

**LAPORAN TUGAS BESAR I**  
**IF3270 PEMBELAJARAN MESIN**  
**FEEDFORWARD NEURAL NETWORK**

Disusun untuk memenuhi tugas mata kuliah Pembelajaran Mesin  
pada Semester II Tahun Akademik 2024/2025



Oleh Kelompok 42:

Thea Josephine Halim	13522012
Raffael Boymian Siahaan	13522046

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

## DAFTAR ISI

<b>DAFTAR ISI</b>	<b>1</b>
<b>BAB I</b>	
DESKRIPSI MASALAH	3
<b>BAB II</b>	
LANDASAN TEORI	5
2.1. Dasar Teori	5
2.1.1. Fungsi Aktivasi	5
2.1.1.1. Fungsi Aktivasi Linear	5
2.1.1.2. Fungsi Aktivasi ReLU (Rectified Linear Unit)	6
2.1.1.3. Fungsi Aktivasi Sigmoid	7
2.1.1.4. Fungsi Aktivasi Hyperbolic Tanh	8
2.1.1.5. Fungsi Aktivasi Softmax	9
2.1.1.6. Fungsi Aktivasi Softplus	10
2.1.1.7. Fungsi Aktivasi Swish	10
2.1.1.8. Fungsi Aktivasi ELU	11
2.1.2. Fungsi Loss	11
2.1.3. Inisialisasi Berat dan Bias	12
2.1.4. L1 dan L2 Regularization	13
2.1.4.1. L1 (LASSO Regression)	13
2.1.4.2. L2 (Ridge Regression)	14
2.1.5. Normalisasi RMSNorm	14
2.1.6. FFNN	14
<b>BAB III</b>	
IMPLEMENTASI DAN PENGUJIAN	15
3.1. Implementasi Program	15
3.1.1. Utils.py	15
3.2. Hasil Pengujian	18
3.2.1. Variasi Depth dan Width	18
3.2.2. Variasi Fungsi Aktivasi	21
3.2.3. Variasi Learning Rate	26
3.2.4. Variasi Metode Inisialisasi Bobot	32
3.2.5. Pengujian Regularisasi L1 dan L2	34
3.2.6. Pengujian dengan RMSNorm	35
3.2.7. Pengujian Berbanding library MLP SkLearn	36
<b>BAB IV</b>	
ANALISIS	37
4.1. Pengaruh depth dan width	37
4.2. Pengaruh fungsi aktivasi	37

4.3. Pengaruh learning rate	38
4.4. Pengaruh inisialisasi bobot	38
4.5. Pengaruh Regularisasi L1 dan L2	38
4.6. Perbandingan Normalisasi RMSNorm	38
4.7. Perbandingan dengan library sklearn	38
<b>BAB V</b>	<b>40</b>
<b>KESIMPULAN DAN SARAN</b>	<b>40</b>
5.1. Kesimpulan	40
5.2. Saran	40
<b>PEMBAGIAN TUGAS</b>	<b>41</b>
<b>LAMPIRAN</b>	<b>41</b>
Repository	41
<b>DAFTAR PUSTAKA</b>	<b>41</b>

## BAB I

### DESKRIPSI MASALAH

Dalam pengembangan model kecerdasan buatan, khususnya neural network, Feedforward Neural Network (FFNN) menjadi salah satu arsitektur yang paling banyak digunakan karena kesederhanaannya dan kemampuannya dalam menangani berbagai tugas klasifikasi maupun regresi. FFNN bekerja dengan mengalirkan data secara searah dari input layer ke output layer melalui satu atau lebih hidden layer, di mana setiap neuron dalam jaringan memiliki bobot dan bias yang diperbarui selama proses pelatihan.

Ada beberapa parameter pilihan penting yang dapat mempengaruhi performansi dari model, seperti:

- **Pemilihan Fungsi Aktivasi:** Fungsi aktivasi memainkan peran penting dalam memperkenalkan non-linearitas ke dalam jaringan. Oleh karena itu, perlu dianalisis bagaimana berbagai fungsi aktivasi (Linear, ReLU, Sigmoid, Tanh, dan Softmax) mempengaruhi performa model.
- **Pemilihan Fungsi Loss:** Fungsi loss menentukan bagaimana kesalahan dihitung selama pelatihan. Perbedaan antara Mean Squared Error (MSE), Binary Cross-Entropy, dan Categorical Cross-Entropy dapat berdampak pada konvergensi model dan akurasi prediksi.
- **Inisialisasi Bobot:** Bobot awal jaringan sangat mempengaruhi jalannya pelatihan. Oleh karena itu, perlu dilakukan eksperimen dengan berbagai metode inisialisasi, termasuk zero initialization, random uniform, dan random normal, untuk melihat pengaruhnya terhadap kinerja model.
- **Pengaruh Hyperparameter:** Faktor seperti jumlah neuron per layer (*width*), jumlah hidden layer (*depth*), serta learning rate dapat mempengaruhi akurasi dan stabilitas pelatihan model. Oleh karena itu, diperlukan analisis terhadap kombinasi hyperparameter untuk menentukan konfigurasi optimal.

Pada tugas besar ini, akan dikembangkan modul FFNN yang fleksibel dan dapat dikonfigurasi dengan berbagai pilihan mulai dari fungsi aktivasi, fungsi loss, metode inisialisasi

bobot, hingga parameter pelatihan. Implementasi ini juga akan dianalisis untuk memahami bagaimana setiap komponen memengaruhi performa model secara keseluruhan dalam dataset MNIST 784.

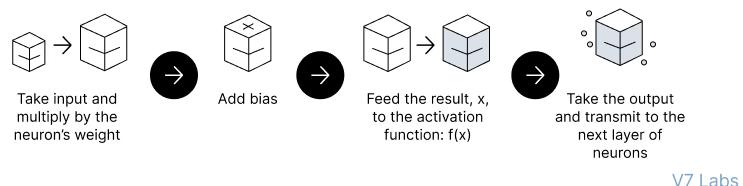
## BAB II

### LANDASAN TEORI

#### 2.1. Dasar Teori

##### 2.1.1. Fungsi Aktivasi

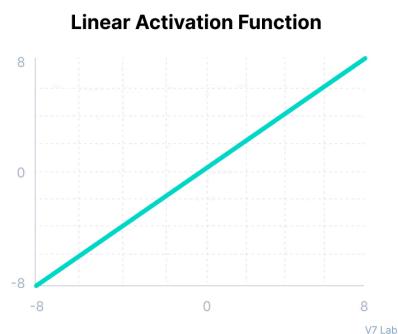
Fungsi aktivasi adalah fungsi untuk menentukan apakah suatu neuron teraktivasi atau tidak dengan cara membatasi nilai output menjadi suatu range nilai. Fungsi aktivasi pada tugas besar ini dapat dibagi menjadi 2: fungsi aktivasi linear dan non linear. Fungsi aktivasi linear, benar-benar proporsional pada input, tidak mengubah nilainya dan hanya mengeluarkan nilai input awal sebagai output. Sedangkan fungsi aktivasi non linear akan memperkenalkan non linearitas pada neural network. Non linearitas ini penting untuk memodelkan masalah yang kompleks dan menggeneralisasinya menjadi pola tertentu. Tanpa adanya non linearitas ini semua neuron akan berlaku sama, walaupun sebagai *trade off*-nya akan menambah waktu komputasi.



Sumber: <https://www.v7labs.com/>

Akan ada beberapa fungsi aktivasi yang digunakan pada tugas besar ini:

###### 2.1.1.1. Fungsi Aktivasi Linear



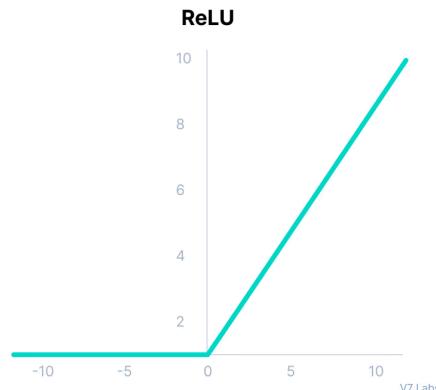
Sumber: <https://www.v7labs.com>

$$\text{Linear}(x) = x$$

$$\frac{d(\text{Linear}(x))}{dx} = 1$$

Fungsi yang tidak mengubah inputnya, cocok untuk regresi linear atau pada layer output dalam jaringan regresi. Penggunaan linear activation function ini berarti kita tidak bisa melakukan tahap backpropagation, sebab hasil penurunan fungsi linear adalah konstanta 1 yang tidak berkaitan dengan nilai x. Nilai output yang sama dengan input juga membuat jumlah layer tidak berdampak karena nilai setiap layer akan tetap sama dengan layer pertama.

#### 2.1.1.2. Fungsi Aktivasi ReLU (Rectified Linear Unit)



Sumber: <https://www.v7labs.com>

$$\text{ReLU}(x) = \max(0, x)$$

$$\frac{d(\text{ReLU}(x))}{dx} = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$

ReLU activation function tidak akan mengaktifasi neuron sekaligus. Neuron akan tidak akan diaktifasi apabila inputnya bernilai negatif dan langsung konversi menjadi 0. Seleksi aktivasi neuron ini memberikan ReLU kelebihan untuk menghemat waktu komputasi. Walaupun begitu, muncul masalah apabila terus-menerus mendapat gradien 0 (input negatif), sebab ketika fase backpropagation nilai bobot dan bias beberapa neuron tidak bisa diupdate, menjadikannya tidak aktif. Masalah ini juga disebut Dying ReLU Problem.

### The Dying ReLU problem



Untuk mengatasi masalah tersebut muncullah beberapa variasi fungsi aktivasi ReLU lainnya, seperti Leaky ReLU, yang tetap memberikan sedikit gradien apabila input negatif, dan menghilangkan masalah neuron tidak aktif.

#### 2.1.1.3. Fungsi Aktivasi Sigmoid



Sumber: <https://www.v7labs.com>

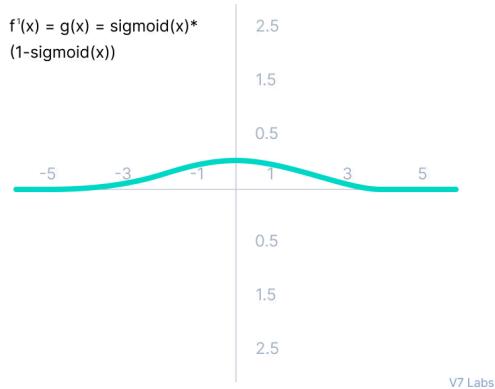
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d(\sigma(x))}{dx} = \sigma(x)(1 - \sigma(x))$$

Semakin tinggi nilai input positif, nilai sigmoidnya semakin mendekati 1, sebaliknya apabila semakin negatif nilai sigmoidnya semakin mendekati 0. Fungsi sigmoid selalu memberikan output antara 0 dan 1, dan memberikan gradien yang mulus tanpa adanya lompatan nilai output.

Ada kalanya fungsi aktivasi sigmoid memunculkan masalah Vanishing gradient problem. Vanishing gradient problem adalah kondisi ketika gradien dari

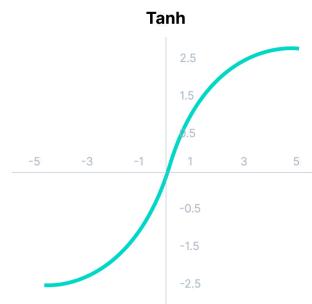
fungsi aktivasi menjadi sangat kecil saat melewati beberapa layer. Hal ini menyebabkan update bobot di lapisan awal menjadi sangat kecil atau hampir tidak berubah selama proses pelatihan menggunakan *backpropagation*. Akibatnya, neural network sulit untuk belajar pola kompleks dan konvergensi menjadi sangat lambat atau bahkan terhenti. Contohnya pada gambar di bawah, gradien untuk input yang melebihi antara 3 dan -3 semakin mengecil mendekati nol.



Turunan Sigmoid Mengalami Vanishing Gradient Problem

Sumber: <https://www.v7labs.com>

#### 2.1.1.4. Fungsi Aktivasi Hyperbolic Tanh



Sumber: <https://www.v7labs.com>

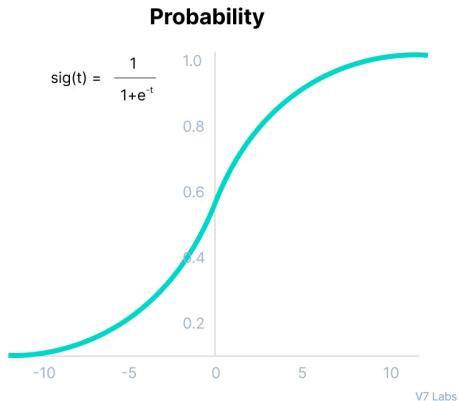
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\frac{d(\tanh(x))}{dx} = \left( \frac{2}{e^x - e^{-x}} \right)^2$$

Fungsi aktivasi tanh membatasi output di antara -1 dan 1. Outputnya terpusat pada 0 dan seringnya digunakan dalam hidden layer, membuat nilai

rata-rata di sekitar 0 dan *centering* data. Sama seperti sigmoid, fungsi tanh mengalami vanishing gradient problem.

### 2.1.1.5. Fungsi Aktivasi Softmax



Sumber: <https://www.v7labs.com>

$$\text{Untuk vector } \vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n, \\ \text{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

Untuk vector  $\vec{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$ ,

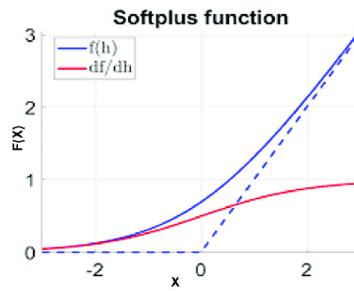
$$\frac{d(\text{softmax}(\vec{x}))_i}{d\vec{x}} = \begin{bmatrix} \frac{\partial(\text{softmax}(\vec{x}))_1}{\partial x_1} & \dots & \frac{\partial(\text{softmax}(\vec{x}))_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial(\text{softmax}(\vec{x}))_n}{\partial x_1} & \dots & \frac{\partial(\text{softmax}(\vec{x}))_n}{\partial x_n} \end{bmatrix}$$

Dimana untuk  $i, j \in \{1, \dots, n\}$ ,

$$\frac{\partial(\text{softmax}(\vec{x}))_i}{\partial x_j} = \text{softmax}(\vec{x})_i (\delta_{i,j} - \text{softmax}(\vec{x})_j) \\ \delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

Fungsi softmax Mengubah vektor menjadi distribusi probabilitas (semua nilai positif dan jumlahnya 1). Softmax biasanya digunakan untuk klasifikasi multi-kelas dan lebih kompleks karena output satu neuron bergantung pada semua input

### 2.1.1.6. Fungsi Aktivasi Softplus



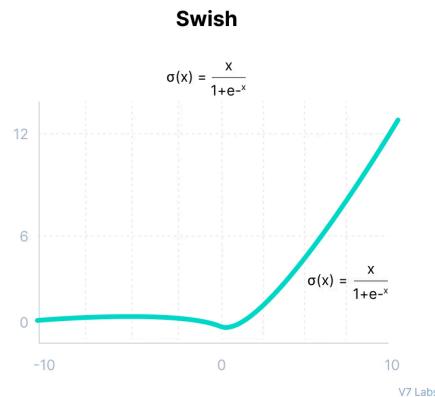
Sumber: <https://www.researchgate.net>

$$f(x) = \ln(1 + \exp(x))$$

$$f(x) = \sigma(x)$$

Softplus merupakan versi *smooth* dari ReLU sehingga gradiennya lebih stabil dalam pelatihan. Fungsi ini memastikan bahwa nilai output selalu positif tetapi tidak memiliki perubahan mendadak seperti ReLU di  $x = 0$ . Fungsi ini juga menghindari Dying ReLU Problem karena tidak ada bagian yang sepenuhnya nol.

### 2.1.1.7. Fungsi Aktivasi Swish



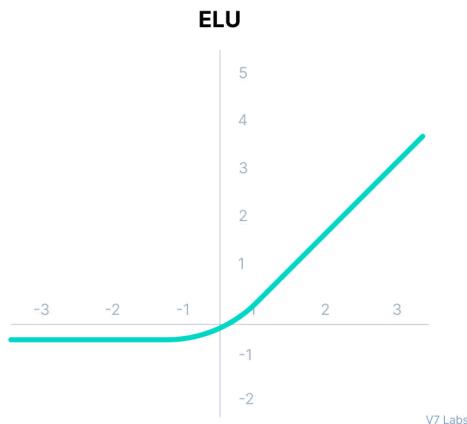
Sumber: <https://www.v7labs.com>

$$f(x) = x * \text{sigmoid}(x)$$

Swish adalah fungsi aktivasi yang dikembangkan Google. Swish punya batas bawah terbatas, tetapi tidak memiliki batas atas. Saat  $x$  menuju negatif tak hingga, output mendekati nilai konstan, sedangkan saat  $x$  menuju positif tak

hingga, output terus meningkat. Swish disebut lebih akurat daripada ReLU karena tidak mengalami perubahan arah secara tiba-tiba di sekitar nol seperti ReLU. Swish juga tetap mempertahankan nilai negatif kecil yang mungkin penting dalam menangkap pola data.

#### 2.1.1.8. Fungsi Aktivasi ELU



Sumber: <https://www.v7labs.com>

$$\begin{cases} x & \text{for } x \geq 0 \\ \alpha(e^x - 1) & \text{for } x < 0 \end{cases}$$

Exponential Linear Unit (ELU) adalah salah satu fungsi aktivasi yang diperkenalkan untuk mengatasi beberapa kelemahan dari fungsi ReLU (Rectified Linear Unit), terutama terkait dengan *Vanishing Gradient Problem* dan *Dying ReLU Problem*. Tidak seperti ReLU, yang menetapkan semua nilai negatif menjadi nol, ELU tetap memiliki output negatif untuk input negatif tetapi dengan nilai yang terbatas, sehingga dapat menghindari *Dying ReLU Problem*.

#### 2.1.2. Fungsi Loss

Fungsi loss dalam Feedforward Neural Network (FFNN) digunakan untuk mengukur seberapa baik model dalam memprediksi output yang diharapkan. Fungsi ini membantu dalam proses backpropagation untuk memperbarui bobot jaringan agar hasil prediksi semakin akurat. Berikut adalah tiga jenis fungsi loss yang akan diimplementasikan:

Nama Fungsi Loss	Deskripsi	Definisi Fungsi
<a href="#">MSE</a>	Mengukur kesalahan antara nilai yang diprediksi dan nilai sebenarnya dalam regresi	$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
<a href="#">Binary Cross-Entropy</a>	Untuk klasifikasi biner, mengukur perbedaan antara distribusi probabilitas prediksi dengan distribusi probabilitas sebenarnya	$\mathcal{L}_{BCE} = -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$ <p> <math>y_i</math> = Actual binary label (0 or 1)  <math>\hat{y}_i</math> = Predicted value of <math>y_i</math>  <math>n</math> = Batch size     </p>
<a href="#">Categorical Cross-Entropy</a>	Untuk klasifikasi multikelas	$\mathcal{L}_{CCE} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^C (y_{ij} \log \hat{y}_{ij})$ <p> <math>y_{ij}</math> = Actual value of instance <math>i</math> for class <math>j</math>  <math>\hat{y}_{ij}</math> = Predicted value of <math>y_{ij}</math>  <math>C</math> = Number of classes  <math>n</math> = Batch size     </p>

### 2.1.3. Inisialisasi Berat dan Bias

- Berat dan bias akan diinisialisasi sesuai dengan jumlah lapisan layer dan neuron. Inisialisasi yang digunakan ada 3 macam, dengan tambahan 2 sebagai implementasi bonus:
- Zero initialization
  - Random dengan distribusi uniform.

- Menerima parameter lower bound (batas minimal) dan upper bound (batas maksimal)
- Menerima parameter seed untuk reproducibility
- Random dengan distribusi normal.
  - Menerima parameter mean dan variance
  - Menerima parameter seed untuk reproducibility
- Xavier Initialization dengan distribusi normal  
Dilakukan dengan menyetel nilai limit dengan merata-ratakan Fin dan Fout bersama-sama dan kemudian mengambil akar kuadratnya.
- He Initialization dengan distribusi normal  
Biasa digunakan untuk melatih model neural network dengan layer yang dalam.  
Untuk distribusi normal akan digunakan  $\eta = 0$  dan  $\sigma = \sqrt{2/Fin}$ .

#### 2.1.4. L1 dan L2 Regularization

L1 dan L2 Regularization adalah teknik yang digunakan untuk mencegah overfitting pada neural network dengan menambahkan penalti pada fungsi loss agar model tidak terlalu bergantung pada bobot-bobot besar. Tujuan utama regularisasi adalah untuk mengurangi kompleksitas model dan membantu model mempelajari fungsi yang lebih sederhana untuk meningkatkan generalisasi.

##### **L1 Regularization**

$$\text{Modified loss function} = \text{Loss function} + \lambda \sum_{i=1}^n |W_i|$$

##### **L2 Regularization**

$$\text{Modified loss function} = \text{Loss function} + \lambda \sum_{i=1}^n W_i^2$$

Sumber: <https://medium.com/@alejandro.itoaramendia>

#### 2.1.4.1. L1 (LASSO Regression)

Pada regularisasi L1, penalti yang diberikan adalah jumlah nilai absolut dari semua bobot. Ini memiliki efek membuat bobot yang tidak penting

menjadi nol, sehingga L1 cenderung menghasilkan model yang lebih *sparse*, bagus apabila kita memiliki jumlah fitur yang banyak.

#### 2.1.4.2. **L2 (Ridge Regression)**

Pada regularisasi L2, penalti berupa jumlah kuadrat dari bobot-bobot, yang tidak membuat bobot menjadi nol tetapi mendorongnya untuk menjadi kecil. L2 cenderung meratakan bobot agar lebih kecil secara merata, membuat model lebih stabil dan lebih tahan terhadap fluktuasi pada data.

#### 2.1.5. **Normalisasi RMSNorm**

RMSNorm menormalisasi nilai aktivasi berdasarkan nilai RMS (*Root Mean Square*) dari nilai aktivasi itu sendiri. RMSNorm membantu menghilangkan ketergantungan terhadap nilai rata-rata (mean) dan hanya fokus pada skala atau besar kecilnya input sehingga membuat model lebih stabil. RMSNorm yang hanya memanfaatkan RMS menjadi lebih ringan dalam komputasi, bagus untuk dataset atau model yang besar.

#### 2.1.6. **FFNN**

Feedforward Neural Network (FFNN) terdiri dari beberapa lapisan neuron yang terhubung secara searah, dari input layer menuju output layer tanpa adanya umpan balik (*feedback*). Setiap neuron dalam jaringan ini menerima input, mengalikan dengan bobot, menambahkan bias, lalu menerapkan fungsi aktivasi sebelum diteruskan ke lapisan berikutnya.

## BAB III

### IMPLEMENTASI DAN PENGUJIAN

#### 3.1. Implementasi Program

Program dibagi dalam 2 file: utils.py dan main.py. Utils.py akan berisi semua activation function, loss function, dan derivative function dari masing-masing activation function. File main akan berisi algoritma untuk FFNN dengan memanfaatkan fungsi-fungsi pilihan dari Utils.py.

##### 3.1.1. Utils.py

<b>Fungsi Aktivasi</b>	<pre> ● ● ● 1  class ActivationFunction: 2      def __init__(self): 3          pass 4 5      @staticmethod 6      def linear(self, x): 7          return x 8 9      @staticmethod 10     def relu(self, x): 11         return np.maximum(0, x) 12 13     @staticmethod 14     def sigmoid(self, x): 15         return 1 / (1 + np.exp(-x)) 16 17     @staticmethod 18     def tanh(self, x): 19         return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x)) 20 21     @staticmethod 22     def softmax(self, x): 23         return np.exp(x) / np.sum(np.exp(x), axis=1, keepdims=True) 24 25     # Bonus: Swish, softplus, and ELU 26     @staticmethod 27     def swish(self, x): 28         return x * ActivationFunction.sigmoid(x) 29 30     @staticmethod 31     def softplus(self, x): 32         return np.log(1 + np.exp(x)) 33 34     @staticmethod 35     def elu(self, x, alpha=1.0): 36         return np.where(x &gt; 0, x, alpha * (np.exp(x) - 1)) </pre>
<b>Fungsi Loss</b>	<pre> ● ● ● 1  class LossFunction: 2      def __init__(self): 3          pass 4 5      @staticmethod 6      def mse(self, yi, y_hat): 7          return np.mean((yi - y_hat) ** 2) 8 9      @staticmethod 10     def binCrossEntropy(self, yi, y_hat): 11         return -np.mean(yi * np.log(y_hat) + (1 - yi) * np.log(1 - y_hat)) 12 13     @staticmethod 14     def catCrossEntropy(self, yi, y_hat): 15         eps = 1e-15 16         y_hat = np.clip(y_hat, eps, 1 - eps) 17 18         n = yi.shape[0] 19 20         loss = -np.sum(yi * np.log(y_hat)) / n 21 22         return loss 23 </pre>

## Turunan Pertama Fungsi Aktivasi

```

1  class Derivative:
2
3
4      @staticmethod
5      def linear(x):
6          return np.ones_like(x)
7
8      @staticmethod
9      def relu(x):
10         return np.where(x > 0, 1, 0)
11
12     @staticmethod
13     def sigmoid(x):
14         s = ActivationFunction.sigmoid(x)
15         return s * (1 - s)
16
17     @staticmethod
18     def tanh(x):
19         return 1 - np.tanh(x)**2
20
21     @staticmethod
22     def softmax(x):
23         s = ActivationFunction.softmax(x)
24         # Shape: (batch_size, num_classes)
25         batch_size, num_classes = s.shape
26
27         # Buat matriks turunan untuk setiap sampel dalam batch
28         jacobian = np.zeros((batch_size, num_classes, num_classes))
29
30         for i in range(batch_size):
31             s_i = s[i].reshape(-1, 1)
32             jacobian[i] = np.diagflat(s_i) - np.dot(s_i, s_i.T)
33
34         # Shape: (batch_size, num_classes, num_classes)
35         return jacobian
36
37     # Bonus: Swish, softplus, and ELU
38     @staticmethod
39     def swish(x):
40         s = ActivationFunction.sigmoid(x)
41         return s + x * s * (1 - s)
42
43     @staticmethod
44     def softplus(x):
45         return 1 / (1 + np.exp(-x))
46
47     @staticmethod
48     def elu(x, alpha=1.0):
49         return np.where(x > 0, 1, alpha * np.exp(x))
50

```

### 3.1.2. Class FFNN

### Init FFNN

```

1 class FFNN:
2     def __init__(self, layers, activation_functions, loss_function="mse",
3                  weight_method='xavier', seed=None, low_bound = -1, up_bound = 1,
4                  mean = 0.0, variance = 0.1, regularization=None, lambda_reg=0.01,):
5         self.layers = layers
6         self.activation_functions = activation_functions
7         self.loss_function = loss_function
8         self.weight_method = weight_method
9         self.num_layers = len(layers) - 1
10        self.seed = seed
11        self.low_bound = low_bound
12        self.up_bound = up_bound
13        self.mean = mean
14        self.variance = variance
15        self.regularization = regularization
16        self.lambda_reg = lambda_reg
17
18        self.weights = []
19        self.biases = []
20        self.gradients = []
21
22        # agar hasil random bisa sama untuk seed sama
23        if seed is not None:
24            np.random.seed(seed)
25
26        self.weights, self.biases = self.initialize_weights(weight_method)

```

### 3.1.3. Main.py

```

1 class FFNN:
2     def __init__(self, layers, activation_functions, loss_function="mse",
3                  weight_method='xavier', seed=None, low_bound = -1, up_bound = 1,
4                  mean = 0.0, variance = 0.1, regularization=None, lambda_reg=0.01,
5                  rms_norm=False):
6         self.layers = layers
7         self.activation_functions = activation_functions
8         self.loss_function = loss_function
9         self.weight_method = weight_method
10        # karena tidak menghitung layer input
11        self.num_layers = len(layers) - 1
12        self.seed = seed
13        self.low_bound = low_bound
14        self.up_bound = up_bound
15        self.mean = mean
16        self.variance = variance
17        self.regularization = regularization
18        self.lambda_reg = lambda_reg
19        self.rms_norm = rms_norm
20
21        self.weights = []
22        self.biases = []
23        self.w_gradients = []
24        self.b_gradients = []
25        self.rms_layer = []
26        self.loss_history = []
27        self.val_loss_history = []
28
29        # agar hasil random bisa sama untuk seed sama
30        if seed is not None:
31            np.random.seed(seed)
32
33        self.weights, self.biases = self.initialize_weights(weight_method)
34
35        if self.rms_norm:
36            self.rms_layer = [RMSNorm(dim=self.layers[i+1]) for i in range(self.num_layers)]
37        else:
38            self.rms_layer = None
39

```

Inisialisasi kelas FFNN dengan atribut terkait. Melakukan inisialisasi weight dan bias. Jika rms\_norm=True, maka jaringan akan menggunakan RMS Norm pada setiap hidden layer

```

1 def save_model(self, file_path):
2     model_data = {
3         "layers": self.layers,
4         "activation_functions": self.activation_functions,
5         "loss_function": self.loss_function,
6         "weight_method": self.weight_method,
7         "num_layers": self.num_layers,
8         "seed": self.seed,
9         "low_bound": self.low_bound,
10        "up_bound": self.up_bound,
11        "mean": self.mean,
12        "variance": self.variance,
13        "regularization": self.regularization,
14        "lambda_reg": self.lambda_reg,
15        "rms_norm": self.rms_norm,
16
17        "weights": [w.tolist() for w in self.weights],
18        "biases": [b.tolist() for b in self.biases],
19        "w_gradients": [wg.tolist() for wg in self.w_gradients],
20        "b_gradients": [bg.tolist() for bg in self.b_gradients],
21
22        "loss_history": self.loss_history,
23        "val_loss_history": self.val_loss_history
24    }
25
26    with open(file_path, "w") as f:
27        json.dump(model_data, f)
28    print("Model saved successfully to " + file_path)

```

Fungsi untuk menyimpan model dan parameternya ke dalam sebuah json file.

```

1 def load_model(self, file_path):
2     with open(file_path, "r") as f:
3         model_data = json.load(f)
4
5         self.layers = model_data["layers"]
6         self.activation_functions = model_data["activation_functions"]
7         self.loss_function = model_data["loss_function"]
8         self.weight_method = model_data["weight_method"]
9         self.num_layers = model_data["num_layers"]
10        self.seed = model_data["seed"]
11        self.low_bound = model_data["low_bound"]
12        self.up_bound = model_data["up_bound"]
13        self.mean = model_data["mean"]
14        self.variance = model_data["variance"]
15        self.regularization = model_data["regularization"]
16        self.lambda_reg = model_data["lambda_reg"]
17        self.rms_norm = model_data["rms_norm"]
18
19        self.weights = [np.array(w) for w in model_data["weights"]]
20        self.biases = [np.array(b) for b in model_data["biases"]]
21        self.w_gradients = [np.array(wg) for wg in model_data["w_gradients"]]
22        self.b_gradients = [np.array(bg) for bg in model_data["b_gradients"]]
23
24        self.loss_history = model_data["loss_history"]
25        self.val_loss_history = model_data["val_loss_history"]
26
27    print(f"Model loaded successfully from {file_path}")

```

Fungsi untuk load model dari file json, dan set parameter dengan hasil variabel yang diload.

```

1  def initialize_weights(self, method):
2      weights = []
3      biases = []
4      for i in range (self.num_layers):
5          if (method == "zero"):
6              w = np.zeros((self.layers[i],self.layers[i+1]))
7              b = np.zeros((1,self.layers[i+1]))
8          elif (method == "uniform"):
9              w = np.random.uniform(self.low_bound, self.up_bound, (self.layers[i],self.layers[i+1]))
10             b = np.random.uniform(self.low_bound, self.up_bound, (1,self.layers[i+1]))
11         elif (method == "normal"):
12             deviation = np.sqrt(self.variance)
13             w = np.random.normal(self.mean, deviation, (self.layers[i],self.layers[i+1]))
14             b = np.random.normal(self.mean, deviation, (1, self.layers[i+1]))
15         elif method == 'xavier':
16             fan_in = self.layers[i]
17             fan_out = self.layers[i+1]
18             limit = np.sqrt(2 / (fan_in + fan_out))
19             w = np.random.normal(0, limit, (self.layers[i], self.layers[i+1]))
20             b = np.zeros((1, self.layers[i+1]))
21         elif method == 'he':
22             fan_in = self.layers[i]
23             limit = np.sqrt(2 / float(fan_in))
24             w = np.random.normal(0, limit, (self.layers[i], self.layers[i+1]))
25             b = np.zeros((1, self.layers[i+1]))
26         else:
27             raise ValueError ("Method initialize weight unknown")
28
29         weights.append(w)
30         biases.append(b)
31     return weights, biases
32

```

Fungsi untuk menginisialisasi berat dan bias sesuai dengan parameter metode yang sesuai

```

1 def forward(self, input):
2     a = input
3     activations = [a] #out sudah dikasih aktivasi
4     pre_activations = [] #net
5
6     for i in range(self.num_layers):
7         sigma = np.dot(a, self.weights[i]) + self.biases[i]
8
9         if self.rms_norm:
10             sigma = self.rms_layer[i](sigma)
11
12         pre_activations.append(sigma)
13
14         if self.activation_functions[i] == 'linear':
15             a = ActivationFunction.linear(sigma)
16         elif self.activation_functions[i] == 'relu':
17             a = ActivationFunction.relu(sigma)
18         elif self.activation_functions[i] == 'sigmoid':
19             a = ActivationFunction.sigmoid(sigma)
20         elif self.activation_functions[i] == 'tanh':
21             a = ActivationFunction.tanh(sigma)
22         elif self.activation_functions[i] == 'softmax':
23             a = ActivationFunction.softmax(sigma)
24         elif self.activation_functions[i] == 'swish':
25             a = ActivationFunction.swish(sigma)
26         elif self.activation_functions[i] == 'softplus':
27             a = ActivationFunction.softplus(sigma)
28         elif self.activation_functions[i] == 'elu':
29             a = ActivationFunction.elu(sigma)
30         else:
31             raise ValueError("Method activation unknown")
32         activations.append(a)
33
34     return activations, pre_activations
35

```

Fungsi forward propagation.  
 Menyimpan hasil net yang sudah diaktivasi ke dalam list activations, dan yang belum ke dalam pre\_activations. Melakukan iterasi hingga tercapai output layer.

<pre> 1 def count_loss(self, observe, pred): 2     base_loss = None 3     if self.loss_function == "mse": 4         base_loss = LossFunction.mse(observe, pred) 5     elif self.loss_function == "binary_crossentropy": 6         base_loss = LossFunction.binCrossEntropy(observe, pred) 7     elif self.loss_function == "categorical_crossentropy": 8         base_loss = LossFunction.catCrossEntropy(observe, pred) 9     else: 10        raise ValueError("Loss method unknown") 11 12    if self.regularization == "l1": 13        penalty = self.lambda_reg * sum([np.sum(np.abs(w)) for w in self.weights]) 14    return base_loss + penalty 15    elif self.regularization == "l2": 16        penalty = self.lambda_reg * sum([np.sum(w ** 2) for w in self.weights]) 17    return base_loss + penalty 18    else: 19        return base_loss </pre>	<p>Fungsi untuk menghitung loss sesuai dengan metode terpilih setelah melakukan forward propagation. Apabila menggunakan regularisasi maka akan ditambah dengan regularization term di kalkulasi loss.</p>
<pre> 1 def plot_model_structure(self, show_weights=True, show_gradients=True): 2     G = nx.DiGraph() 3     edge_labels = {} 4 5     # Hidden layer nodes 6     for hidden_idx in range(1, self.num_layers): 7         for neuron_idx in range(0, len(self.layers[hidden_idx])): 8             G.add_node(f"b[{hidden_idx}]{neuron_idx}", layer="bias") 9             G.add_node(f"n[{hidden_idx}]{neuron_idx}", layer="hidden") 10 11    # Output nodes 12    for neuron_idx in range(len(self.layers[-1])): 13        G.add_node(f"o[{neuron_idx}]", layer="output") 14 15 16    # Edge dari setiap hidden layer 17    for i in range(1, self.num_layers-1): 18        for j in range(self.layers[i]): 19            for k in range(self.layers[i+1]): 20                from_node = f"n[{i}]{j}" 21                to_node = f"n[{i+1}]{k}" 22                G.add_edge(from_node, to_node) 23 24                if show_weights: 25                    weight = self.weights[i][j][k] 26                    label = f"w{weight:2f}" 27                    if show_gradients and len(self.w_gradients) &gt; i: 28                        grad = self.w_gradients[i][j][k] 29                        label += f" w{grad:2f}" 30                    edge_labels[(from_node, to_node)] = label 31 32    # Edge dari hidden terakhir ke output 33    last_hidden = self.num_layers - 1 34    for i in range(last_hidden-1, last_hidden): 35        for j in range(self.layers[i]): 36            for k in range(self.layers[i+1]): 37                from_node = f"n[{last_hidden}]{(i-1)}" 38                to_node = f"o[{j}]" 39                G.add_edge(from_node, to_node) 40 41                if show_weights: 42                    weight = self.weights[-1][i][j] 43                    label = f"w{weight:2f}" 44                    if show_gradients and len(self.w_gradients) &gt; last_hidden - 1: 45                        grad = self.w_gradients[-1][i][j] 46                        label += f" w{grad:2f}" 47                    edge_labels[(from_node, to_node)] = label 48 49    # Edge dari bias ke masing-masing hidden layer 50    for i in range(1, self.num_layers): 51        for j in range(self.layers[i]): 52            from_node = f"b[{i}]" 53            to_node = f"n[{i}]{j}" 54            G.add_edge(from_node, to_node) 55            # Bias ini hanya just bias value 56            if show_weights: 57                bias_val = self.biases[i][0][j] 58                label = f"b:{bias_val:2f}" 59                if show_gradients and len(self.b_gradients) &gt; i-1: 60                    bias_grad = self.b_gradients[i-1][0][j] 61                    label += f" b{bias_grad:2f}" 62                edge_labels[(from_node, to_node)] = label 63 64    pos = {} 65    vertical_spacing = 15000 66    horizontal_spacing = 50000 67 68    for hidden_idx in range(1, self.num_layers): 69        for neuron_idx in range(len(self.layers[hidden_idx])): 70            pos[f"n[{hidden_idx}]{neuron_idx}"] = (hidden_idx * horizontal_spacing, neuron_idx * vertical_spacing) 71 72 73    for neuron_idx in range(len(self.layers[-1])): 74        pos[f"o[{neuron_idx}]"] = ((self.num_layers * horizontal_spacing, neuron_idx * vertical_spacing)) 75 76    for hidden_idx in range(0, self.num_layers): 77        pos[f"b[{hidden_idx}]"] = ((hidden_idx * horizontal_spacing) - 30000, -15000) 78 79 80    # Visualisasi grafik menggunakan matplotlib 81    plt.figure(figsize=(8, 6)) 82    nx.draw(G, pos, with_labels=True, node_size=700, node_color='skyblue', font_size=10, font_weight='bold', arrows=True) 83    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8) 84 85    # Tampilkan plot 86    plt.title("FFNN Graph") 87    plt.show() </pre>	<p>Fungsi untuk menampilkan plot network lengkap dengan gradien dan bobotnya</p>

```

1 def backward(self, input, output, activations, pre_activations, learning_rate):
2     weight_gradient = []
3     biases_gradient = []
4
5     # BACKWARD OUTPUT
6     # dErr/dOut
7     # print("Counting Err/Out\n")
8     if self.loss_function == "mse":
9         derr_dOut = activations[-1] - output
10    elif self.loss_function == "binary_crossentropy":
11        derr_dOut = activations[-1] - output
12    elif self.loss_function == "categorical_crossentropy":
13        derr_dOut = activations[-1] - output
14    else:
15        raise ValueError("Loss method unknown")
16
17     # dOut/dNet
18     # print("Counting Out/Net\n")
19     if self.activation_functions[-1] == 'sigmoid':
20         dout_dNet = Derivative.sigmoid(pre_activations[-1])
21     elif self.activation_functions[-1] == 'relu':
22         dout_dNet = Derivative.relu(pre_activations[-1])
23     elif self.activation_functions[-1] == 'tanh':
24         dout_dNet = Derivative.tanh(pre_activations[-1])
25     elif self.activation_functions[-1] == 'linear':
26         dout_dNet = Derivative.linear(pre_activations[-1])
27     elif self.activation_functions[-1] == 'swish':
28         dout_dNet = Derivative.swish(pre_activations[-1])
29     elif self.activation_functions[-1] == 'softplus':
30         dout_dNet = Derivative.softplus(pre_activations[-1])
31     elif self.activation_functions[-1] == 'elu':
32         dout_dNet = Derivative.elu(pre_activations[-1], alpha=1.0)
33     elif self.activation_functions[-1] == 'softmax':
34         # print(pre_activations[-1].shape)
35         s = ActivationFunction.softmax(pre_activations[-1])
36         # print(s.shape)
37         dout_dNet = Derivative.softmax(pre_activations[-1])
38     else:
39         raise ValueError("Unknown Activation Method")
40
41     # dNet/dW
42     # print("Counting Net/W\n")
43     dNet_dW = activations[-2]
44
45     # Count weight and bias
46     if (self.activation_functions[-1] == 'softmax'
47         and self.loss_function == 'categorical_crossentropy'):
48         delta = derr_dOut
49     else:
50         delta = derr_dOut * dout_dNet
51
52     weight_gradient.append(np.dot(dNet_dW.T, delta))
53     biases_gradient.append(np.sum(delta, axis=0, keepdims=True))
54
55     # BACKWARD HIDDEN
56     # output layer already processed, start from num_layers-2
57     for l in range(self.num_layers - 2, -1, -1):
58         activation_func = self.activation_functions[l]
59
60         # dErr/dNet
61         delta = np.dot(delta, self.weights[l+1].T) * getattr(Derivative, activation_func)(pre_activations[l])
62
63         # dNet/dW
64         dNet_dW = activations[l]
65
66         # Compute gradients, insert in beginning since we went from behind
67         weight_gradient.insert(0, np.dot(dNet_dW.T, delta))
68         biases_gradient.insert(0, np.sum(delta, axis=0, keepdims=True))
69
70     for l in range(self.num_layers):
71         if self.regularization == "l1":
72             weight_gradient[l] += self.lambda_reg * np.sign(self.weights[l])
73         elif self.regularization == "l2":
74             weight_gradient[l] += 2 * self.lambda_reg * self.weights[l]
75             self.weights[l] -= learning_rate * weight_gradient[l]
76             self.biases[l] -= learning_rate * biases_gradient[l]
77
78     self.w_gradients = weight_gradient
79     self.b_gradients = biases_gradient
80
81

```

Fungsi untuk melakukan backward propagation. Fungsi pertama akan melakukan backward prop pada output layer, dilanjutkan dengan iterasi ke semua hidden layer yang ada.

Gradien dihitung dengan chain rule, di mana gradien error dari layer berikutnya dikalikan dengan turunan fungsi aktivasi pada layer saat ini. Gradien bobot dan bias kemudian disimpan dalam list dan digunakan untuk memperbarui bobot jaringan

<pre> ● ● ● 1 def plot_loss_curve(self, loss_history, val_loss_history): 2     plt.plot(loss_history, label="Training Loss") 3     plt.plot(val_loss_history, label="Validation Loss") 4     plt.xlabel("Epoch") 5     plt.ylabel("Loss") 6     plt.title("Traning Loss vs Val Loss Curve") 7     plt.legend() 8     plt.show() </pre>	<p>Fungsi untuk menampilkan grafik loss baik validasi maupun training loss</p>
<pre> ● ● ● 1 def weight_dist_plot(self, layer_idx=None): 2     if layer_idx is None: 3         layer_idx = range(self.num_layers) 4 5     for i in layer_idx: 6         plt.figure(figsize=(5, 3)) 7         plt.hist(self.weights[i].flatten(), bins=50) 8         plt.title(f'Weight Distribution for Layer {i+1}') 9         plt.xlabel('Weight Value') 10        plt.ylabel('Frequency') 11        plt.show() </pre>	<p>Fungsi untuk menampilkan grafik distribusi weight pada layer terpilih</p>
<pre> ● ● ● 1 def grad_dist_plot(self, layer_idx=None): 2     if layer_idx is None: 3         layer_idx = range(self.num_layers) 4 5     for i in layer_idx: 6         plt.figure(figsize=(5, 3)) 7         if i &lt; len(self.w_gradients): 8             plt.hist(self.w_gradients[i].flatten(), bins=50) 9             plt.title(f'Gradient Distribution for Layer {i+1}') 10            plt.xlabel('Gradient Value') 11            plt.ylabel('Frequency') 12            plt.show() 13 </pre>	<p>Fungsi untuk menampilkan distribusi gradien pada layer terpilih</p>

```

1  def train(self, X_train, y_train, epochs, learning_rate, batch_size, verbose, X_val=None, y_val=None):
2      num_samples = X_train.shape[0]
3
4      encoder = OneHotEncoder(sparse_output=False)
5      y_train_encoded = encoder.fit_transform(y_train.reshape(-1, 1))
6
7      if X_val is not None and y_val is not None:
8          y_val_encoded = encoder.transform(y_val.reshape(-1, 1))
9
10     for epoch in range(epochs):
11         # Shuffle data at each epoch
12         indices = np.arange(num_samples)
13         np.random.shuffle(indices)
14         X_train, y_train_encoded = X_train[indices], y_train_encoded[indices]
15
16         epoch_loss = 0
17         for i in range(0, num_samples, batch_size):
18             # Mini-batch selection
19             X_batch = X_train[i:i + batch_size]
20             y_batch = y_train_encoded[i:i + batch_size]
21
22             # Forward Propagation
23             activations, pre_activations = self.forward(X_batch)
24
25             # Compute loss
26             loss = self.count_loss(y_batch, activations[-1])
27             epoch_loss += np.mean(loss)
28
29             # Backpropagation
30             self.backward(X_batch, y_batch, activations, pre_activations, learning_rate)
31
32         # Average loss per epoch
33         epoch_loss /= (num_samples / batch_size)
34         self.loss_history.append(epoch_loss)
35
36         # Calculate validation loss
37         if X_val is not None and y_val is not None:
38             activations_val, _ = self.forward(X_val)
39             val_loss = self.count_loss(y_val_encoded, activations_val[-1])
40             self.val_loss_history.append(np.mean(val_loss))
41
42         if verbose == 1:
43             print(f"Epoch {epoch + 1}/{epochs} - Training Loss: {epoch_loss:.4f}", end="")
44             if X_val is not None and y_val is not None:
45                 print(f" - Validation Loss: {self.val_loss_history[-1]:.4f}", end="")
46             print()
47
48         self.plot_loss_curve(self.loss_history, self.val_loss_history)
49

```

Fungsi train bertanggung jawab untuk melatih model FFNN menggunakan algoritma mini-batch gradient descent.

Setiap epoch, data pelatihan dirandom agar proses training lebih efektif, lalu dibagi menjadi batch kecil. Untuk setiap batch, dilakukan forward propagation untuk menghitung output, diikuti dengan perhitungan loss, dan backpropagation untuk memperbarui bobot menggunakan gradient descent.

### 3.2. Hasil Pengujian

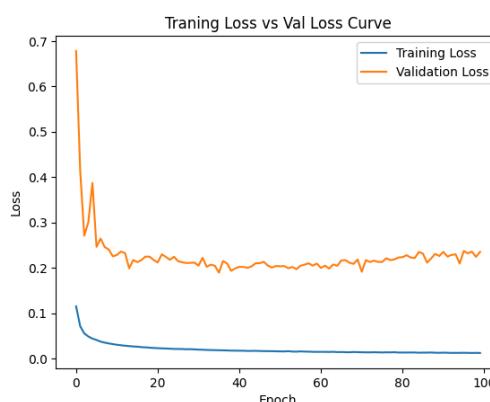
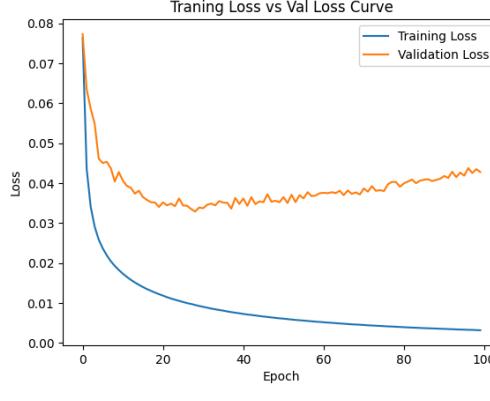
Hasil Pengujian dilakukan pada dataset [MNIST](#).

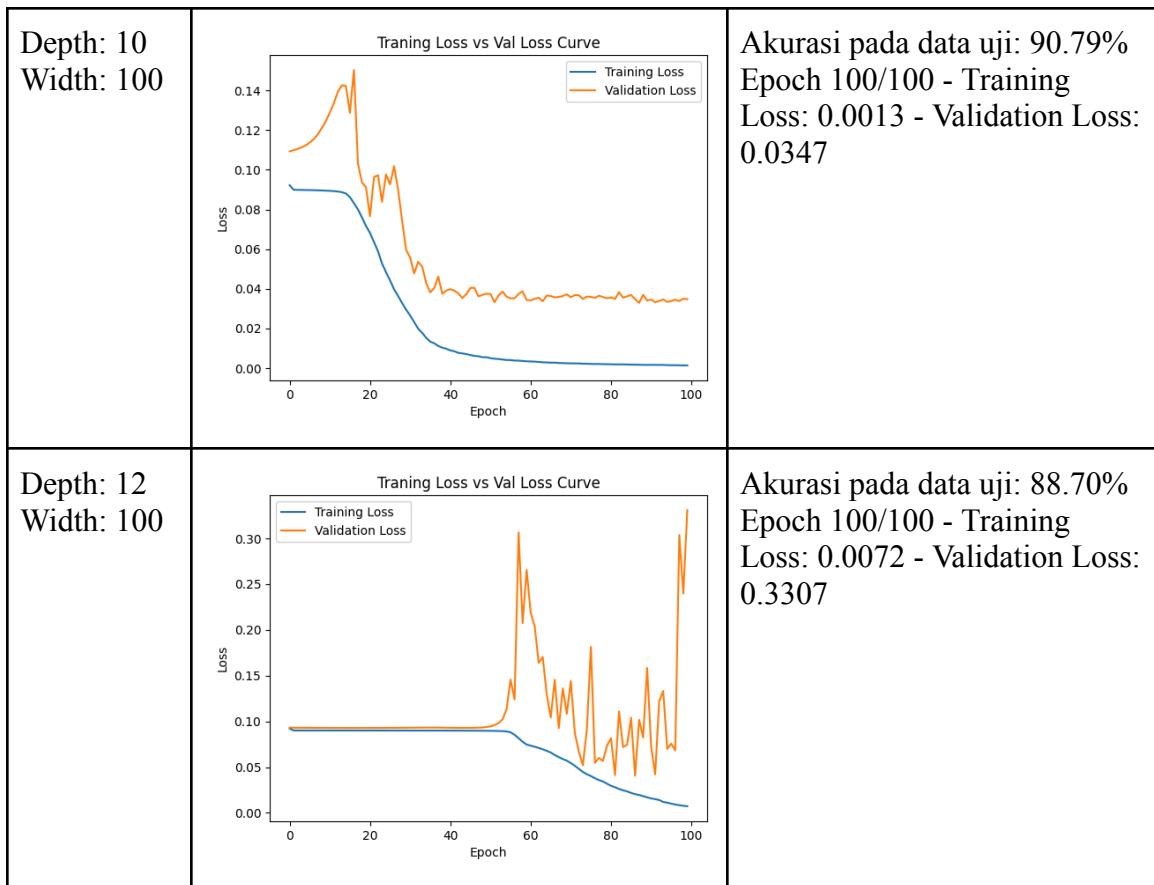
#### 3.2.1. Variasi Depth dan Width

Pengujian dilakukan dengan mengubah parameter depth dan width, parameter lain disetel sama dengan nilai sebagai berikut:

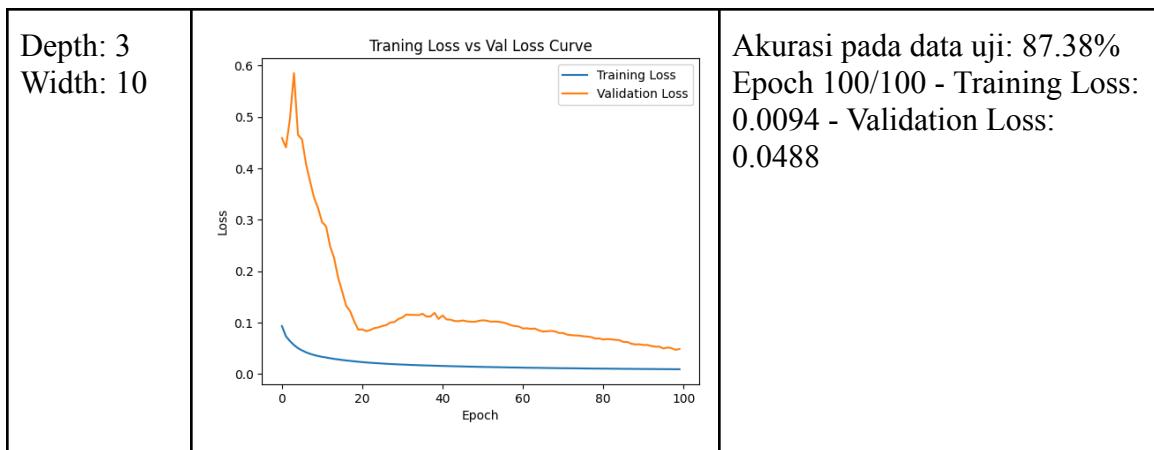
- Fungsi aktivasi (setiap layer): swish
- Fungsi loss: MSE
- Fungsi inisialisasi berat: Xavier
- Learning rate: 0.001
- Epoch: 100
- Batch size: 200
- Regularisation: None
- RMSNorm: False

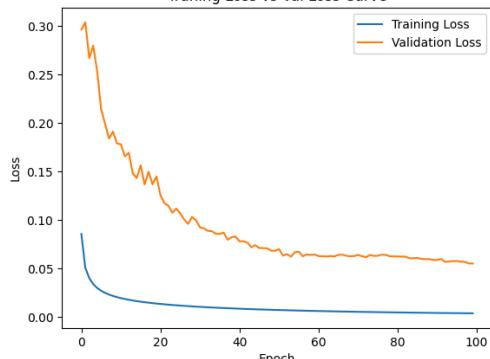
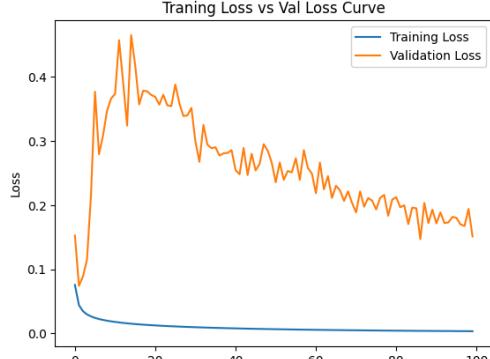
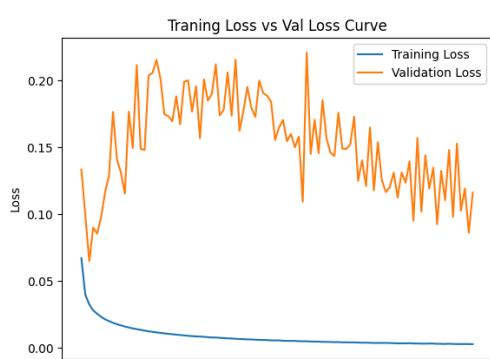
Tabel 3.2.1.1 Hasil Pengujian Depth

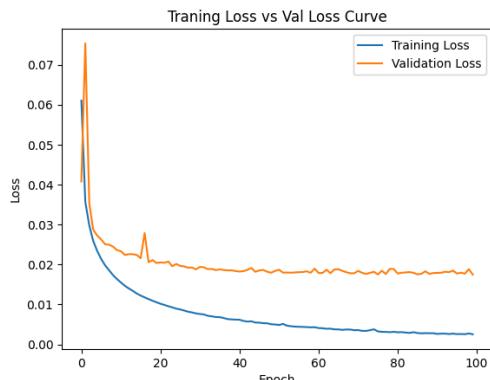
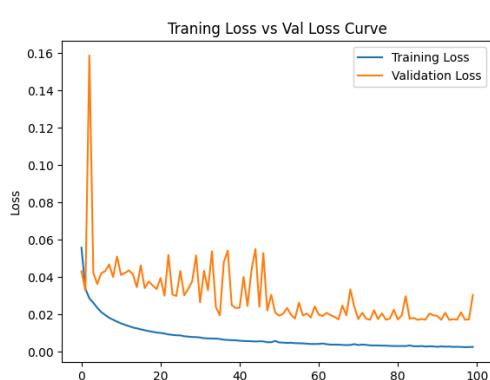
Depth: 1 Width: 100	 <p>Traning Loss vs Val Loss Curve</p> <p>Training Loss      Validation Loss</p> <p>Epoch</p>	Akurasi pada data uji: 88.55% Epoch 100/100 - Training Loss: 0.0119 - Validation Loss: 0.2350
Depth: 3 Width: 100	 <p>Traning Loss vs Val Loss Curve</p> <p>Training Loss      Validation Loss</p> <p>Epoch</p>	Akurasi pada data uji: 93.58% Epoch 100/100 - Training Loss: 0.0031 - Validation Loss: 0.0428



Tabel 3.2.2.1 Hasil Pengujian Width



Depth: 3 Width: 50	 A line graph titled "Traning Loss vs Val Loss Curve". The x-axis is labeled "Epoch" and ranges from 0 to 100. The y-axis is labeled "Loss" and ranges from 0.00 to 0.30. There are two lines: a blue line for "Training Loss" which starts at approximately 0.08 and drops to near 0.00 by epoch 10; and an orange line for "Validation Loss" which starts at approximately 0.30 and decreases more slowly, reaching about 0.055 at epoch 100.	Akurasi pada data uji: 92.93% Epoch 100/100 - Training Loss: 0.0037 - Validation Loss: 0.0551
Depth: 3 Width: 100	 A line graph titled "Traning Loss vs Val Loss Curve". The x-axis is labeled "Epoch" and ranges from 0 to 100. The y-axis is labeled "Loss" and ranges from 0.00 to 0.40. There are two lines: a blue line for "Training Loss" which starts at approximately 0.08 and drops to near 0.00 by epoch 10; and an orange line for "Validation Loss" which starts at approximately 0.10, peaks at 0.40 around epoch 15, and then fluctuates between 0.20 and 0.30 until epoch 100.	Akurasi pada data uji: 93.77% Epoch 100/100 - Training Loss: 0.0032 - Validation Loss: 0.1510
Depth: 3 Width: 300	 A line graph titled "Traning Loss vs Val Loss Curve". The x-axis is labeled "Epoch" and ranges from 0 to 100. The y-axis is labeled "Loss" and ranges from 0.00 to 0.20. There are two lines: a blue line for "Training Loss" which starts at approximately 0.08 and drops to near 0.00 by epoch 10; and an orange line for "Validation Loss" which starts at approximately 0.10, peaks at 0.20 around epoch 15, and then fluctuates between 0.10 and 0.20 until epoch 100.	Akurasi pada data uji: 93.89% Epoch 100/100 - Training Loss: 0.0027 - Validation Loss: 0.1161

Depth: 3 Width: 500		Akurasi pada data uji: 94.16% Epoch 100/100 - Training Loss: 0.0025 - Validation Loss: 0.0175
Depth: 3 Width: 1000		Akurasi pada data uji: 93.88% Epoch 100/100 - Training Loss: 0.0026 - Validation Loss: 0.0304

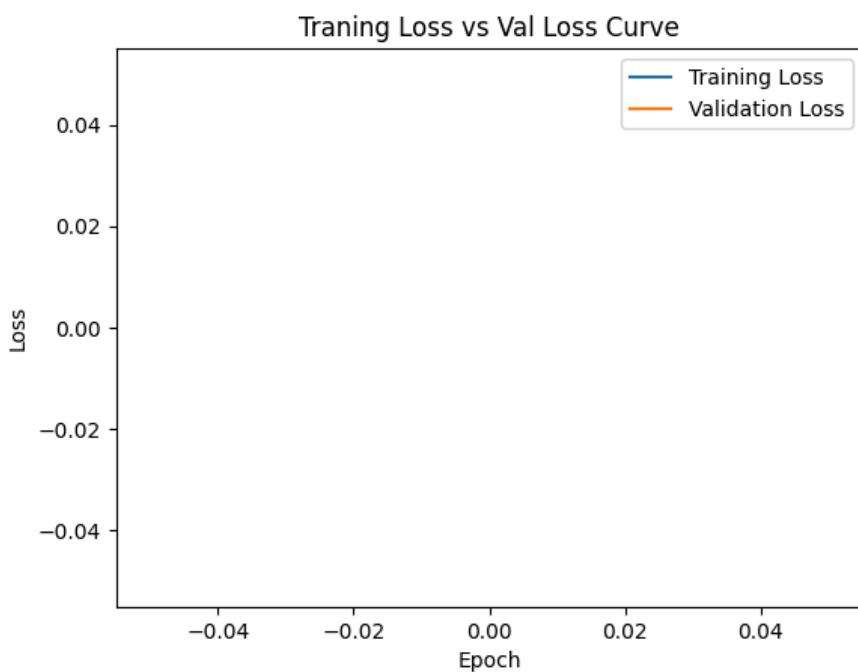
### 3.2.2. Variasi Fungsi Aktivasi

Pengujian dilakukan dengan mengubah parameter fungsi aktivasi (tiap layer sama), parameter lain disetel sama dengan nilai sebagai berikut:

- Depth: 3
- Width: 100
- Fungsi loss: MSE
- Fungsi inisialisasi berat: Xavier
- Learning rate: 0.001
- Epoch: 100
- Batch size: 200
- Regularisation: None
- RMSNorm: False

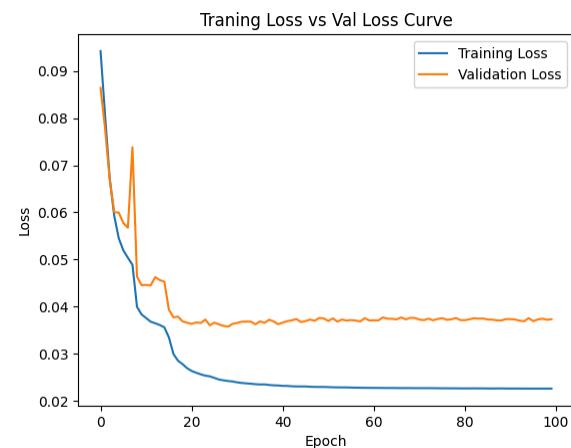
Tabel 3.2.2.1 Hasil Pengujian Fungsi Aktivasi

## Linear

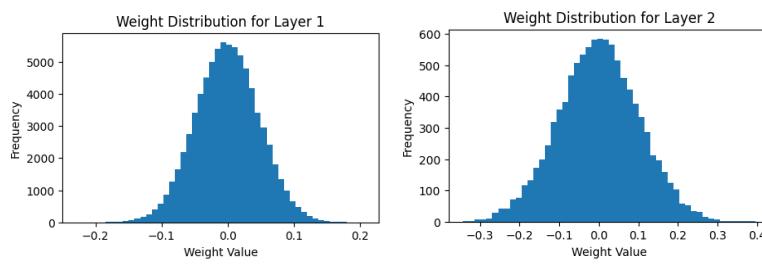


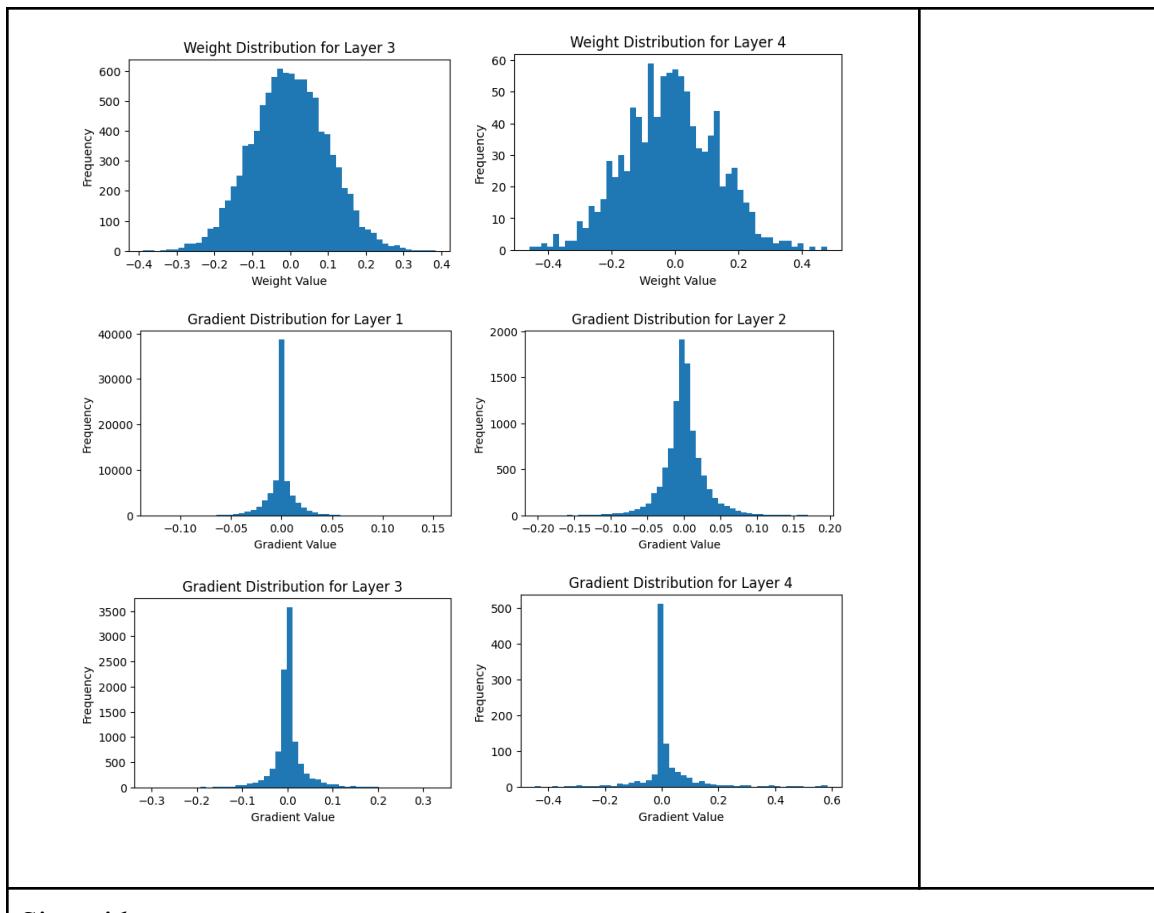
Akurasi pada data uji: 10.38%  
Epoch 100/100 - Training Loss: nan - Validation Loss: nan

## RELU

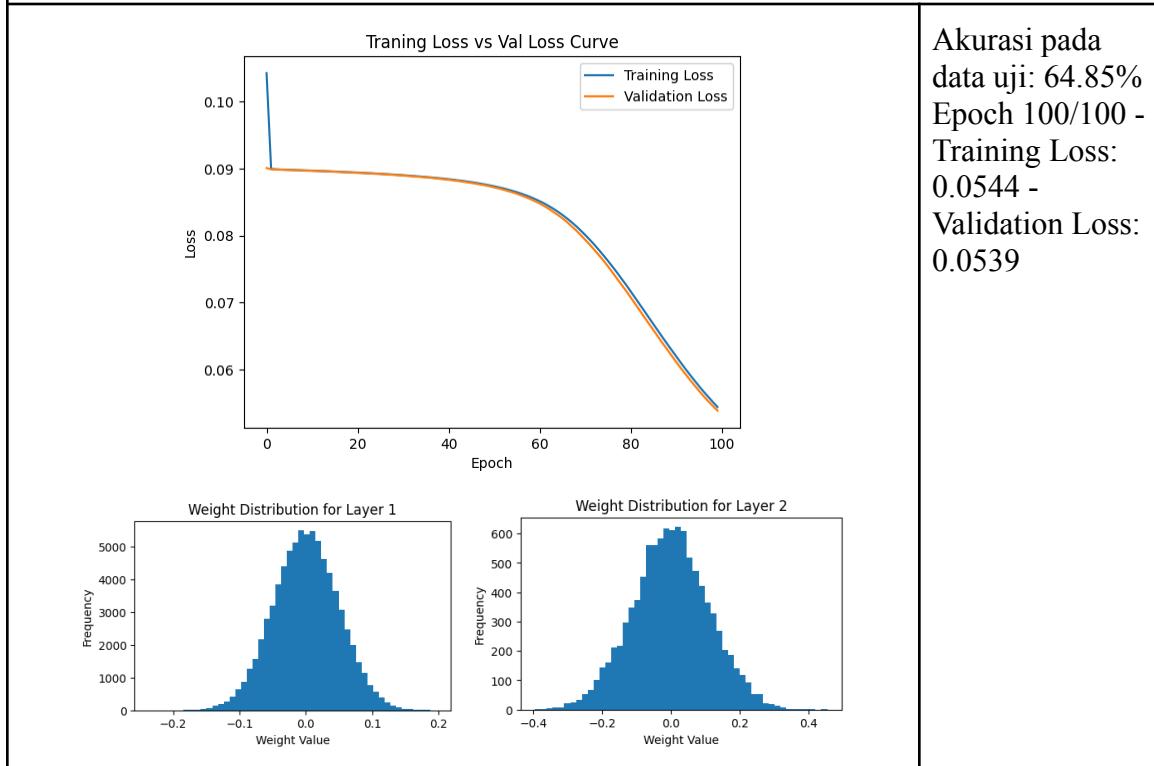


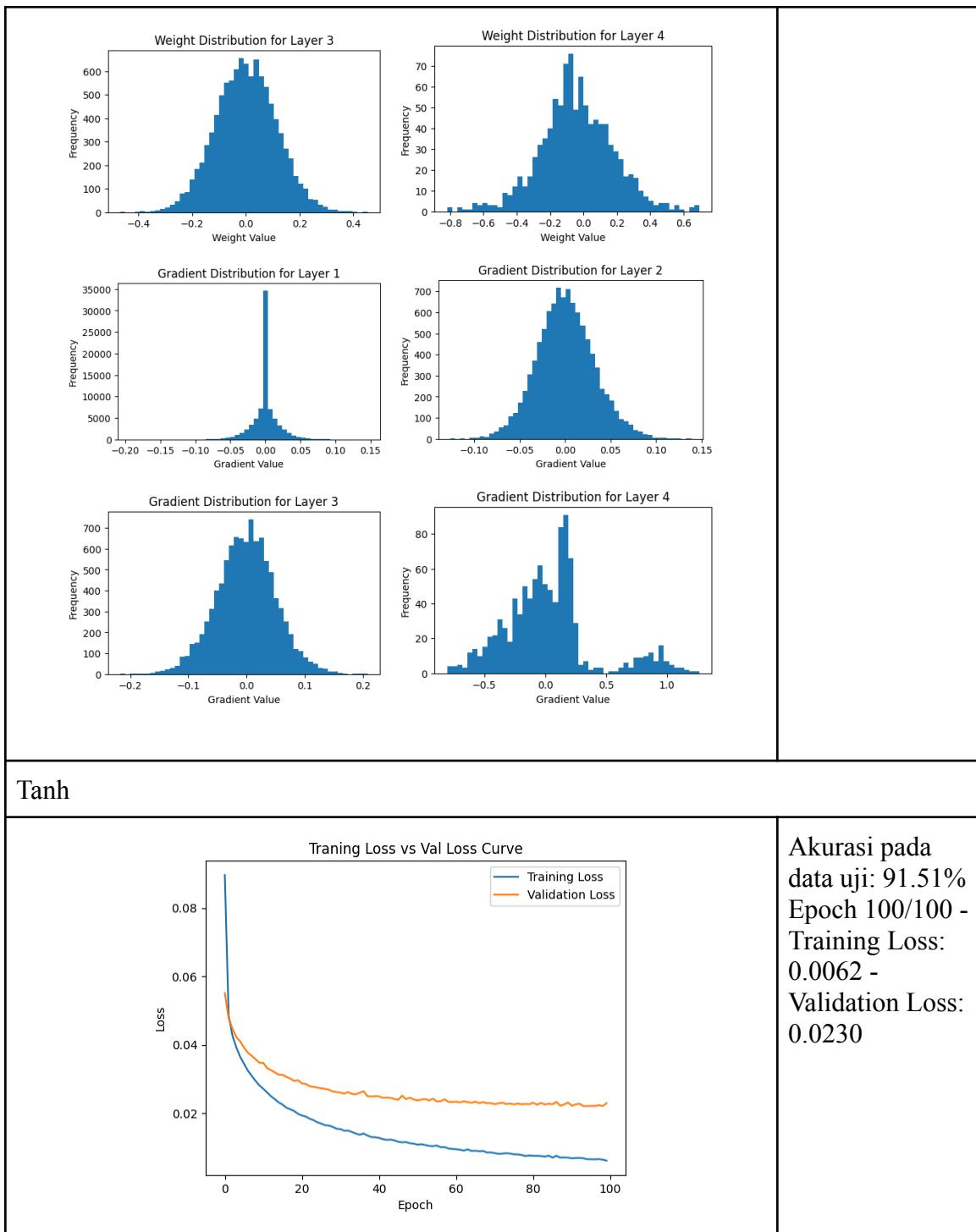
Akurasi pada data uji: 73.61%  
Epoch 100/100 - Training Loss: 0.0226 - Validation Loss: 0.0373

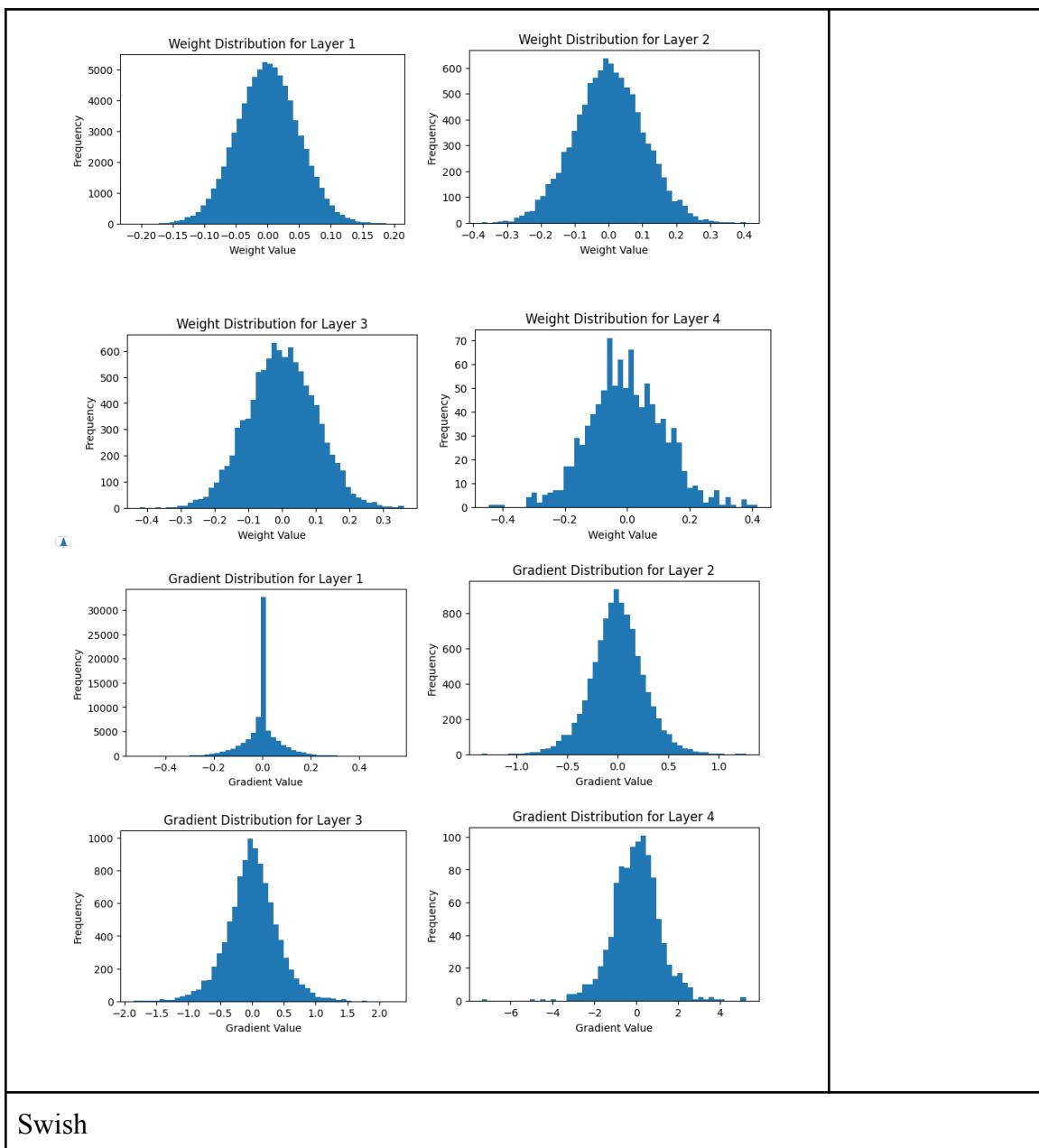


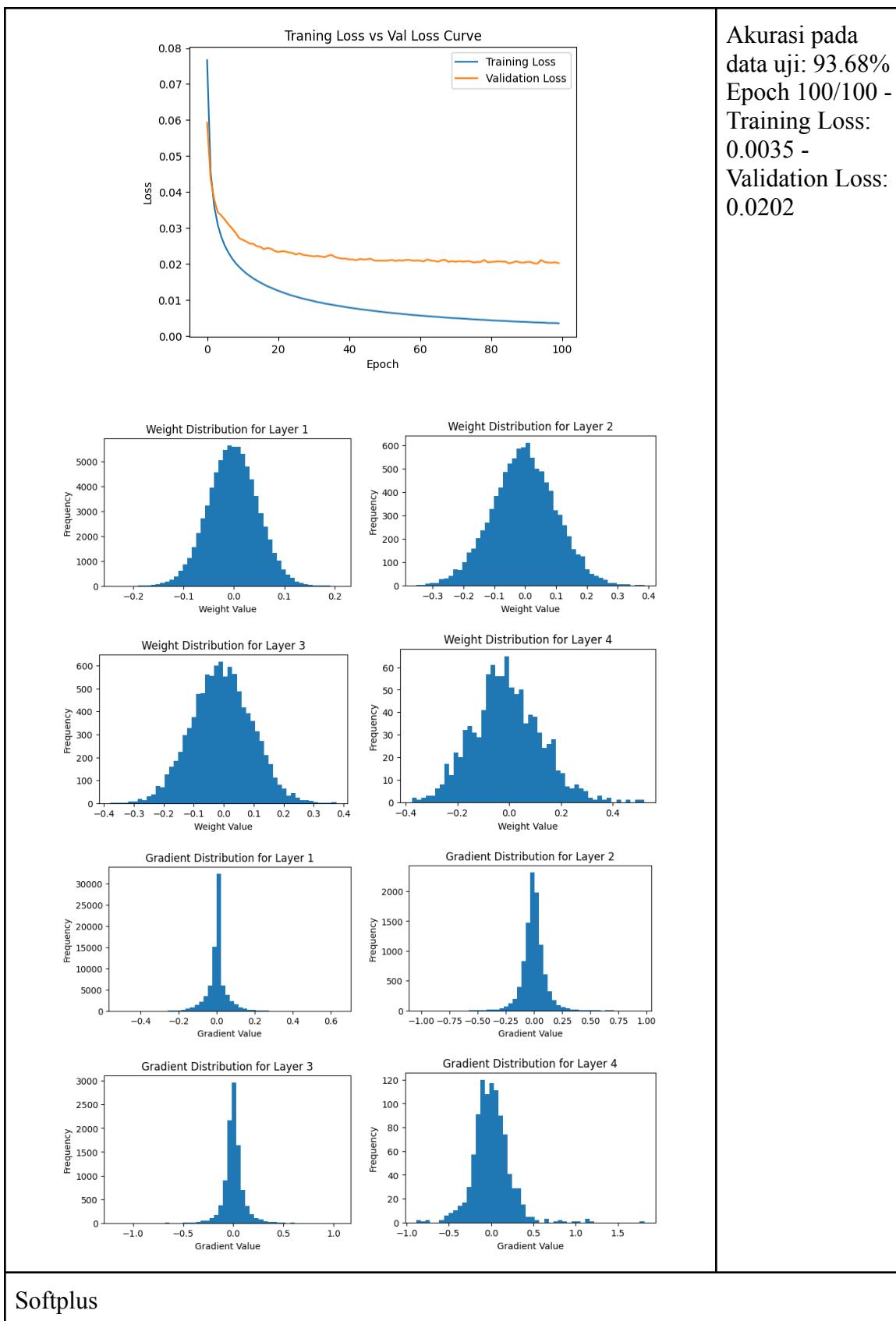


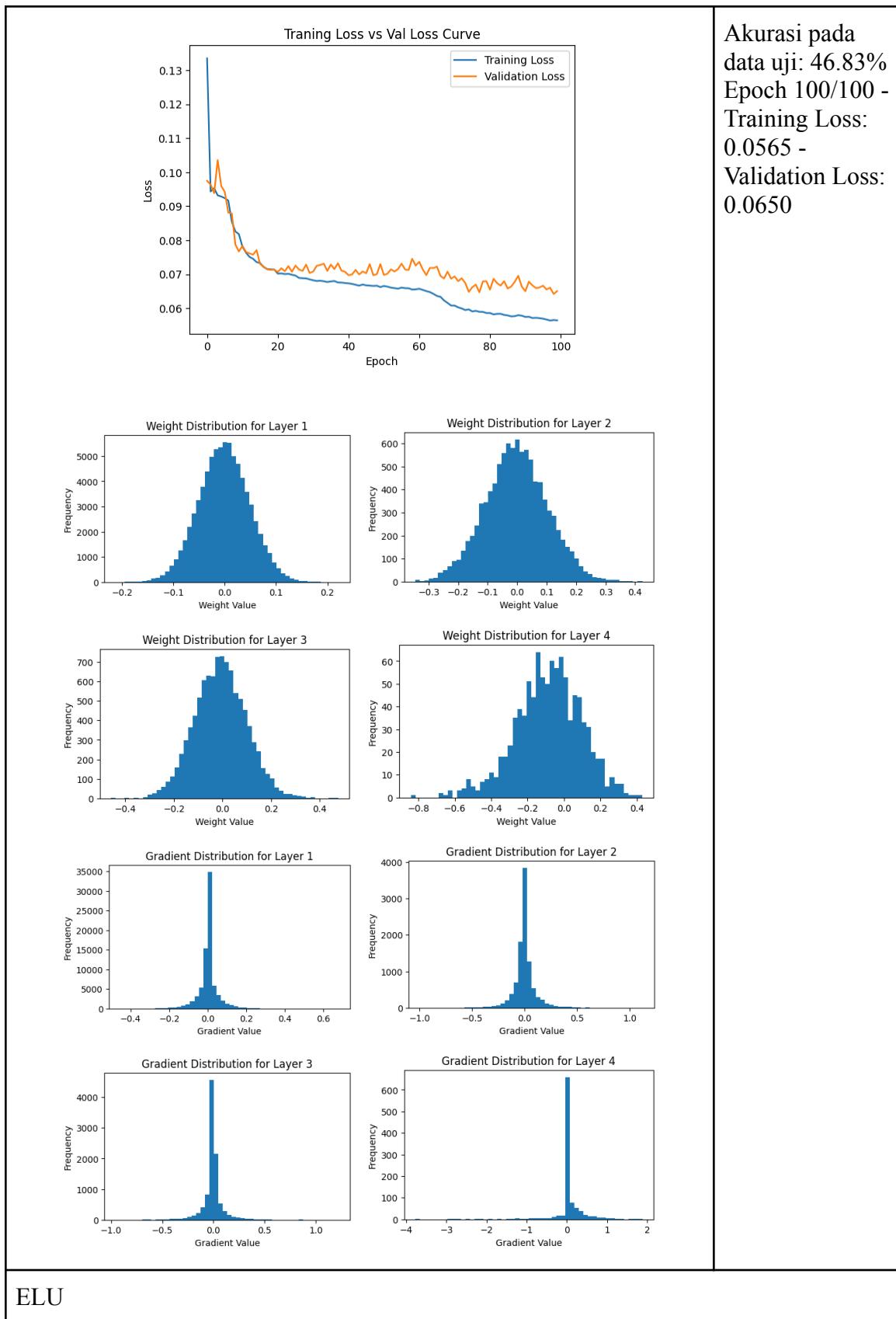
### Sigmoid

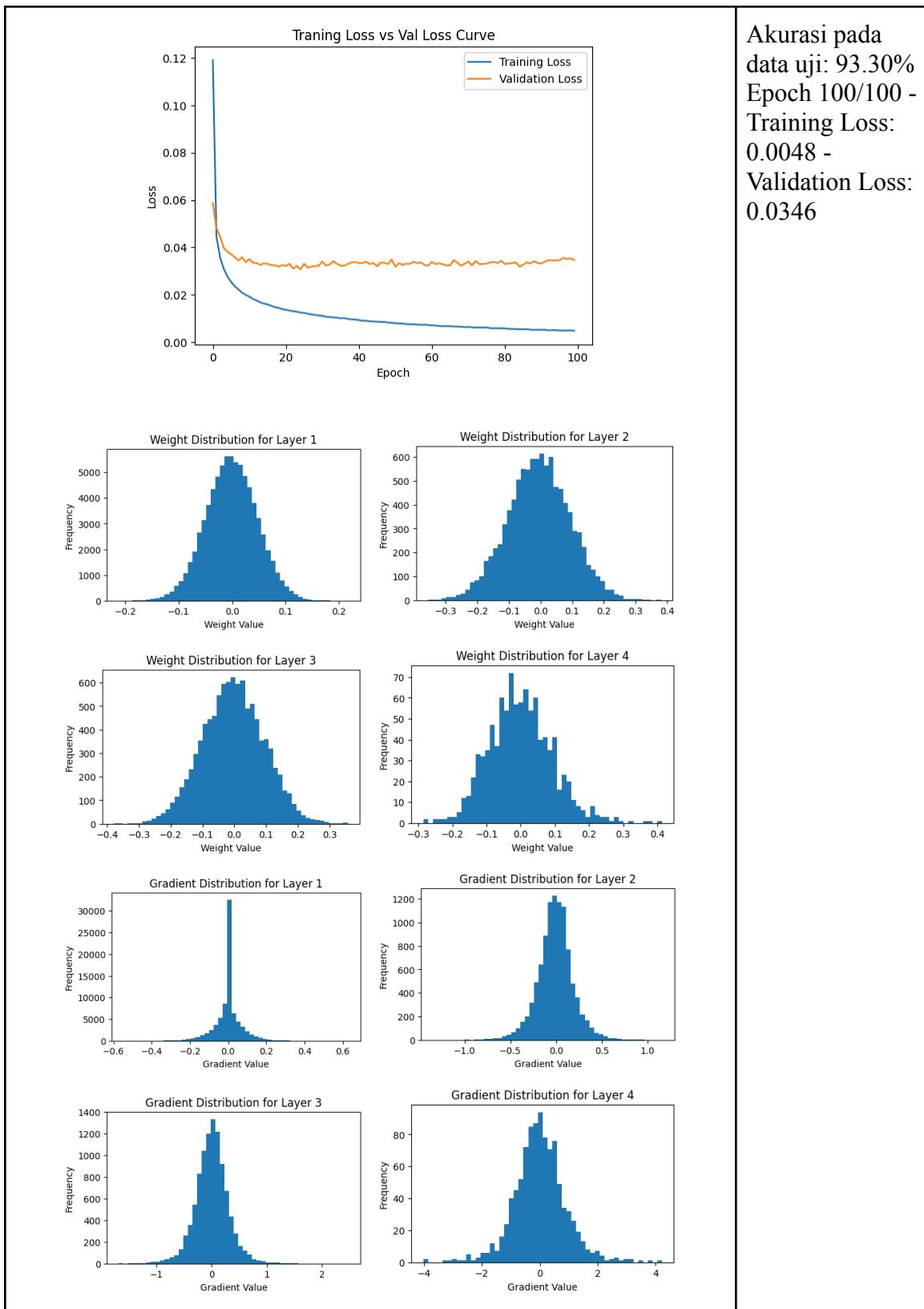










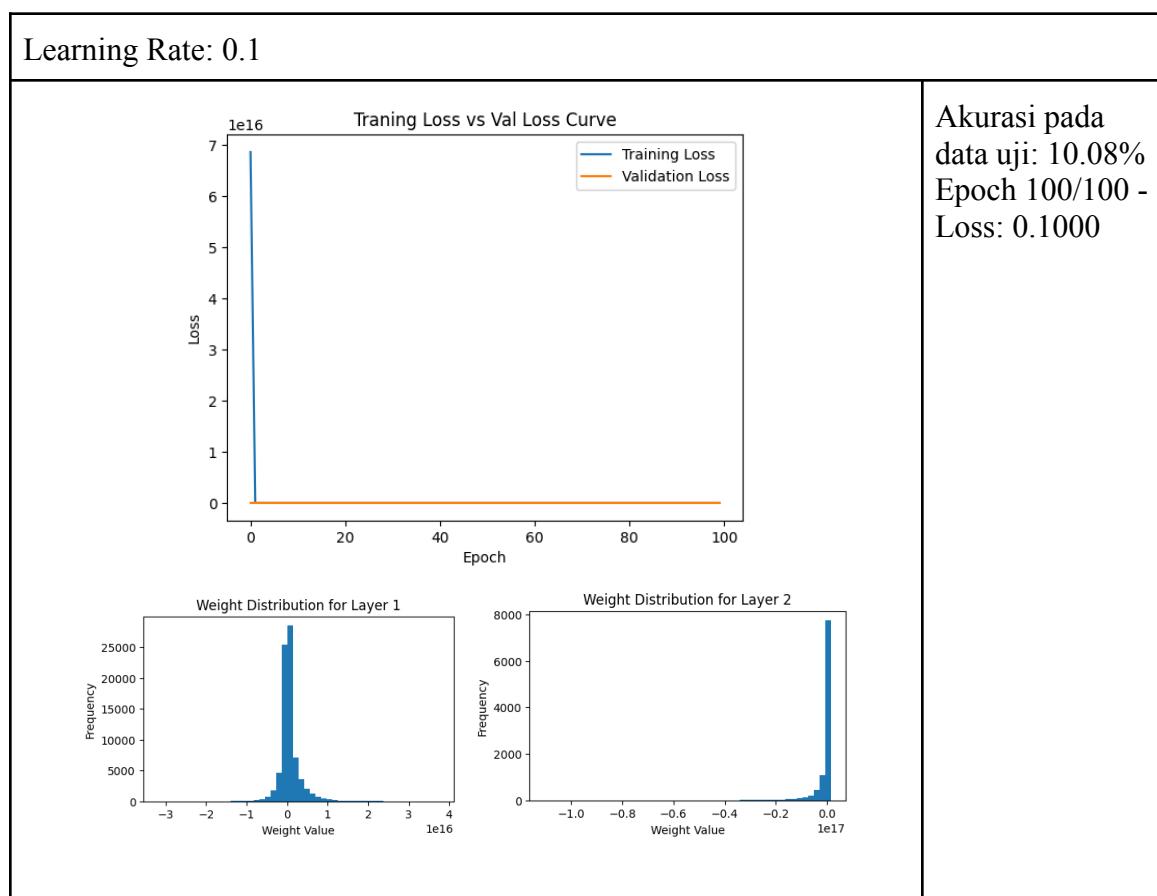


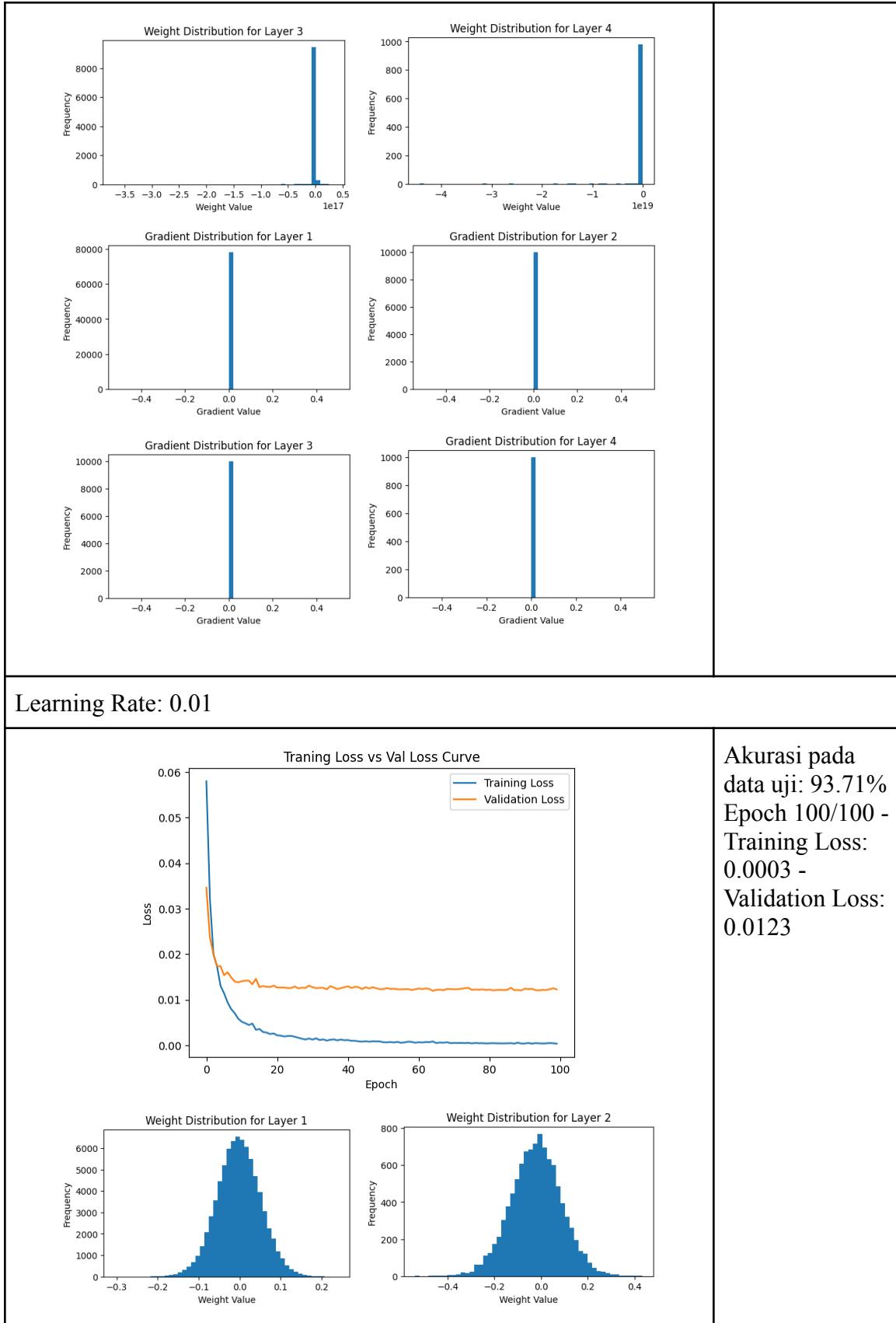
### 3.2.3. Variasi Learning Rate

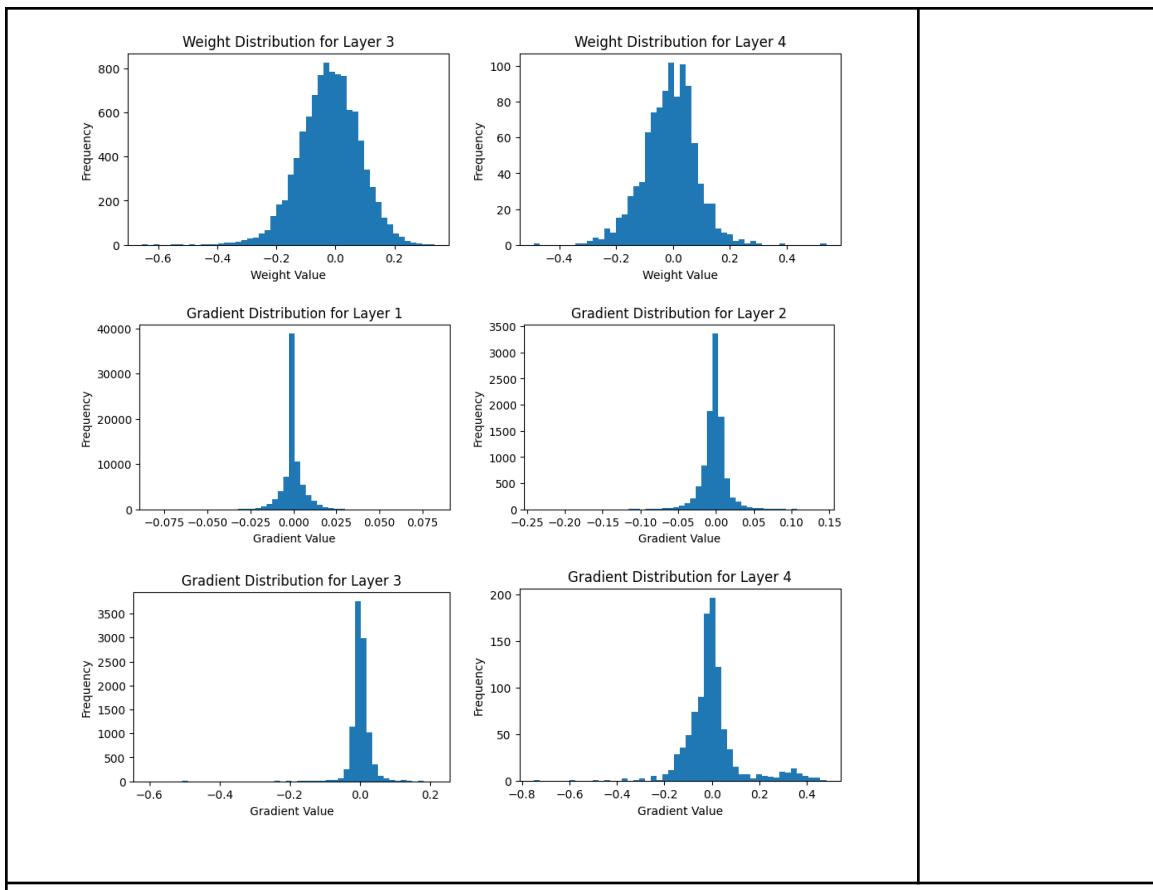
Pengujian ini dilakukan 2 kali untuk fungsi aktivasi swish dan sigmoid. Pada Pengujian dilakukan dengan mengubah parameter besar learning rate, parameter lain disetel sama dengan nilai sebagai berikut:

- Depth: 3
- Width: 100
- Fungsi loss: MSE
- Fungsi inisialisasi berat: Xavier
- Epoch: 100
- Batch size: 200
- Regularisation: None
- RMSNorm: False

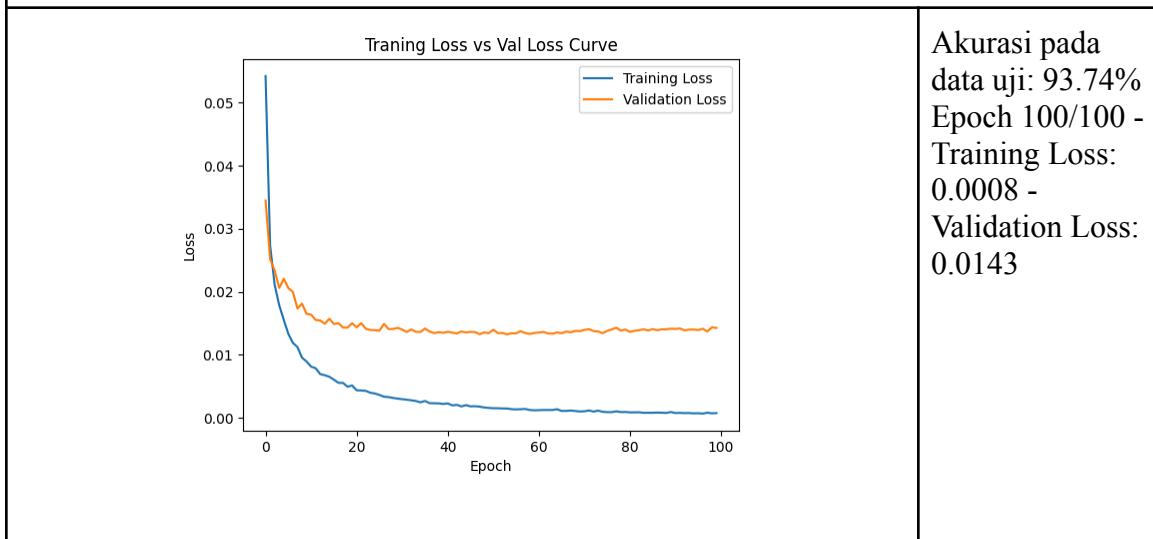
Tabel 3.2.3.1 Hasil Pengujian Learning Rate (dengan Swish)

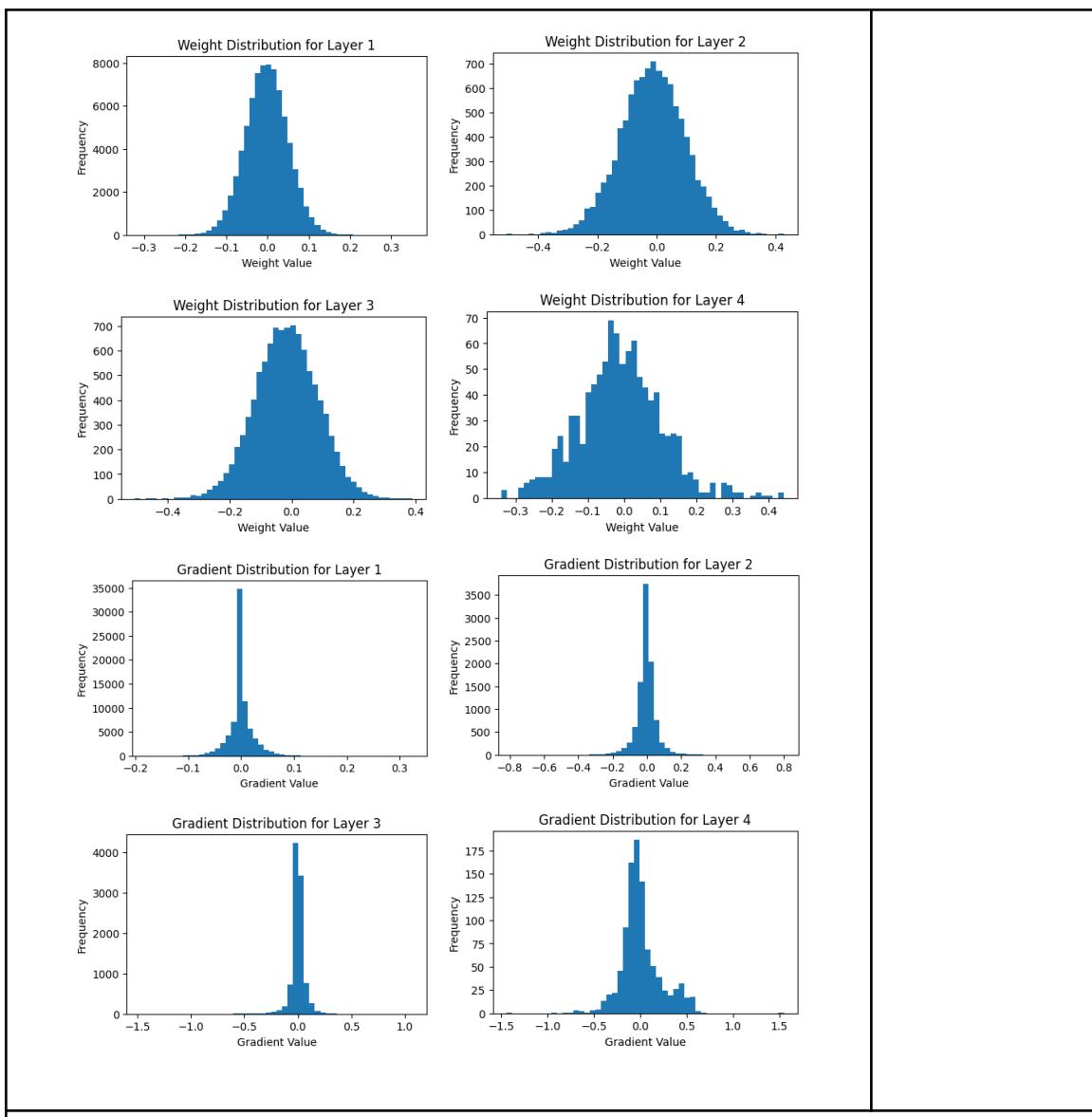




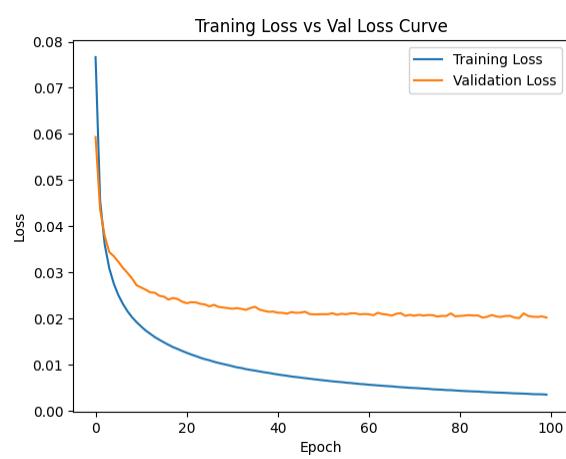


Learning Rate: 0.005

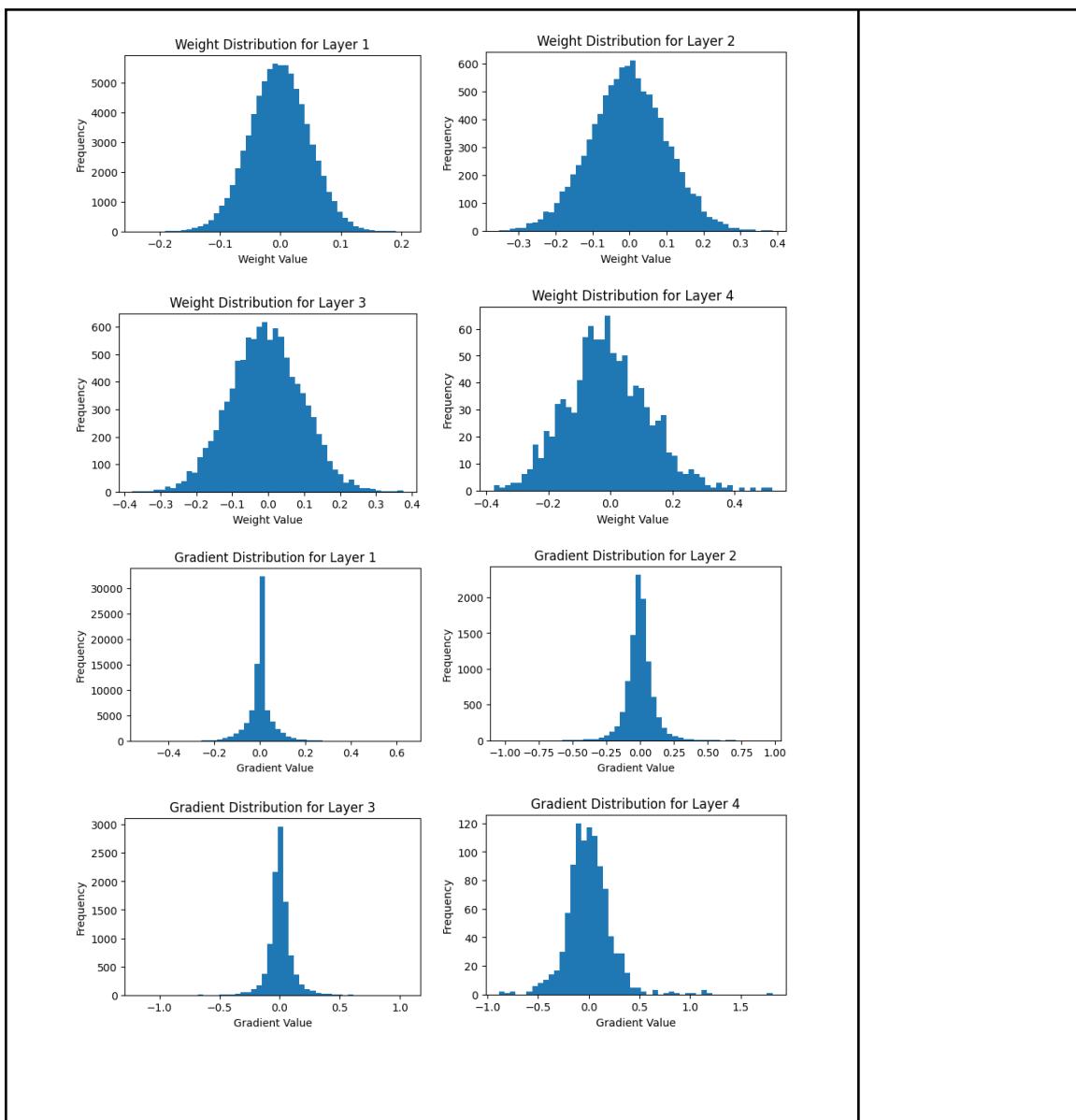




Learning Rate: 0.001

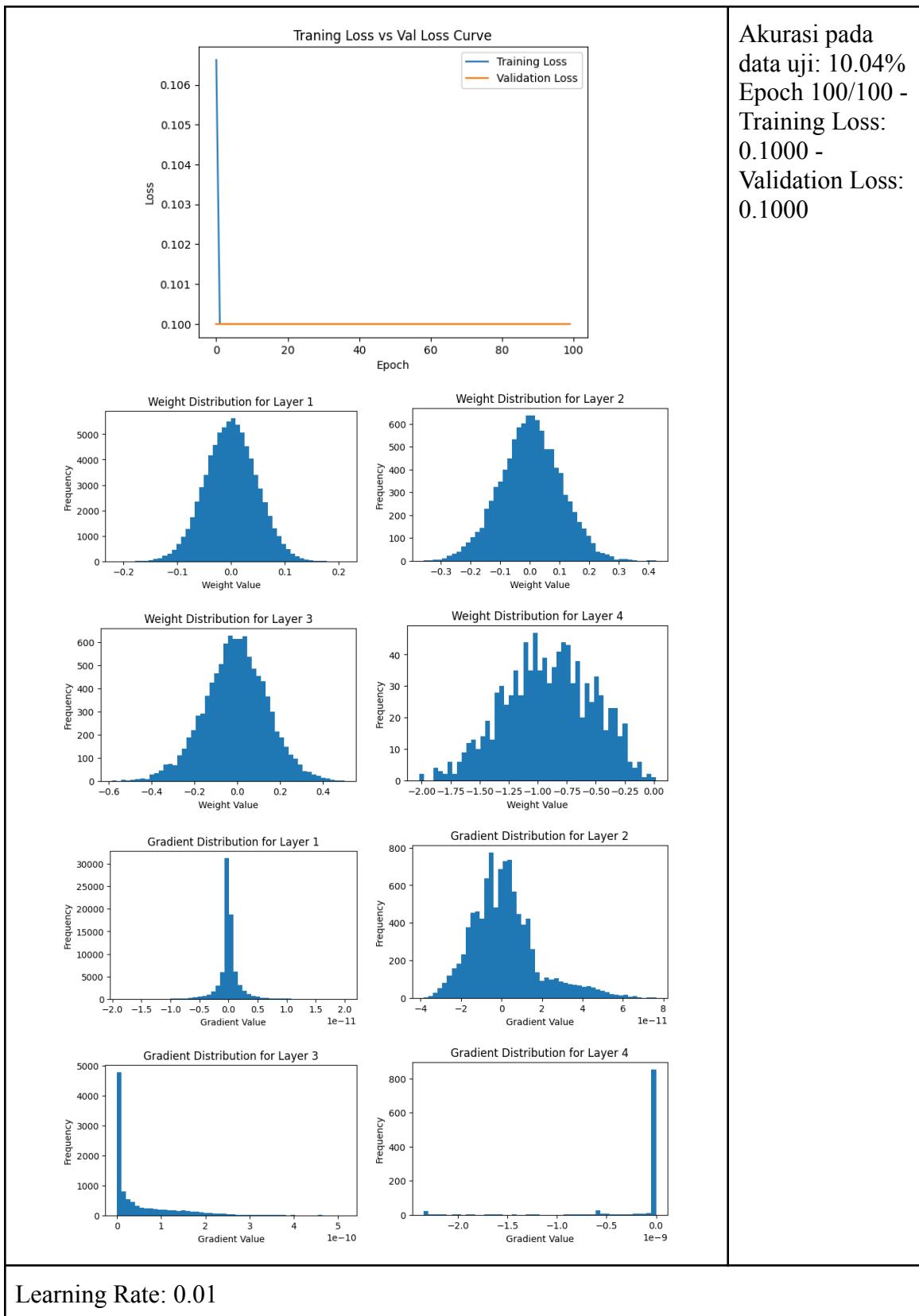


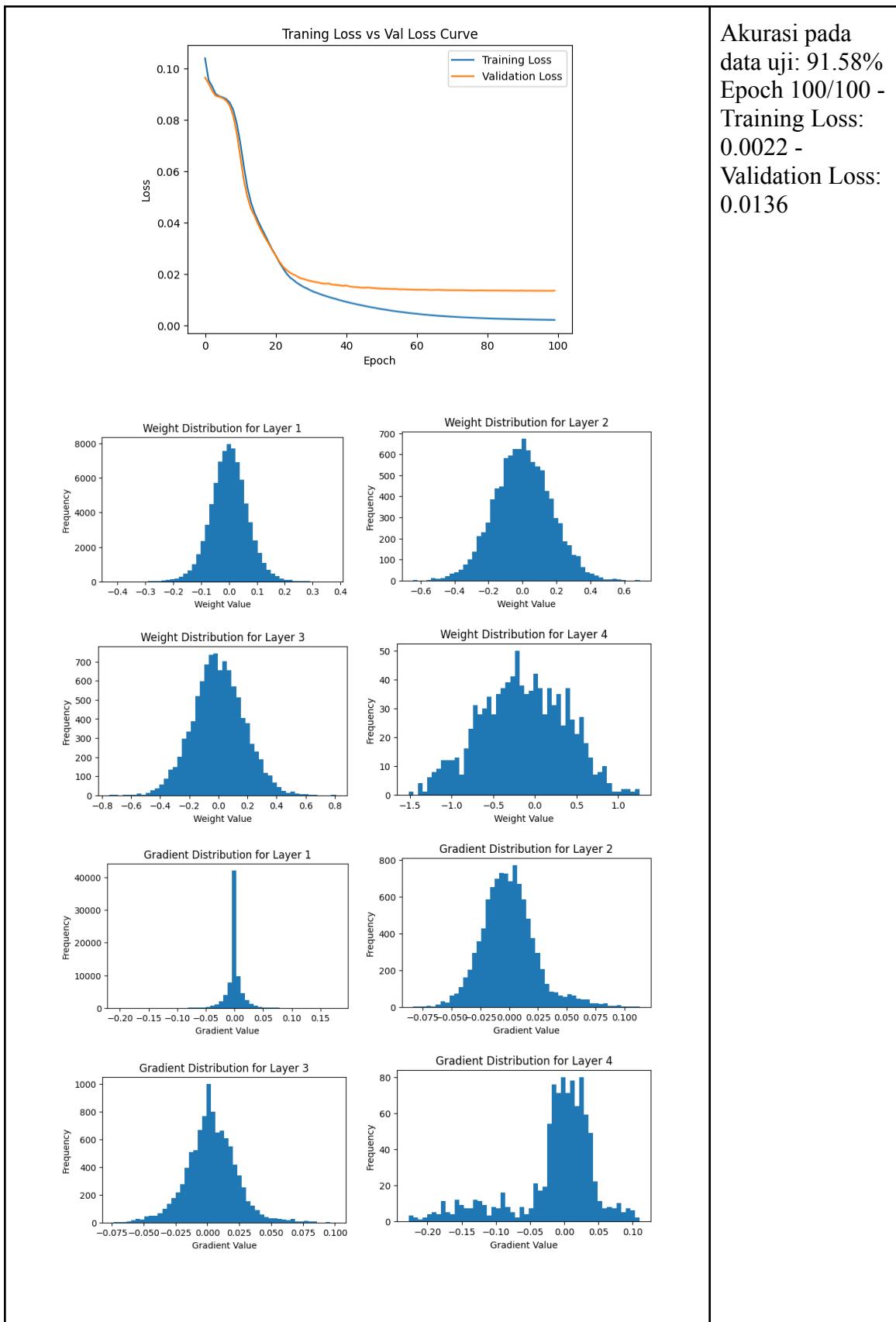
Akurasi pada data uji: 93.68%  
Epoch 100/100 - Training Loss: 0.0035 - Validation Loss: 0.0202



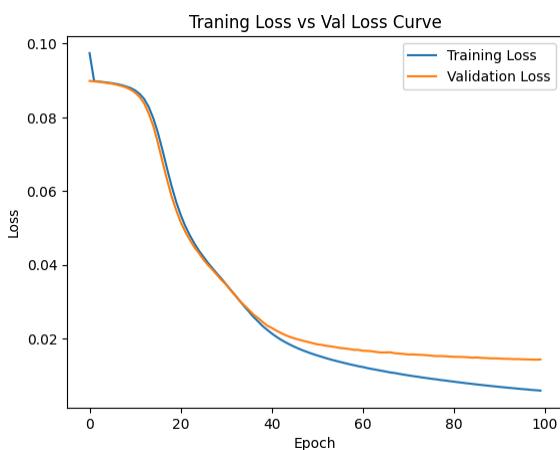
Tabel 3.2.3.2 Hasil Pengujian Learning Rate (dengan Sigmoid)

Learning Rate: 0.1
--------------------

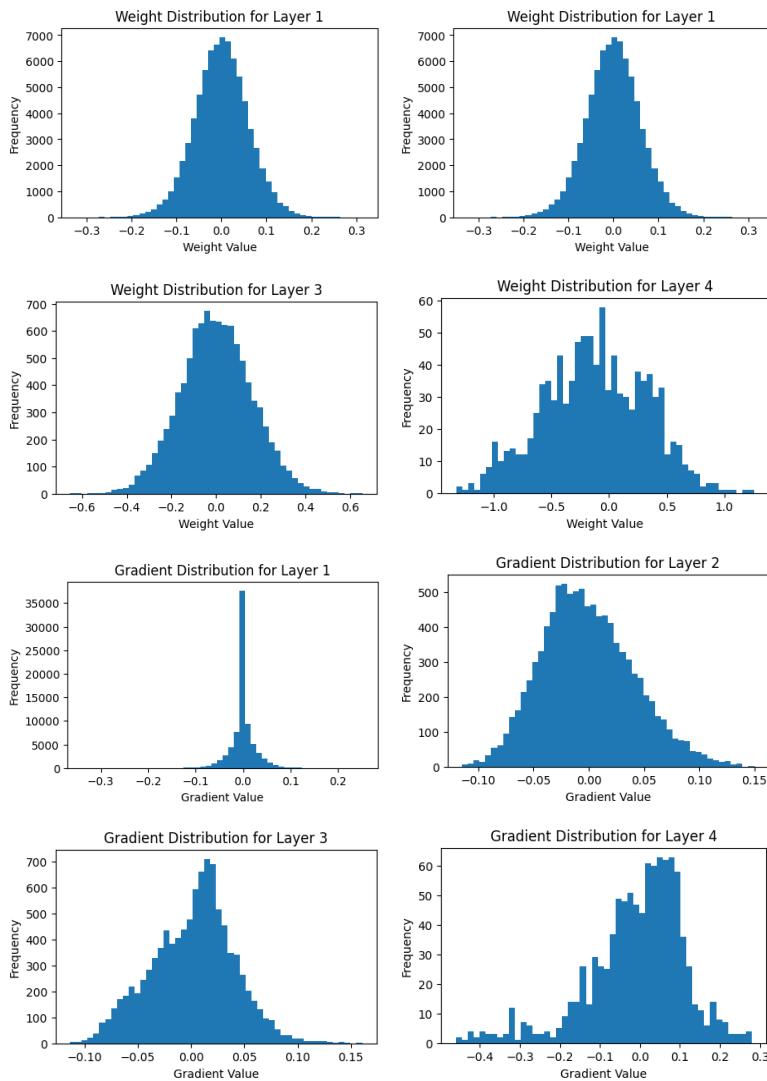




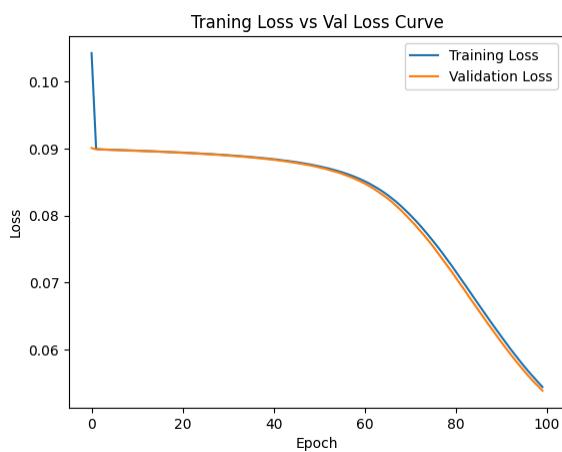
Learning Rate: 0.005



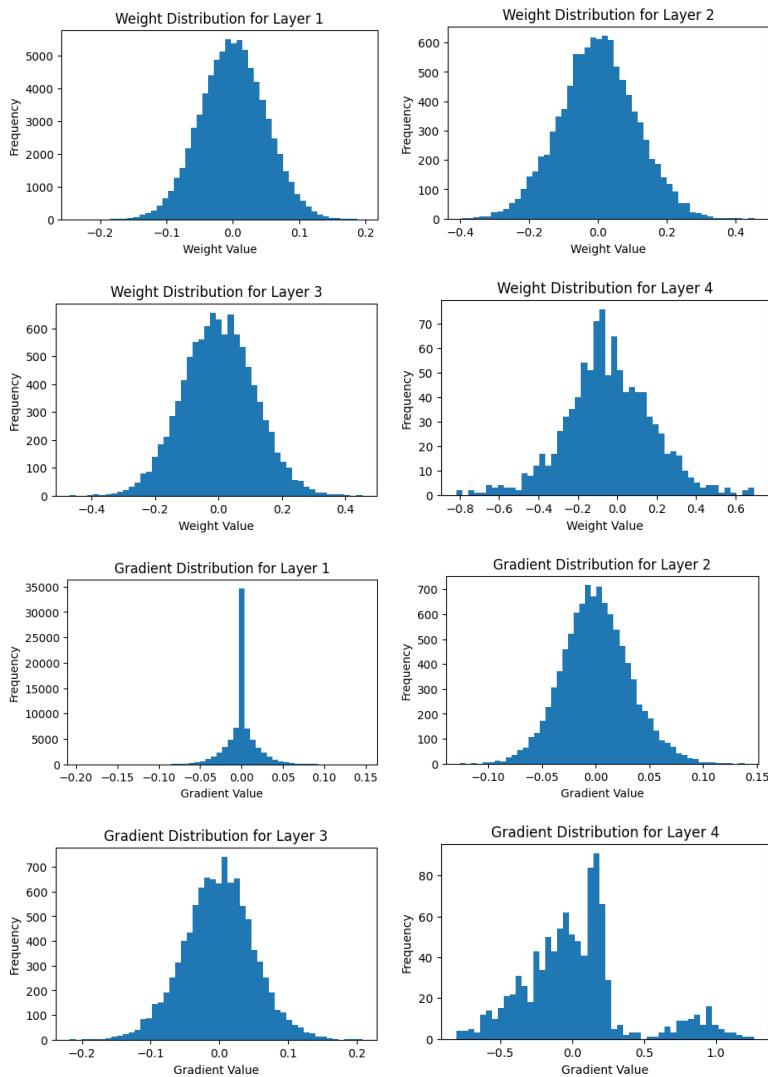
Akurasi pada data uji: 91.08%  
Epoch 100/100 - Training Loss: 0.0058 - Validation Loss: 0.0143



Learning Rate: 0.001



Akurasi pada data uji: 64.85%  
Epoch 100/100 - Training Loss: 0.0544 - Validation Loss: 0.0539

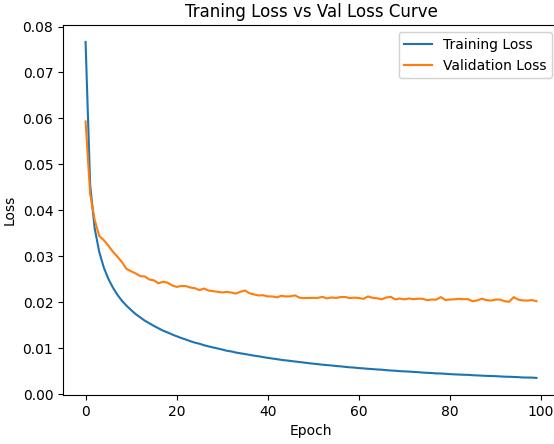
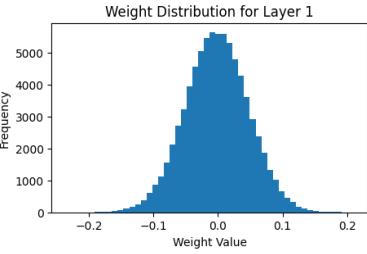
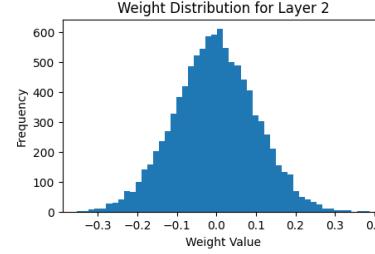


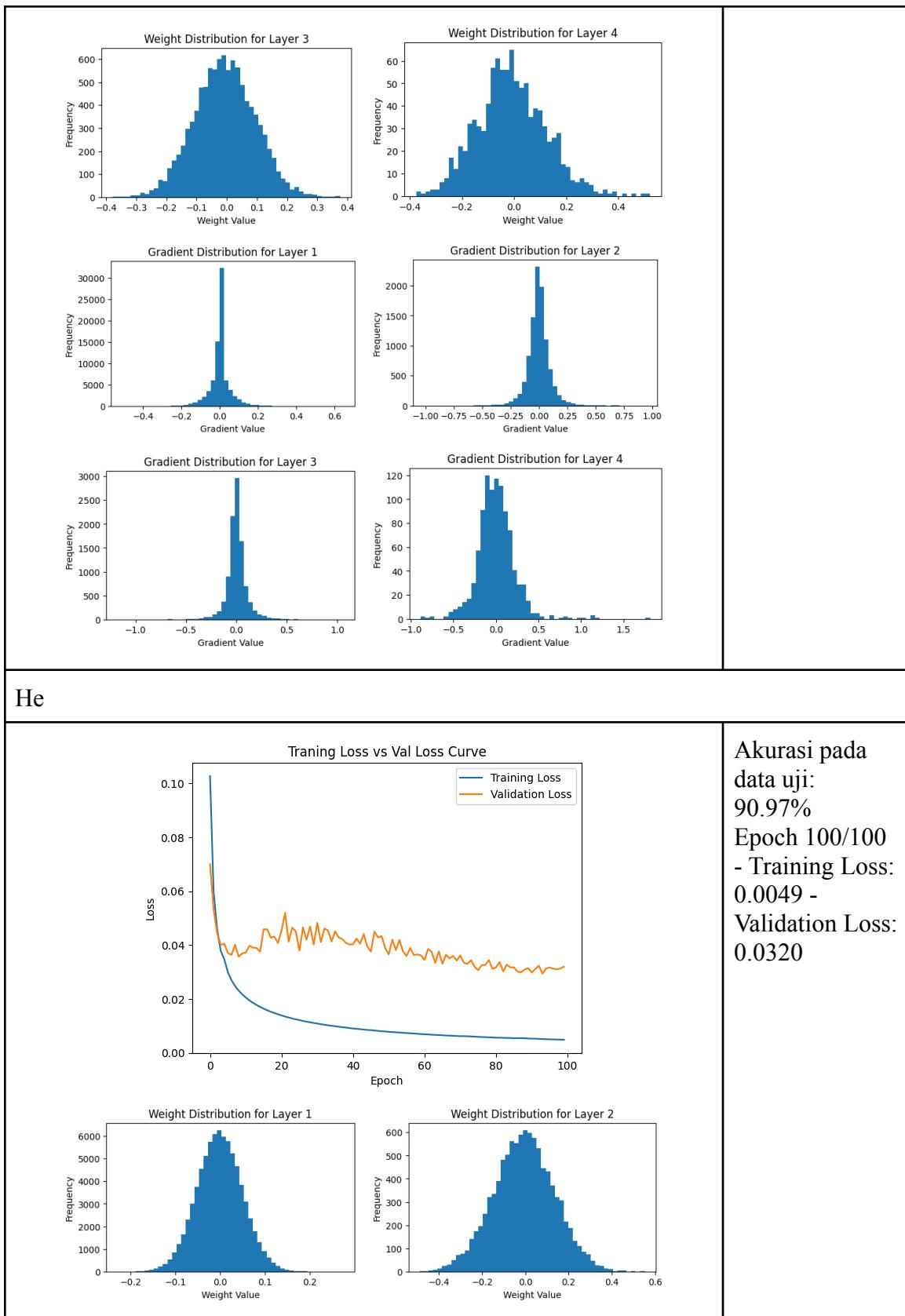
### 3.2.4. Variasi Metode Inisialisasi Bobot

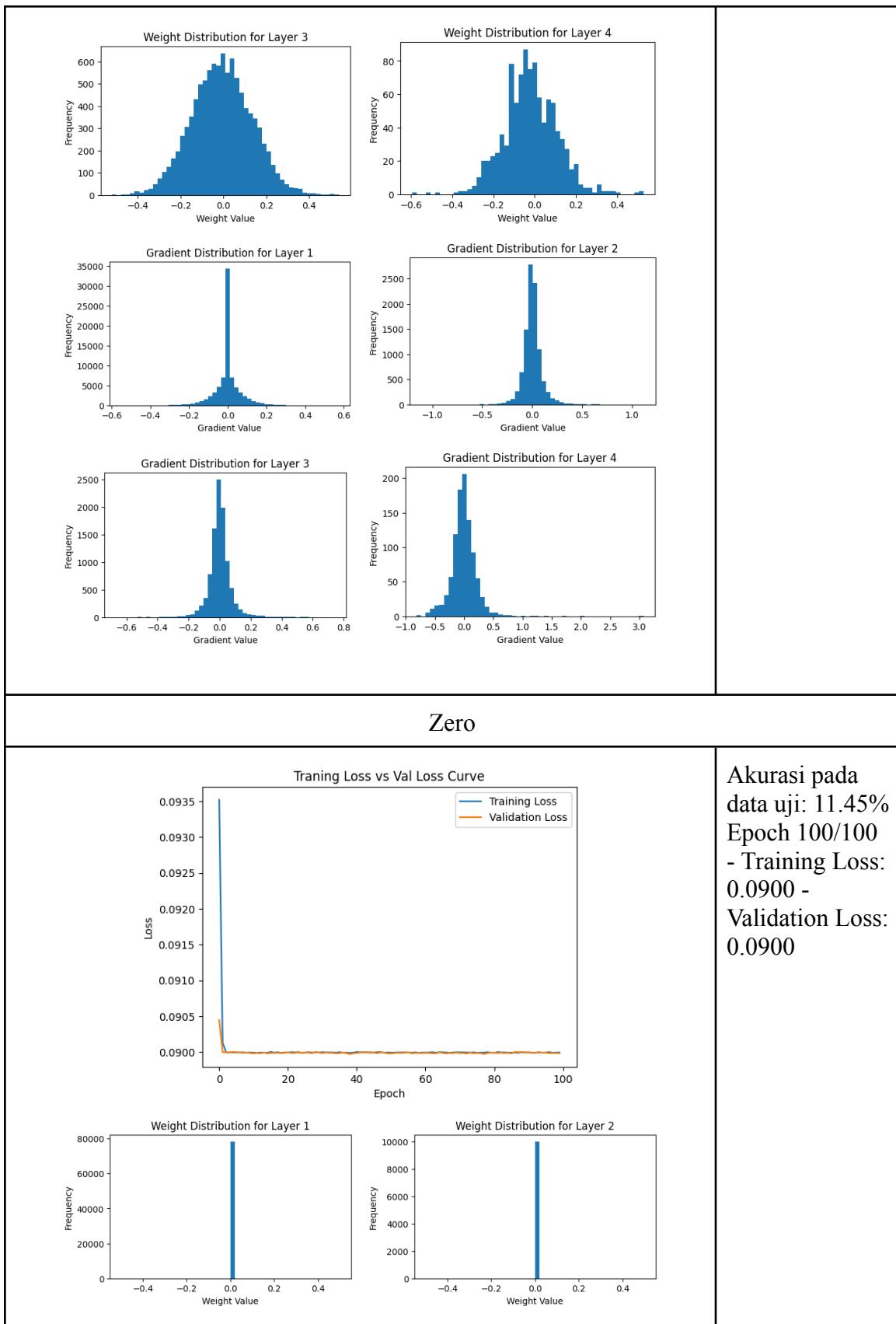
Pengujian ini dilakukan dengan mengubah parameter jenis fungsi inisialisasi bobot, parameter lain disetel sama dengan nilai sebagai berikut:

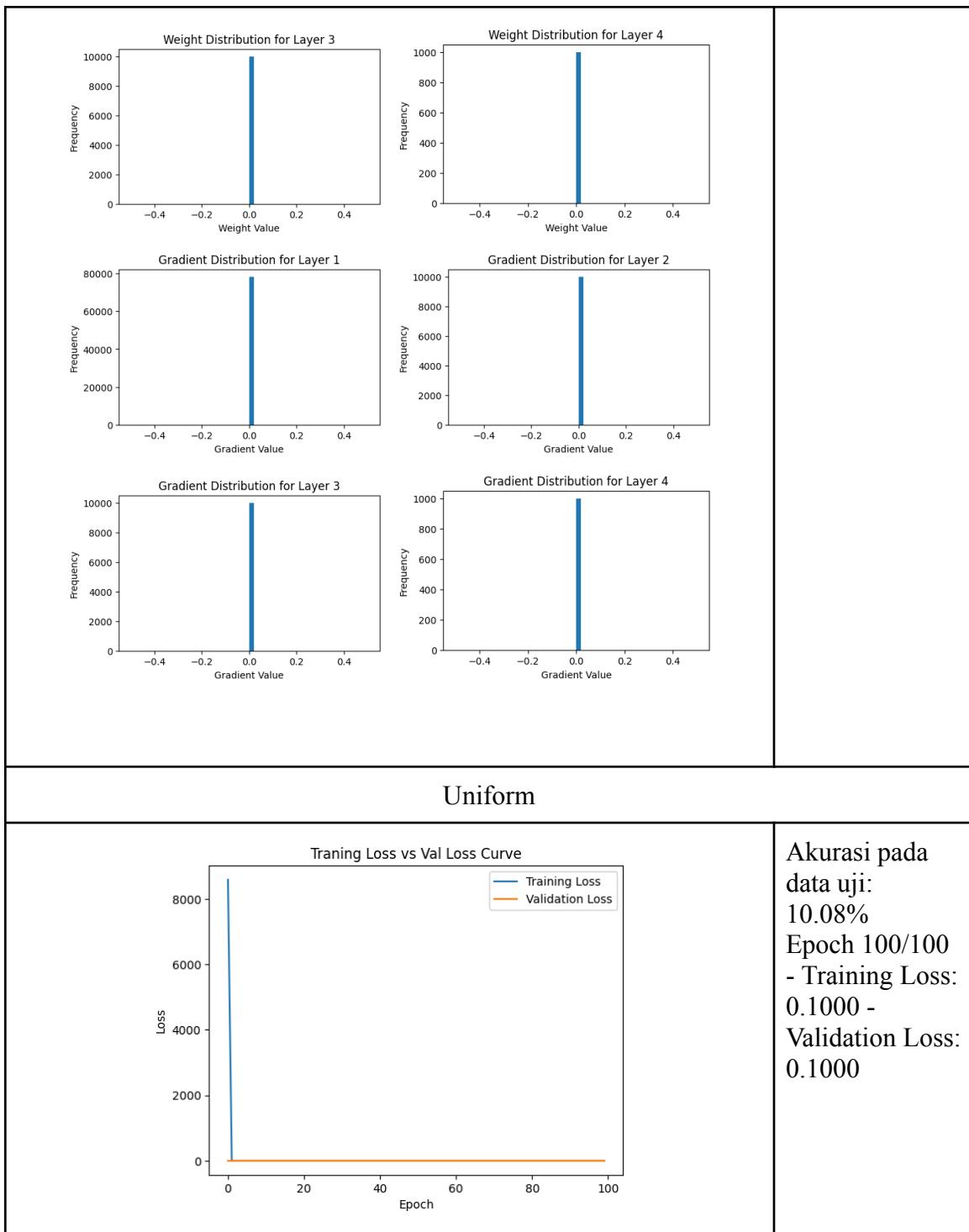
- Depth: 3
- Width: 100
- Fungsi aktivasi: swish
- Fungsi loss: MSE
- Epoch: 100
- Learning rate: 0.001
- Batch size: 200
- Regularisation: None
- RMSNorm: False

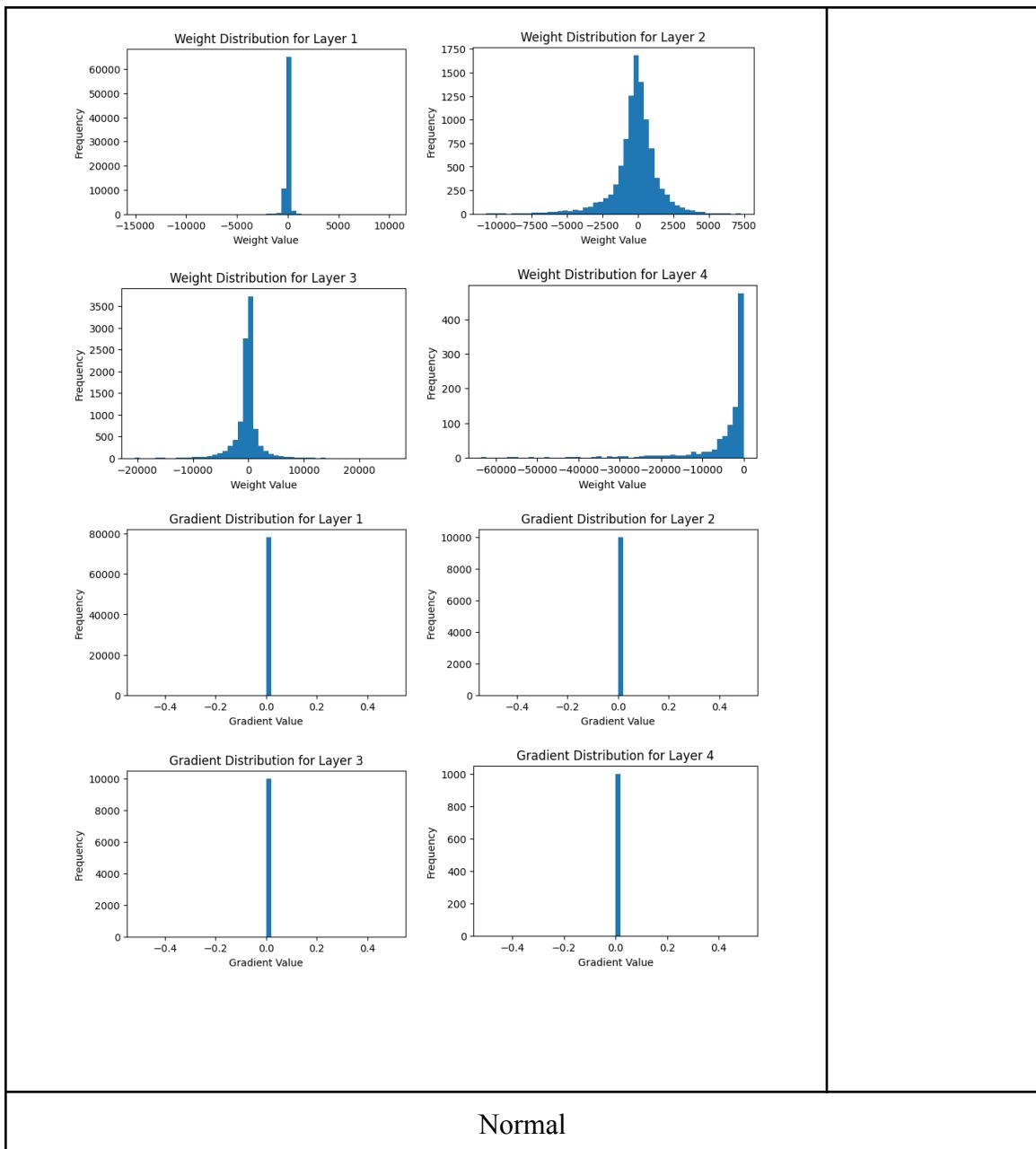
Tabel 3.2.4.1 Hasil Pengujian Inisialisasi Bobot

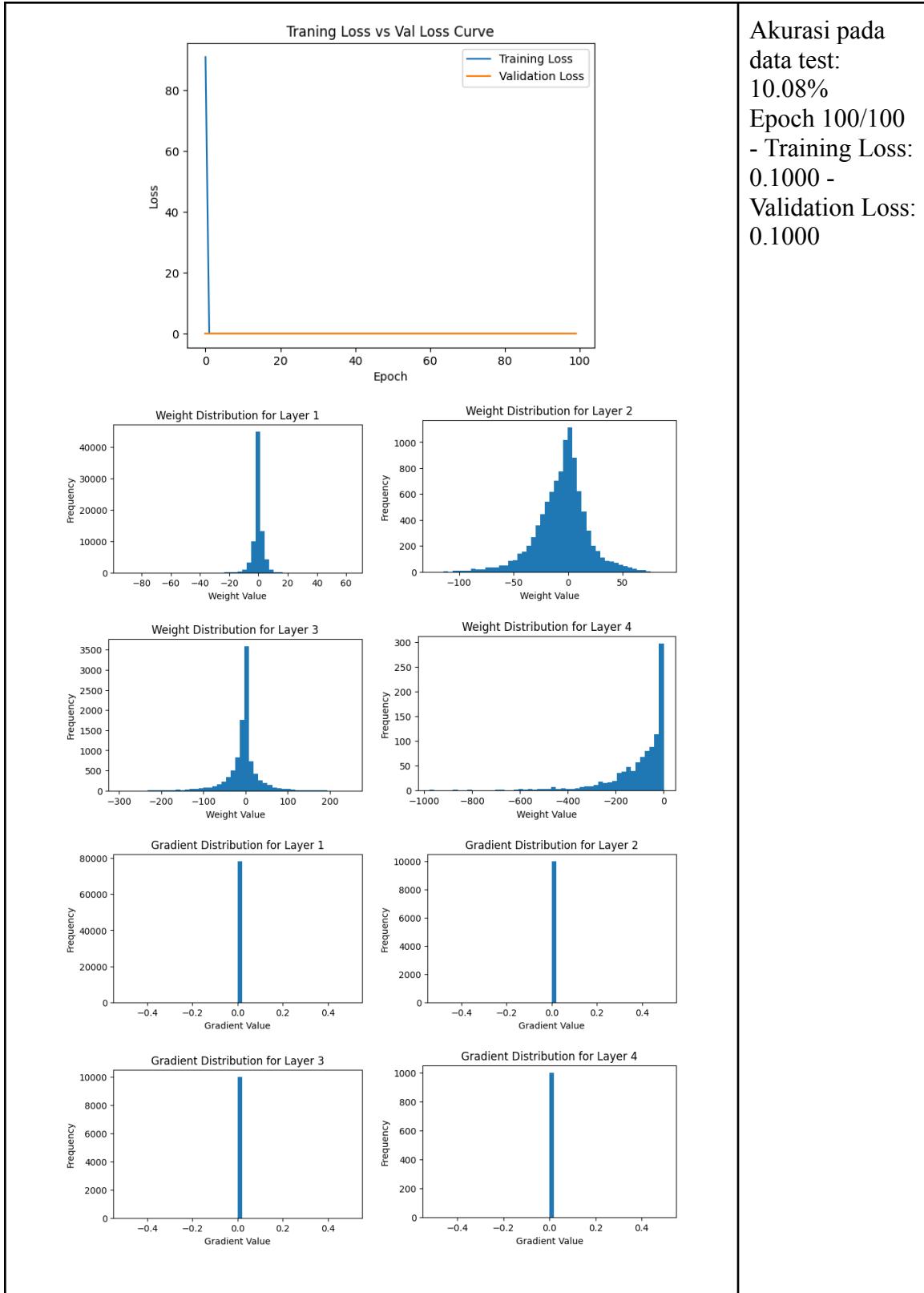
Xavier	
  	<p>Akurasi pada data uji: 93.68%  Epoch 100/100  - Training Loss: 0.0035 -  Validation Loss: 0.0202</p>









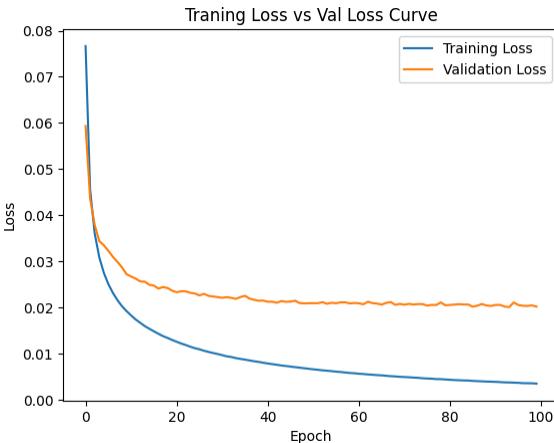
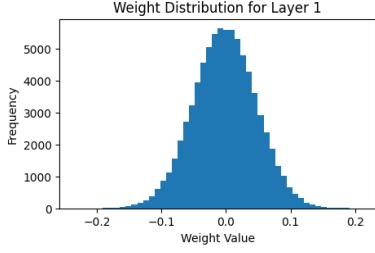
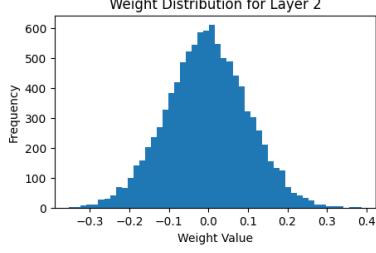


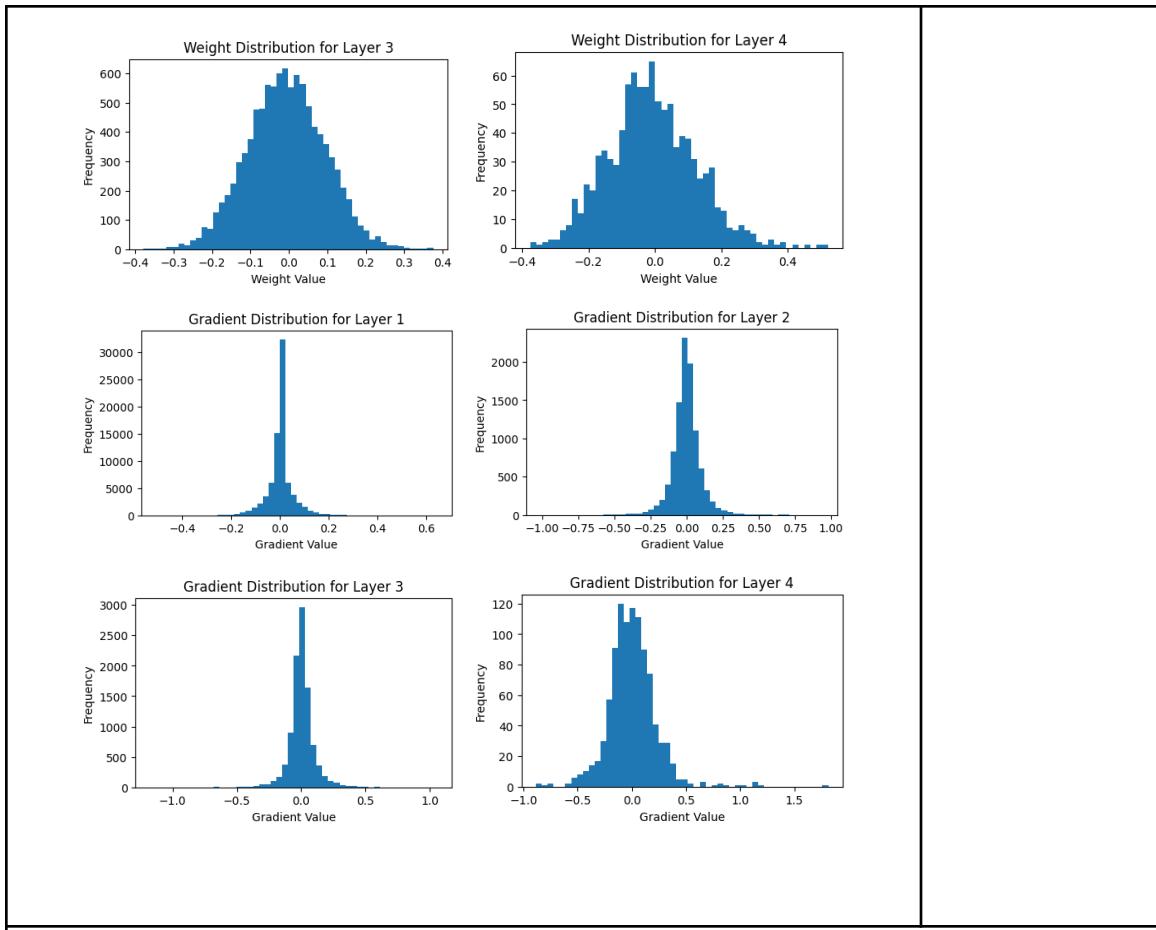
### 3.2.5. Pengujian Regularisasi L1 dan L2

Pengujian ini dilakukan dengan membandingkan model tanpa regularisasi, dengan regularisasi L1, dan dengan regularisasi L2. parameter lain disetel sama dengan nilai sebagai berikut:

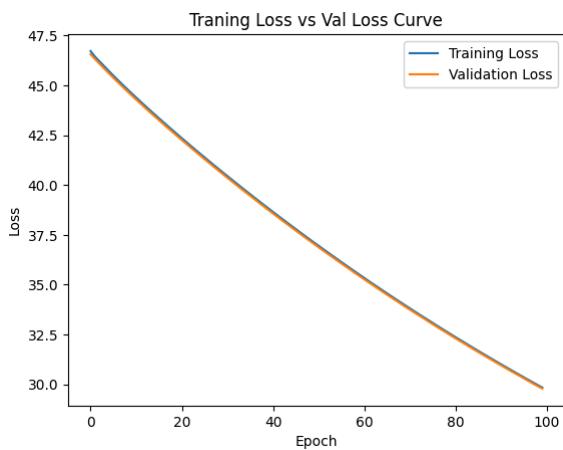
- Depth: 3
- Width: 100
- Fungsi aktivasi: swish
- Fungsi loss: MSE
- Epoch: 100
- Learning rate: 0.001
- Batch size: 200
- RMSNorm: False

Tabel 3.2.5.1 Hasil Pengujian Regularisasi

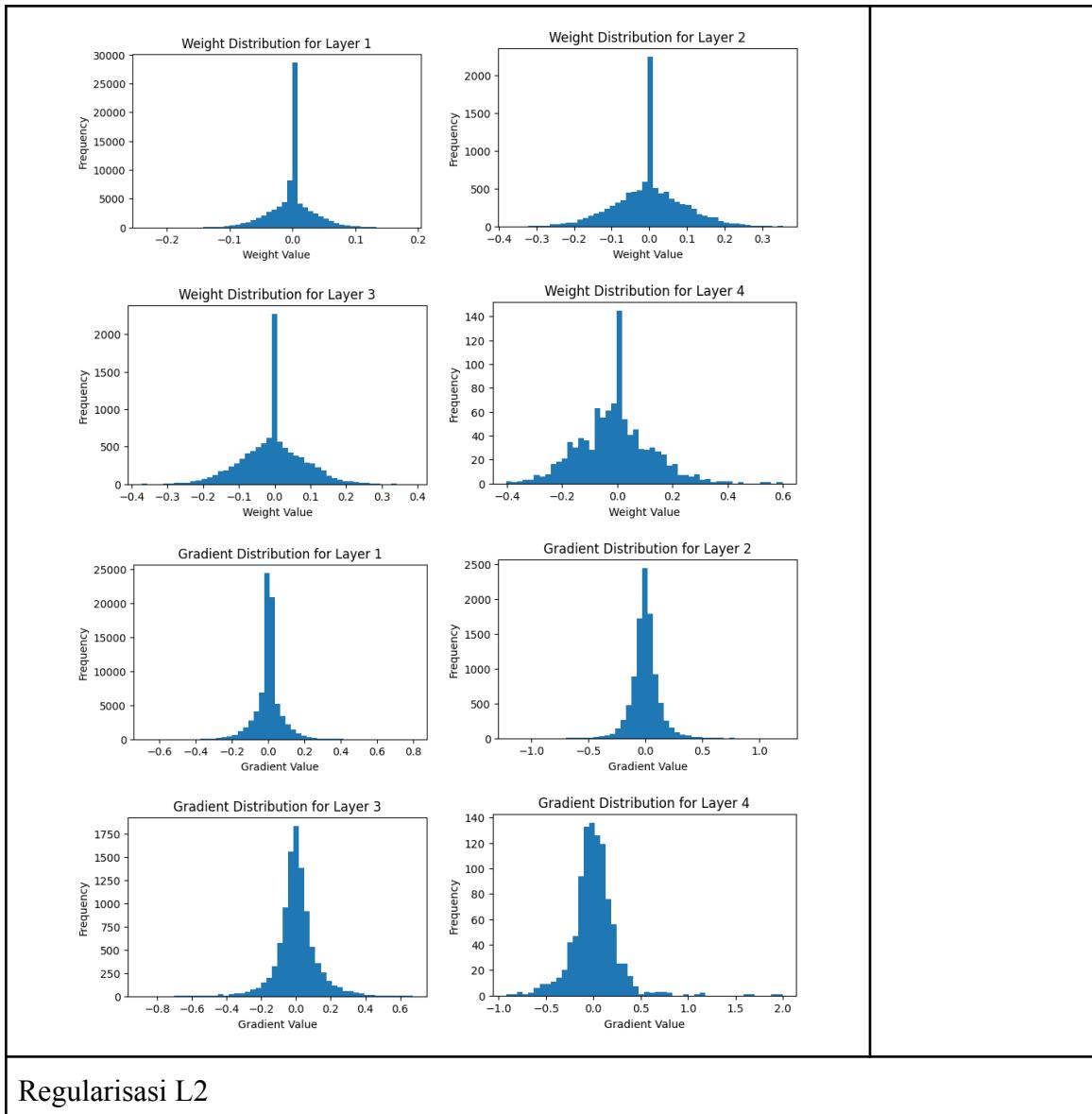
Tanpa Regularisasi		Akurasi pada data uji: 93.68% Epoch 100/100 - Training Loss: 0.0035 - Validation Loss: 0.0202
  		

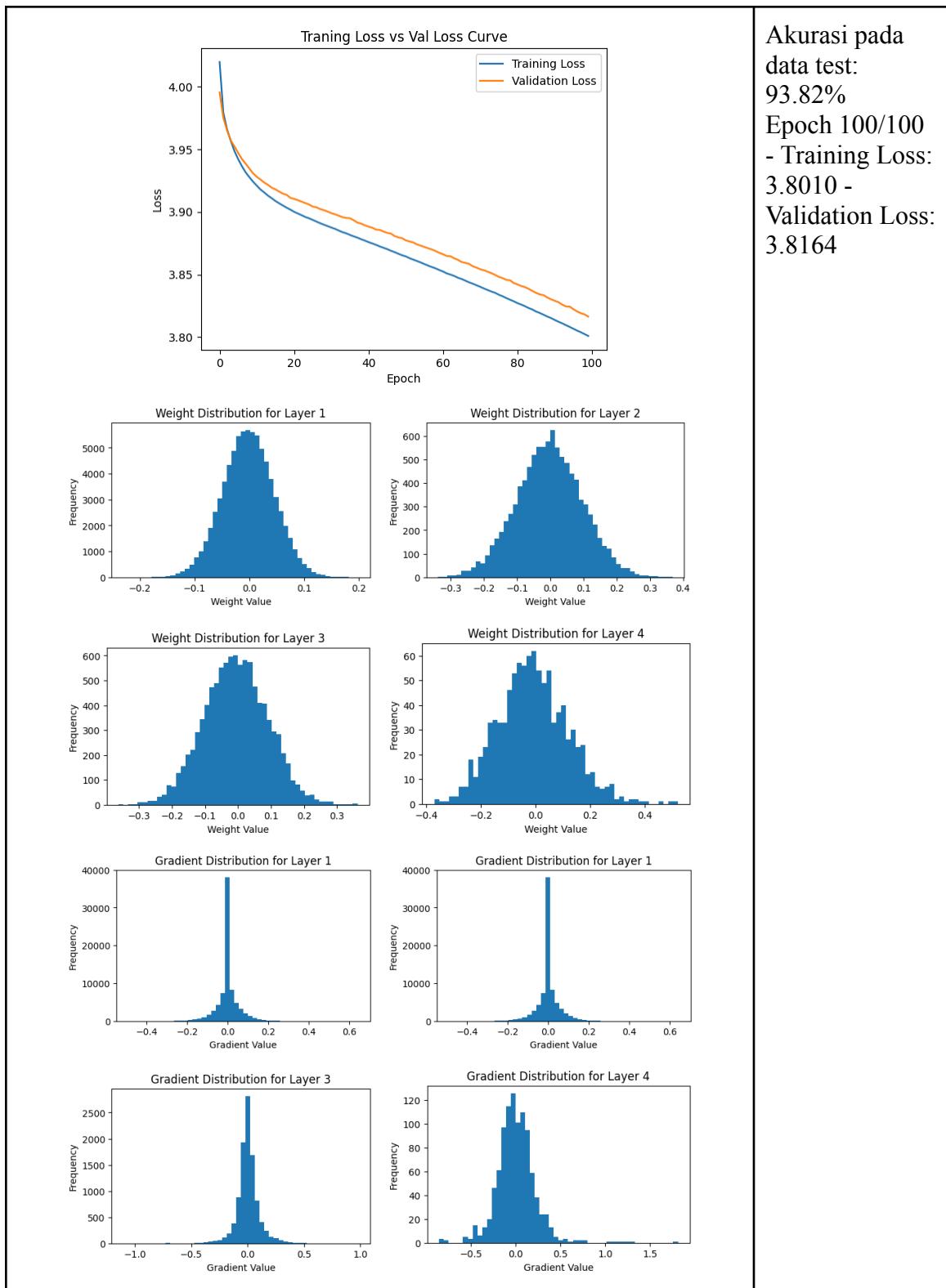


### Regularisasi L1



Akurasi pada data test:  
94.39%  
Epoch 100/100  
- Training Loss:  
29.8387 -  
Validation Loss:  
29.7871



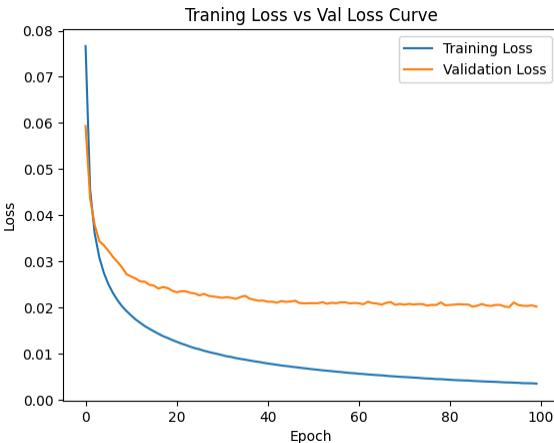
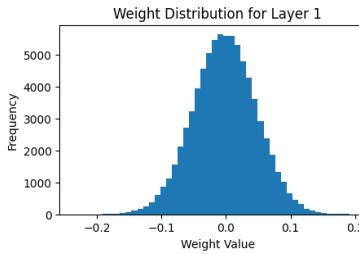
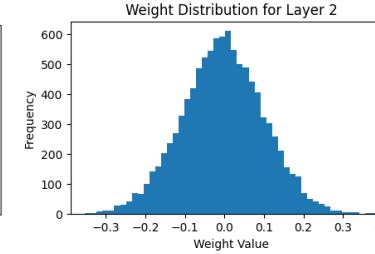


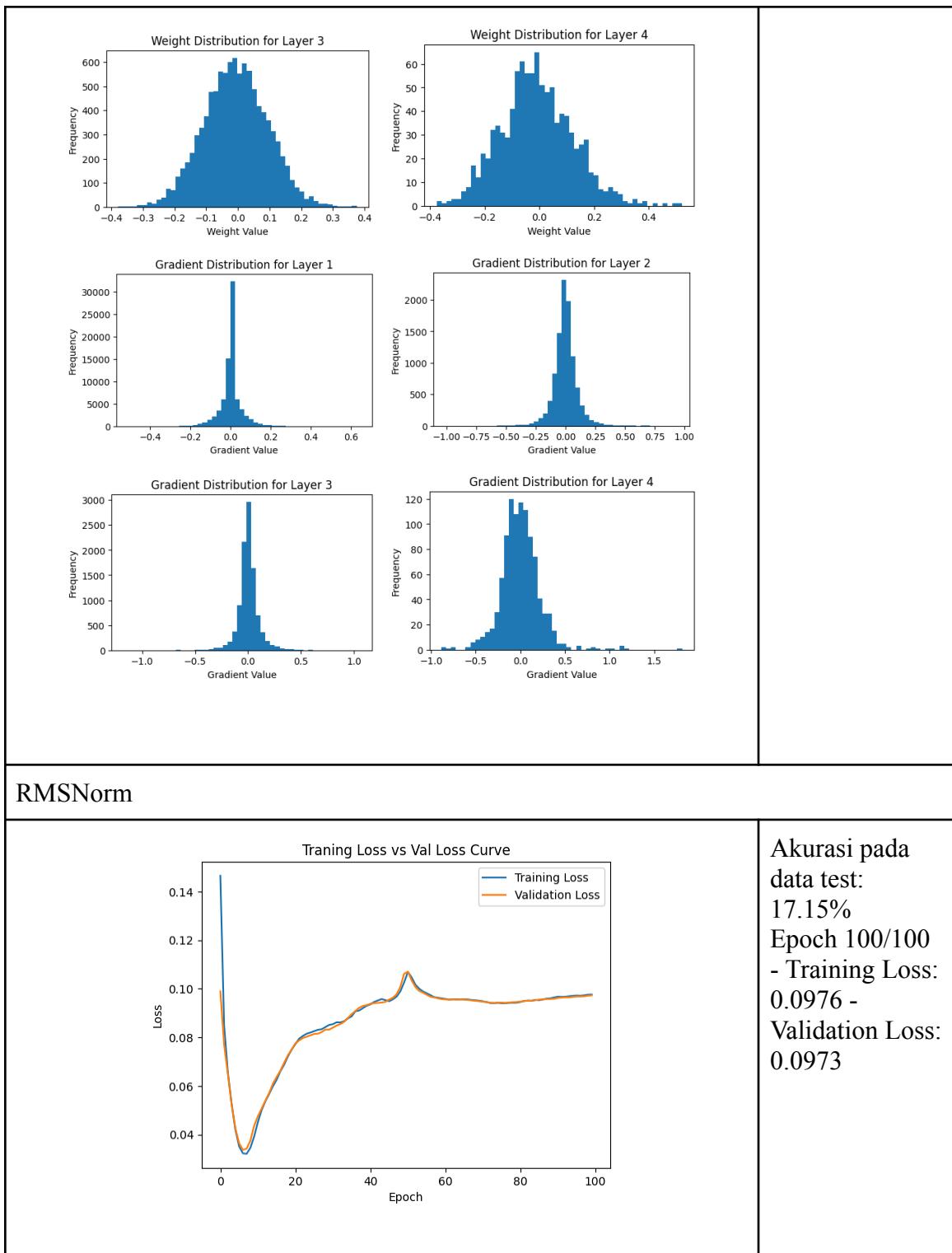
### 3.2.6. Pengujian dengan RMSNorm

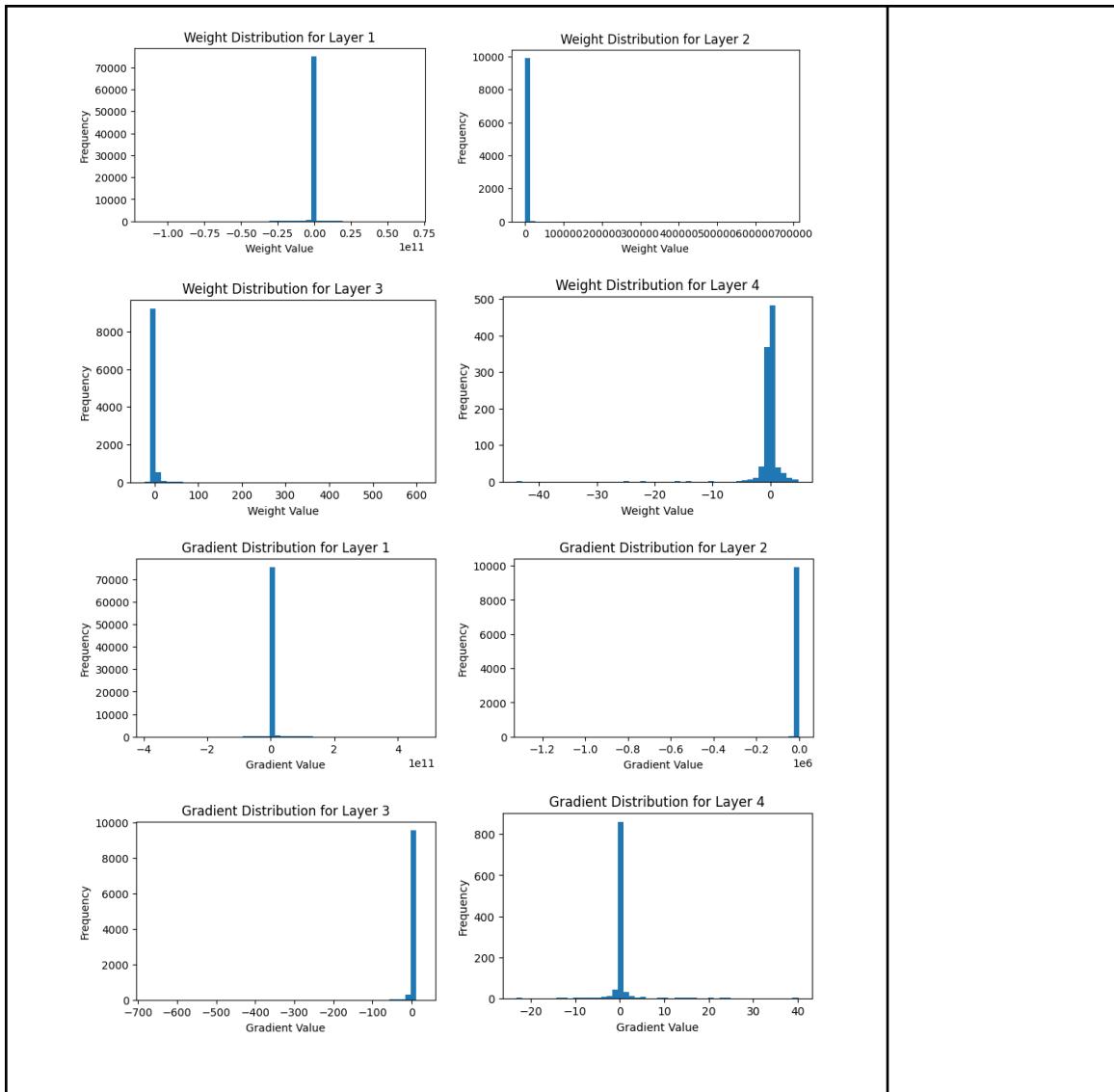
Pada pengujian ini dilakukan dengan membandingkan model tanpa RMSNorm, dan dengan RMSNorm. parameter lain disetel sama dengan nilai sebagai berikut:

- Depth: 3
- Width: 100
- Fungsi aktivasi: swish
- Fungsi loss: MSE
- Epoch: 100
- Learning rate: 0.01
- Batch size: 200
- Regularisation: None
- RMSNorm: False

Tabel 3.2.6.1 Hasil Pengujian RMSNorm

Tanpa RMSNorm	
 <p>Traning Loss vs Val Loss Curve</p> <p>Training Loss (Blue line) starts at approximately 0.075 and decreases steadily to about 0.0035. Validation Loss (Orange line) starts at approximately 0.055 and decreases rapidly to about 0.0202 by epoch 100.</p>  <p>Weight Distribution for Layer 1</p> <p>A histogram showing the frequency distribution of weight values for Layer 1. The x-axis ranges from -0.2 to 0.2, and the y-axis (Frequency) ranges from 0 to 5000. The distribution is centered around 0.0, with a peak frequency of approximately 5500.</p>  <p>Weight Distribution for Layer 2</p> <p>A histogram showing the frequency distribution of weight values for Layer 2. The x-axis ranges from -0.3 to 0.4, and the y-axis (Frequency) ranges from 0 to 600. The distribution is centered around 0.0, with a peak frequency of approximately 600.</p>	<p>Akurasi pada data uji: 93.68% Epoch 100/100 - Training Loss: 0.0035 - Validation Loss: 0.0202</p>





### 3.2.7. Pengujian Berbanding library MLP SkLearn

Pengujian dilakukan dengan mengubah parameter fungsi aktivasi. SKLearn hanya mendukung 4 fungsi aktivasi: linear, sigmoid, tanh, dan ReLU Parameter lain disetel sama dengan nilai sebagai berikut:

- Depth: 3
- Width: 100
- Fungsi loss: MSE
- Epoch: 100
- Weight Initialisation: Xavier
- Learning rate: 0.001
- Batch size: 200
- Regularisation: None
- RMSNorm: False

Tabel 3.2.7.1 Hasil Perbandingan Library MLP SkLearn

Sigmoid	Akurasi pada data test: 61.67% Akurasi MLPClassifier pada data test: 90.93% Perbandingan Akurasi: FFNN Accuracy: 61.67% MLPClassifier Accuracy: 90.93%
ReLU	Akurasi pada data test: 73.61% Akurasi MLPClassifier pada data test: 93.17% Perbandingan Akurasi: FFNN Accuracy: 73.61% MLPClassifier Accuracy: 93.17%
Tanh	Akurasi pada data test: 91.42% Akurasi MLPClassifier pada data test: 91.39% Perbandingan Akurasi: FFNN Accuracy: 91.42% MLPClassifier Accuracy: 91.39%

## BAB IV

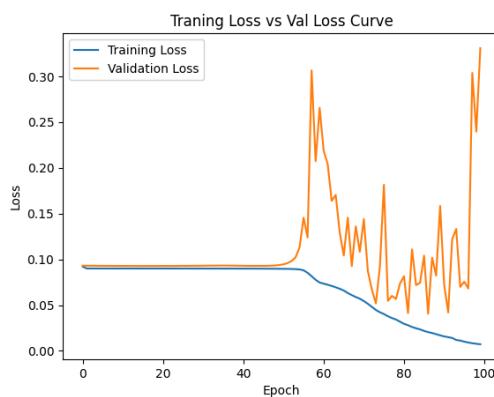
### ANALISIS

#### 4.1. Pengaruh depth dan width

Depth menandakan banyaknya jumlah hidden layer yang digunakan. Semakin banyak hidden layer, semakin dalam juga pembelajaran model (*deep learning*). Semakin meningkatnya kedalaman, semakin membantu model untuk mempelajari pattern yang kompleks, tetapi apabila tidak ditangani dengan benar dapat memunculkan masalah, seperti overfitting, vanishing/exploding gradient, serta kompleksitas.

Width menandakan banyaknya jumlah neuron dalam satu layer. Semakin banyak neuron, model semakin baik dalam menangkap variasi dan fitur yang lebih luas pada setiap layer. Namun, apabila tidak ditangani dengan benar dapat memunculkan beberapa masalah, seperti overfitting, beban komputasi yang berat, serta redundansi karena dapat terjadi pengulangan pembelajaran yang telah dilakukan neuron yang lain dalam satu layer.

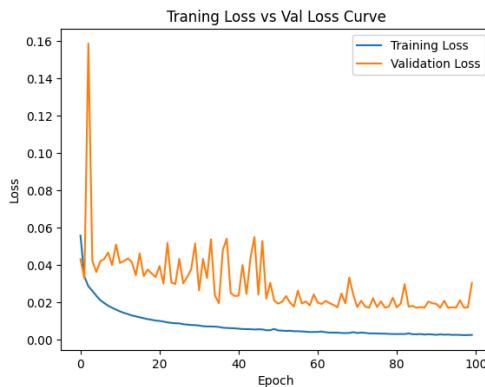
Berdasarkan hasil pengujian, kita dapat melihat pola bahwa akurasi paling tinggi terjadi pada kedalaman 3, meningkat jauh dari kedalaman 1, tetapi mengalami penurunan saat kedalaman 10 dan semakin menurun di kedalaman 12.



Berdasarkan grafik di atas (pada depth 12), dapat dilihat bahwa training loss menurun secara stabil dan cukup datar setelah dilakukan beberapa epoch. Hal ini menunjukkan bahwa model terus belajar dengan baik pada data train dan tidak mengalami kesulitan dalam meminimalkan error. Sebaliknya, validation loss berfluktuasi dengan sangat tajam, dengan lonjakan besar yang terjadi pada sekitar epoch ke-50. Hal ini mengindikasikan bahwa model mulai mengalami kesulitan dalam melakukan generalisasi data validasi. Ketidakstabilan ini menandakan bahwa

model mengalami *overfitting*, di mana model terlalu fokus pada *data train* dan tidak dapat beradaptasi dengan baik pada *unseen data*.

Sedangkan untuk hasil pengujian *width* atau jumlah neuron per layer mendapatkan nilai akurasi tertinggi pada banyak neuron 500 per layer, merupakan dari peningkatan pengujian neuron yang lebih sedikit, namun mengalami penurunan saat jumlah neuron 1000.



Berdasarkan grafik di atas (pada width 1000), training loss mengalami penurunan yang stabil dan halus seiring bertambahnya epoch, yang menandakan bahwa model telah belajar dengan baik pada data pelatihan dan secara konsisten memperbaiki kesalahan. Sebaliknya, validation loss menunjukkan fluktuasi yang cukup besar, terutama pada awal pelatihan, tetapi akhirnya mulai stabil pada nilai yang lebih rendah setelah beberapa epoch. Meskipun demikian, ada kecenderungan validation loss untuk sedikit meningkat pada akhir pelatihan, yang tentunya menimbulkan kecurigaan bahwa model mengalami *overfitting*, di mana model mulai kehilangan kemampuan untuk melakukan generalisasi pada *data validation* meskipun terus memperbaiki kesalahan pada *data train*.

Berdasarkan analisis yang dilakukan di atas, terbukti bahwa peningkatan *depth* dan *width* tidak selalu berdampak baik. Setelah titik tertentu, model justru mengalami kesulitan dalam pelatihan dan akurasi menurun.

#### 4.2. Pengaruh fungsi aktivasi

Dari hasil pengujian yang dilakukan, didapatkan urutan peringkat performa fungsi aktivasi sebagai berikut dari terendah ke terbaik:

- Softplus (46.83%)
- ReLU (73.61%)
- Sigmoid (64.85%)

- Tanh (91.51%)
- ELU (93.30%)
- Swish (93.68%)

Pada pengujian dengan fungsi aktivasi ReLU, akurasi model mencapai 73.61%, yang mana ini lebih baik dibandingkan dengan fungsi sigmoid yang hanya menghasilkan akurasi 64.85%. Hal ini terjadi karena ReLU memiliki kemampuan untuk menghindari masalah *vanishing gradient* yang sering terjadi pada sigmoid, di mana gradien yang sangat kecil dapat memperlambat proses pembelajaran dan menghambat kinerja model, terutama pada data yang lebih kompleks. Selain itu, ReLU juga lebih efektif dalam menangani input yang nilainya positif karena memberikan nilai gradien konstan untuk input yang lebih besar dari nol, yang tentunya memungkinkan *updating* bobot yang jauh lebih stabil dan lebih cepat. Sebaliknya, sigmoid cenderung mengalami nilai input yang sangat besar atau bahkan sangat kecil, yang menyebabkan gradien menjadi lebih dari ‘sangat kecil’ dan dapat memperlambat pelatihan model. Meskipun memiliki kelebihan, ReLU juga rentan mengalami masalah, yaitu *dying ReLU*, dimana ketika nilai output bernilai dominan negatif, maka saat melakukan *backpropagation*, neuron akan ‘mati’ atau istilahnya tidak aktif karena gradien akan selalu bernilai 0, sehingga bobot tidak akan mengalami perubahan dan membuat model sulit untuk belajar di *epoch* berikutnya.

Sementara itu, fungsi aktivasi tanh menghasilkan akurasi 91.51%. Kelebihan dari fungsi aktivasi tanh adalah outputnya terpusat di nol, yang membuat distribusi gradien lebih seimbang dan membantu model belajar pola yang lebih kompleks dengan lebih efektif. Kemudian, ELU (Exponential Linear Unit) memberikan akurasi 93.30%, yang mana ELU dapat berfungsi dengan baik karena mencegah masalah *vanishing gradient* dan *dying ReLU* pada input negatif, berkat komponennya yang eksponensial.

Selanjutnya, fungsi aktivasi swish menghasilkan akurasi tertinggi (93.68%). Hal ini terjadi karena swish memberikan gradien yang lebih lancar dan tidak cepat jenuh, yang memungkinkan *updating* bobot yang lebih stabil selama pelatihan.

Sebaliknya, Softplus hanya menghasilkan akurasi sekitar 46.83%. Meskipun softplus merupakan pendekatan yang lebih *smooth* dari ReLU, gradien yang dihasilkan mungkin terlalu lemah untuk menangkap kompleksitas data secara efektif, sehingga menghambat proses pembelajaran yang optimal.

Namun perlu diperhatikan lagi bahwa Setiap fungsi aktivasi memiliki kelebihan dan kekurangannya masing-masing, sehingga efektivitasnya tergantung pada masalah yang ingin diselesaikan, *hyperparameter* (misalkan pengaturan *learning rate*, *depth*, *width*, *batch size*, jumlah epoch, dsb), atau teknik optimisasi yang digunakan (apakah pakai regularisasi, normalisasi, metode inisialisasi, dsb).

#### **4.3. Pengaruh learning rate**

*Learning rate* menentukan seberapa besar langkah pembaruan bobot yang dilakukan setiap kali gradien dihitung. Jika learning rate terlalu tinggi, pembaruan bobot menjadi terlalu agresif sehingga model dapat melewati minimum global (konvergen), yang menyebabkan loss tidak stabil atau bahkan divergen. Di sisi lain, jika *learning rate* terlalu kecil, perubahan bobot di setiap iterasi menjadi sangat lambat, sehingga model membutuhkan lebih banyak epoch untuk mencapai konvergensi. Akibatnya, meskipun loss mungkin turun secara perlahan, waktu pelatihan menjadi sangat lama dan model mungkin terjebak di *local minimum* yang kurang optimal.

Hasil percobaan menunjukkan bahwa *learning rate* yang terlalu tinggi (0.1) menyebabkan performa model sangat buruk dengan akurasi hanya 10.08% untuk kedua fungsi aktivasi, baik Swish maupun Sigmoid, menunjukkan bahwa pembaruan bobot yang terlalu agresif dan liar sehingga menyebabkan proses training tidak stabil.

Pada saat *learning rate* 0.001, sigmoid menunjukkan performa yang lebih buruk (hanya mencapai sekitar 64%), dibandingkan dengan Swish yang menghasilkan akurasi sekitar 93.68%. Namun terdapat hal yang cukup menarik, yaitu pada learning rate 0.01 dan 0.005, Swish tetap mempertahankan performa tinggi, model dengan sigmoid justru mengalami penurunan performa.

Perbandingan ini mengindikasikan bahwa fungsi aktivasi Swish cenderung lebih stabil terhadap variasi learning rate, menghasilkan loss yang lebih rendah pada LR moderat, sedangkan Sigmoid lebih sensitif dan memerlukan learning rate yang sangat rendah untuk mengatasi masalah *vanishing gradient*. Swish menunjukkan sedikit keunggulan dalam hal stabilitas loss dibandingkan Sigmoid.

#### **4.4. Pengaruh inisialisasi bobot**

Pada percobaan menggunakan Swish, inisialisasi bobot dengan metode Xavier menghasilkan performa terbaik dengan akurasi 93.13% dan loss 0.0032, sedangkan He

initialization menghasilkan performa yang sedikit lebih rendah (akurasi 90.75% dan loss 0.0041). Metode inisialisasi Zero, Uniform, dan Normal memberikan hasil yang jauh lebih buruk dengan akurasi sekitar 9-11% dan loss yang tinggi (sekitar 0.0899–0.1000). Hal ini menunjukkan bahwa inisialisasi yang tepat sangat krusial untuk konvergensi yang baik. Xavier, yang mengatur skala bobot berdasarkan jumlah neuron input dan output, membantu menjaga distribusi aktivasi tetap seimbang sehingga model dapat belajar secara efektif dengan fungsi aktivasi Swish. Sebaliknya, inisialisasi Zero gagal memecah simetri antar neuron, sedangkan Uniform dan Normal (dengan parameter yang digunakan) menghasilkan bobot dengan skala yang tidak sesuai sehingga menyebabkan saturasi pada aktivasi, yang pada akhirnya menghambat learning.

#### 4.5. Pengaruh Regularisasi L1 dan L2

Hasil percobaan menunjukkan bahwa tanpa regularisasi, model mencapai akurasi 93.13% dengan nilai loss yang sangat rendah (0.0032). Penerapan regularisasi L1 meningkatkan akurasi hingga 94.03%, namun nilai loss meningkat drastis menjadi 29.7164. Hal ini terjadi karena pada regularisasi L1, terdapat penambahan komponen penalti pada penghitungan loss berupa jumlah absolut bobot, sehingga meskipun model menjadi lebih general dan mampu menghasilkan akurasi yang sedikit lebih baik. Perlu dicantumkan bahwa total loss meningkat karena adanya penambahan komponen penalti pada penghitungan lossnya. Sedangkan pada regularisasi L2, akurasi sedikit membaik menjadi 93.42% dengan loss sebesar 3.7880.

Ketika regularisasi diterapkan, loss tidak turun secara drastis di awal karena penalti regularisasi menambahkan bobot pada fungsi loss. Penalti ini membuat pembaruan bobot lebih stabil, sehingga gradien yang dihasilkan tidak hanya berasal dari error prediksi, tetapi juga dari penalti yang berusaha menjaga bobot tetap kecil. Akibatnya, proses optimisasi menjadi lebih stabil namun lambat, dengan penurunan loss yang lebih landai dibandingkan tanpa regularisasi. Dengan regularisasi, model lebih fokus pada generalisasi dan menghindari overfitting.

#### 4.6. Perbandingan Normalisasi RMSNorm

Berdasarkan hasil eksperimen penggunaan RMSNorm pada model FFNN, pada model tanpa RMSNorm, akurasi pada data uji mencapai 93.68%, dengan nilai training loss sebesar 0.0035 dan validation loss 0.0202 setelah 100 epoch. Grafik loss menunjukkan penurunan yang stabil dan konvergen, di mana training loss menurun tajam di awal dan terus menurun secara

konsisten, sedangkan validation loss menurun lalu melandai stabil, menunjukkan *training* tidak overfitting.

Sebaliknya, pada model dengan RMSNorm, performanya justru turun drastis dengan akurasi hanya 17.15%. Training loss dan validation loss relatif tinggi, masing-masing 0.0976 dan 0.0973, serta grafik loss-nya menunjukkan fluktuasi tajam yang tidak stabil (tiba-tiba ada kenaikan menanjak di tengah *training*), menunjukkan adanya ketidakstabilan dalam proses pelatihan atau eksplosinya nilai aktivasi.

#### 4.7. Perbandingan dengan library sklearn

Berdasarkan pengujian yang dilakukan dengan membandingkan model FFNN sendiri dengan library MLP dari Scikit-learn, terlihat bahwa fungsi aktivasi memiliki pengaruh signifikan terhadap akurasi model. Dengan fungsi aktivasi sigmoid, FFNN implementasi hanya mencapai akurasi 61.67%, jauh tertinggal dari MLPClassifier yang mencapai 90.93%. Hal ini kemungkinan besar disebabkan oleh efek *vanishing gradient* yang lebih parah di implementasi FFNN manual, terutama jika tidak menggunakan optimisasi atau regularisasi lanjutan.

Dengan menggunakan ReLU, FFNN implementasi meningkat signifikan ke 73.61%, namun tetap lebih rendah dari MLPClassifier (93.17%). Ini menunjukkan bahwa ReLU memang membantu mengatasi *vanishing gradient*, namun implementasi dari Sklearn jauh lebih stabil dan mungkin menggunakan teknik tambahan seperti *early stopping* dan *adaptive learning rate* (Adam). Untuk Tanh, kedua model tampil sangat baik dan hampir setara: FFNN 91.42%, dan Sklearn 91.39%. Ini menunjukkan bahwa fungsi tanh dalam kasus ini sangat cocok dengan arsitektur dan data, serta FFNN buatan berhasil mengoptimalkan model secara efisien.

## BAB V

### KESIMPULAN DAN SARAN

#### **5.1. Kesimpulan**

Selama proses mengerjakan tugas besar ini, kami semakin memahami proses untuk membangun model prediksi yang tepat dalam analisis data. Berdasarkan analisis yang telah dilakukan, kami menemukan bahwa pemilihan arsitektur model, seperti jumlah layer (depth) dan jumlah neuron per layer (width), sangat memengaruhi performa model. Penambahan depth dan width memang dapat meningkatkan kemampuan model dalam menangkap pola kompleks, tetapi dapat menimbulkan overfitting apabila terlalu berlebihan. Selain itu, fungsi aktivasi juga berperan penting; fungsi seperti Swish, ELU, dan tanh menunjukkan performa terbaik karena mampu menjaga kestabilan gradien dan mendukung proses pembelajaran yang efektif. Di sisi lain, fungsi seperti sigmoid dan softplus cenderung mengalami penurunan performa akibat *vanishing gradient*.

Pemilihan learning rate yang tepat menjadi faktor krusial dalam proses training, di mana nilai yang terlalu besar menyebabkan pelatihan gagal, sementara nilai terlalu kecil membuat proses pelatihan berjalan sangat lambat. Penggunaan Xavier dan He menghasilkan model yang lebih stabil dan cepat konvergen dibandingkan metode zero, uniform, atau normal konvensional. Penerapan regularisasi L1 dan L2 terbukti mampu meningkatkan kemampuan generalisasi model, meskipun menambah penalti pada fungsi loss. Namun, tidak semua teknik modern memberikan hasil yang diharapkan—misalnya penggunaan RMSNorm yang justru menurunkan performa karena kemungkinan menyebabkan ketidakstabilan selama pelatihan.

Meskipun implementasi manual FFNN memiliki keterbatasan dibandingkan SKLearn, hasil yang kami capai sudah cukup kompetitif, khususnya pada konfigurasi dan fungsi aktivasi tertentu seperti tanh.

#### **5.2. Saran**

Selama penggerjaan tugas besar ini kami memiliki beberapa saran untuk penggerjaan tugas besar berikutnya, di antaranya:

- Mencari referensi lebih banyak lagi sebagai bahan literasi penambah wawasan.

- Membuat sheets berisi link-link terintegrasi yang dibutuhkan selama penggerjaan tugas serta jadwal yang jelas terkait waktu pertemuan maupun *soft deadline* untuk setiap task.

## PEMBAGIAN TUGAS

NIM	Nama	Kontribusi
13522012	Thea Josephine Halim	Kode, Laporan
13522046	Raffael Boymian Siahaan	Kode, Laporan

## LAMPIRAN

### Repository

[https://github.com/pandaandsushi/IF3270\\_Tubes1\\_FFNN](https://github.com/pandaandsushi/IF3270_Tubes1_FFNN)

## DAFTAR PUSTAKA

3Blue1Brown. (2017, 5 Oktober). But what is a neural network? | Deep learning chapter 1 [Video]. <https://youtu.be/aircArUvnKk>

Asisten, S. A. (2024). Tugas Besar 1 IF3270 Pembelajaran Mesin Feedforward Neural Network. docx. Diakses dari Google Dokumen: <https://docs.google.com/document>

Baheti, P. (2021). Activation Functions in Neural Networks [12 Types & Use Cases]. Diakses pada tanggal 7 Maret 2025, dari <https://www.v7labs.com>

GeeksforGeeks. (2024). Feedforward neural network. Diakses pada tanggal 7 Maret 2025, dari <https://www.geeksforgeeks.org/feedforward-neural-network/>

GeeksforGeeks. (2025). Activation functions in Neural Networks. Diakses pada tanggal 7 Maret 2025, dari <https://www.geeksforgeeks.org/activation-functions-neural-networks/>

Khelli, M. (2022). Introduction to Artificial Neural Network in Deep Learning. Diakses pada tanggal 7 Maret 2025, dari <https://khelli07.medium.com>

Khelli, M. (2022). Feedforward Neural Network (FNN) Implementation from Scratch Using Python. Diakses pada tanggal 7 Maret 2025, dari <https://khelli07.medium.com>

Neuralthreads. (2021). SELU and ELU — Exponential Linear Units. Diakses pada tanggal 8 Maret 2025, dari <https://neuralthreads.medium.com>

Rosebrock, A. (2021). Understanding weight initialization for neural networks. Diakses pada tanggal 8 Maret 2025, dari <https://pyimagesearch.com>

Tewari, U. (2021). Regularization — Understanding L1 and L2 regularization for Deep Learning. Diakses pada tanggal 27 Maret 2025, dari <https://medium.com/analytics-vidhya>