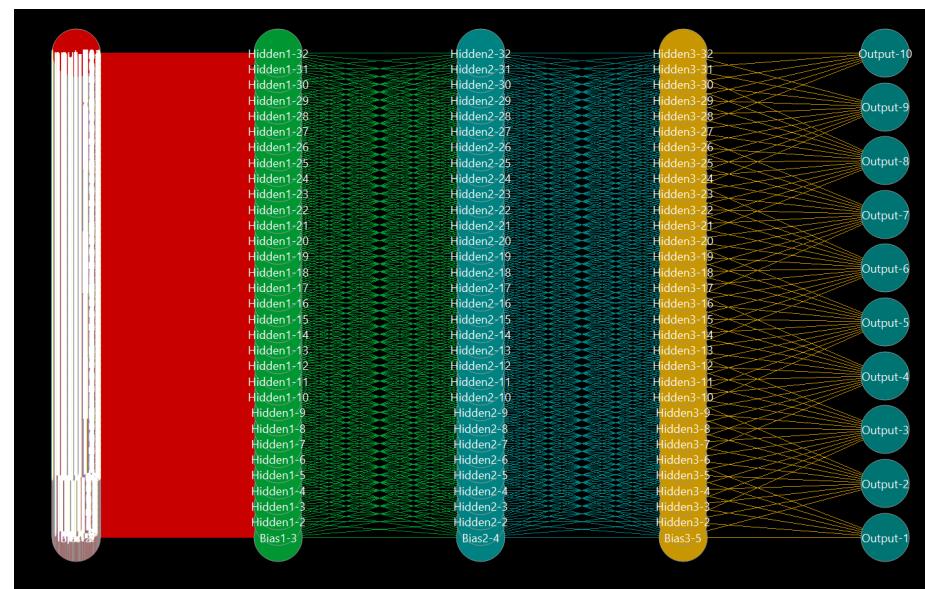


c. Width = 32, 32,32 neuron dengan depth = 3

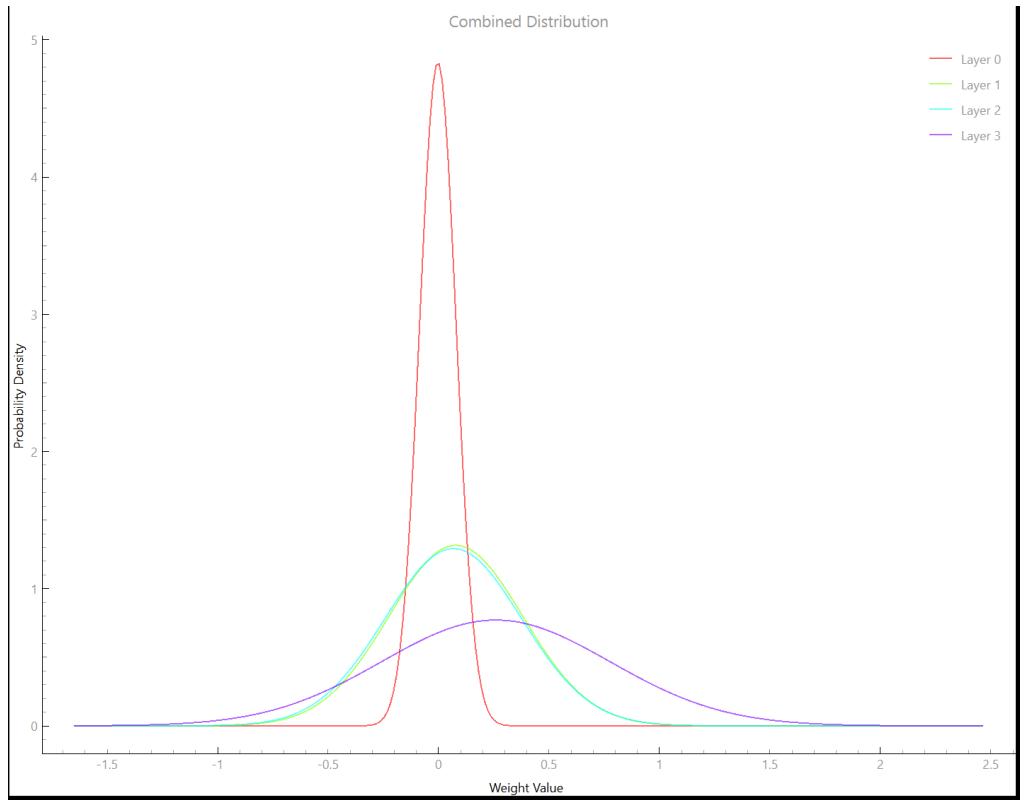
- Prediction Score



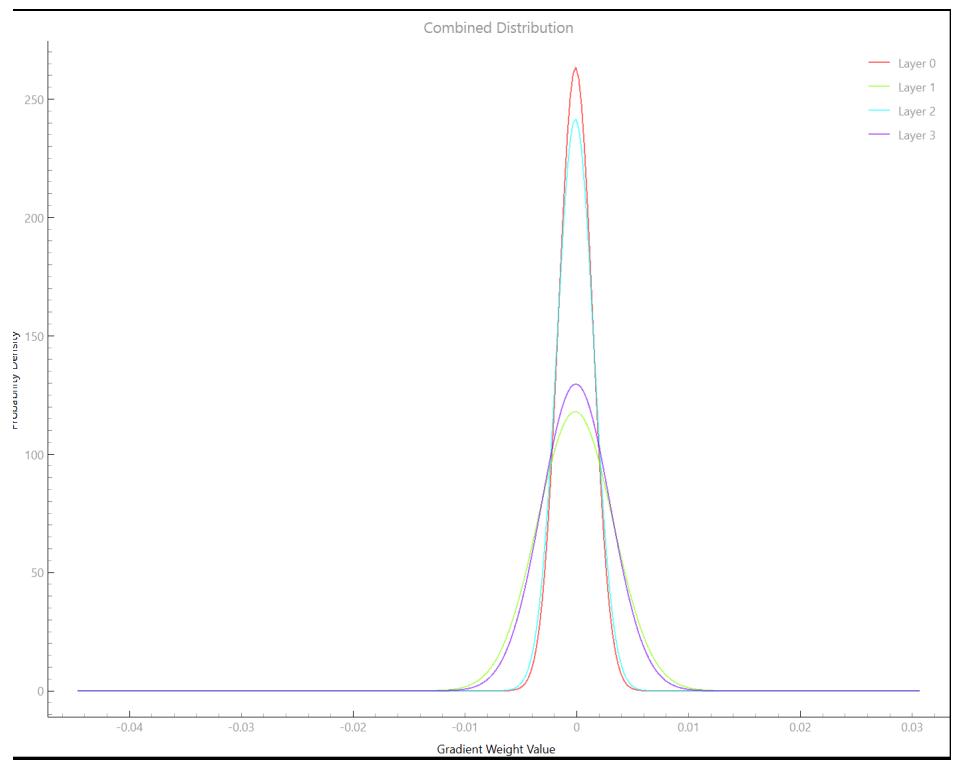
- Neuron Model



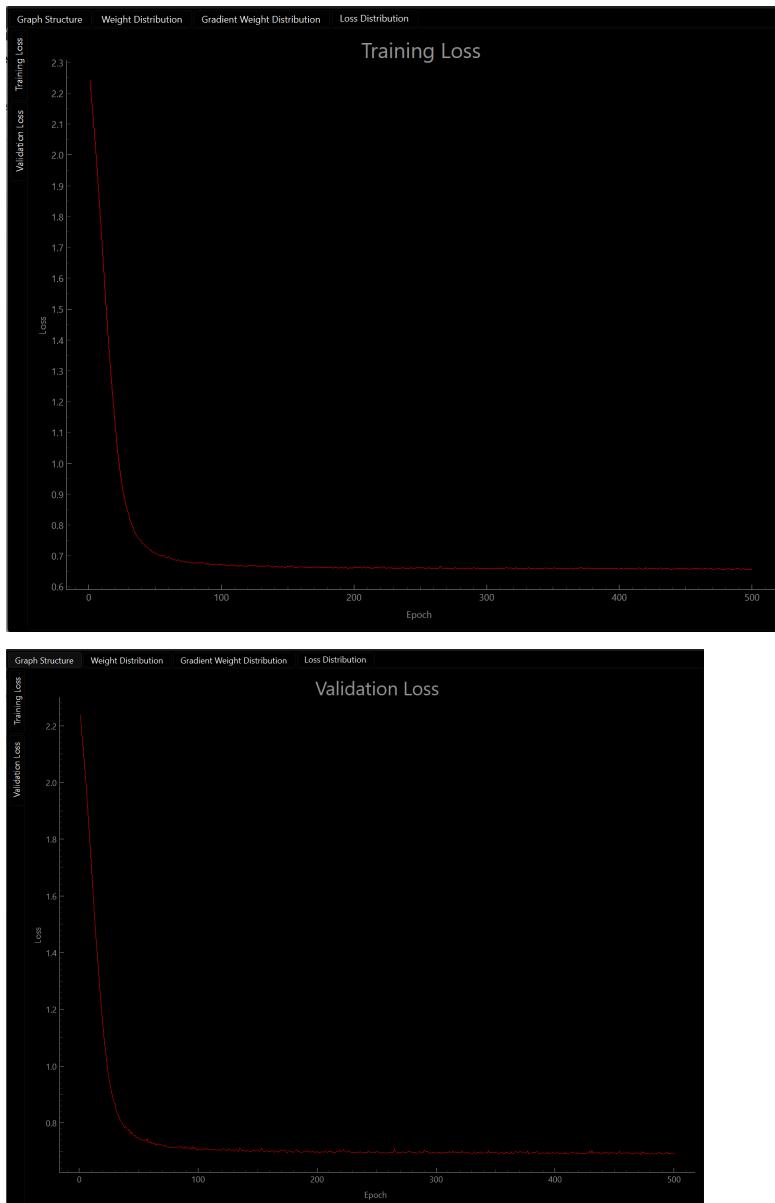
● Weight Distribution



● Gradient Weight Distribution



● Perbandingan Training dengan Validation



2.2.1.3 Hasil Analisis

Berdasarkan hasil pengujian dengan berbagai kedalaman jaringan, model dengan satu hidden layer berukuran 128 menunjukkan performa terbaik dengan F1-score tertinggi sebesar 0.9193. Model ini memiliki distribusi bobot yang lebih stabil, dengan gradien yang tetap terdistribusi dengan baik, sehingga proses pembelajaran dapat berlangsung secara optimal tanpa mengalami permasalahan vanishing (fenomena dalam pelatihan jaringan saraf di mana gradien dari fungsi aktivasi menjadi sangat kecil saat backpropagation berlangsung) atau exploding gradient.

Ketika kedalaman jaringan bertambah menjadi tiga hidden layer dengan ukuran 128, 64, dan 64, terjadi penurunan F1-score menjadi 0.8803. Walaupun model ini

masih menunjukkan performa yang cukup baik, distribusi bobotnya mulai menyempit, yang dapat menyebabkan kesulitan dalam pembelajaran layer terdalam. Distribusi gradien juga menunjukkan kecenderungan menyusut pada layer-layer terdalam, menandakan potensi permasalahan vanishing gradient yang dapat memperlambat terjadinya konvergensi.

Pada model dengan empat hidden layer (128, 64, 64, 128), performa menurun drastis dengan F1-score sebesar 0.2436. Penurunan ini menunjukkan bahwa jaringan dengan kedalaman lebih besar justru mengalami kesulitan dalam mempertahankan distribusi bobot dan gradien yang optimal. Distribusi bobot menjadi semakin sempit, sedangkan distribusi gradien menunjukkan kecenderungan menghilang di layer yang lebih dalam, yang semakin memperparah efek vanishing gradient.

Secara keseluruhan, hasil ini menunjukkan bahwa peningkatan kedalaman jaringan tidak selalu meningkatkan performa model. Model dengan satu hidden layer memberikan hasil terbaik dengan keseimbangan distribusi bobot dan gradien yang baik, sementara model yang lebih dalam mengalami penurunan performa akibat vanishing gradient dan ketidakseimbangan distribusi bobot.

Sedangkan, jika dilihat dari pengaruh width nya, model dengan lebar 128-128-128 memiliki F1-score tertinggi, yaitu 0.9319, yang menunjukkan performa paling optimal dibandingkan model lainnya. Model ini juga memiliki nilai loss yang lebih kecil dan waktu pelatihan yang lebih lama dibandingkan arsitektur lainnya. Sementara itu, model dengan lebar 64-64-64 memiliki F1-score 0.9309, sedikit lebih rendah dibandingkan model 128-128-128, tetapi dengan waktu pelatihan yang lebih cepat. Model dengan lebar 32-32-32 menunjukkan performa paling rendah dengan F1-score 0.8819, ini menunjukkan jaringan dengan jumlah neuron yang lebih sedikit kurang mampu menangkap pola yang kompleks.

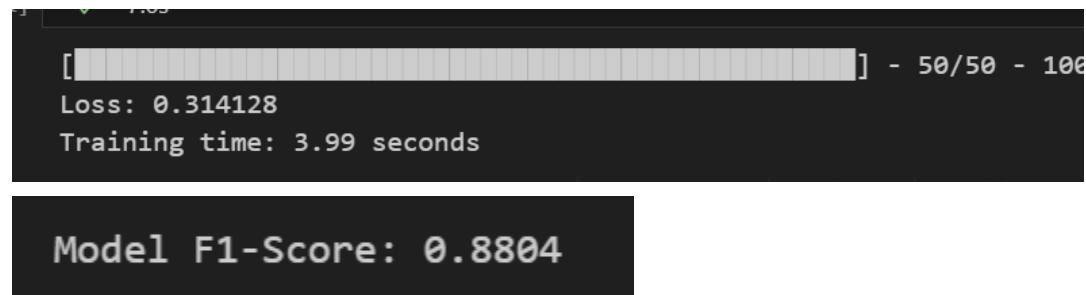
Dari aspek distribusi bobot, model dengan lebih banyak neuron cenderung memiliki distribusi yang lebih tersebar, menunjukkan adanya lebih banyak parameter yang terlibat dalam pembelajaran. Distribusi gradien juga menunjukkan pola yang serupa, di mana model yang lebih besar memiliki distribusi gradien yang lebih tajam, menunjukkan bahwa masing-masing parameter mengalami perubahan yang lebih signifikan selama proses pembelajaran. Dari analisis tersebut dapat dikatakan bahwa model dengan lebih banyak neuron memiliki kapasitas yang lebih besar untuk belajar, tetapi dengan konsekuensi meningkatnya waktu komputasi. Model yang lebih kecil, meskipun lebih cepat untuk dilatih, tampaknya tidak memiliki kapasitas yang cukup untuk menangkap kompleksitas data secara optimal, seperti yang terlihat dari skor F1 yang lebih rendah.

Berdasarkan grafik *Training Loss* dan *Validation Loss* yang menunjukkan pola penurunan yang sangat mirip dan stabil, dapat disimpulkan bahwa proses pelatihan model berlangsung dengan baik tanpa mengalami *overfitting* maupun *underfitting*. Kedua loss menurun tajam di awal pelatihan dan mencapai titik konvergensi setelah sekitar 100 epoch, lalu melandai secara konsisten hingga akhir pelatihan. Tidak adanya perubahan besar secara tiba-tiba atau selisih yang mencolok antara training dan validation loss mengindikasikan bahwa pembelajaran berlangsung stabil, dengan distribusi bobot dan gradien yang terjaga dengan baik sepanjang proses. Hal ini memperkuat penjelasan sebelumnya bahwa arsitektur model dengan keseimbangan antara kedalaman dan lebar jaringan sangat penting untuk memastikan kestabilan proses pelatihan serta pencapaian performa model yang optimal.

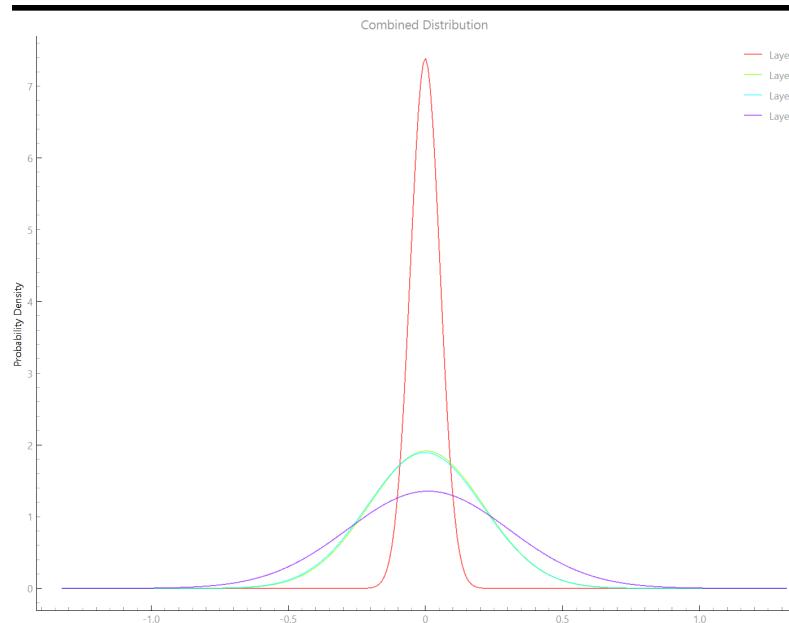
2.2.2 Pengaruh fungsi aktivasi

a. Fungsi Aktivasi Linear

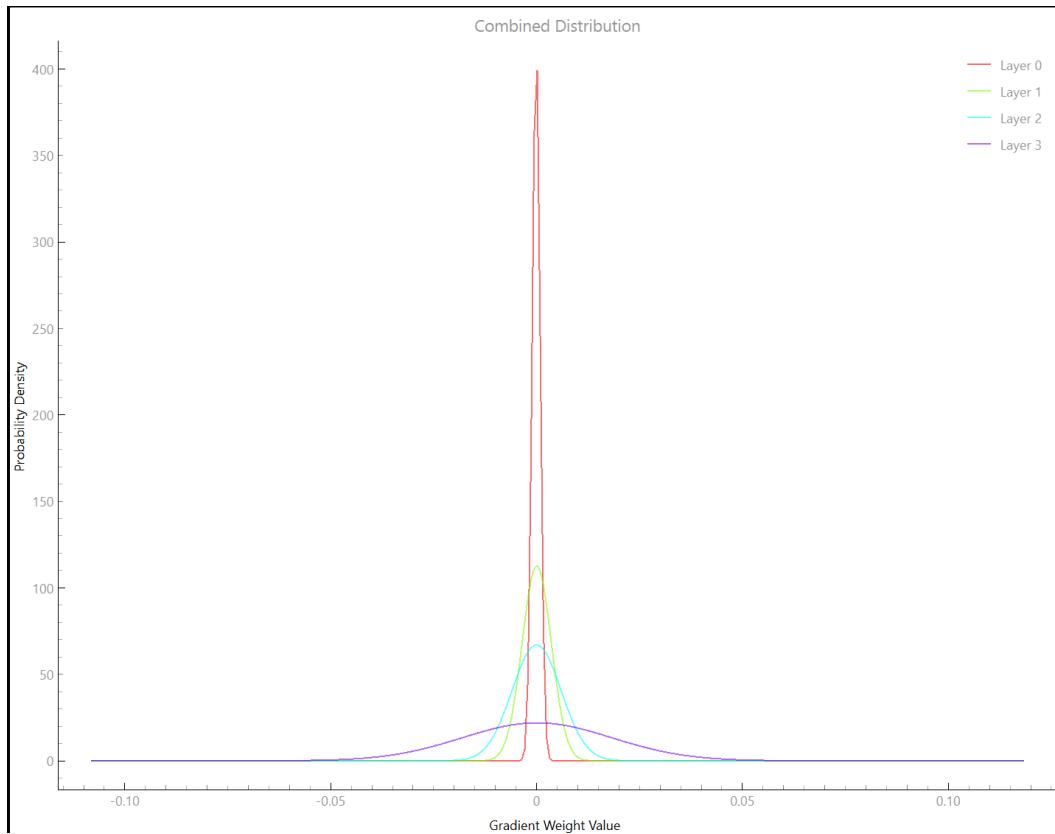
- **Prediction Score**



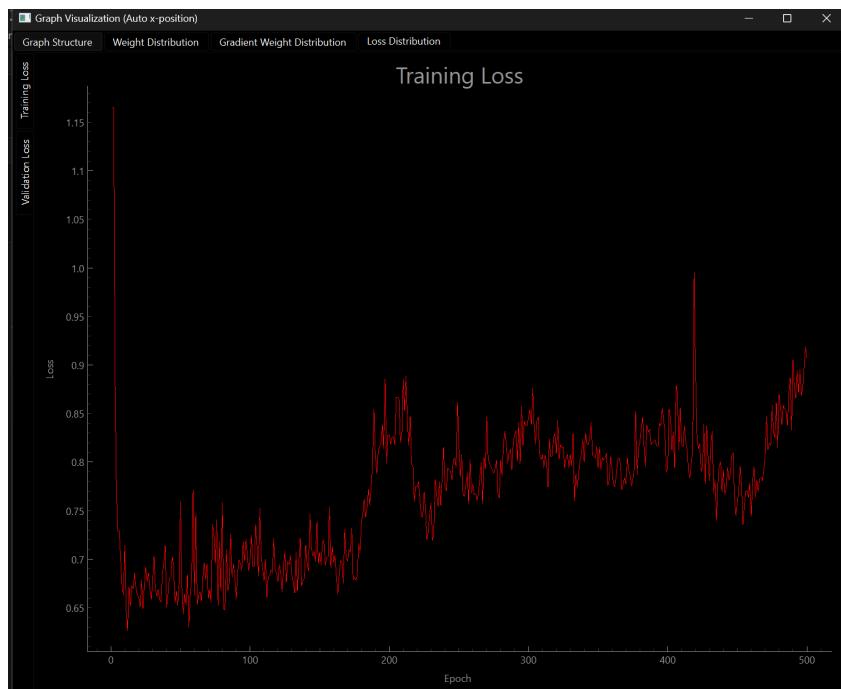
- **Distribusi Bobot**

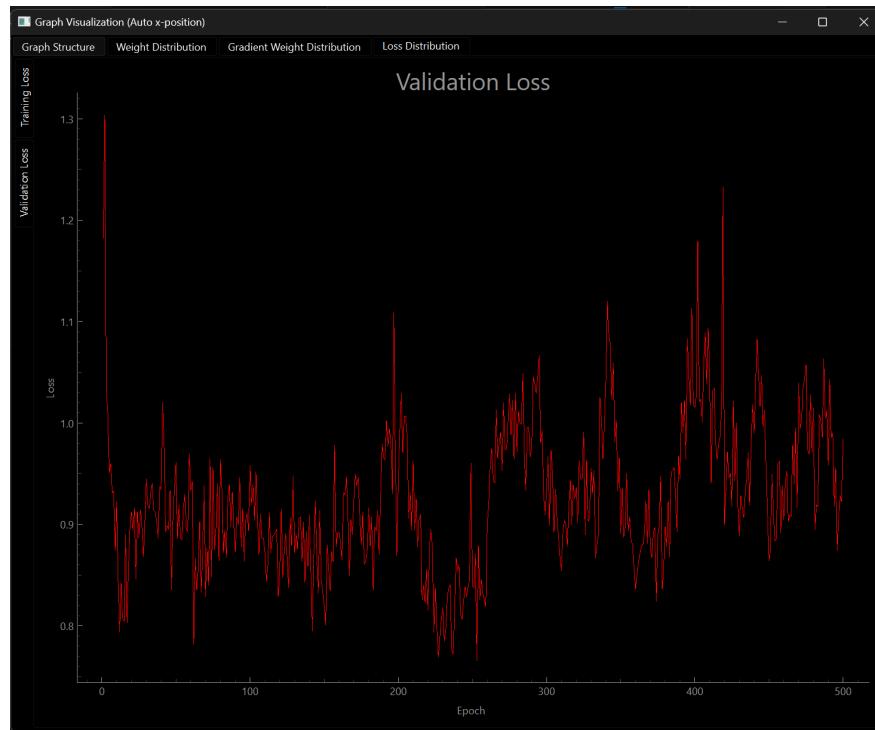


● Distribusi Gradien



● Perbandingan Training dengan Validation





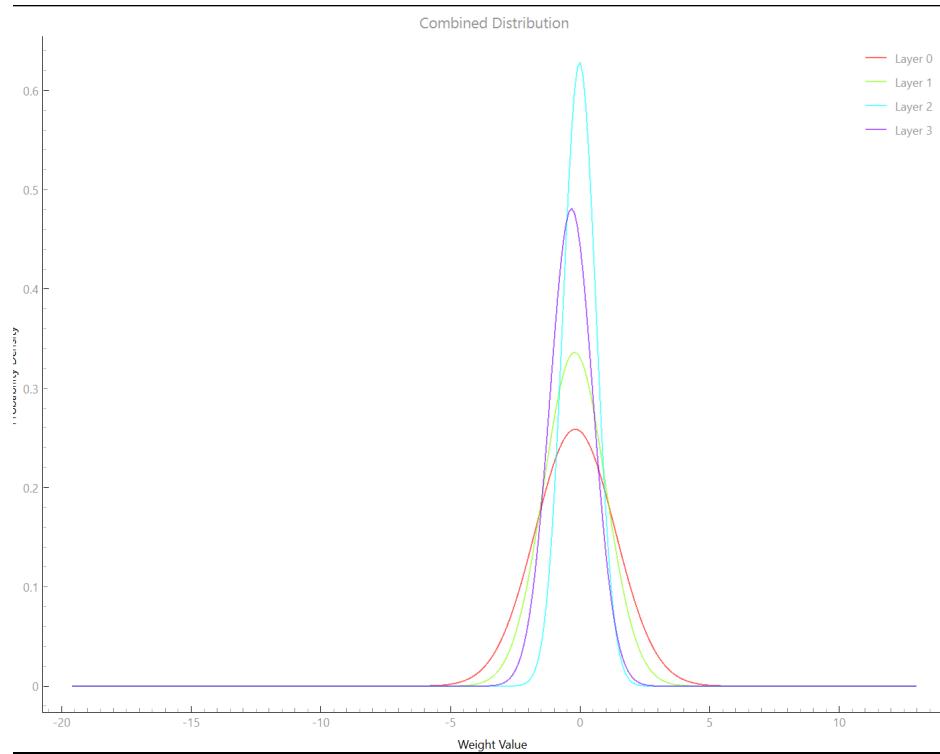
b. Fungsi Aktivasi Sigmoid

- **Prediction Score**

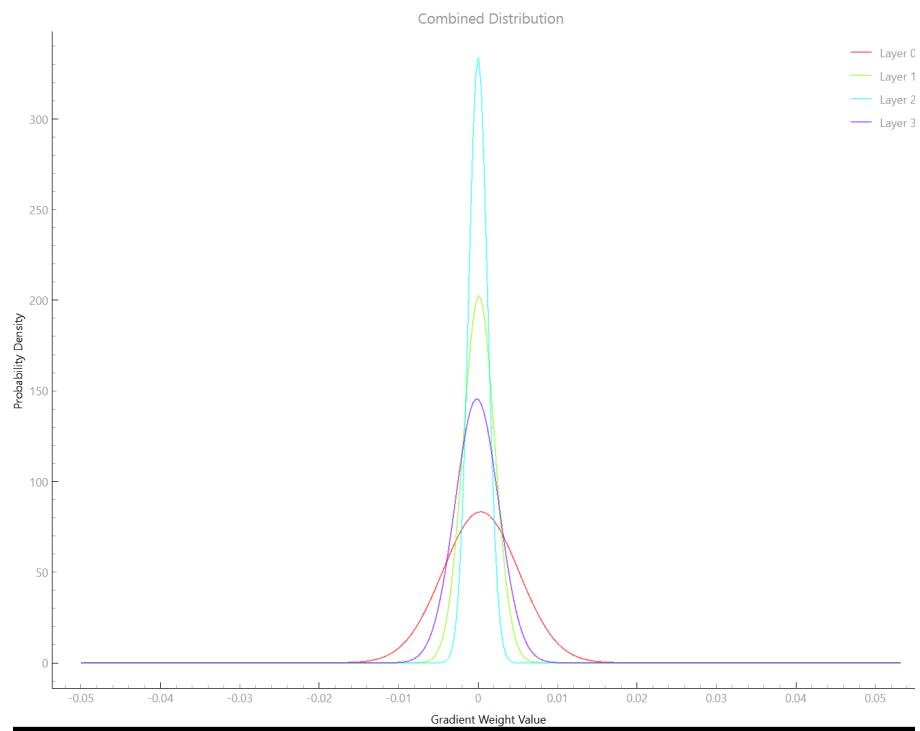
```
[  
Loss: 0.873165  
Training time: 59.11 seconds
```

```
Model F1-Score: 0.6976
```

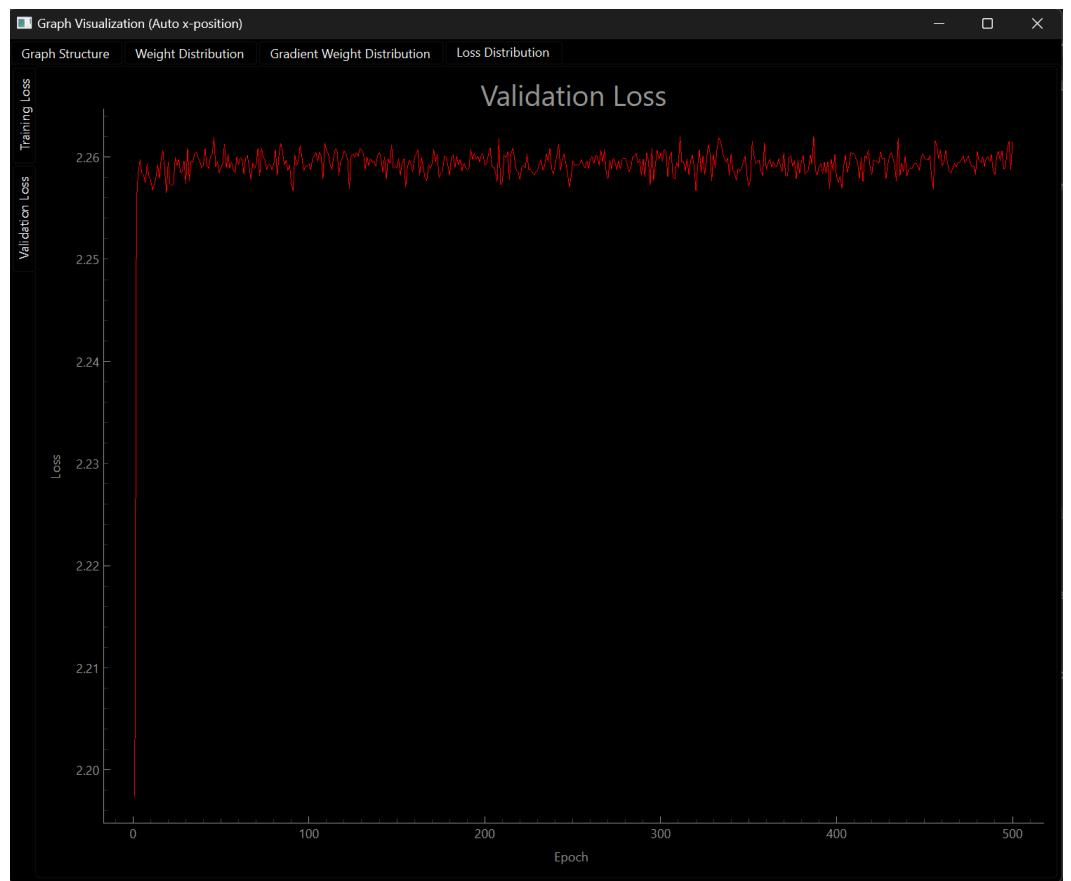
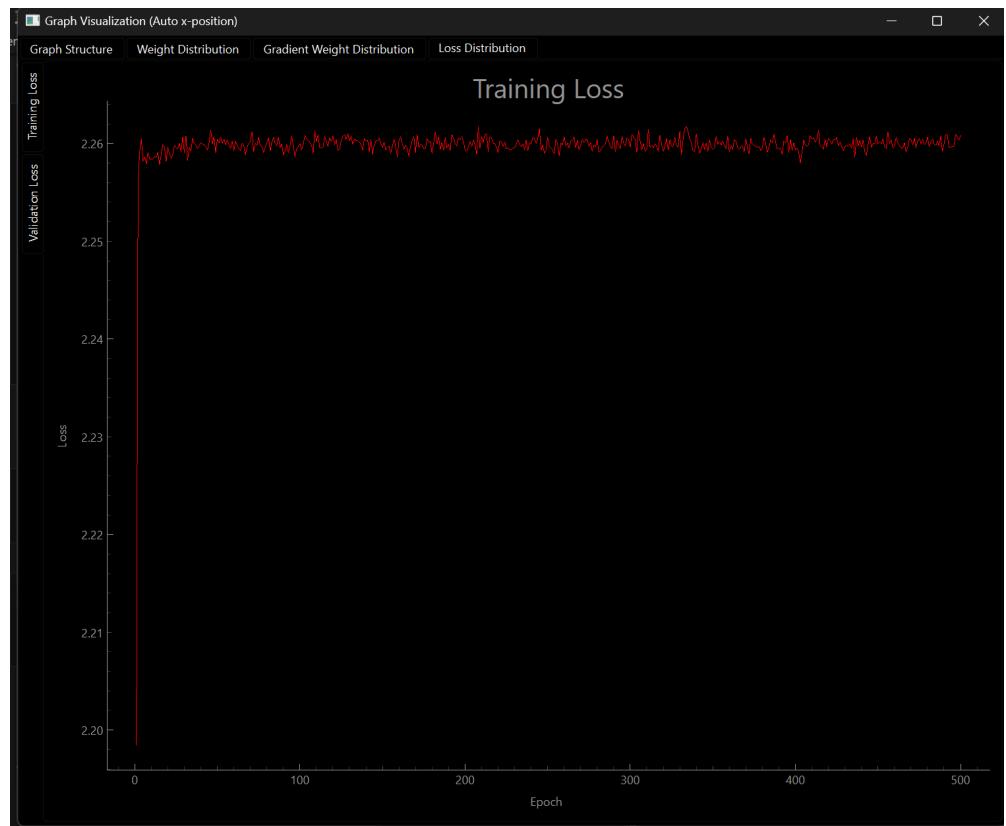
- **Weight Distribution**



● Gradient Weight Distribution

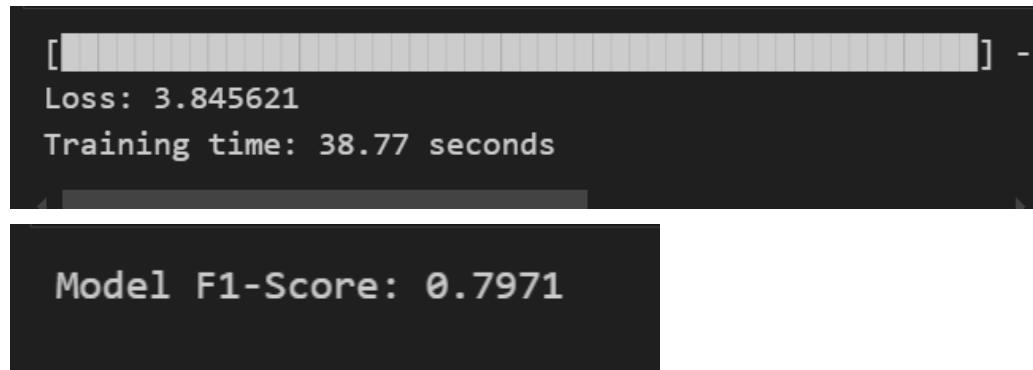


● Perbandingan Training dengan Validation

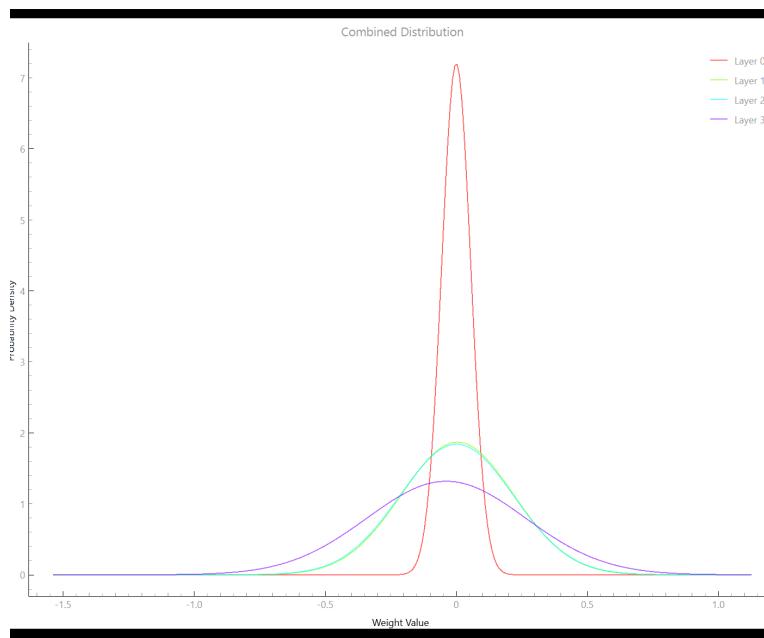


c. Fungsi Aktivasi ReLu

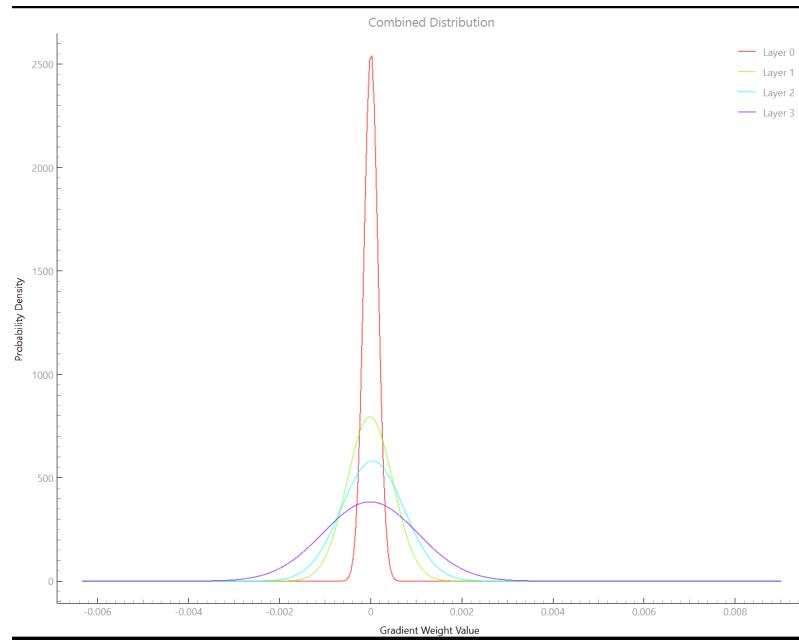
- **Prediction Score**



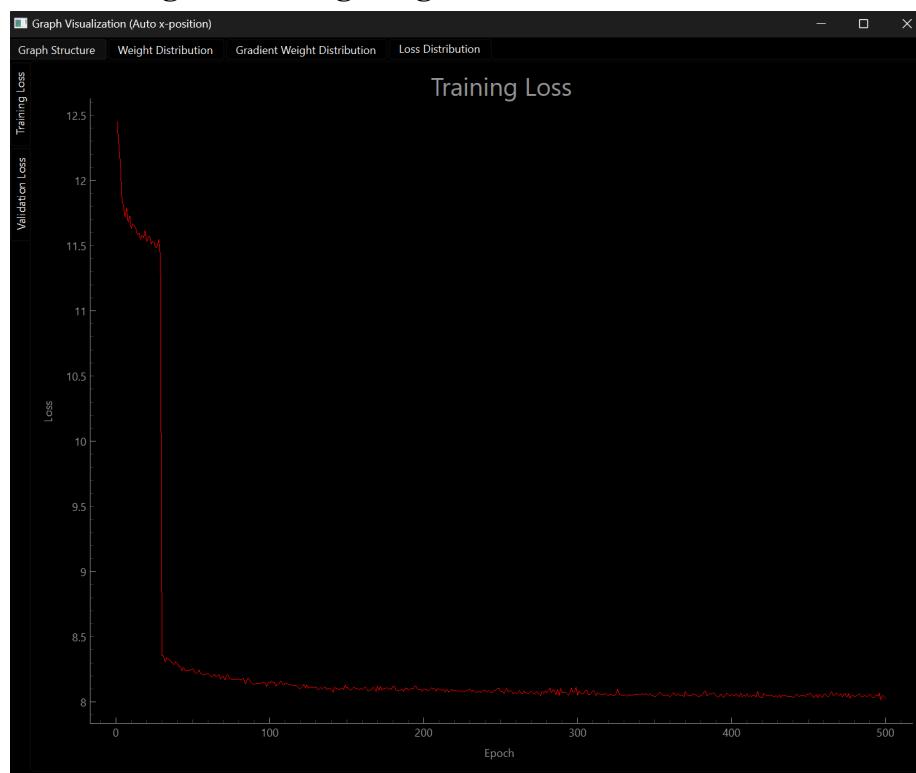
- **Weight Distribution**

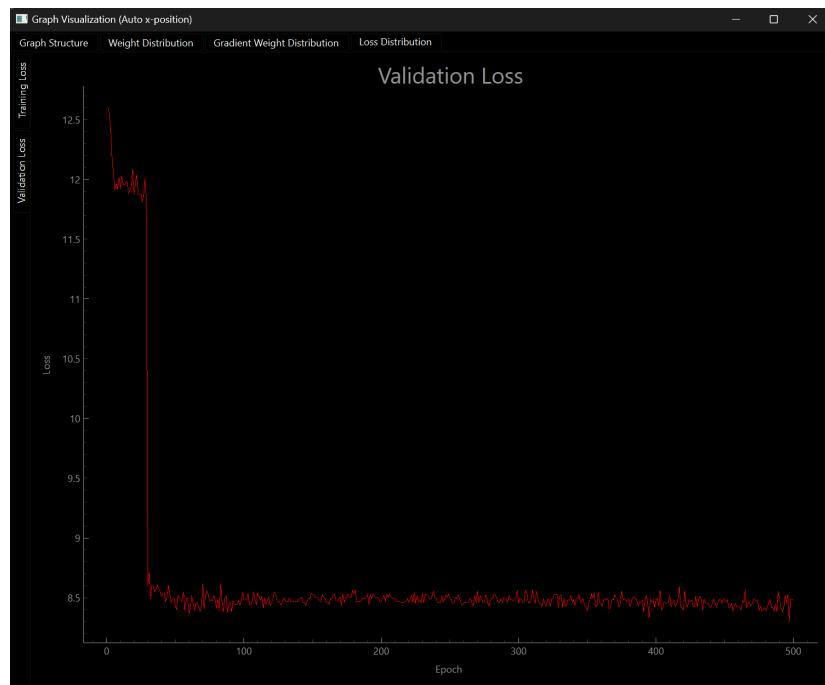


- **Gradient Weight Distribution**



- Perbandingan Training dengan Validation

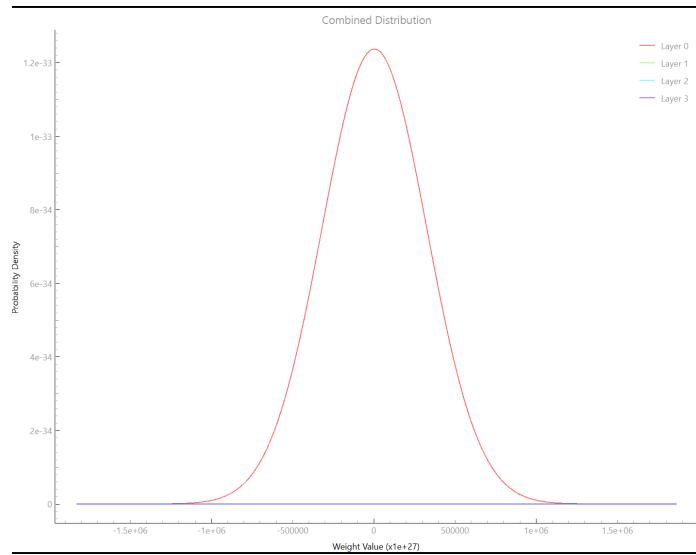




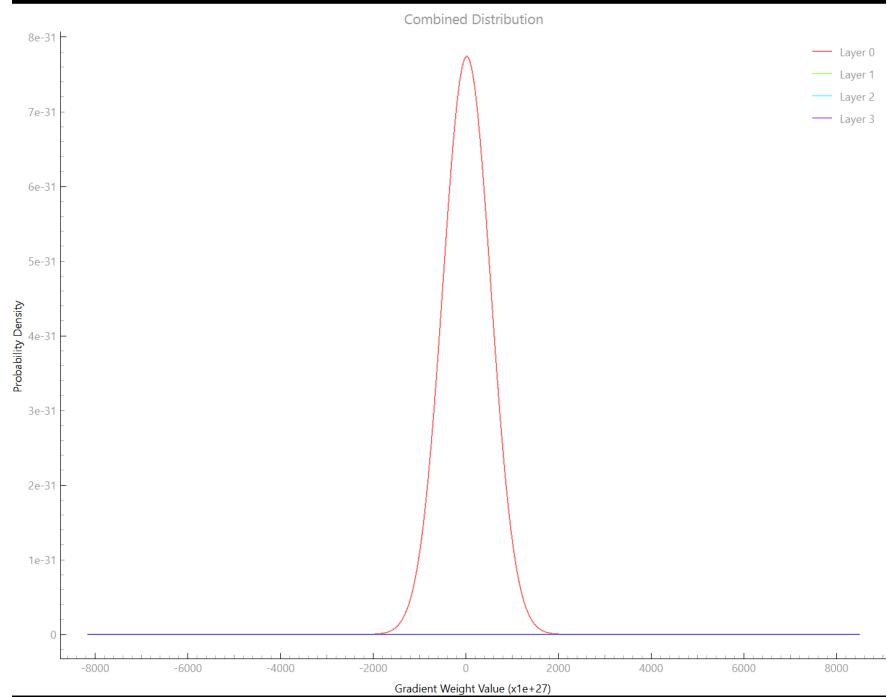
- d. Fungsi Aktivasi TanH
- **Prediction Score**

```
[  
Loss: 22.318957  
Training time: 47.91 seconds  
  
Model F1-Score: 0.0445
```

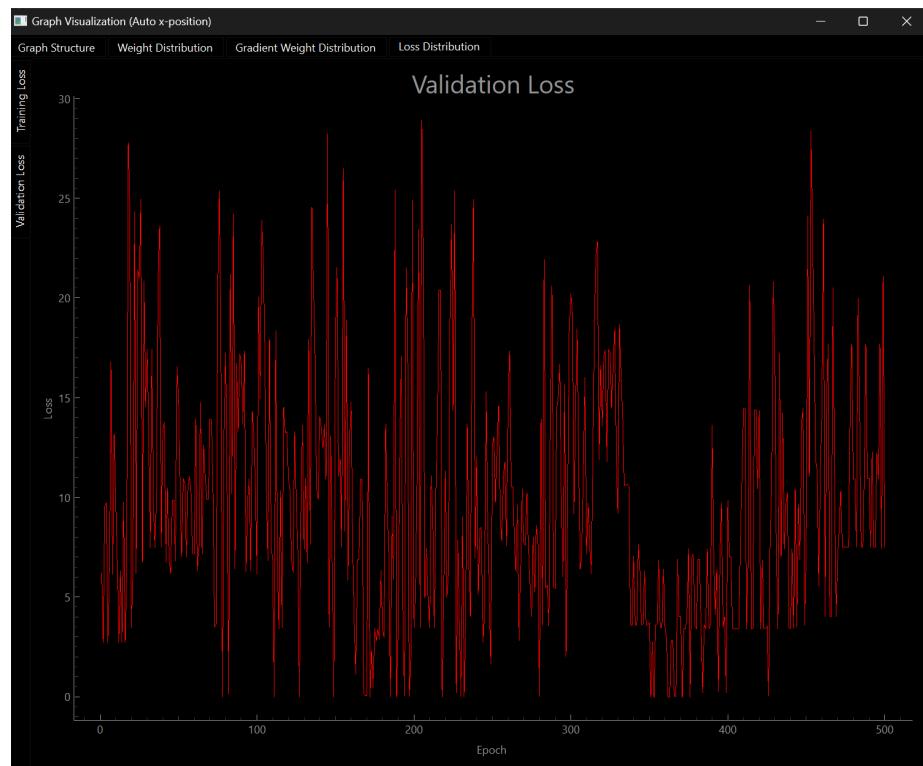
- **Weight Distribution**



● Gradient Weight Distribution



● Perbandingan Training dengan Validation



e. Hasil Analisis

Dari hasil pengujian di atas, dapat disimpulkan bahwa fungsi aktivasi Linear memiliki F1-score tertinggi dibandingkan yang lain, yaitu 0.8804 dengan epoch 50. Namun, saat jumlah epoch diperbesar, model mengalami masalah karena nilai loss menjadi NaN, yang menyebabkan kegagalan dalam proses pembelajaran. Hal ini terjadi karena Linear tidak memiliki sifat non-linearitas yang bisa digunakan untuk memahami hubungan yang kompleks dalam data. Fungsi aktivasi Sigmoid, meskipun lebih stabil, memiliki F1-score yang lebih rendah, yaitu 0.6976. Bobotnya lebih menyebar dibandingkan Linear, tetapi gradiennya sangat kecil dan mendekati nol, yang menunjukkan adanya masalah vanishing gradient. Vanishing gradient adalah fenomena dalam pelatihan jaringan saraf di mana gradien dari fungsi aktivasi menjadi sangat kecil saat backpropagation berlangsung, terutama dalam layer-layer yang lebih dalam. Akibatnya, bobot pada lapisan awal diperbarui dengan sangat lambat atau bahkan berhenti berubah, sehingga model kesulitan belajar pola yang lebih kompleks.

Selanjutnya, fungsi aktivasi ReLU menunjukkan performa yang lebih baik daripada Sigmoid dengan F1-score sebesar 0.7971. Distribusi bobotnya lebih menyebar, dan gradiennya lebih besar dibandingkan Sigmoid, meskipun tetap memiliki beberapa titik nol akibat dying ReLU. Keunggulan ReLU terletak pada ketahanannya terhadap vanishing gradient, sehingga pelatihan berjalan lebih cepat dengan waktu 38.77 detik. Namun, tetap ada risiko unit yang mati karena selalu menghasilkan nol. Di sisi lain, fungsi aktivasi Tanh memiliki performa paling buruk, dengan F1-score yang hanya mencapai 0.0445. Gradiennya lebih baik dibandingkan Sigmoid tetapi masih rentan terhadap vanishing gradient.

Berdasarkan grafik *training loss* dan *validation loss* untuk masing-masing fungsi aktivasi, terlihat bahwa ReLU memiliki penurunan *loss* yang paling konsisten dan signifikan seiring bertambahnya epoch. Hal ini menunjukkan proses pelatihan yang stabil dan efektif. Hal ini mencerminkan kemampuan ReLU dalam menghindari masalah *vanishing gradient* dan mempercepat konvergensi. Sebaliknya, fungsi aktivasi Tanh menunjukkan variasi loss yang sangat ekstrem baik pada data training maupun validation, yang mengindikasikan pelatihan yang tidak stabil dan ketidakmampuan model untuk belajar secara efektif. Pada fungsi aktivasi Sigmoid, meskipun nilai loss tampak lebih stabil dibanding Tanh, namun nilainya cenderung tinggi dan tidak mengalami penurunan yang terlalu signifikan, yang berarti memperkuat adanya masalah *vanishing gradient* yang menghambat pembelajaran. Untuk fungsi aktivasi Linear, pada awalnya loss mengalami penurunan, tetapi seiring bertambahnya epoch, nilainya justru meningkat atau bahkan menjadi tidak terdefinisi (NaN), ini menandakan ketidakstabilan pelatihan akibat kurangnya kemampuan untuk menangkap hubungan non-linear dalam data.

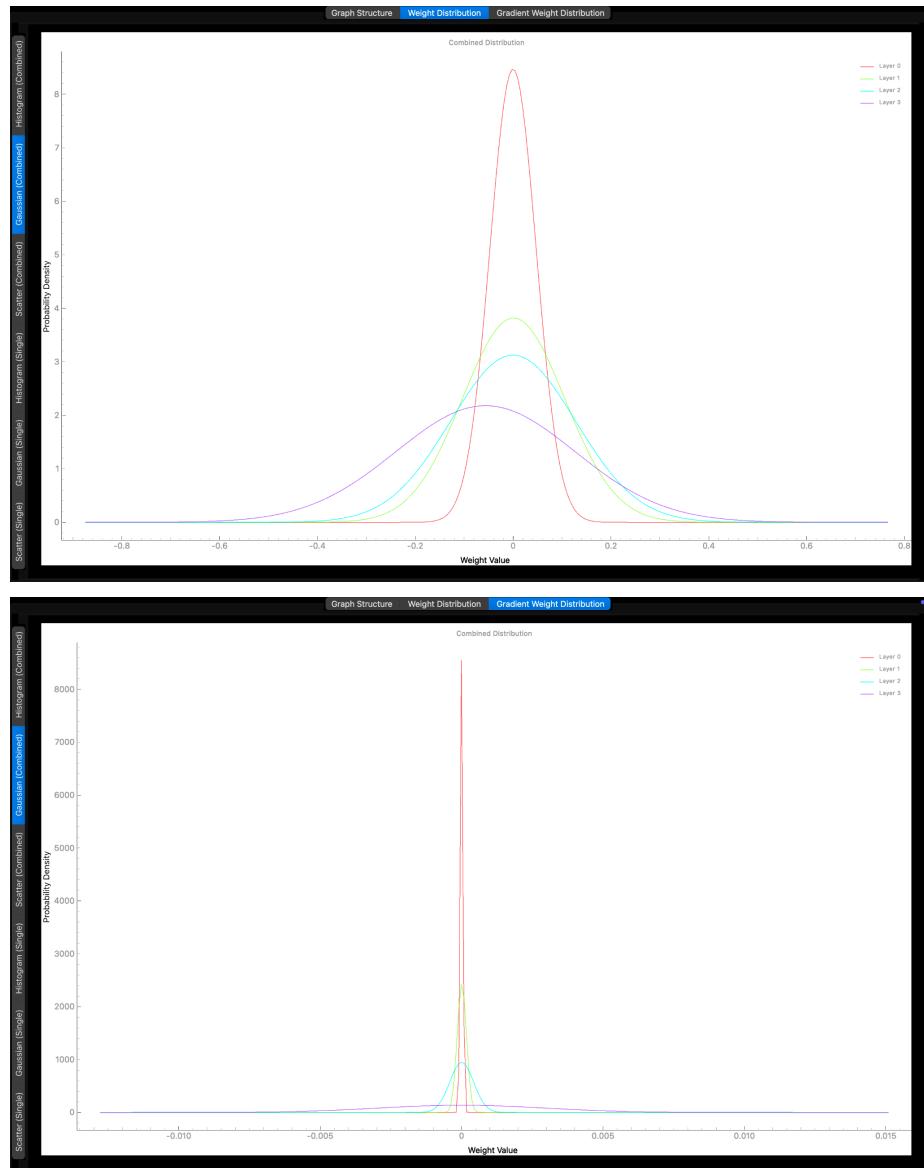
Secara keseluruhan, meskipun Linear memiliki skor terbaik dalam kondisi awal, fungsinya tidak stabil untuk pelatihan jangka panjang. Sigmoid dan Tanh mengalami masalah vanishing gradient yang memperlambat proses pembelajaran, sedangkan ReLU menjadi pilihan terbaik karena memberikan keseimbangan antara akurasi, waktu pelatihan, dan kestabilan gradien.

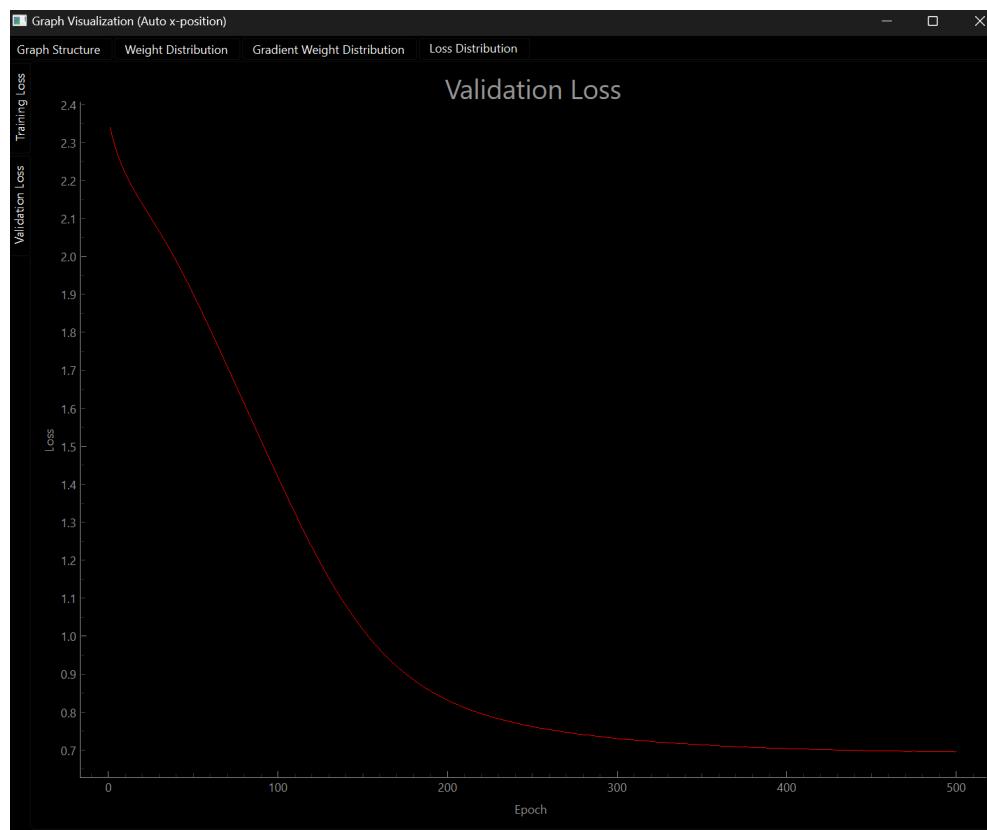
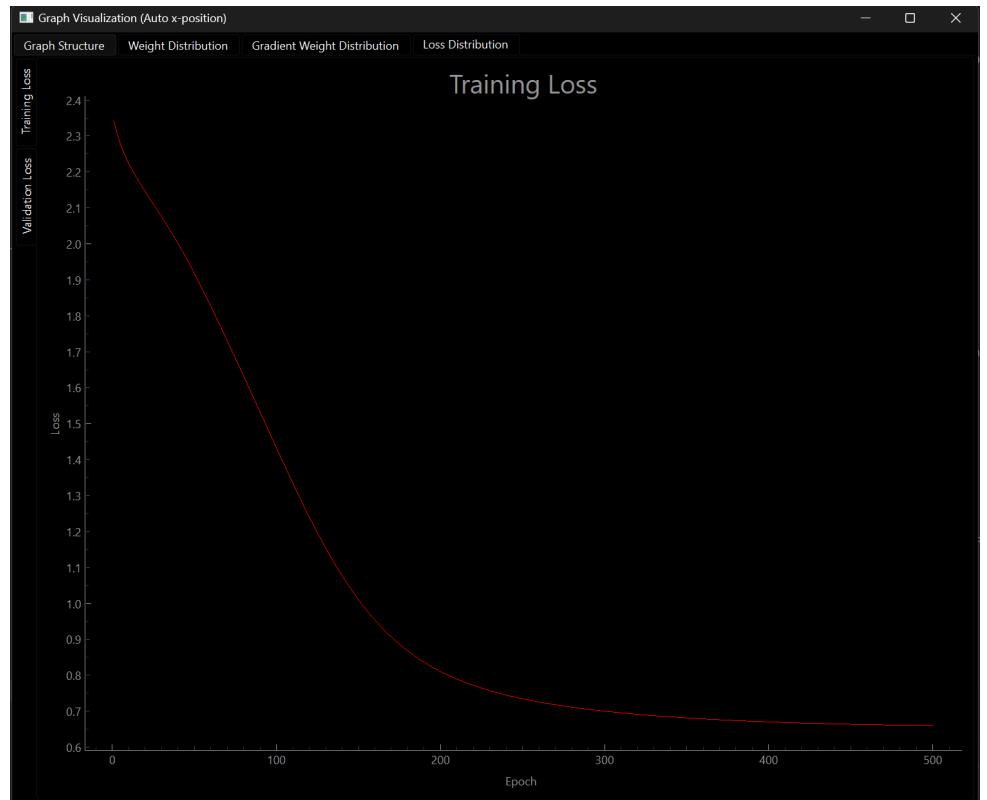
2.2.3 Pengaruh learning rate

a. Learning rate = 0.01

```
[██████████] - 500/500 - 100.0% - 40.71 s - Loss: 2.218330
Loss: 2.218330
Training time: 40.71 seconds
Model berhasil disimpan ke ffnn_model.pkl
```

Model F1-Score: 0.3122
Model Accuracy: 0.4260

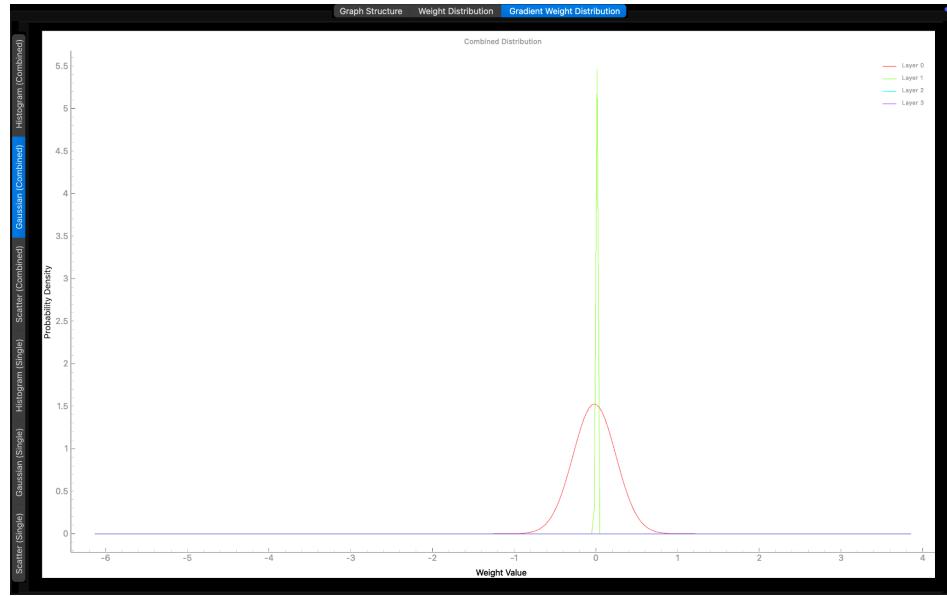
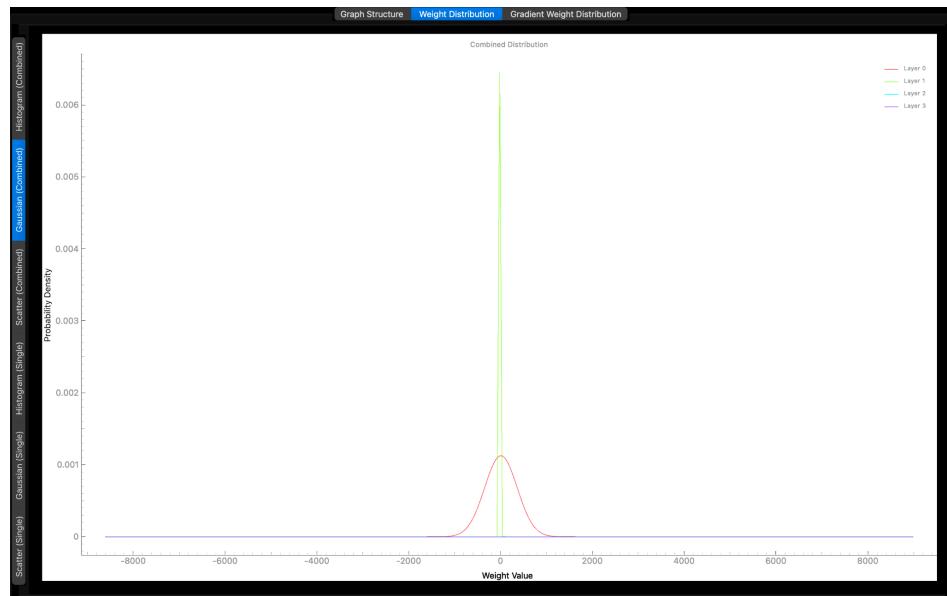


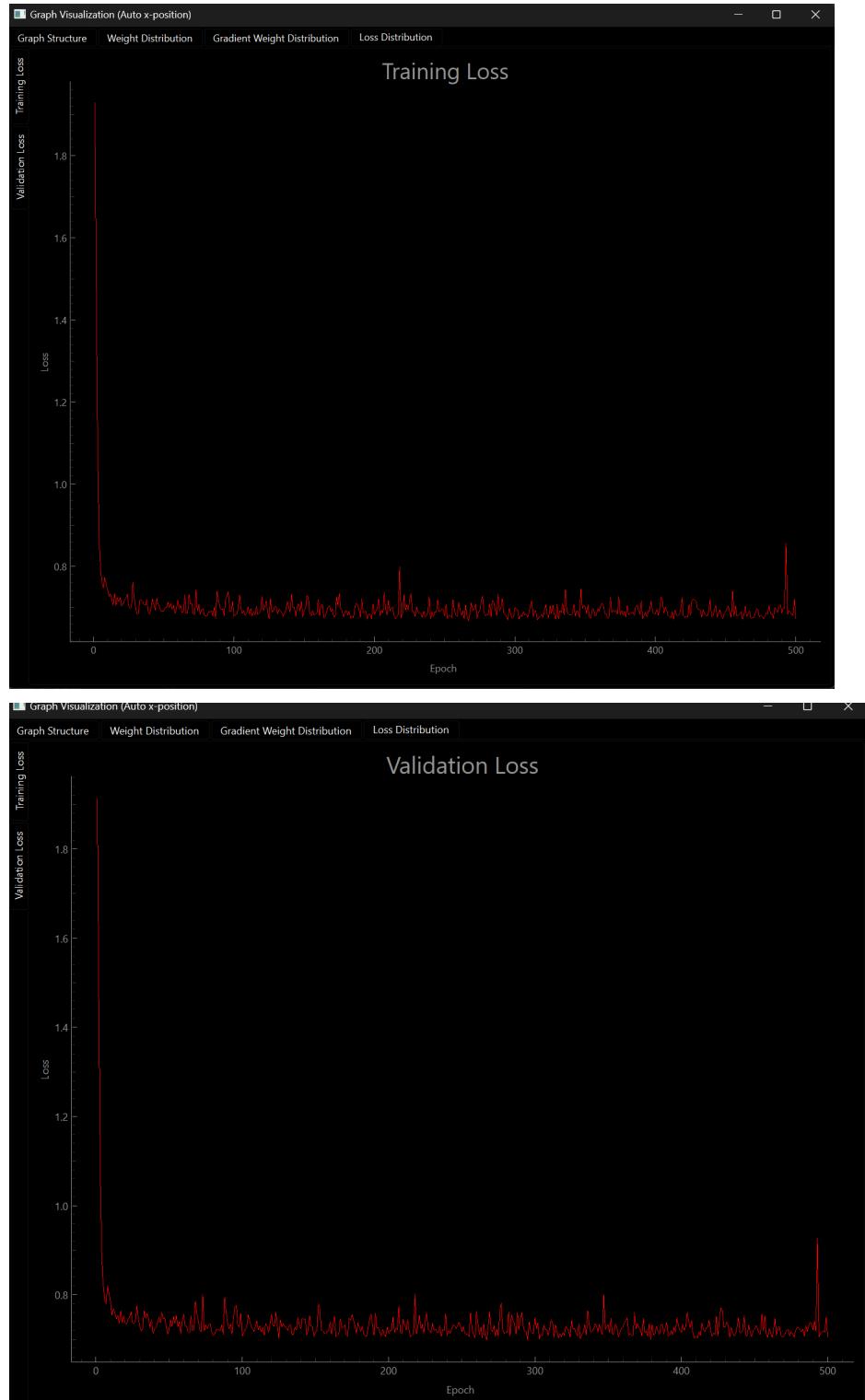


b. Learning rate = 0.5

```
[██████████] - 500/500 - 100.0% - 36.38 s - Loss: 1.273574
Loss: 1.273574
Training time: 36.38 seconds
Model berhasil disimpan ke ffnn_model.pkl
```

Model F1-Score: 0.5265
Model Accuracy: 0.5589

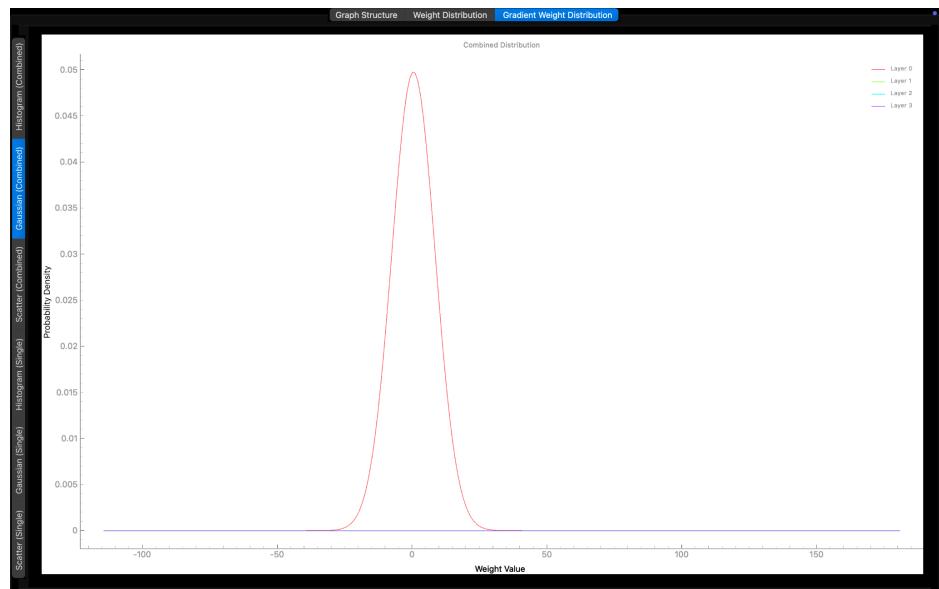
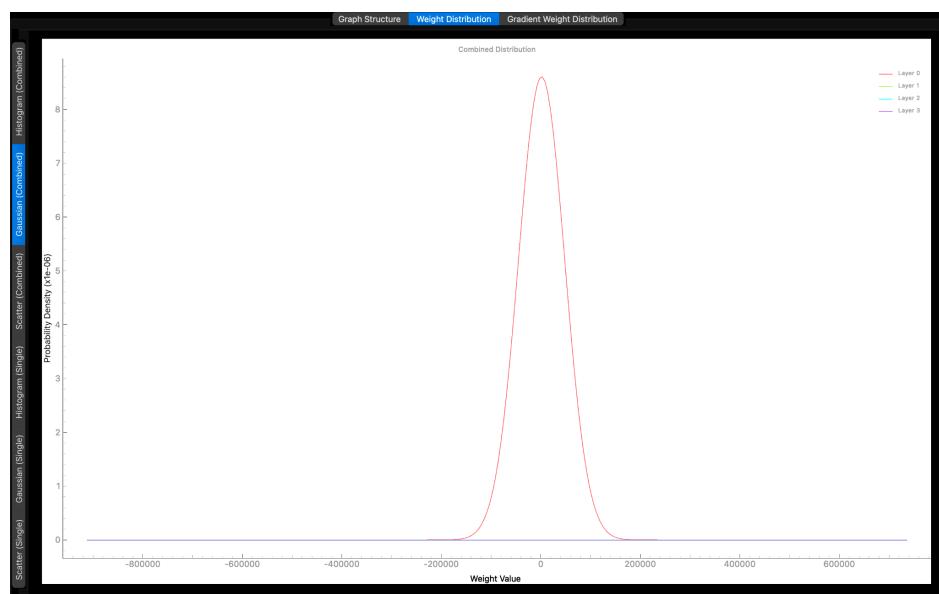


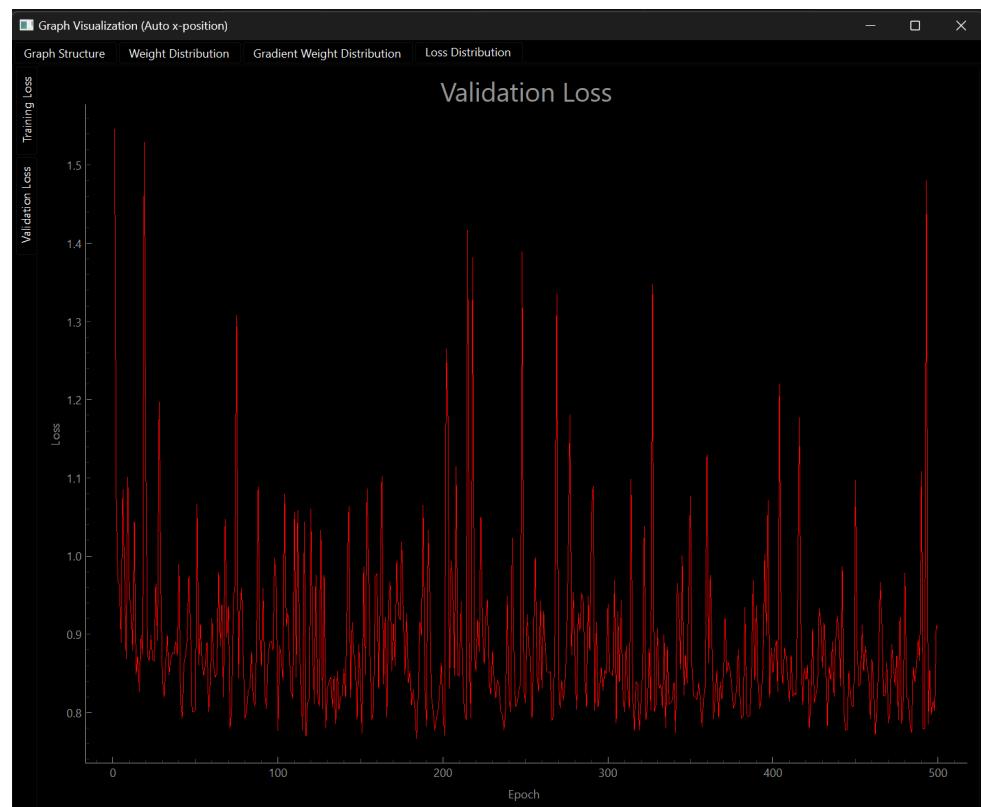


c. Learning rate = 1

```
[-----] - 500/500 - 100.0% - 36.61 s - Loss: 1.495791  
Loss: 1.495791  
Training time: 36.61 seconds  
Model berhasil disimpan ke ffnn_model.pkl
```

Model F1-Score: 0.4397
Model Accuracy: 0.4456





d. Hasil Analisis

Berdasarkan hasil eksperimen dengan tiga nilai learning rate yang berbeda, terlihat bahwa learning rate 0.5 menghasilkan performa terbaik dengan F1-score 0.5265 dan akurasi 0.5589, lebih tinggi dibandingkan learning rate 0.01 yang hanya mencapai F1-score 0.3122 dan akurasi 0.4260, serta learning rate 1 yang memiliki F1-score 0.4397 dan akurasi 0.4456. Learning rate dengan angka 0.01 ini terlalu kecil, sehingga model sulit melakukan pembaruan bobot secara efektif, bisa terlihat juga dari distribusi bobot yang lebih menyebar dan gradien yang kecil. Sebaliknya, learning rate 1 cenderung terlalu besar, menyebabkan perubahan bobot yang lebih agresif dan berisiko membuat model kurang stabil atau bahkan *overshoot*. Sementara itu, learning rate 0.5 menunjukkan distribusi bobot yang lebih terpusat dengan puncak yang tajam, menandakan *update* bobot yang lebih stabil dan optimal. Distribusi gradien pada nilai ini juga lebih terkontrol dibandingkan dua nilai lainnya, sehingga model dapat belajar secara lebih efisien tanpa mengalami osilasi atau merupakan pilihan yang paling optimal dalam eksperimen ini karena mampu menyeimbangkan antara stabilitas dan kecepatan untuk mencapai konvergensi. Jadi, learning rate 0.5 yang paling menghasilkan performa terbaik di antara ketiga nilai yang diuji.

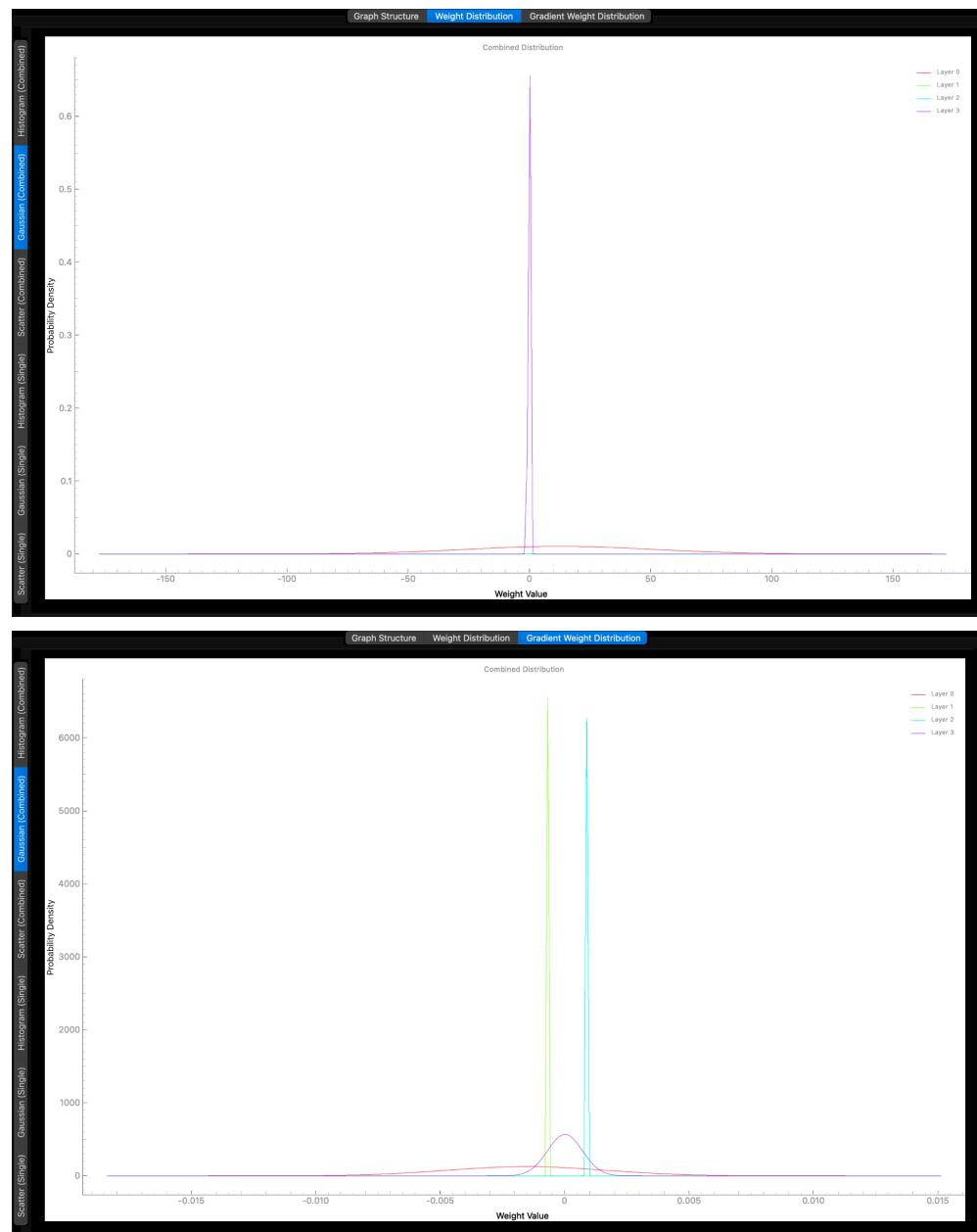
Dari grafik loss yang dihasilkan untuk ketiga nilai learning rate, terlihat bahwa model dengan learning rate 0.01 menunjukkan penurunan training dan validation loss yang halus dan stabil, namun cenderung lambat dan berakhir pada nilai loss yang masih relatif tinggi, ini menunjukkan proses pembelajaran yang kurang optimal. Sementara itu, pada learning rate 0.5, baik training loss maupun validation loss menurun drastis di awal dan kemudian cenderung stabil di angka yang lebih rendah, menunjukkan bahwa model mampu belajar secara cepat dan konvergen dengan baik, tanpa gejala overfitting yang terlalu signifikan. Sebaliknya, pada learning rate 1, kurva training dan validation loss tampak sangat tidak stabil dengan banyak lonjakan tajam sepanjang proses pelatihan, yang mengindikasikan bahwa model mengalami kesulitan untuk konvergen akibat pembaruan bobot yang terlalu besar. Hal ini memperkuat bahwa learning rate 0.5 memberikan keseimbangan terbaik dalam kecepatan belajar dan stabilitas pelatihan.

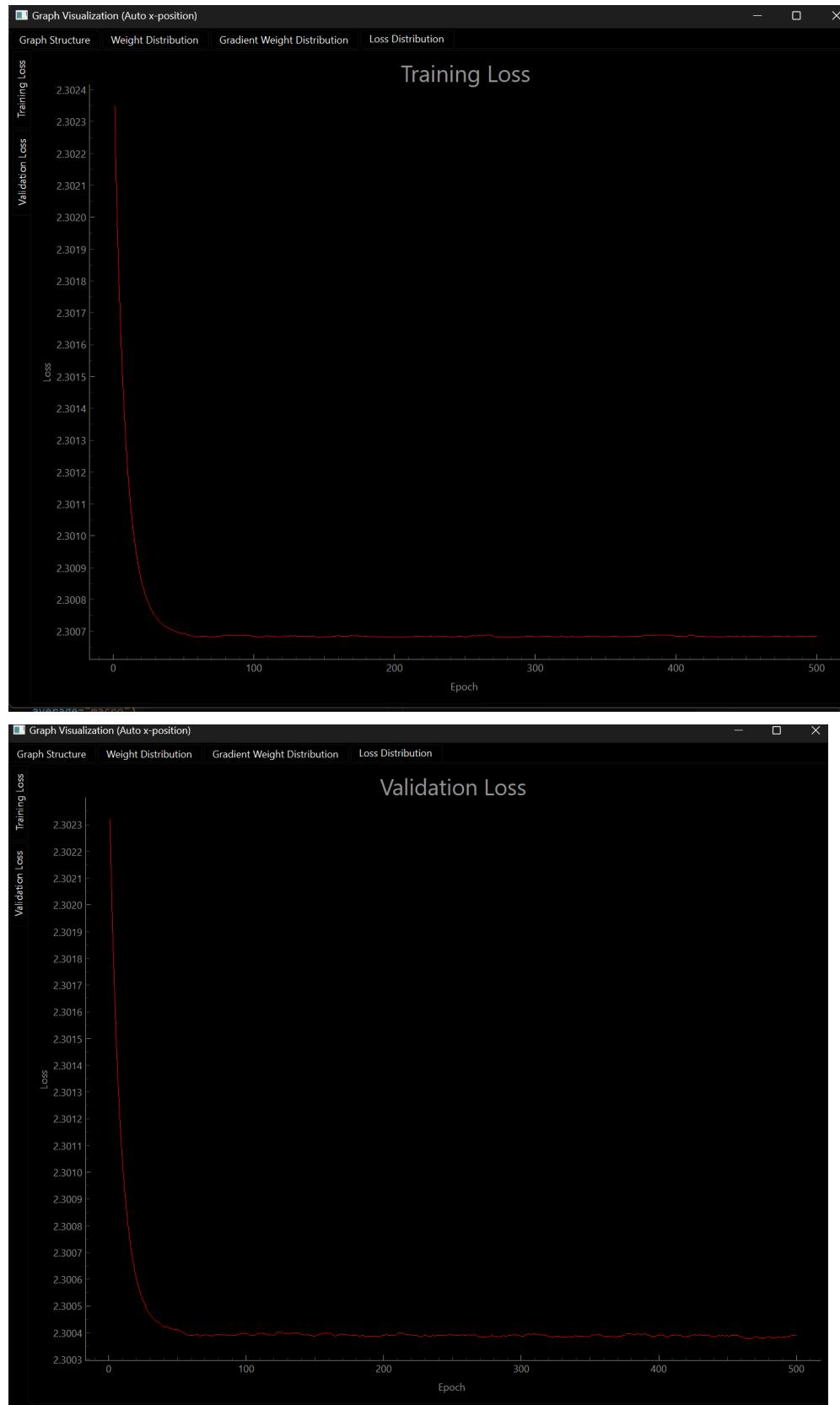
2.2.4 Pengaruh inisialisasi bobot

a. Zero initialization

```
[██████████] - 500/500 - 100.0% - 35.71 s - Loss: 2.062614
Loss: 2.062614
Training time: 35.71 seconds
Model berhasil disimpan ke ffnn_model.pkl
```

Model F1-Score: 0.0986
Model Accuracy: 0.2079

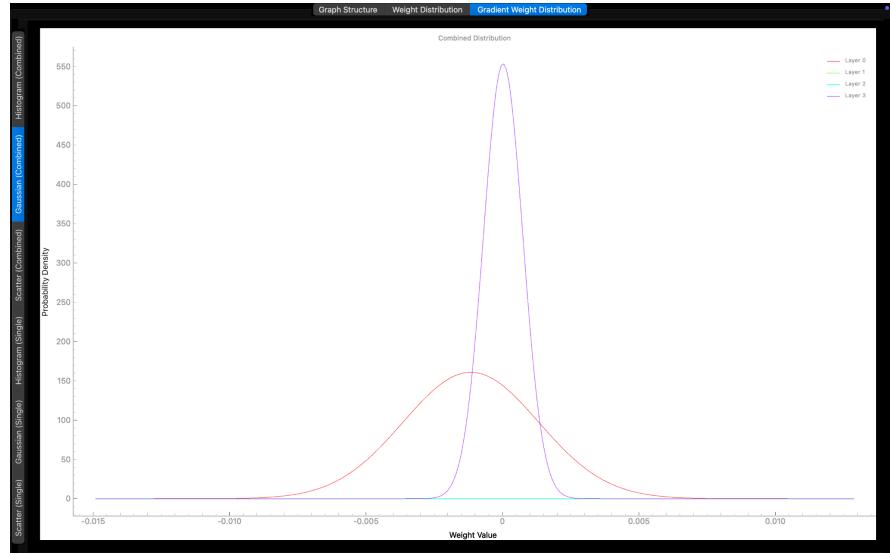
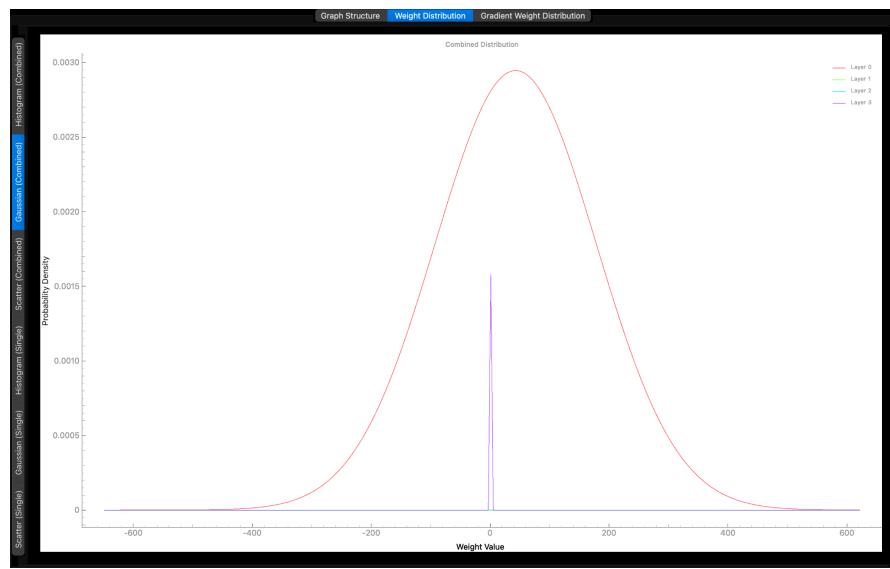


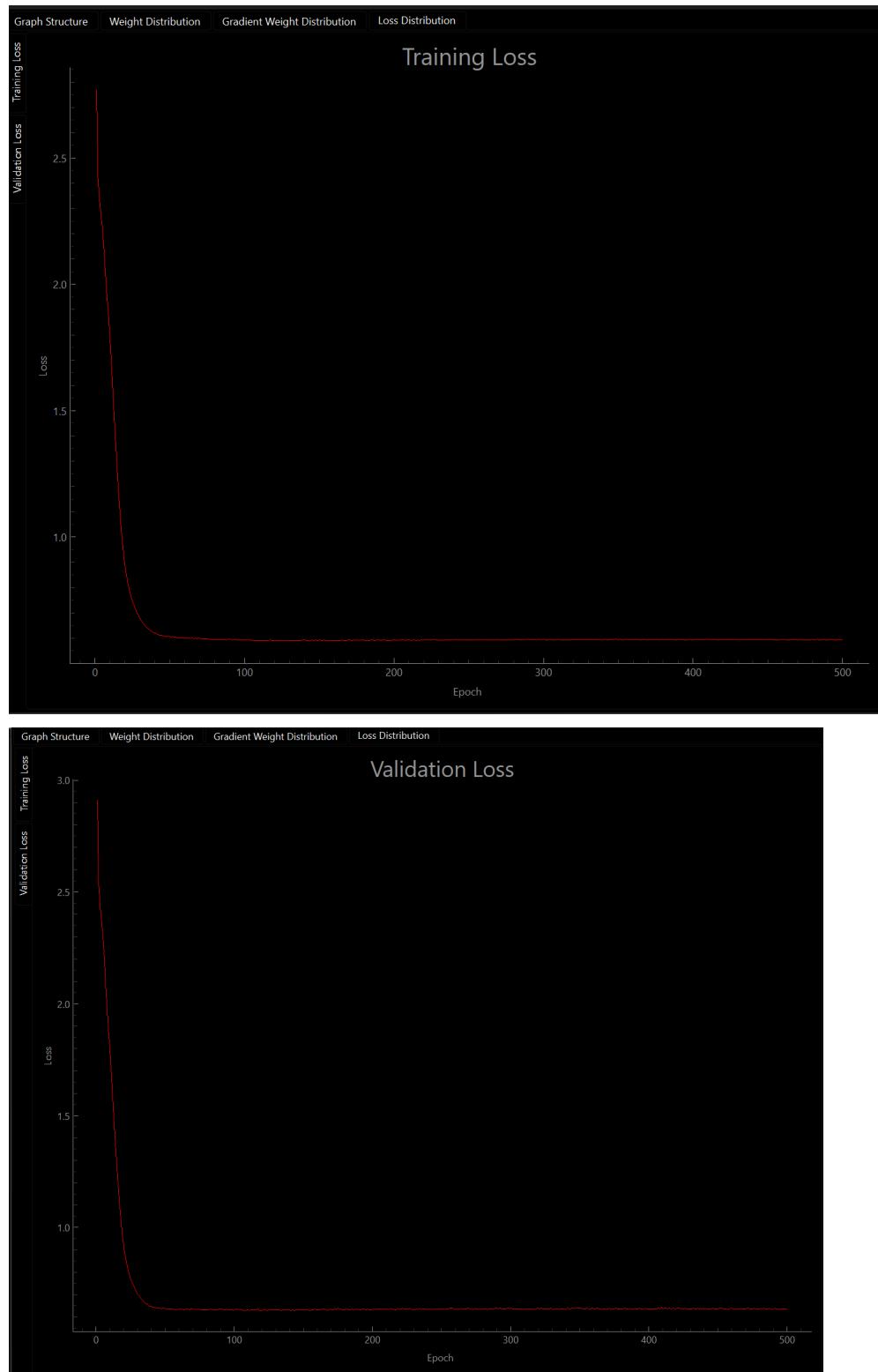


b. Random dengan Distribusi Uniform

```
[-----] - 500/500 - 100.0% - 35.84 s - Loss: 2.085923  
Loss: 2.085923  
Training time: 35.84 seconds  
Model berhasil disimpan ke ffnn_model.pkl
```

Model F1-Score: 0.0959
Model Accuracy: 0.2075

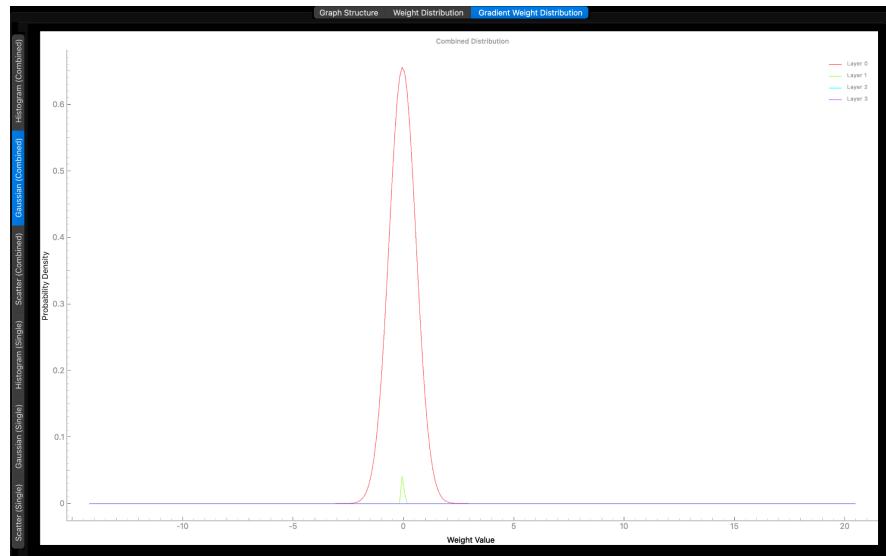
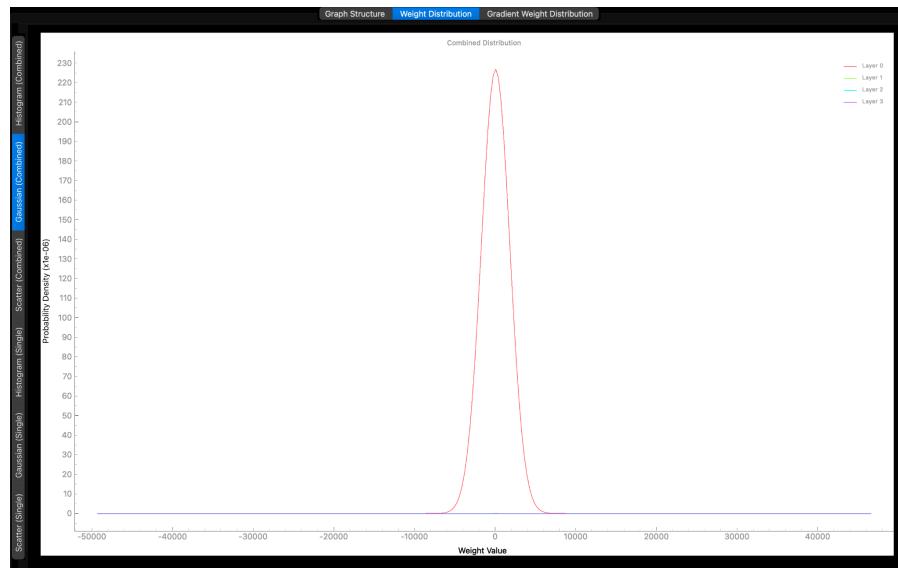


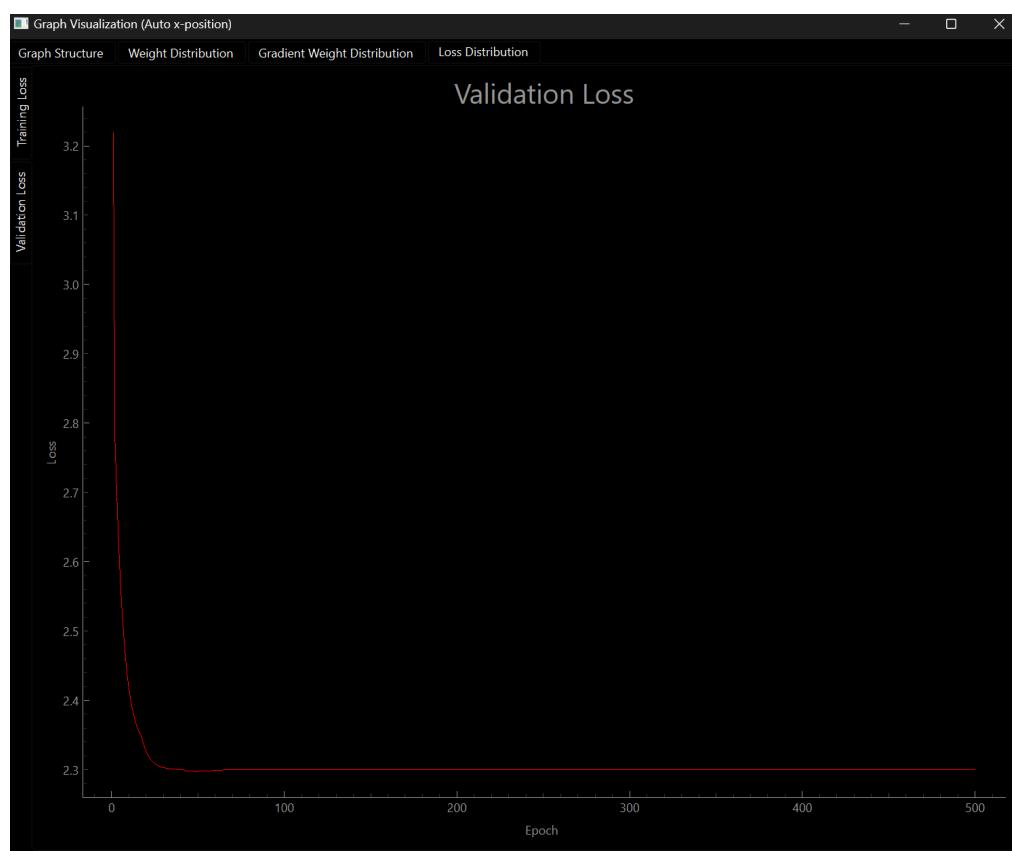
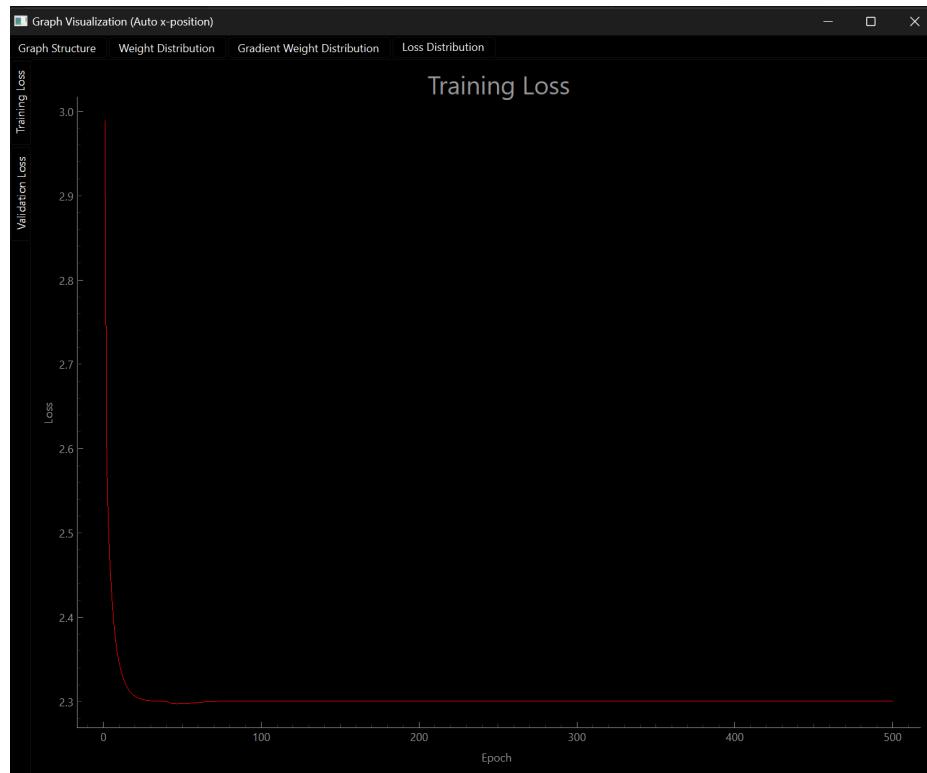


c. Random dengan Distribusi Normal

```
[-----] - 500/500 - 100.0% - 35.50 s - Loss: 1.093076  
Loss: 1.093076  
Training time: 35.50 seconds  
Model berhasil disimpan ke ffnn_model.pkl
```

Model F1-Score: 0.6692
Model Accuracy: 0.6696





d. Hasil Analisis

Berdasarkan hasil pengujian di atas, inisialisasi bobot Zero initialization menghasilkan performa yang sangat buruk dengan F1-score sebesar 0.0986 dan akurasi 0.2079. Hal ini kemungkinan besar terjadi karena bobot yang diinisialisasi dengan nol menyebabkan semua neuron dalam lapisan yang sama mendapatkan gradien yang identik, sehingga tidak ada perbedaan dalam pembaruan bobot, yang bisa menghambat proses pembelajaran. Distribusi bobot dan gradien yang ditampilkan menunjukkan bahwa mayoritas bobot tetap mendekati nol tanpa adanya variasi yang cukup untuk mendukung pembelajaran dengan baik.

Sementara itu, inisialisasi bobot rando dengan distribusi uniform menghasilkan performa yang hampir sama buruknya dengan F1-score sebesar 0.0959 dan akurasi 0.2075. Distribusi bobot awalnya lebih tersebar dibandingkan dengan zero initialization, tetapi distribusi gradien menunjukkan bahwa banyak bobot tetap dalam rentang kecil tanpa penyebaran yang cukup luas. Hal ini menunjukkan bahwa meskipun ada sedikit variasi dalam bobot awal, pembelajaran model tetap tidak optimal karena bobot awal tidak memberikan informasi yang cukup kaya untuk mendorong pembelajaran yang lebih efektif.

Di sisi lain, inisialisasi bobot dengan distribusi normal menunjukkan hasil yang jauh lebih baik dengan F1-score sebesar 0.6692 dan akurasi 0.6696. Hal ini menunjukkan bahwa distribusi normal memberikan bobot awal yang lebih baik dalam menangkap pola dalam data. Distribusi bobot dan gradien memperlihatkan bahwa model dapat melakukan pembaruan bobot yang lebih efektif selama proses pelatihan. Persebaran gradien yang lebih baik ini memungkinkan pembelajaran yang lebih stabil dan memungkinkan model untuk lebih cepat mencapai konvergensi.

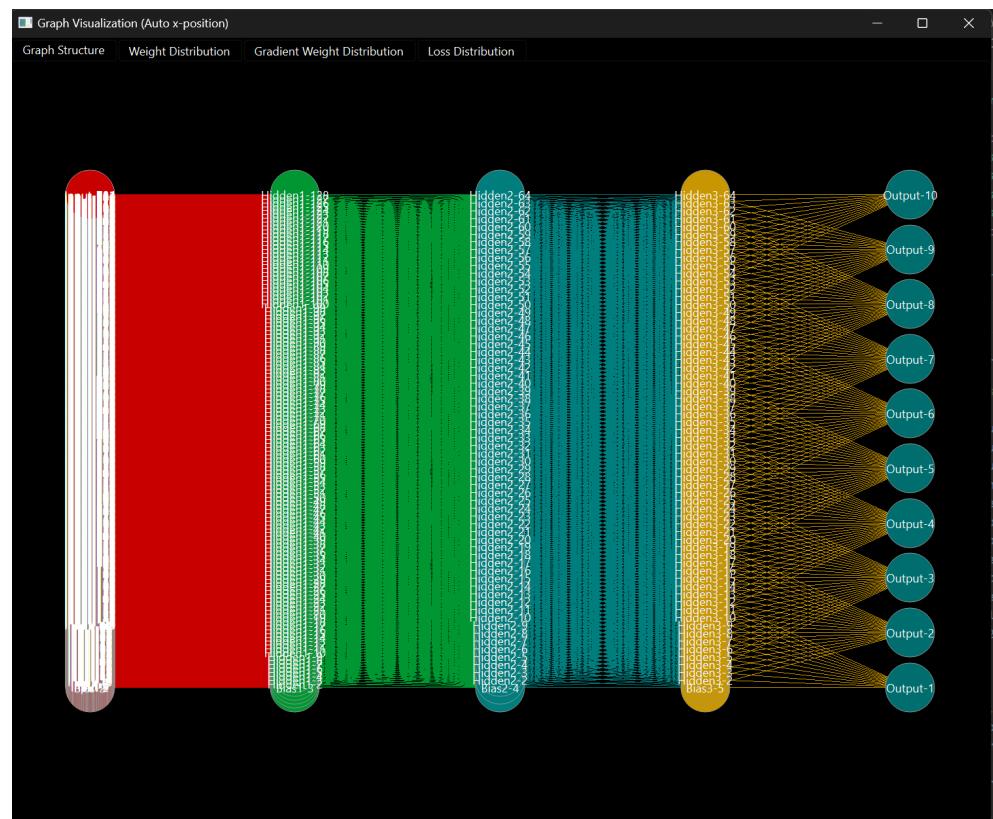
Dari grafik training loss dan validation loss pada ketiga metode inisialisasi, terlihat bahwa semua metode mengalami penurunan nilai loss seiring bertambahnya epoch, yang mengindikasikan proses pembelajaran tetap berjalan. Namun, pada inisialisasi zero dan random uniform, meskipun loss terlihat turun tajam di awal dan cenderung stabil di akhir pelatihan, performa model tetap sangat rendah. Hal ini menunjukkan bahwa meskipun nilai loss terlihat rendah, bukan berarti model mampu memahami data dengan baik, apalagi jika bobot awalnya tidak mendukung proses belajar fitur-fitur penting dari data. Sebaliknya, inisialisasi dengan distribusi normal tidak hanya menunjukkan penurunan loss yang serupa, tetapi juga menghasilkan performa evaluasi yang jauh lebih tinggi. Hal ini menunjukkan bahwa distribusi bobot awal yang baik memungkinkan model untuk belajar representasi yang lebih banyak, yang tercermin dari nilai training loss dan validation loss yang rendah serta hasil evaluasi yang lebih baik.

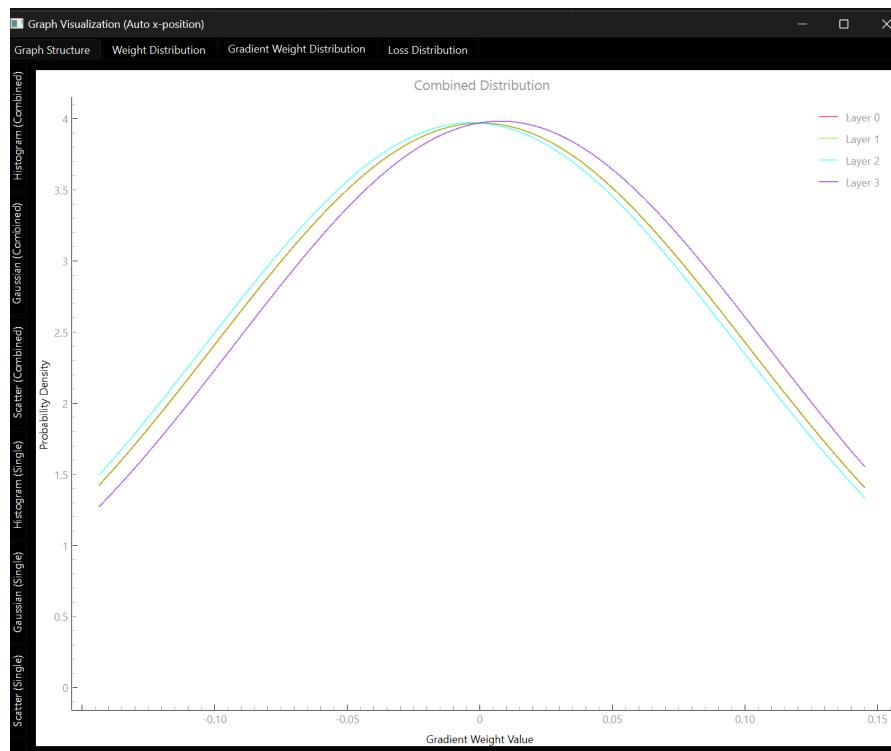
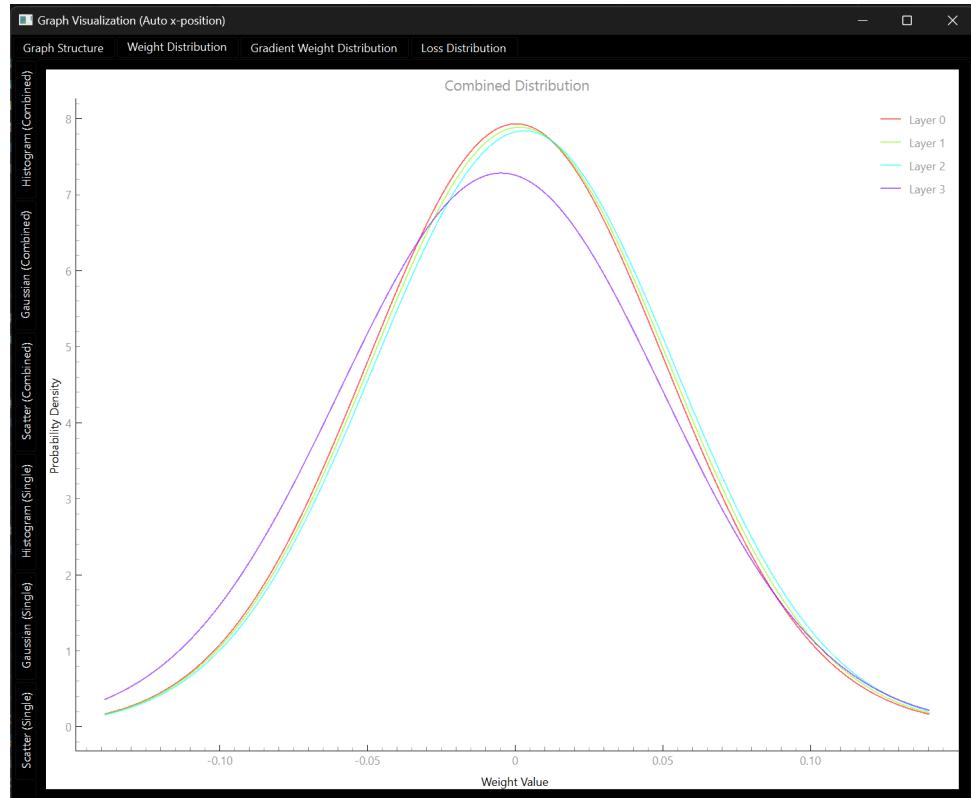
2.2.5 Pengaruh regularisasi

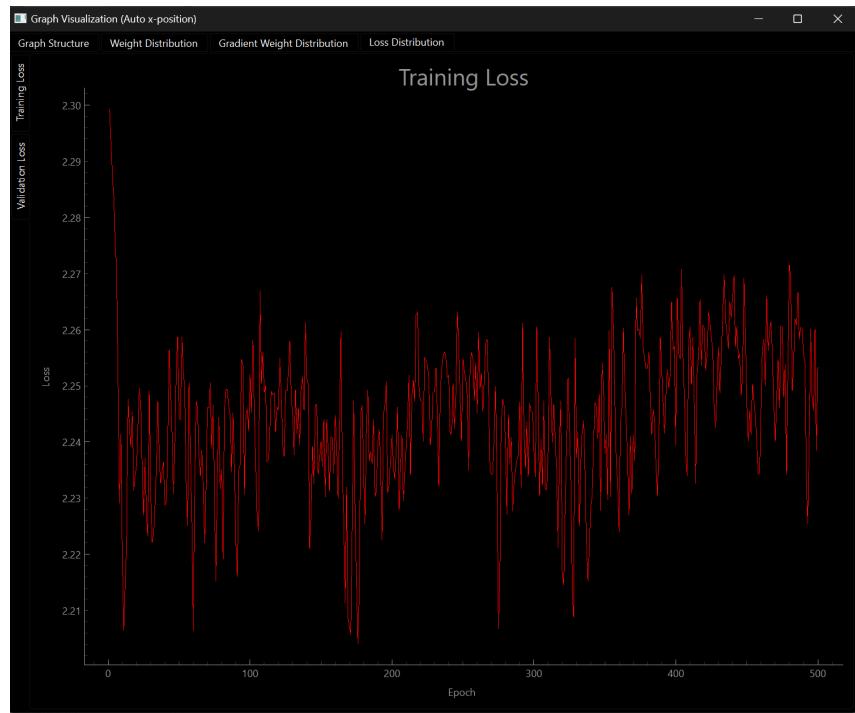
a. Regularisasi L1

```
[ ] - 500/500 - 100.0% - 173  
Final Training Loss: 2.253272  
Final Validation Loss: 2.248702  
Training time: 173.35 seconds
```

Model F1-Score: 0.1017
Model Accuracy: 0.1429



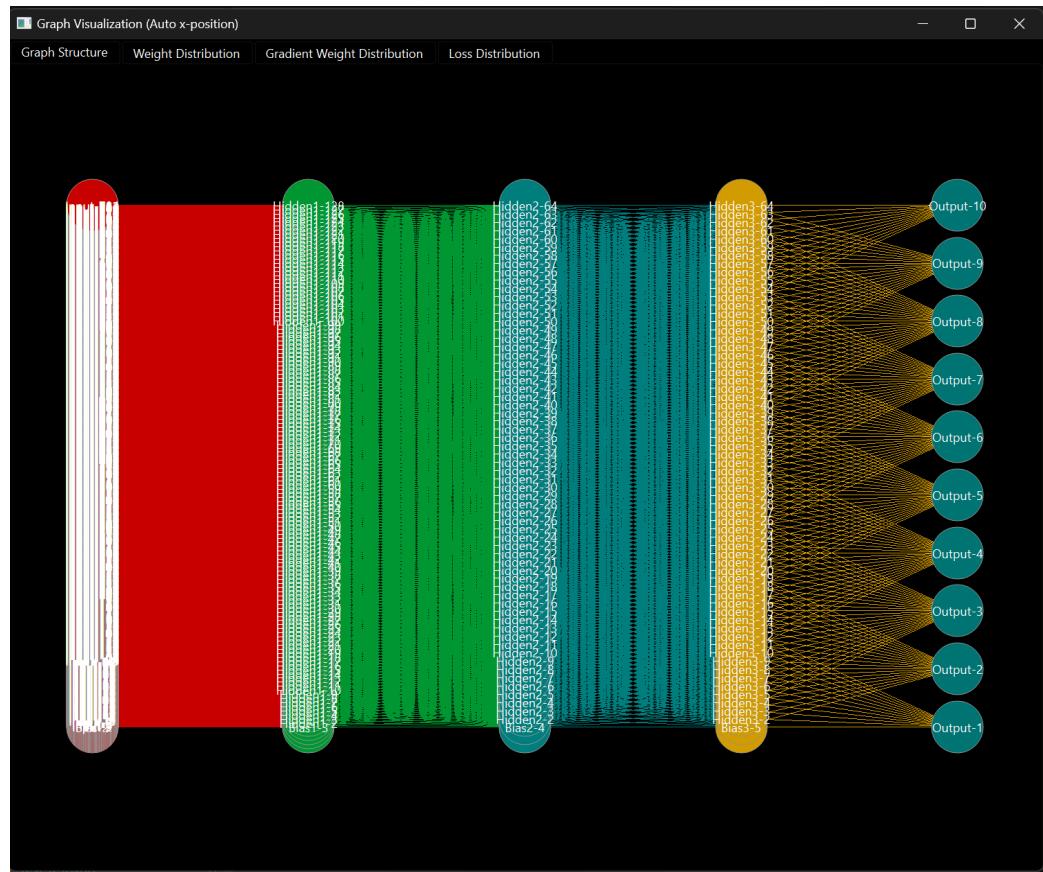


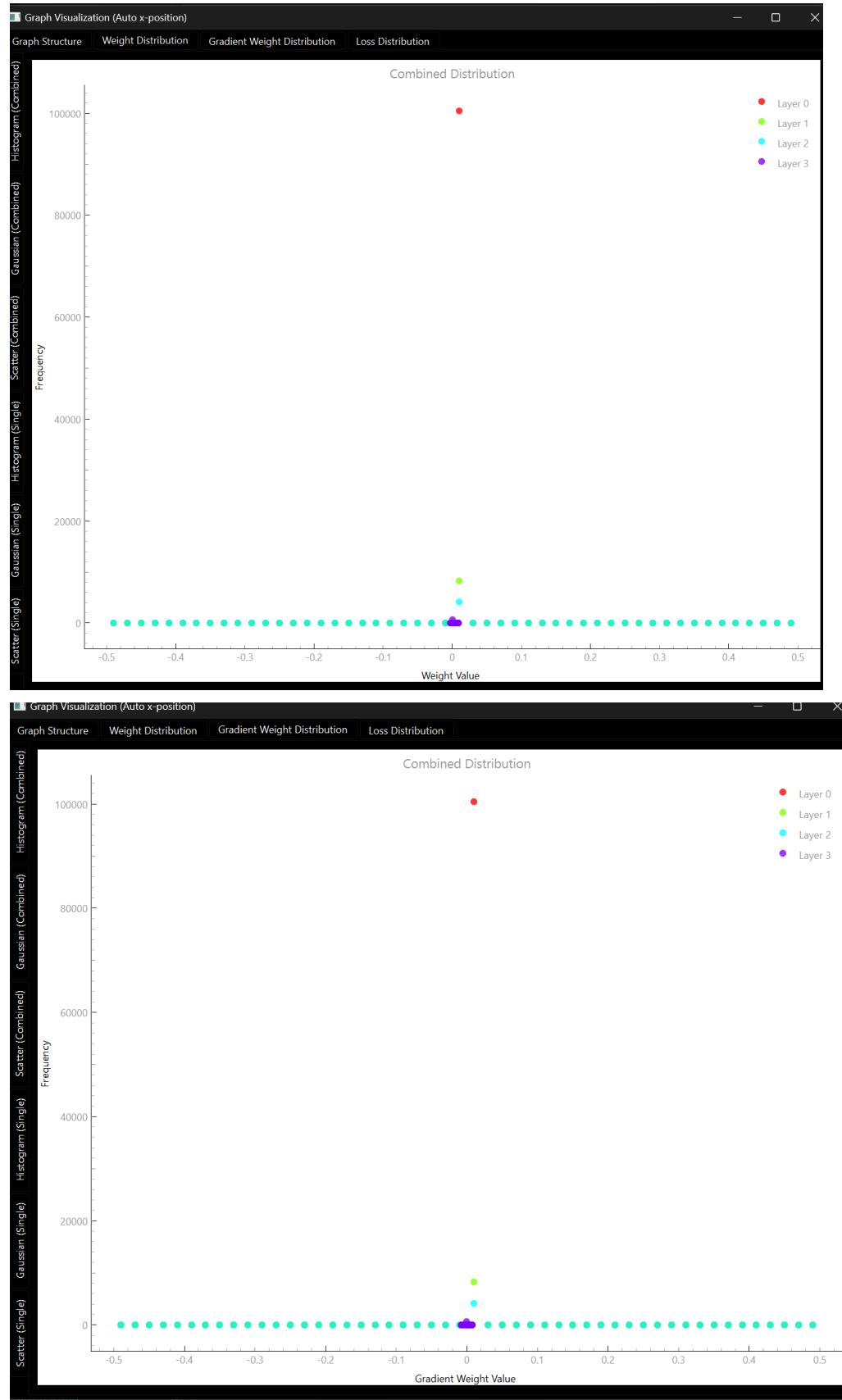


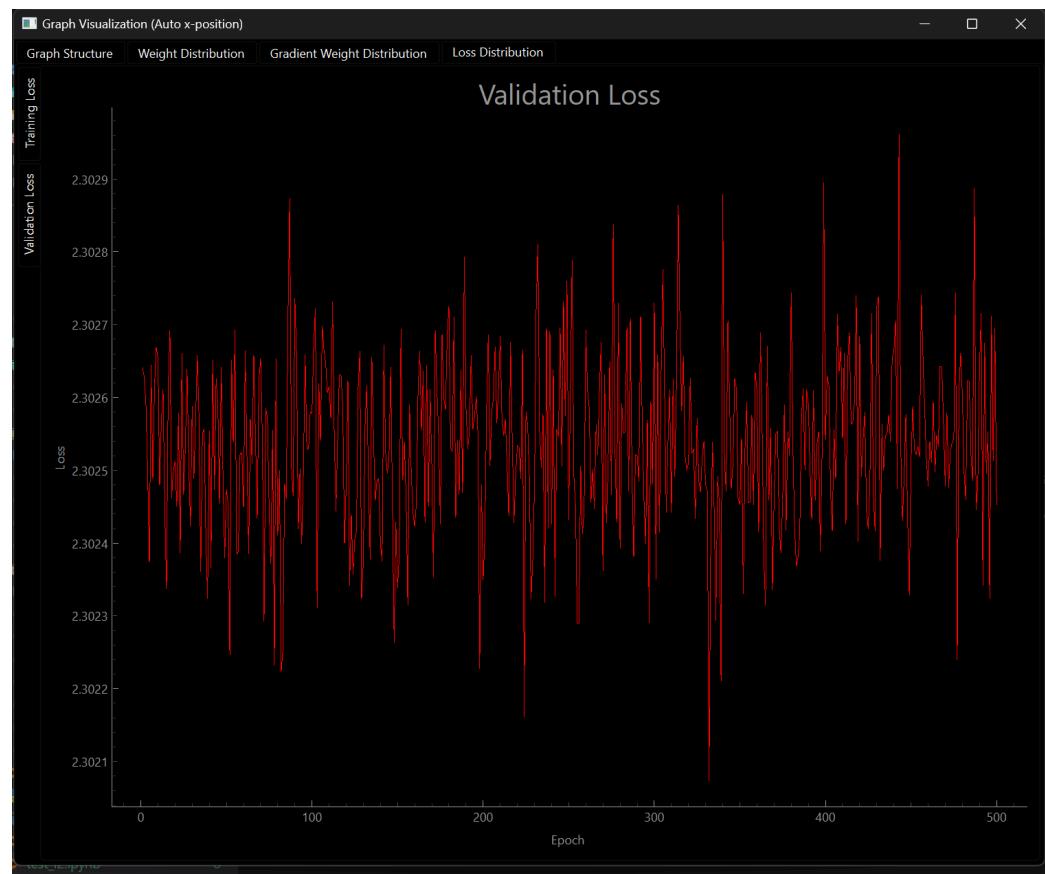
b. Regularisasi L2

```
[██████████] - 500/500 - 100.0% - 176.16s - Tr
Final Training Loss: 2.302405
Final Validation Loss: 2.302453
Training time: 176.16 seconds
```

Model F1-Score: 0.0200
Model Accuracy: 0.1113







c. Hasil Analisis

Berdasarkan hasil pengujian di atas, regularisasi L1 menghasilkan performa yang sedikit lebih baik dibandingkan dengan L2, dengan F1-score sebesar 0.1017 dan akurasi 0.1429. Meskipun nilai ini masih tergolong sangat rendah, namun menunjukkan bahwa regularisasi L1 sedikit lebih efektif dalam membantu model mengenali pola dalam data. Salah satu penyebabnya adalah sifat dari L1 yang mendorong *sparsity*, yaitu membuat sebagian besar bobot menjadi nol sehingga hanya fitur-fitur penting yang dipertahankan. Hal ini terlihat pada distribusi bobot yang memperlihatkan bahwa banyak bobot berada di sekitar angka nol, hal ini mencerminkan proses regularisasi yang cukup kuat dalam menyaring fitur.

Distribusi gradien pada model dengan regularisasi L1 menunjukkan variasi loss yang cukup tinggi, yang berarti pembaruan bobot terjadi secara agresif, terutama pada bobot yang tidak terlalu berkontribusi terhadap loss. Namun, agresivitas ini juga membuat proses pelatihan menjadi tidak stabil, terlihat dari training loss dan validation loss yang menunjukkan pola naik turun tanpa pola konvergensi yang jelas. Tetapi, efek sparsity dari L1 masih memberikan sedikit peningkatan performa dibandingkan L2.

Di sisi lain, regularisasi L2 memberikan hasil yang lebih rendah, dengan F1-score sebesar 0.0200 dan akurasi 0.1111. L2 bekerja dengan mengecilkan semua bobot secara proporsional, tanpa mendorong bobot ke nol. Akibatnya, fitur-fitur yang sebenarnya tidak penting tetap dipertahankan, meskipun dengan kontribusi kecil. Hal ini terlihat pada distribusi bobot yang lebih merata dan tidak terlalu terkonsentrasi di sekitar nol. Distribusi gradien pun tampak lebih smooth dan stabil, namun kurang selektif terhadap bobot-bobot yang tidak penting.

Training loss dan *validation loss* pada regularisasi L2 juga menunjukkan pola yang fluktuatif (variasi loss yang tinggi), mirip dengan L1, namun tidak disertai peningkatan signifikan pada metrik evaluasi. Ini menunjukkan bahwa meskipun model mampu melakukan pembaruan bobot secara konsisten, kemampuan untuk mempelajari fitur-fitur yang penting tetap rendah, kemungkinan karena bobot awal atau arsitektur model yang tidak optimal untuk data yang digunakan.

Dari keseluruhan hasil, dapat disimpulkan bahwa regularisasi L1 lebih berhasil dalam menyaring fitur dan sedikit meningkatkan performa model dibandingkan L2. Namun, performa keduanya secara umum masih tergolong sangat rendah. Hal ini menunjukkan bahwa peran regularisasi saja tidak cukup untuk menghasilkan model yang baik, diperlukan juga strategi inisialisasi bobot, preprocessing data, dan pemilihan arsitektur model yang lebih tepat untuk mendorong proses pembelajaran yang optimal.

2.2.5 Perbandingan dengan library sklearn

a. From Scratch (Model Accuracy), Tensor Accuracy, dan Sklearn Accuracy

```
[-----] - 500/500 - 100.0% - 34.70 s - Loss: 1.088230
Loss: 1.088230
Training time: 34.70 seconds
Model berhasil disimpan ke ffnn_model.pkl
Model Accuracy: 0.7120
313/313 0s 693us/step - accuracy: 0.9451 - loss: 0.4237
Tensor Accuracy: 0.9476
SKLearn Accuracy: 0.9152
```

b. Hasil Analisis

Dari hasil pengujian, ada perbedaan akurasi kurang lebih 20% antara implementasi *from scratch* dengan library yang sudah ada. Hal ini bisa terjadi karena beberapa faktor. Salah satunya adalah cara masing-masing framework menangani data dan melakukan perhitungan. TensorFlow dan Scikit-learn punya optimasi bawaan yang membuat pengolahan data dan perhitungan matematis lebih efisien dibandingkan dengan implementasi manual, sehingga hasilnya lebih stabil dan lebih sedikit kesalahan numerik.

Selain itu, terkait inisialisasi bobot, setiap framework punya cara sendiri untuk menentukan bobot awal agar model lebih stabil. Perbedaan lain juga terlihat di cara fungsi aktivasi dan proses backpropagation diterapkan. Contohnya, TensorFlow menggunakan teknik otomatis untuk menghitung turunan (automatic differentiation), yang membuat perhitungan gradien lebih akurat dan mengurangi kemungkinan error akibat keterbatasan angka desimal dalam komputasi.

Selain itu, cara training dilakukan juga bisa memengaruhi hasil. TensorFlow dan Scikit-learn sudah punya optimasi untuk pengolahan batch, yang membantu mempercepat dan menstabilkan pembaruan bobot model. Misalnya, Scikit-learn dengan MLPClassifier menggunakan metode batch yang sudah diatur agar lebih efisien, sedangkan implementasi *from scratch* bisa jadi sedikit berbeda dalam cara memproses batch, sehingga hasil akhirnya pun tidak sama. Jadi, meskipun semua hyperparameter sudah dibuat sama, perbedaan dalam optimasi komputasi, stabilitas angka, dan cara pembaruan bobot di tiap framework bisa membuat hasil akurasi berbeda.

BAB 3 KESIMPULAN DAN SARAN

3.1 Kesimpulan

Secara keseluruhan, hasil analisis menunjukkan bahwa model dengan satu hidden layer berukuran 128 memiliki performa terbaik dengan F1-score 0.9193, sedangkan model yang lebih dalam mengalami penurunan akibat vanishing gradient. Dari segi lebar (width) jaringan, model dengan struktur 128-128-128 mencapai F1-score tertinggi (0.9319), menunjukkan bahwa menambah jumlah neuron lebih efektif dibanding memperdalam jaringan. Fungsi aktivasi ReLU memberikan keseimbangan terbaik antara akurasi, waktu pelatihan, dan kestabilan gradien, sementara fungsi Linear, meskipun awalnya memiliki skor tinggi, tidak stabil untuk pelatihan jangka panjang. Selain itu, learning rate 0.5 menghasilkan performa terbaik karena memungkinkan pembaruan bobot yang lebih stabil dibandingkan learning rate yang terlalu kecil atau terlalu besar.

Dari segi inisialisasi bobot, distribusi normal menunjukkan hasil terbaik dengan F1-score 0.6692 dibandingkan zero initialization atau distribusi uniform, yang menyebabkan kesulitan dalam pembelajaran. Perbandingan dengan framework seperti Scikit-learn dan TensorFlow menunjukkan bahwa implementasi dari library ini memiliki akurasi sekitar 20% lebih tinggi dibandingkan model *from scratch*, kemungkinan karena optimasi bawaan dalam pengolahan data dan komputasi numerik. Faktor seperti inisialisasi bobot, metode backpropagation, dan cara pengolahan batch juga mempengaruhi perbedaan hasil ini, dan menjadi latar belakang mengapa model yang dibuat *from scratch* terkadang kurang optimal dibandingkan framework yang telah dioptimasi dengan baik.

3.1 Saran

Untuk meningkatkan tingkat akurasi agar lebih mendekati hasil dari TensorFlow dan Scikit-learn, berikut beberapa saran untuk pengembangan implementasinya:

1. Menggunakan teknik inisialisasi bobot yang lebih baik dengan mengadopsi metode seperti Xavier atau He initialization untuk membantu distribusi bobot yang lebih optimal sejak awal, mengurangi risiko vanishing atau exploding gradient.
2. Menggunakan optimasi backpropagation yang lebih efisien dengan mengadopsi algoritma optimasi seperti Adam atau RMSprop untuk meningkatkan kecepatan dan stabilitas pembelajaran dibandingkan dengan metode dasar seperti stochastic gradient descent (SGD).
3. Menggunakan automatic differentiation yaitu mengimplementasikan perhitungan gradien secara otomatis seperti yang digunakan dalam TensorFlow untuk meningkatkan akurasi perhitungan dan stabilitas dalam pembaruan bobot.

Dengan saran-saran ini, diharapkan implementasi *from scratch* dapat lebih optimal dalam menangani pelatihan *artificial neural network* dan mendekati performa *framework* yang telah dioptimasi.

Pembagian Tugas

Tugas	Anggota
-------	---------

Forward Propagation	13522164 - Valentino Chryslie Triadi
Backward Propagation	13522134 - Shabrina Maharani
Loss Function	13522157 - Muhammad Davis Adhipramana 13522134 - Shabrina Maharani
Activation Function	13522164 - Valentino Chryslie Triadi 13522134 - Shabrina Maharani
Fit, Predict, Save, dan Load	13522164 - Valentino Chryslie Triadi 13522134 - Shabrina Maharani 13522157 - Muhammad Davis Adhipramana
Visualisasi Graf, Distribusi Bobot, Distribusi Gradien, dan grafik <i>training loss</i> dan <i>validation loss</i>	13522157 - Muhammad Davis Adhipramana
Weight Initiation	13522164 - Valentino Chryslie Triadi
Regularisasi	13522134 - Shabrina Maharani 13522157 - Muhammad Davis Adhipramana 13522164 - Valentino Chryslie Triadi

Referensi

Mitchell, T. (1997). *Machine Learning*. McGraw Hill.

PPT Pembelajaran Mata Kuliah Pembelajaran Mesin Tahun 2025

Raschka, S., et al. (2022). *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing Ltd. (Chapter 11).

Osajima, J. (2018). *Forward Propagation*. Diakses dari <https://www.jasonosajima.com/forwardprop>

Osajima, J. (2018). *Forward Propagation*. Diakses dari <https://www.jasonosajima.com/forwardprop>

Green, E. (2016). *The Softmax Function and Its Derivative*. Diakses dari <https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>

Rosebrock, A. (2021). *Implementing Feedforward Neural Networks with Keras and TensorFlow*. Diakses dari <https://pyimagesearch.com/2021/05/06/implementing-feedforward-neural-networks-with-keras-and-tensorflow/>