Tugas Besar 1 IF3270 Pembelajaran Mesin
Feedforward Neural Network

Oleh:

13522125 - Satriadhikara Panji Yudhistira

13522128 - Mohammad Andhika Fadillah

13522145 - Farrel Natha Saskoro

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

Daftar Isi

Bab 1

Deskripsi Persoalan

Pengimplementasian Feedforward Neural Network (FFNN) from Scratch untuk menyelesaikan permasalahan berdasarkan dataset yang telah diberikan.

- FFNN yang diimplementasikan dapat **menerima jumlah neuron dari tiap layer** (termasuk input layer dan output layer)
- FFNN yang diimplementasikan dapat **menerima fungsi aktivasi dari tiap layer**.
- FFNN yang diimplementasikan dapat **menerima fungsi loss**
- Terdapat mekanisme untuk **inisialisasi bobot** tiap neuron (termasuk bias).
- Instance model yang diinisialisasikan harus bisa **menyimpan bobot** tiap neuron (termasuk bias)
- Instance model yang diinisialisasikan harus bisa **menyimpan gradien bobot** tiap neuron (termasuk bias)
- Instance model memiliki method untuk **menampilkan model** berupa **struktur jaringan** beserta **bobot** dan **gradien bobot** tiap neuron dalam bentuk graf.
- Instance model memiliki method untuk **menampilkan distribusi bobot** dari tiap layer.
- Instance model memiliki method untuk **menampilkan distribusi gradien bobot**
- Instance model memiliki method untuk ***save*** dan ***load***
- Model memiliki implementasi **forward propagation**
- Model memiliki implementasi **backward propagation**
- Model memiliki implementasi **weight update** dengan menggunakan **gradient descent**

Bab 2

Pembahasan

2.1 Penjelasan Implementasi

2.1.1 Deskripsi kelas beserta deskripsi atribut dan methodnya

Activations.py			
Kelas	deskripsi	atribut	method
Activation	Kelas abstrak yang menjadi dasar bagi semua fungsi aktivasi dalam jaringan saraf tiruan	Tidak memiliki atribut instance (hanya menyediakan metode yang harus diimplementasikan oleh subclass).	**1. forward(self, x) → (Abstract Method)** - Menerapkan fungsi aktivasi pada input x. - Harus diimplementasikan oleh subclass. **2. backward(self, x) → (Abstract Method)** - Menghitung turunan fungsi aktivasi terhadap input x. - Harus diimplementasikan oleh subclass.
Linear	Turunan dari kelas Activation. Fungsi aktivasi linear.	-	**1. forward(self, x)** - Mengembalikan nilai input x tanpa perubahan. - f(x)=x **2. backward(self, x)** - Mengembalikan array dengan nilai 1 untuk semua elemen x (karena turunan dari fungsi linear adalah 1). - f'(x) = 1
ReLU	Turunan dari kelas Activation. Fungsi aktivasi ReLU.	-	**1. forward(self, x)** - Mengembalikan x jika positif, 0 jika negatif. - f(x)=max(0,x)

			2. backward(self, x) - Mengembalikan 1 untuk elemen positif dan 0 untuk elemen negatif. - f′(x)=1 jika x>0, dan f′(x)=0f jika x≤0.
Sigmoid	Turunan dari kelas Activation. Fungsi aktivasi Sigmoid.	-	**1. forward(self, x)** - Mengembalikan nilai sigmoid dari x. $\sigma(x) = \frac{1}{1+e^{-x}}$ **2. backward(self, x)** - Mengembalikan turunan fungsi sigmoid $\frac{d(\sigma(x))}{dx} = \sigma(x)(1-\sigma(x))$ - Turunan sigmoid lebih kecil dari 1 sehingga dapat menyebabkan vanishing gradient problem.
Tanh	Turunan dari kelas Activation. Fungsi aktivasi Tanh.	-	**1. forward(self, x)** - Mengembalikan nilai tanh dari x. - f(x)=tanh(x). **2. backward(self, x)** - Mengembalikan turunan fungsi tanh, yaitu 1 - tanh²(x). - f′(x)=1 - tanh²(x).
Softmax	Turunan dari kelas Activation. Digunakan dalam klasifikasi multiclass untuk menghasilkan probabilitas dari beberapa kelas.	-	**1. forward(self, x)** - Mengembalikan distribusi probabilitas dari input x dengan ekspresi Softmax. Untuk vector $\vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$, $softmax(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$

			2. backward(self, x) Untuk vector $\vec{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$, $\frac{d(softmax(\vec{x})_i)}{d\vec{x}} = \begin{bmatrix} \frac{\partial(softmax(\vec{x})_1)}{\partial x_1} & \dots & \frac{\partial(softmax(\vec{x})_1)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial(softmax(\vec{x})_n)}{\partial x_1} & \dots & \frac{\partial(softmax(\vec{x})_n)}{\partial x_n} \end{bmatrix}$ Dimana untuk $i, j \in \{1, \dots, n\}$, $\frac{\partial(softmax(\vec{x})_i)}{\partial x_j} = softmax(\vec{x})_i(\delta_{i,j} - softmax(\vec{x})_j)$ $\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$
Swish	Turunan dari kelas Activation. Fungsi aktivasi Swish.	-	**1. forward(self, x)** - Mengembalikan $x \cdot \text{sigmoid}(x)$ **2. backward(self, x)** Mengembalikan turunan Swish yaitu: $\sigma(x)+x \cdot \sigma(x) \cdot (1-\sigma(x))$
GeLU	Turunan dari kelas Activation. Fungsi aktivasi GeLU.	-	**1. forward(self, x)** - Menggunakan pendekatan tanh untuk perhitungan: **2. backward(self, x)** Turunan GELU menggunakan kombinasi fungsi CDF (Cumulative Density Function) dan PDF (Probability Density Function) dari distribusi normal.
layers.py			
Kelas	deskripsi	atribut	method
Layer	Kelas dasar untuk semua jenis layer dalam neural network.	**parameters:** Dictionary untuk menyimpan parameter layer (seperti bobot dan bias). **gradients:** Dictionary untuk menyimpan gradien dari parameter selama proses backpropagation.	**1. forward(x):** Method abstrak yang diimplementasikan oleh subclass untuk melakukan perhitungan forward propagation. **2. backward(grad_output):** Method abstrak yang diimplementasikan oleh subclass untuk melakukan perhitungan backward propagation.

		cache: Dictionary untuk menyimpan data yang diperlukan dalam forward pass, agar bisa digunakan kembali dalam backward pass.	**3. update_parameters(learning_rate):** Memperbarui parameter layer berdasarkan gradien yang dihitung selama backward pass menggunakan metode gradient descent. **4. get_parameters():** Mengembalikan parameter layer. **5. get_gradients():** Mengembalikan gradien dari parameter layer.
DenseLayer	Kelas yang merepresentasikan lapisan fully connected (dense layer), yang menghubungkan semua neuron dari satu lapisan ke lapisan berikutnya.	**input_dim:** Jumlah neuron pada lapisan input. **output_dim:** Jumlah neuron pada lapisan output. **activation:** Fungsi aktivasi yang digunakan dalam layer ini. **parameters:** Menyimpan bobot (weights) dan bias (biases). **gradients:** Menyimpan gradien dari bobot dan bias yang dihitung selama backpropagation.	**1. __init__(input_dim, output_dim, activation, weight_initializer=None):** - Menginisialisasi bobot menggunakan kaidah He initialization jika weight_initializer tidak diberikan. - Inisialisasi bias dengan nol. **2. forward(x):** - Melakukan perhitungan forward propagation **3. backward(grad_output):** - Melakukan perhitungan backpropagation, menghitung gradien untuk bobot dan bias: - Menyimpan gradien dalam gradients dan mengembalikan gradien untuk lapisan sebelumnya (grad_input).
loss_functions.py			
Kelas	deskripsi	atribut	method
Loss	Kelas abstrak yang mendefinisikan kerangka	-	**1. forward(y_true, y_pred):** - Method abstrak yang

	dasar untuk semua fungsi loss dalam neural network.		diimplementasikan oleh subclass. - Menghitung nilai loss berdasarkan target sebenarnya (y_true) dan prediksi model (y_pred). **2. backward(y_true, y_pred):** - Method abstrak yang diimplementasikan oleh subclass. - Menghitung gradien dari loss terhadap output model untuk digunakan dalam backpropagation.
MeanSquaredError	Kelas yang mengimplementasikan fungsi loss Mean Squared Error (MSE).	-	**1. forward(y_true, y_pred):** - Menghitung nilai Mean Squared Error menggunakan rumus **2. backward(y_true, y_pred):** - Menghitung gradien dari loss terhadap output model menggunakan rumus:
BinaryCrossEntropy	Kelas yang mengimplementasikan fungsi loss Binary Cross-Entropy (BCE)	-	**1. forward(y_true, y_pred):** - Menghitung nilai BinaryCrossEntropy menggunakan rumus **2. backward(y_true, y_pred):** - Menghitung gradien dari loss terhadap output model menggunakan rumus
CategoricalCrossEntropy	Kelas yang mengimplementasikan fungsi loss Categorical Cross-Entropy (CCE)	-	**1. forward(y_true, y_pred):** - Menghitung nilai Categorical Cross-Entropy menggunakan rumus **2. backward(y_true, y_pred):** - Menghitung gradien dari loss terhadap output model menggunakan rumus
neural_network.py			
NeuralNetwork	Kelas NeuralNetwork merupakan implementasi dari neural network yang terdiri dari beberapa lapisan	**layer_sizes** Tipe: list Menyimpan ukuran setiap lapisan dalam	**1. __init__(self, layer_sizes, activation_names, loss_function, weight_initializer)** - Konstruktor yang

	(DenseLayer). Jaringan ini dapat melakukan forward propagation, backward propagation, dan pembaruan parameter selama proses pelatihan.	jaringan saraf. **activation_names** Tipe: list Daftar nama fungsi aktivasi yang digunakan di setiap lapisan jaringan. **loss_function** Tipe: objek fungsi Fungsi loss yang digunakan untuk menghitung error selama pelatihan. **weight_initializer** Tipe: str atau objek fungsi Metode inisialisasi bobot yang digunakan untuk setiap lapisan jaringan. **layers** Tipe: list Menyimpan daftar lapisan DenseLayer yang telah diinisialisasi dalam jaringan.	menginisialisasi jaringan saraf berdasarkan jumlah lapisan, fungsi aktivasi, fungsi loss, dan metode inisialisasi bobot. **2. _initialize_layers(self)** - Membantu inisialisasi setiap lapisan dalam jaringan saraf berdasarkan jumlah neuron dan fungsi aktivasi. **3. forward(self, x)** - Melakukan forward propagation terhadap input x, melewati seluruh lapisan jaringan. **4. backward(self, grad_output)** - Melakukan backward propagation untuk menghitung gradien dan memperbarui bobot jaringan. **5. update_parameters(self, learning_rate)** - Memperbarui parameter bobot dan bias dari setiap lapisan berdasarkan nilai gradien dan learning_rate. **6. display_model(self)** - Menampilkan arsitektur model, termasuk ukuran neuron di setiap lapisan serta bobot dan gradiennya. **7. visualize_network(self)** - Membuat visualisasi struktur jaringan menggunakan networkx dan matplotlib dan juga menampilkan koneksi antar lapisan serta nilai rata-rata bobot dan gradiennya. **8. plot_weight_distribution(self, layers_to_plot=None):** Untuk menampilkan histogram distribusi bobot (weights) dari layer tertentu dalam jaringan saraf.

			9. plot_gradient_distribution(self, layers_to_plot=None): Untuk menampilkan histogram distribusi gradien (weights gradients) dari layer tertentu dalam jaringan saraf. **10. train(self, X_train, y_train, X_val=None, y_val=None, batch_size=32, learning_rate=0.01, epochs=100, l1_lambda=0, l2_lambda=0, momentum=0, use_rmsnorm=False, verbose=1, early_stopping_patience=None)** - Melatih model dengan menggunakan data latih X_train dan y_train, serta X_val dan y_val. - Fitur tambahan: ○ Regularisasi L1 & L2 untuk mencegah overfitting. ○ Momentum & RMSNorm untuk mempercepat konvergensi. ○ Early Stopping untuk menghentikan pelatihan jika performa tidak meningkat. **11. save(self, filename)** - Menyimpan model ke dalam file menggunakan pickle. **12. load(filename) *(static method)*** - Memuat kembali model dari file yang telah disimpan sebelumnya menggunakan pickle.
normalization.py			

Kelas	deskripsi	atribut	method
RMSNorm	Menormalisasi input dengan metode RMS (Root Mean Square).	**epsilon (float)** - Nilai yang digunakan untuk menghindari pembagian dengan nol saat menghitung RMS. - nilai epsilon : 1e-8. **cache (dict)** - Digunakan untuk menyimpan nilai intermediate dalam perhitungan normalisasi jika diperlukan untuk optimasi atau debugging. 1	**1. __init__(self, epsilon=1e-8)** Konstruktor yang menginisialisasi objek RMSNorm dengan nilai epsilon. **2. normalize(self, x)** Melakukan normalisasi RMS terhadap input x. Parameter: x (numpy.ndarray): Matriks input dengan bentuk (batch_size, feature_dim). **3. backward(self, x, grad_output)** Menghitung backpropagation dari operasi normalisasi RMS.
weight_initializers.py			
WeightInitializer	Kelas dasar untuk semua metode inisialisasi bobot.	-	**initialize(self, shape)** diimplementasikan oleh subclass untuk menginisialisasi bobot berdasarkan metode tertentu.
ZeroInitializer	Inisialisasi bobot dengan nilai nol.	-	**initialize(self, shape)** Mengembalikan array NumPy berisi nol dengan ukuran shape. Return: Array NumPy dengan semua elemen bernilai nol.
UniformInitializer	Inisialisasi bobot menggunakan distribusi uniform acak dalam rentang tertentu.	**lower_bound (float):** Batas bawah distribusi uniform (default: -0.1). **upper_bound (float):** Batas atas distribusi uniform (default: 0.1). **seed (int atau None)**	**__init__(self, lower_bound=-0.1, upper_bound=0.1, seed=None)** Menginisialisasi batas distribusi dan seed jika diberikan. **initialize(self, shape)** Mengembalikan array NumPy dengan nilai acak dari distribusi uniform. Return: Array NumPy dengan nilai acak

			dari distribusi uniform dalam rentang [lower_bound, upper_bound].
NormalInitializer	Inisialisasi bobot menggunakan distribusi normal dengan mean dan varians tertentu.	**mean (float):** Mean dari distribusi normal (default: 0). **variance (float):** Varians dari distribusi normal (default: 0.01). **seed (int atau None):** Seed untuk memastikan replikasi hasil acak (opsional).	**__init__(self, mean=0, variance=0.01, seed=None)** Mengatur parameter distribusi normal dan seed jika diberikan. **initialize(self, shape)** Mengembalikan array NumPy dengan nilai acak dari distribusi normal. Return: Array NumPy dengan nilai acak dari distribusi normal dengan mean dan variance.
XavierInitializer	Inisialisasi bobot menggunakan metode Xavier/Glorot Initialization, yang dirancang untuk mempertahankan varians sinyal tetap konstan di seluruh jaringan.	-	**initialize(self, shape)** Mengembalikan array NumPy dengan nilai acak dari distribusi uniform dalam batas Return: Array NumPy dengan nilai acak dari distribusi uniform.
HeInitializer	Inisialisasi bobot menggunakan metode He Initialization, yang dirancang untuk jaringan dengan aktivasi ReLU.	-	**initialize(self, shape)** Mengembalikan array NumPy dengan nilai acak dari distribusi normal dengan standar deviasi Return: Array NumPy dengan nilai acak dari distribusi normal.

2.1.2 Penjelasan forward propagation

Forward propagation adalah proses dimana kita membawa data pada input melewati tiap neuron pada hidden layer sampai kepada output layer yang nanti akan dihitung errornya

Proses ini terdiri dari beberapa langkah:

1. **Input ke Neuron**
   Data input diteruskan ke setiap neuron pada hidden layer.
   Setiap neuron menerima bobot awal dan bias sebagai parameter awal model.
   Implementasi pada kode:

```
def forward(self, x):
    """Melakukan forward pass."""
    output = x
    for layer in self.layers:
        output = layer.forward(output)
    return output
```

2. **Kalkulasi Aktivasi**
   Setiap neuron menghitung z, yaitu kombinasi linear dari input dan bobot:
   Hasil ini kemudian diterapkan ke fungsi aktivasi seperti ReLU, Sigmoid, atau Softmax untuk memperkenalkan non-linearitas.

   Implementasi pada kode:

```
def forward(self, x):
        self.cache["input"] = x

(neuron menghitung z)
        z = np.dot(x, self.parameters["weights"]) + self.parameters["biases"]
        self.cache["z"] = z

(z diterapkan pada fungsi aktivasi)
        output = self.activation.forward(z)

        return output
```

3. **Penerusan ke Lapisan Berikutnya**
   Output dari neuron di satu lapisan menjadi input bagi lapisan berikutnya hingga mencapai lapisan output.
   Setelah sampai di lapisan output, hasilnya digunakan sebagai prediksi model.

4. **Perhitungan Loss (Error)**
   Output jaringan dibandingkan dengan nilai target (label sebenarnya).
   Fungsi loss menghitung error, yang akan digunakan dalam backpropagation.

Tujuan forward propagation adalah mendapatkan prediksi awal, yang nantinya dibandingkan dengan nilai sebenarnya untuk menghitung error dan memperbarui bobot pada proses berikutnya.

2.1.3 Penjelasan backward propagation dan weight update

Backward Propagation adalah proses untuk memperbarui bobot (weights) dan bias dalam artificial neural network dengan cara menggunakan error yang kita dapat pada forward propagation. Proses ini bertujuan untuk meminimalkan selisih antara prediksi jaringan dengan nilai sebenarnya, sehingga model dapat belajar dan meningkatkan akurasinya seiring waktu.

Langkah-langkah Backpropagation:

1. **Hitung Loss Gradient**

   Dari forward propagation, kita telah mendapatkan error antara output model dan label sebenarnya. Gunakan turunan parsial dari fungsi loss untuk menghitung bagaimana perubahan bobot mempengaruhi loss.

   Implementasi pada kode:

```
class MeanSquaredError(Loss):
    def backward(self, y_true, y_pred):
        n_samples = y_true.shape[0]
        return 2 * (y_pred - y_true) / n_samples

class BinaryCrossEntropy(Loss):
    def backward(self, y_true, y_pred):
        epsilon = 1e-15
        y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
        return -(y_true / y_pred - (1 - y_true) / (1 - y_pred)) / len(y_true)

class CategoricalCrossEntropy(Loss):
    def backward(self, y_true, y_pred):
        epsilon = 1e-15
        y_pred = np.clip(y_pred, epsilon, 1.0 - epsilon)
        return (y_pred - y_true) / y_true.shape[0]
```

2. **Turunan Fungsi Aktivasi**

   Gradien dihitung dengan menggunakan turunan dari fungsi aktivasi. Misalnya, untuk ReLU, turunannya adalah 1 untuk nilai positif dan 0 untuk negatif.

   Implementasi pada kode:

```
class Linear(Activation):
    def backward(self, x):
        return np.ones_like(x)


class ReLU(Activation):
    def backward(self, x):
        return (x > 0).astype(float)
```

```
class Sigmoid(Activation):
    def backward(self, x):
        sigmoid_x = self.forward(x)
        return sigmoid_x * (1 - sigmoid_x)


class Tanh(Activation):
    def backward(self, x):
        return 1 - np.tanh(x) ** 2


class Softmax(Activation):
    def backward(self, x):
        return np.ones_like(x)


class Swish(Activation):
    def backward(self, x):
        sigmoid_x = 1 / (1 + np.exp(-x))
        return sigmoid_x + x * sigmoid_x * (1 - sigmoid_x)


class GELU(Activation):
    def backward(self, x):
        cdf = 0.5 * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * np.power(x,
3))))
        pdf = np.exp(-(x**2) / 2) / np.sqrt(2 * np.pi)
        return cdf + x * pdf
```

3. **Distribusi Gradien ke Bobot Tiap Lapisan (Chain Rule)**
   Dengan menggunakan aturan rantai (chain rule) dalam kalkulus, kita menyebarkan error ke setiap bobot dalam neural network mulai dari output ke hidden layers hingga input.

   Implementasi pada kode:

```
def backward(self, grad_output):
    """Melakukan backward pass."""
    for layer in reversed(self.layers):
        grad_output = layer.backward(grad_output)
```

4. **Penyimpanan Gradien untuk Update Bobot**
   Setelah menghitung gradien untuk setiap bobot, nilai ini digunakan dalam weight update menggunakan algoritma optimasi.

Implementasi pada kode:

```
def backward(self, grad_output):
    x = self.cache["input"]
    z = self.cache["z"]
    batch_size = x.shape[0]

    dz = grad_output * self.activation.backward(z)

    self.gradients["weights"] = np.dot(x.T, dz) / batch_size
    self.gradients["biases"] = np.sum(dz, axis=0, keepdims=True) / batch_size

    grad_input = np.dot(dz, self.parameters["weights"].T)

    return grad_input
```

```
def update_parameters(self, learning_rate):
    """Memperbarui parameter model."""
    for layer in self.layers:
        layer.parameters["weights"] -= learning_rate * layer.gradients["weights"]
        layer.parameters["biases"] -= learning_rate * layer.gradients["biases"]
```

Weight Update adalah proses untuk memperbarui bobot (*weights*) dalam artificial neural network agar model dapat belajar dari data dan meningkatkan akurasinya. Pembaruan bobot dilakukan setelah menghitung gradien dari fungsi kerugian selama backpropagation, menggunakan algoritma optimasi seperti Gradient Descent.

2.2 Hasil Pengujian

2.2.1 Pengaruh depth dan width

Depth (kedalaman) mengacu pada jumlah hidden layer dalam jaringan neural, sedangkan width (lebar) mengacu pada jumlah neuron dalam setiap hidden layer.

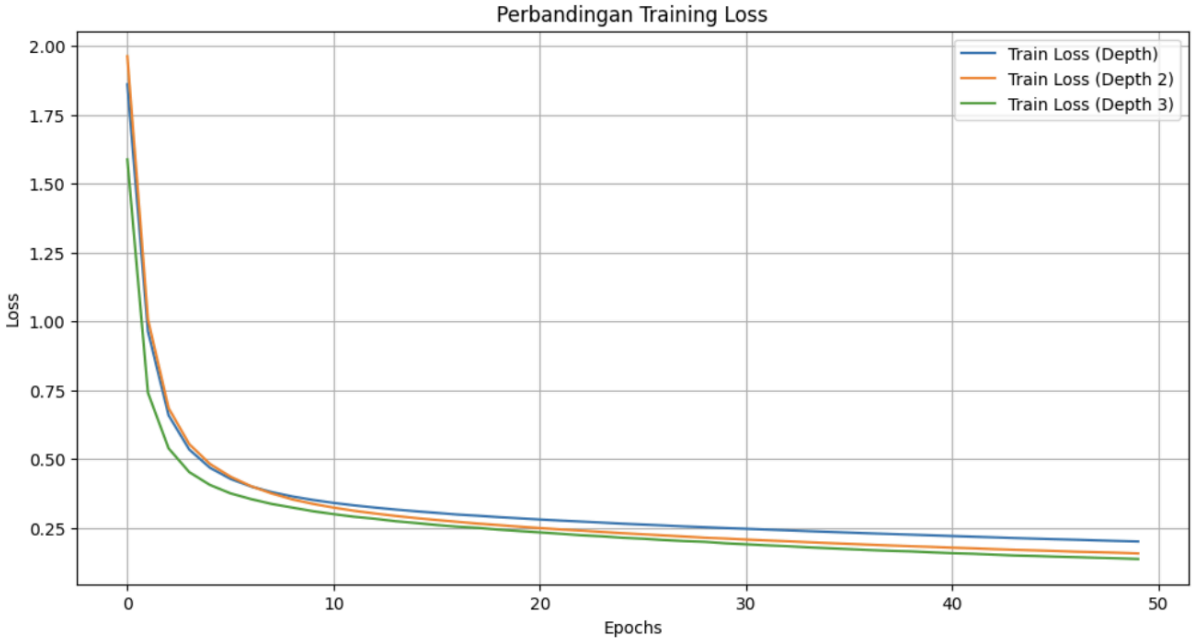
- Meningkatkan depth (kedalaman) dapat meningkatkan kapasitas model untuk menangkap pola kompleks dalam data, tetapi juga meningkatkan risiko overfitting dan waktu komputasi.
- Menambah width (lebar) dalam satu lapisan dapat membantu model belajar representasi fitur yang lebih baik, tetapi juga berpotensi menyebabkan overfitting jika tidak diimbangi dengan regularisasi yang tepat.

Variasi depth :

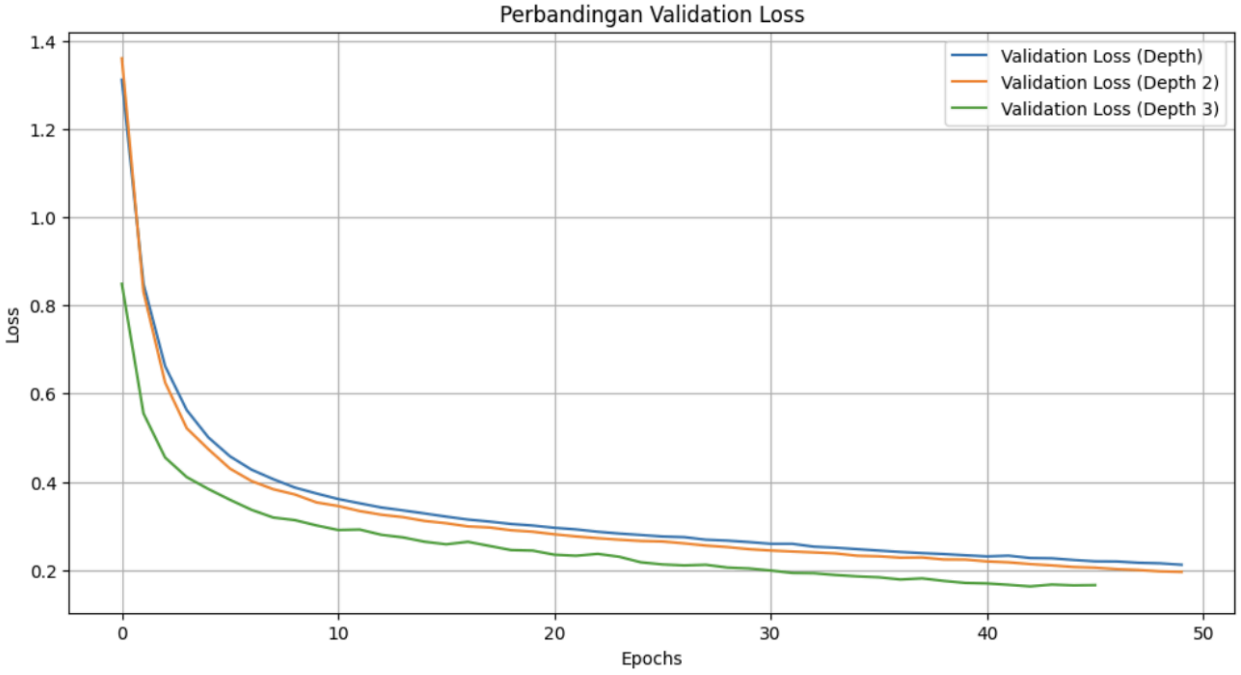
Depth	Test Accuracy
`layer_sizes=[784, 128, 64, 32, 10],`	`Test Accuracy (Depth): 0.9353`

`layer_sizes=[784, 256, 128, 64, 32, 10],`	`Test Accuracy (Depth 2): 0.9484`
`layer_sizes=[784, 512, 256, 128, 64, 32, 10],`	`Test Accuracy (Depth 3): 0.9511`

Grafik Training Loss (Variasi Depth):



Grafik Validation Loss (Variasi Depth):



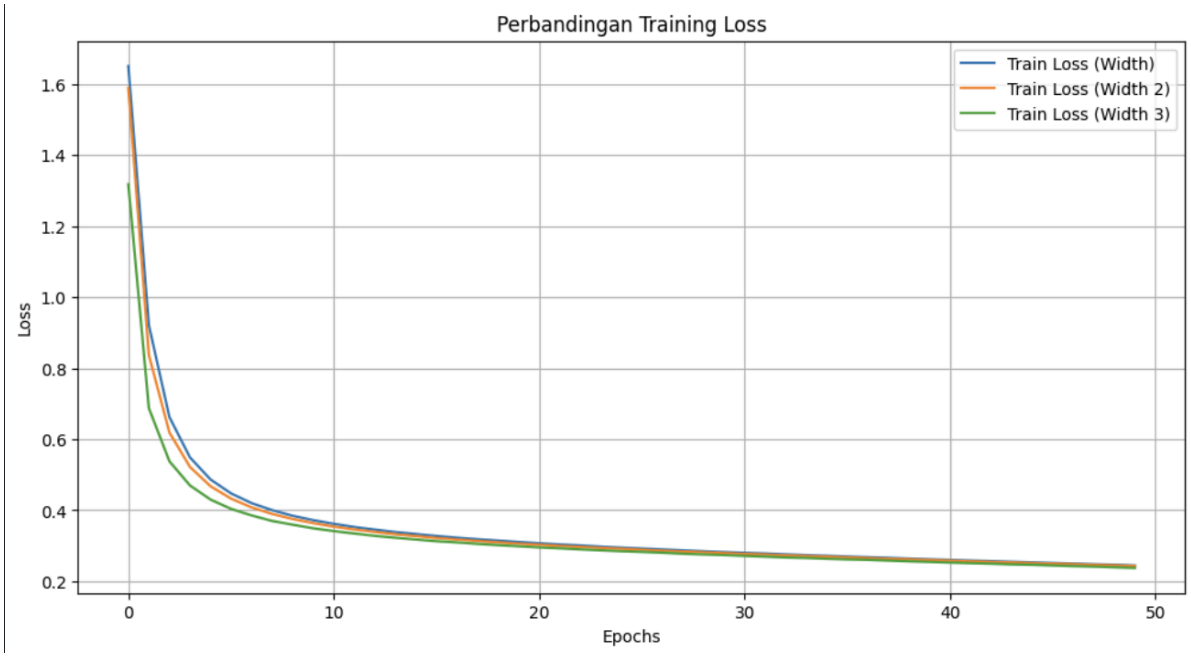
Perbandingan antara dua grafik training loss dan validation loss tiap epoch menunjukan bahwa ketiganya menunjukkan pola penurunan yang sejalan antara training dan validation loss, yang mengindikasikan tidak terjadi overfitting secara signifikan.

Pada grafik training loss, model dengan depth 3 memiliki nilai training loss yang paling rendah dibanding depth lainnya yang menunjukkan performa pelatihan yang lebih baik.
Pada grafik validation loss, model Depth 3 juga menunjukkan nilai validation loss yang paling rendah yang menunjukkan bahwa model ini tidak hanya fit ke data training tapi juga mampu menggeneralisasi ke data validasi.

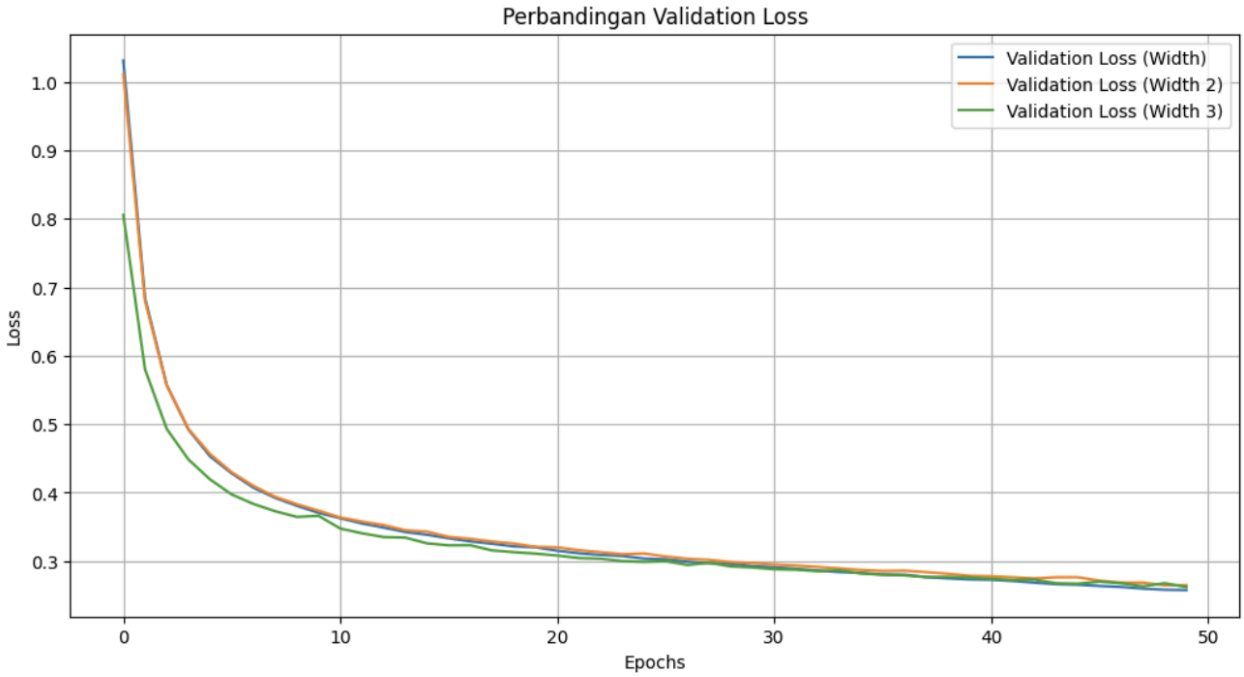
Variasi width :

Width	Test Accuracy
`layer_sizes=[784, 256, 256, 10],`	`Test Accuracy (Width): 0.9261`
`layer_sizes=[784, 512, 512, 10],`	`Test Accuracy (Width 2): 0.9269`
`layer_sizes=[784, 1024, 1024, 10],`	`Test Accuracy (Width 3): 0.9271`

Grafik Training Loss (Variasi Width):



Grafik Validation Loss (Variasi Width):



Perbandingan antara dua grafik training loss dan validation loss tiap epoch menunjukan bahwa ketiganya menunjukkan pola penurunan yang sejalan antara training dan validation loss, yang mengindikasikan tidak terjadi overfitting.

Pada grafik training loss, model dengan width 3 memiliki nilai training loss yang paling rendah dibanding width lainnya yang menunjukkan performa pelatihan yang lebih baik.

Pada grafik validation loss, model width 3 juga menunjukkan bahwa penambahan width juga membantu, tapi tidak se-signifikan penambahan depth sesuai perbandingan dengan grafik sebelumnya.

2.2.2 Pengaruh fungsi aktivasi

Fungsi aktivasi menentukan bagaimana suatu neuron mengubah inputnya menjadi output. Pemilihan fungsi aktivasi yang tepat berpengaruh terhadap konvergensi, stabilitas pelatihan, serta kemampuan model dalam menangkap pola dalam data.

Nama Fungsi Aktivasi	Definisi Fungsi	Penjelasan
Linear	$Linear(x) = x$	Linear(x) merupakan nilai keluaran dari fungsi Linear saat diberikan masukan x.

ReLU	$ReLU(x) = \max(0, x)$	ReLU(x) adalah nilai keluaran dari fungsi ReLU saat diberikan masukan x. Jika x adalah positif, maka f(x) akan mengambil nilai x, dan jika x adalah negatif, maka f(x) akan mengambil nilai 0.
Sigmoid	$\sigma(x) = \frac{1}{1 + e^{-x}}$	σ(x) adalah nilai keluaran dari fungsi Sigmoid saat diberikan masukan x, dan e adalah konstanta Euler
Hyperbolic Tangent (tanh)	$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	tanh(x) adalah nilai keluaran dari fungsi Tanh saat diberikan masukan x, dan e adalah konstanta Euler
Softmax	Untuk vector $\vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$, $softmax(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$	softmax(x) adalahnilai keluaran dari model softmax dengan parameter masukan x dan e
Swish	$f(x) = x \cdot \sigma(\beta \cdot x)$	$f(x)$ adalah nilai keluaran dari fungsi Swish saat diberikan masukan x , dan σ adalah fungsi Sigmoid, serta β adalah parameter yang mempengaruhi kecuraman fungsi.
GELU	$f(x) = 0.5x\left(1 + \text{Tanh}\left(\sqrt{\frac{2}{\pi}} \cdot (x + 0.044715x^3)\right)\right)$	$f(x)$ adalah nilai keluaran dari fungsi GELU saat diberikan masukan.

Analisis :

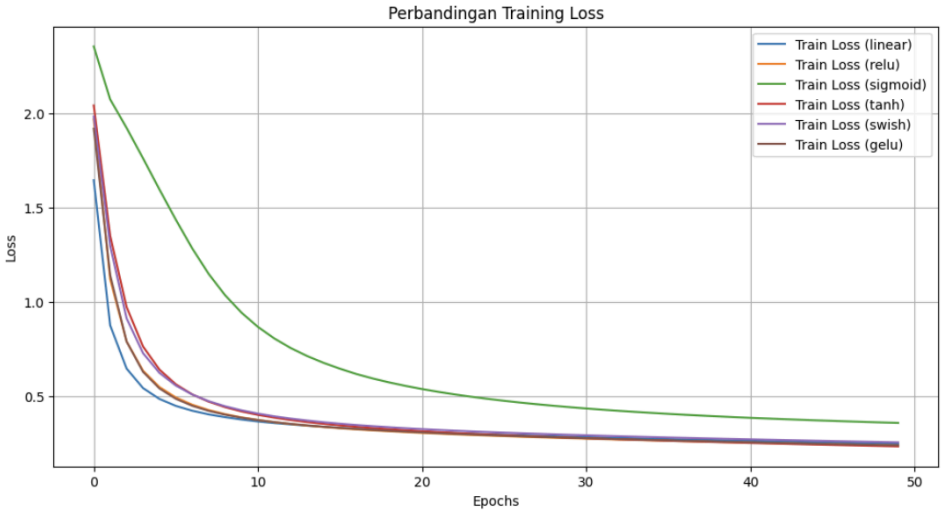
Activation Function	Test Accuracy
Linear	`Test Accuracy (Activation linear): 0.9245`

ReLU	`Test Accuracy (Activation relu): 0.9250`
Sigmoid	`Test Accuracy (Activation sigmoid): 0.8981`
TanH	`Test Accuracy (Activation tanh): 0.9294`
Swish	`Test Accuracy (Activation swish): 0.9230`
GeLU	`Test Accuracy (Activation gelu): 0.9274`

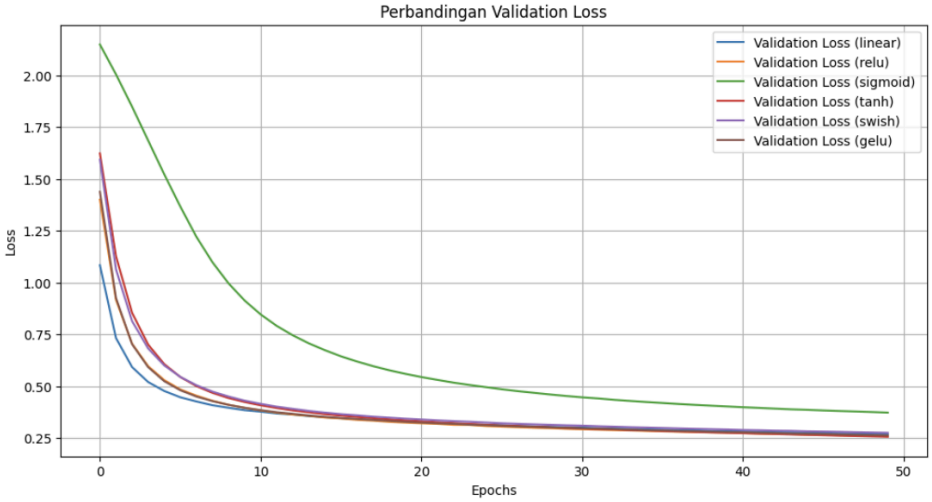
Hasil Pengujian didapatkan bahwa :

TanH merupakan aktivasi yang lebih baik dibandingkan fungsi aktivasi lainnya sehingga bisa membantu model dalam mencapai konvergensi yang lebih cepat. Akurasi yang lebih tinggi menunjukkan bahwa model ini dapat belajar pola lebih kompleks dengan generalisasi yang lebih baik.

Grafik Training Loss :

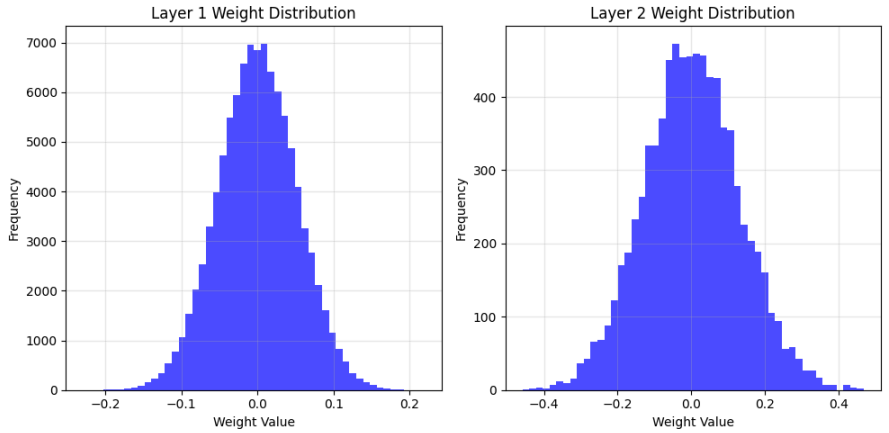
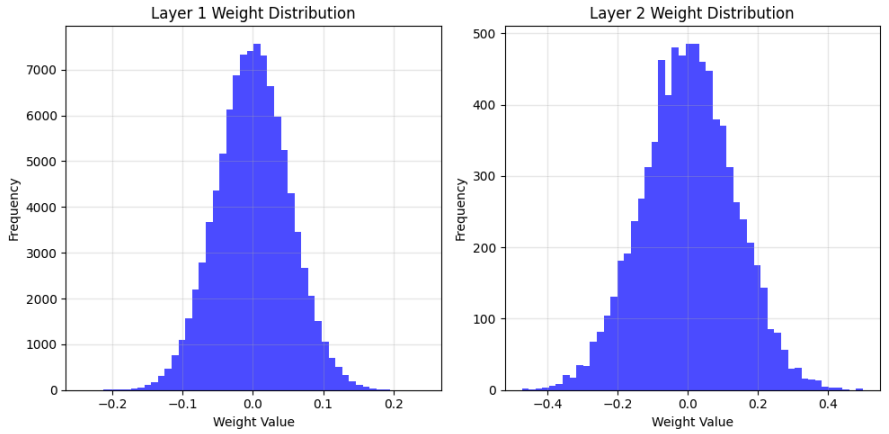
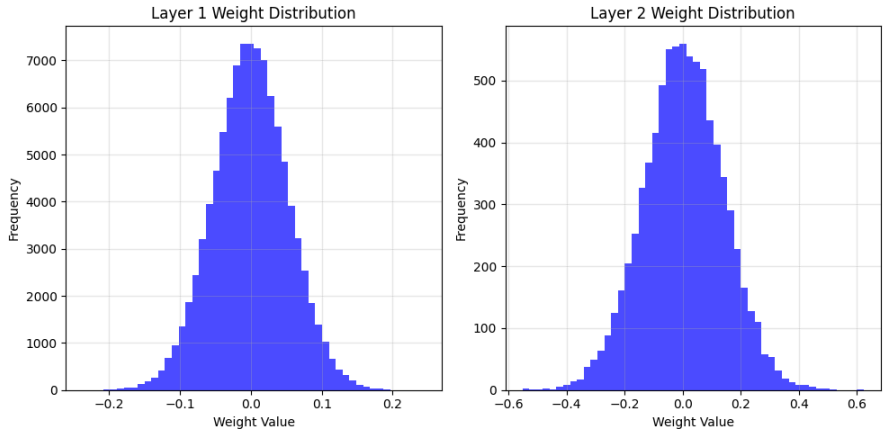


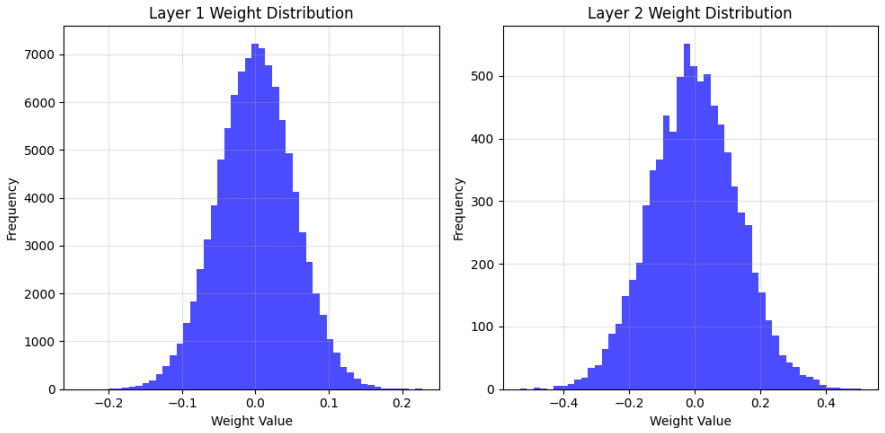
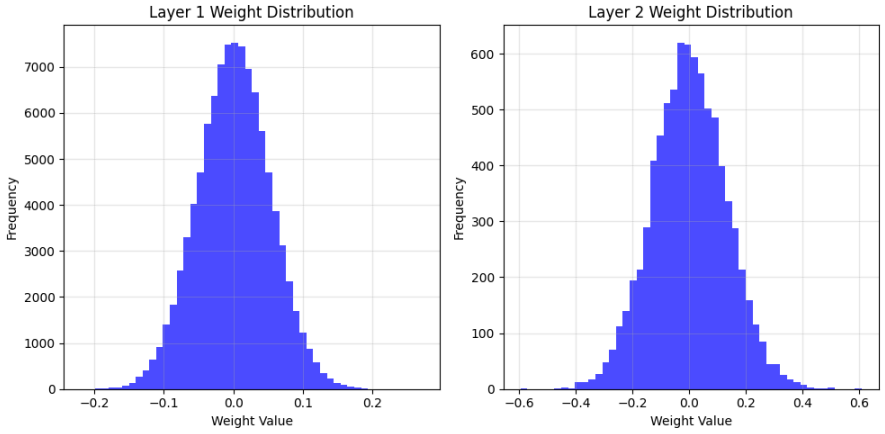
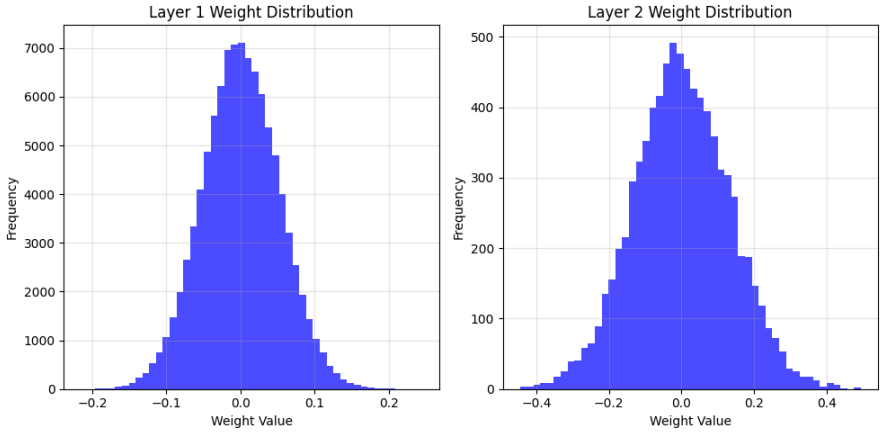
Grafik Validation Loss :



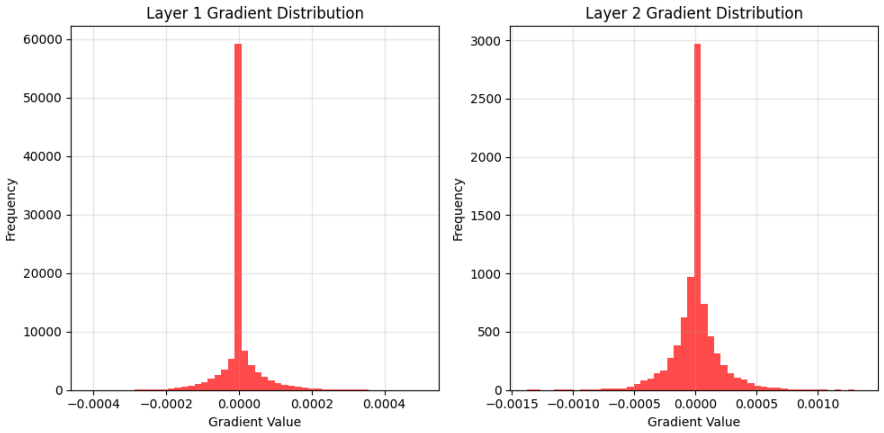
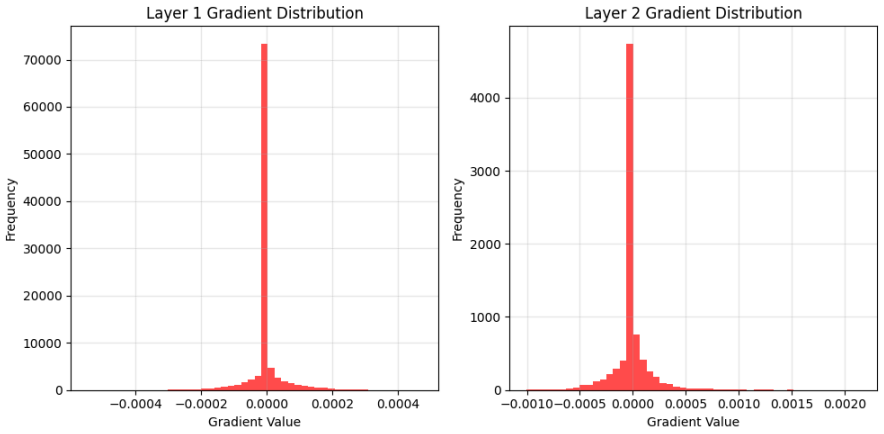
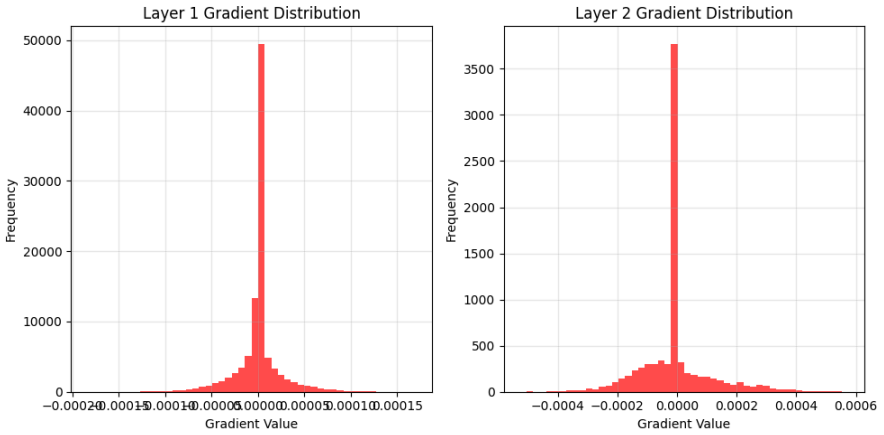
Berdasarkan grafik training loss dan validation loss yang ditampilkan, terlihat perbedaan performa model ketika menggunakan berbagai fungsi aktivasi. Fungsi aktivasi **TanH** dan**ReLU** menunjukkan performa terbaik, ditandai dengan penurunan loss yang paling cepat dan stabil sepanjang proses pelatihan. Hal ini konsisten dengan hasil akurasi uji yang tinggi, yang menunjukkan bahwa kedua fungsi ini mampu menangkap pola kompleks dalam data dengan lebih baik.

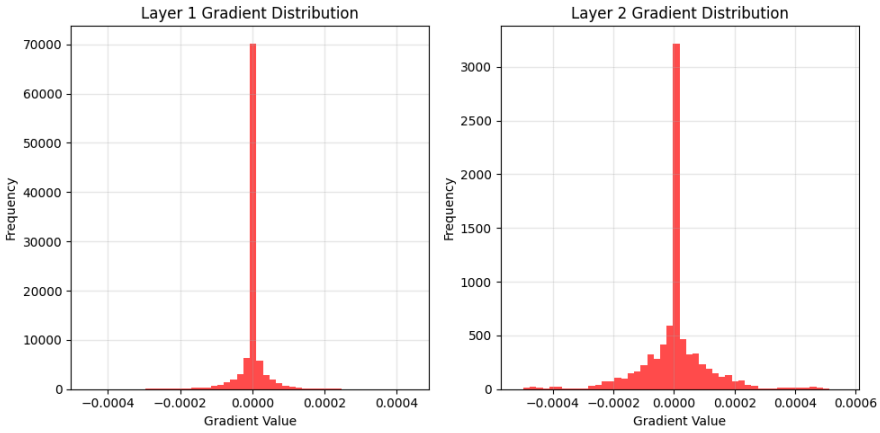
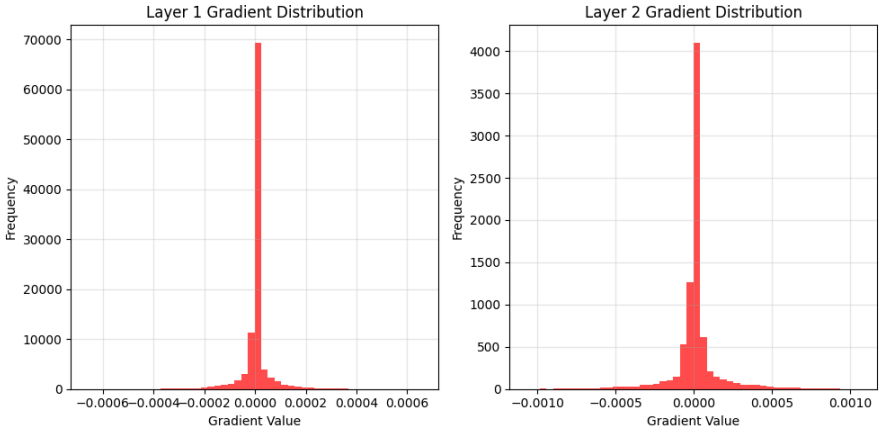
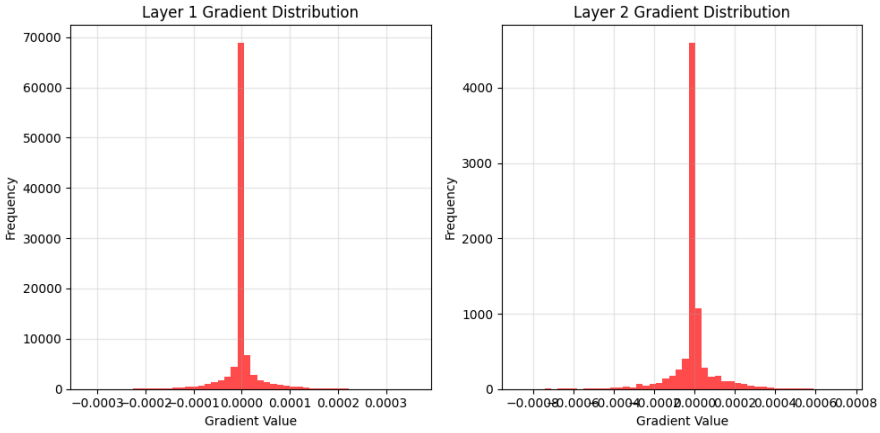
Distribusi Bobot

Linear	
ReLU	
Sigmoid	

TanH	
Swish	
GeLU	

Distribusi Gradien Bobot

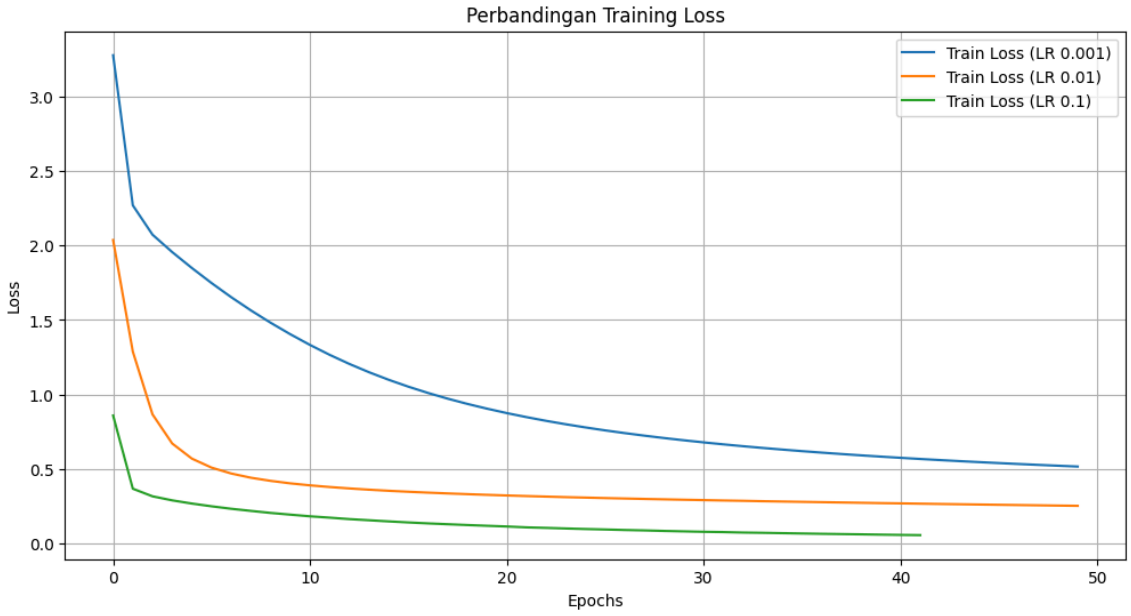
Linear	
ReLU	
Sigmoid	

TanH	
Swish	
GeLU	

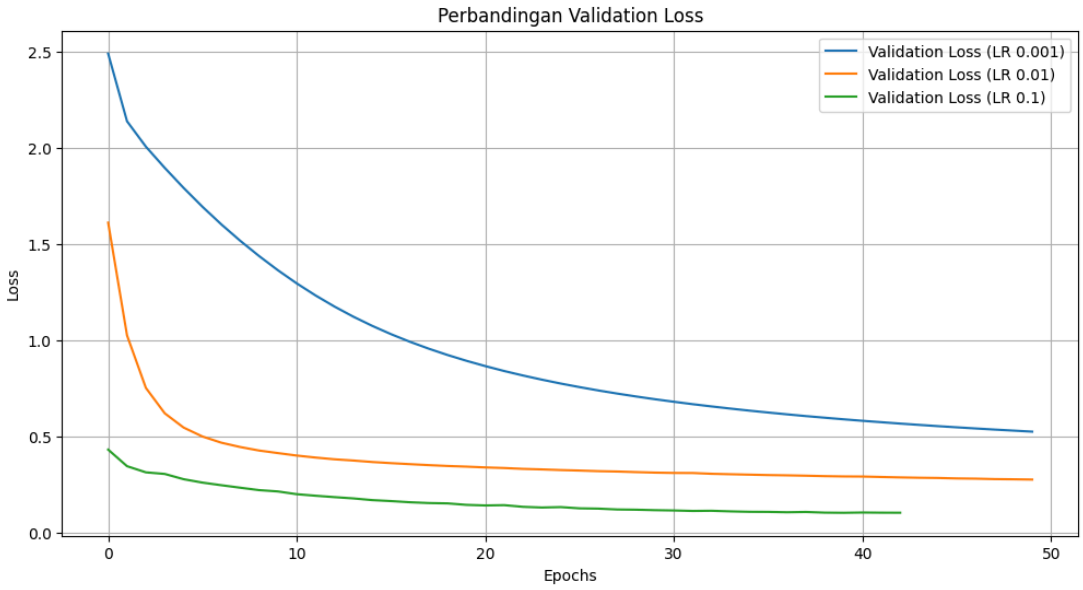
2.2.3 Pengaruh learning rate

Learning Rate	Test Accuracy
0,1	`Test Accuracy (Learning Rate 0.1): 0.9692`
0,01	`Test Accuracy (Learning Rate 0.01): 0.9227`
0,001	`Test Accuracy (Learning Rate 0.001): 0.8654`

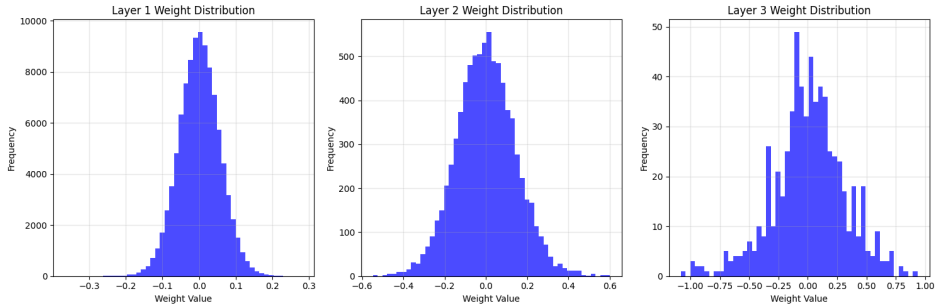
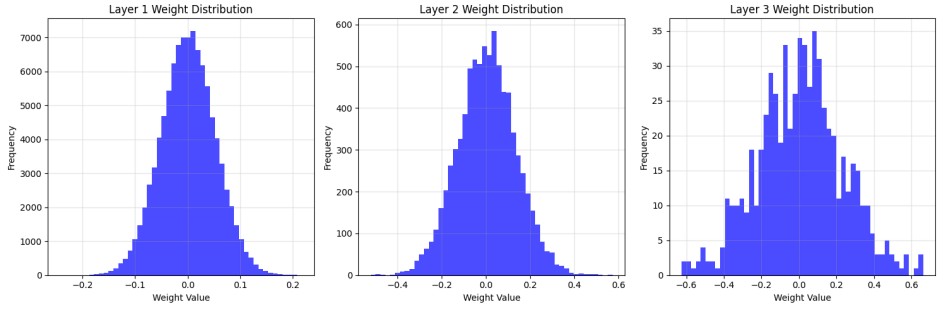
Grafik Training Loss :

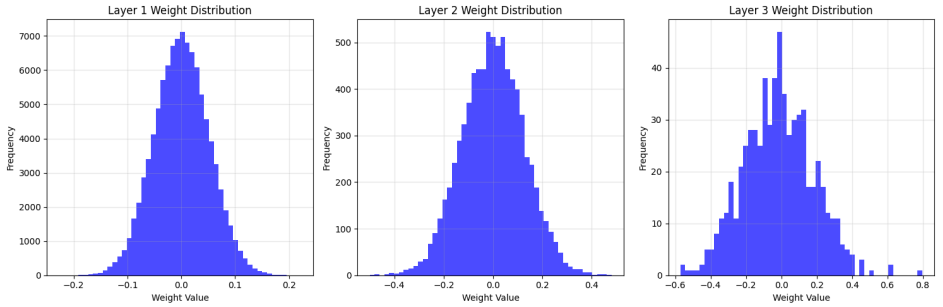


Grafik Validation Loss :

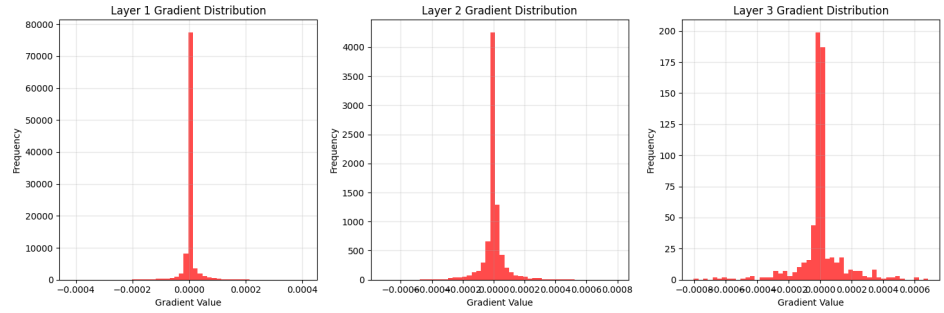
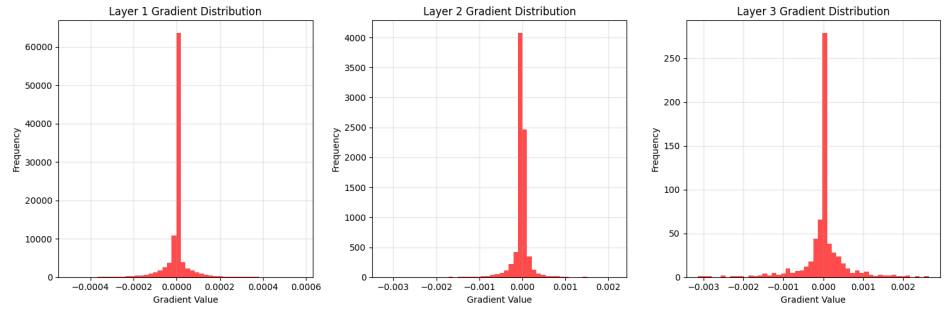
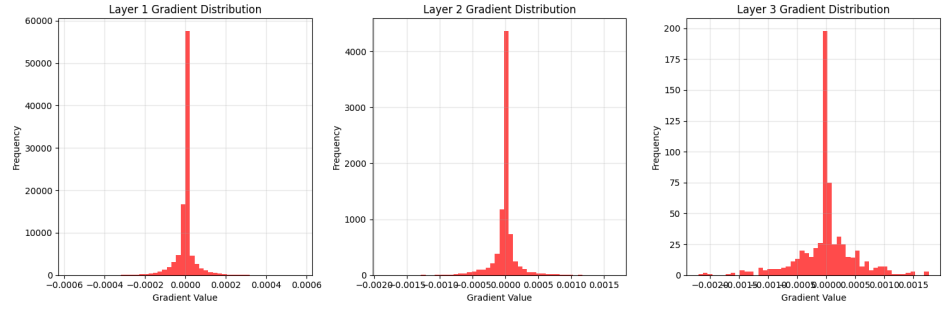


Distribusi Bobot:

Learning Rate	Hasil
0,1	
0,01	

0,001	

Distribusi Gradien Bobot:

Learning Rate	**Hasil**
0,1	
0,01	
0,001	

- Learning rate terlalu besar dapat menyebabkan model melewati solusi optimal dan tidak konvergen.
- Learning rate terlalu kecil dapat membuat proses training menjadi lambat dan berpotensi terjebak dalam local minima.

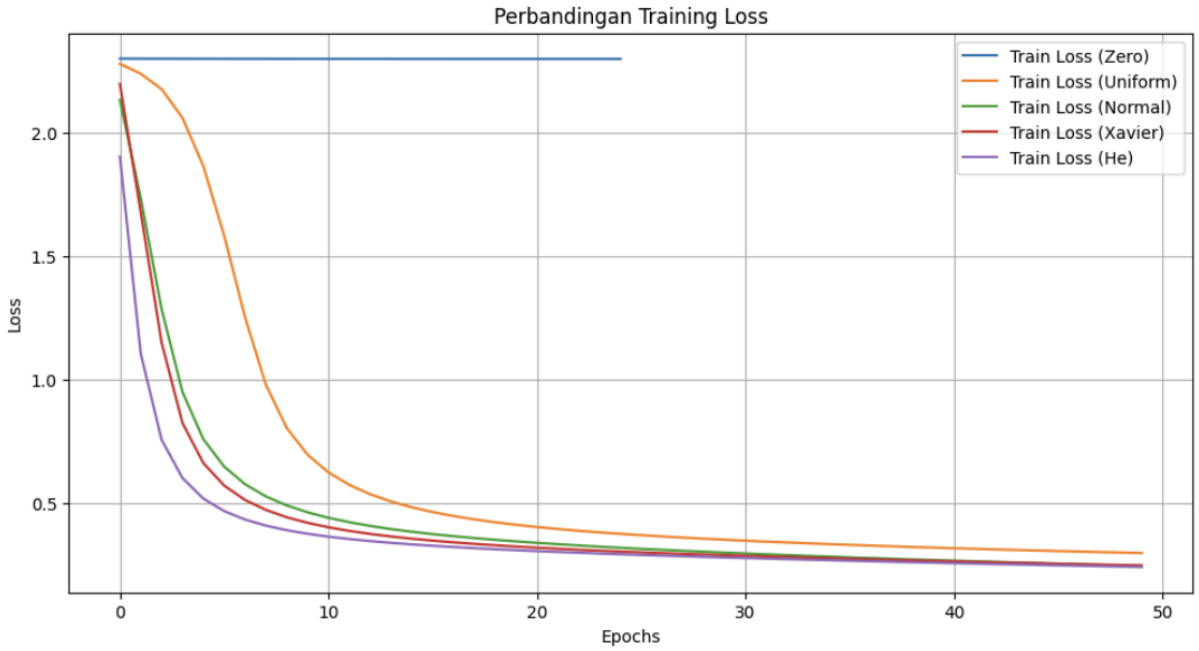
Percobaan dilakukan dengan menggunakan berbagai nilai learning rate (misalnya 0.1, 0.01, 0.001) dan **hasil yang paling optimal pada saat menggunakan learning rate 0,1.**

2.2.4 Pengaruh inisialisasi bobot

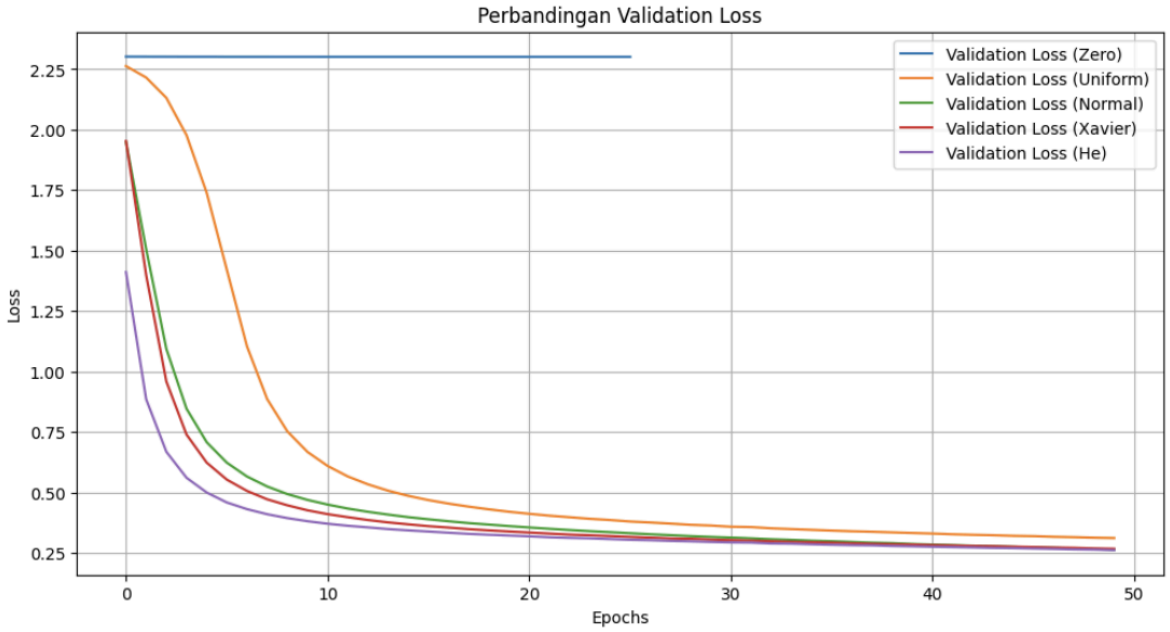
Inisialisasi bobot (weight initialization) merupakan tahap awal dalam pelatihan neural network yang berpengaruh besar terhadap konvergensi, stabilitas, dan kinerja model. Hasil pengujian didapat yaitu :

Inisialisasi Bobot	Test Accuracy
Zero	`Test Accuracy (Zero Initializer): 0.1143`
Uniform	`Test Accuracy (Uniform Initializer): 0.9109`
Normal	`Test Accuracy (Normal Initializer): 0.9274`
Xavier	`Test Accuracy (Xavier Initializer): 0.9261`
He	`Test Accuracy (He Initializer): 0.9275`

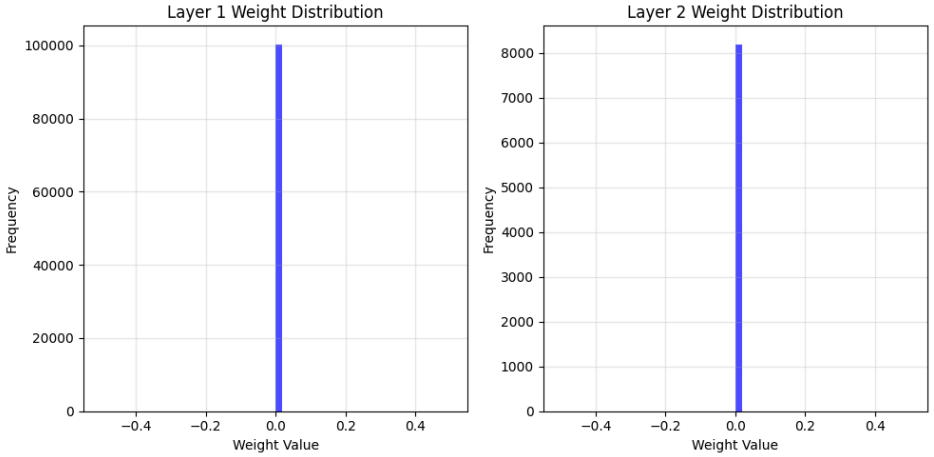
Grafik Training Loss :

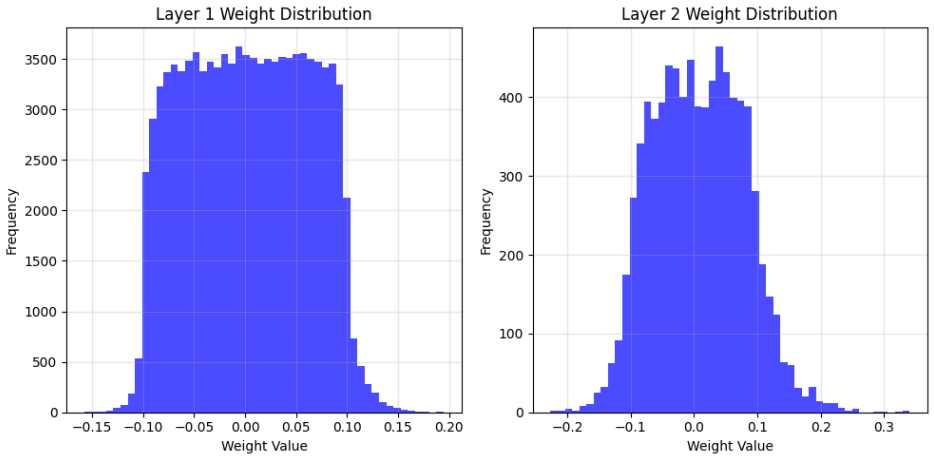
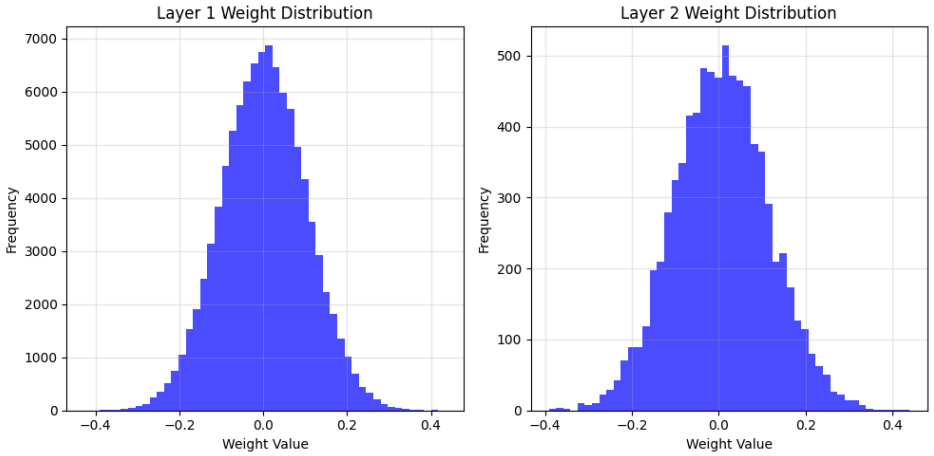
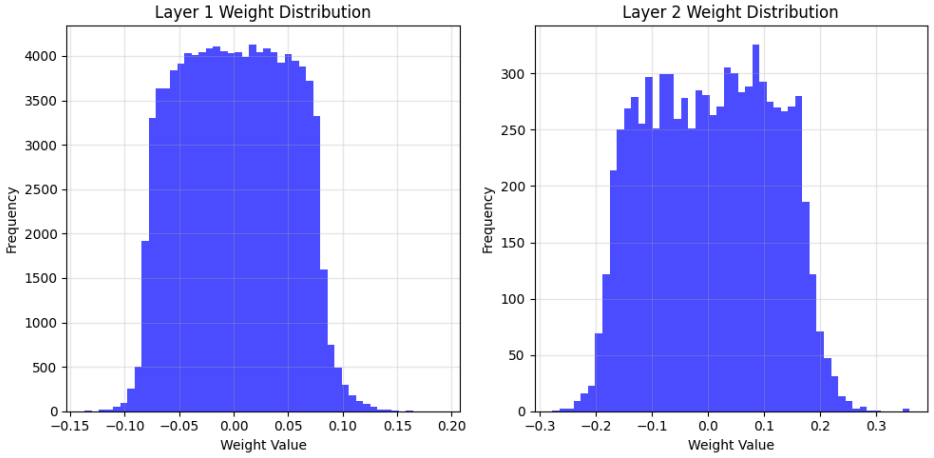


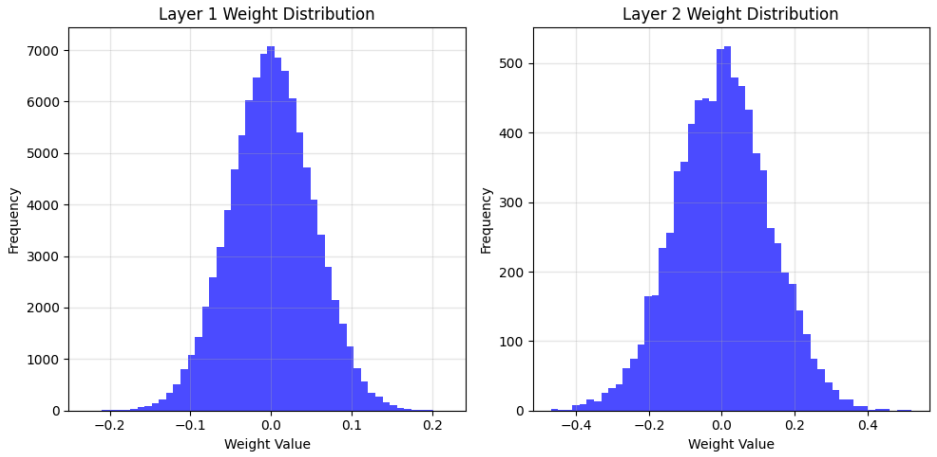
Grafik Validation Loss :



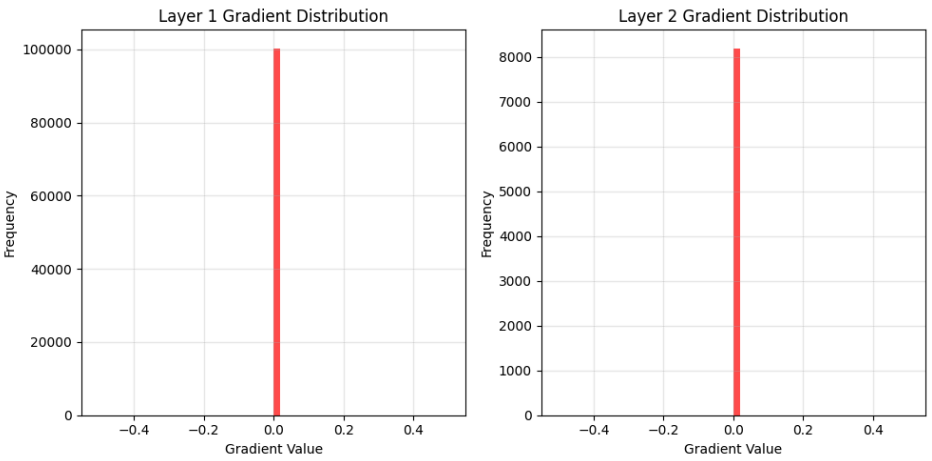
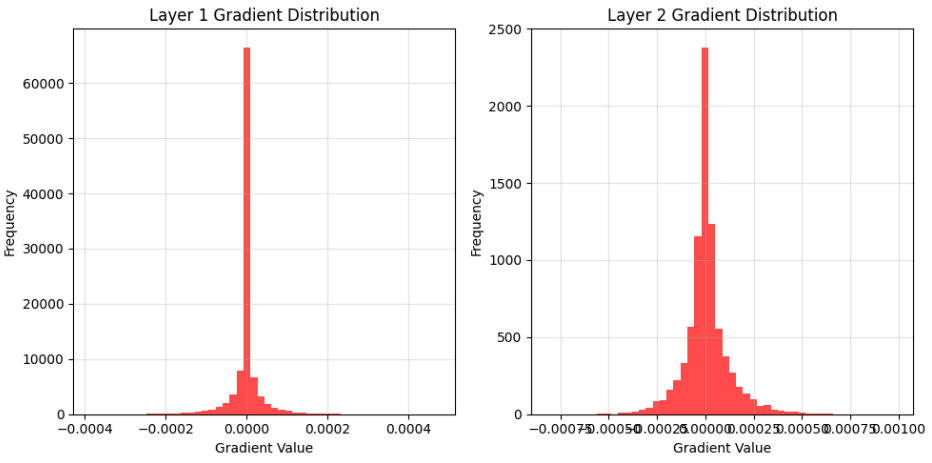
Distribusi Bobot:

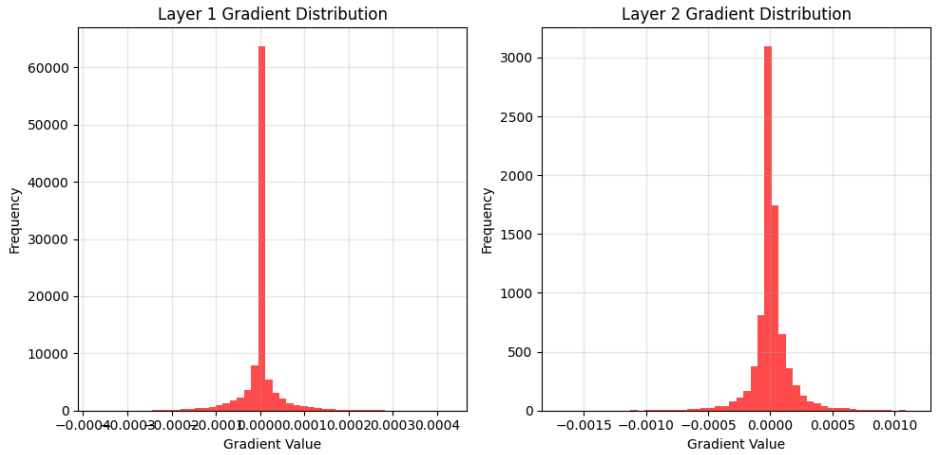
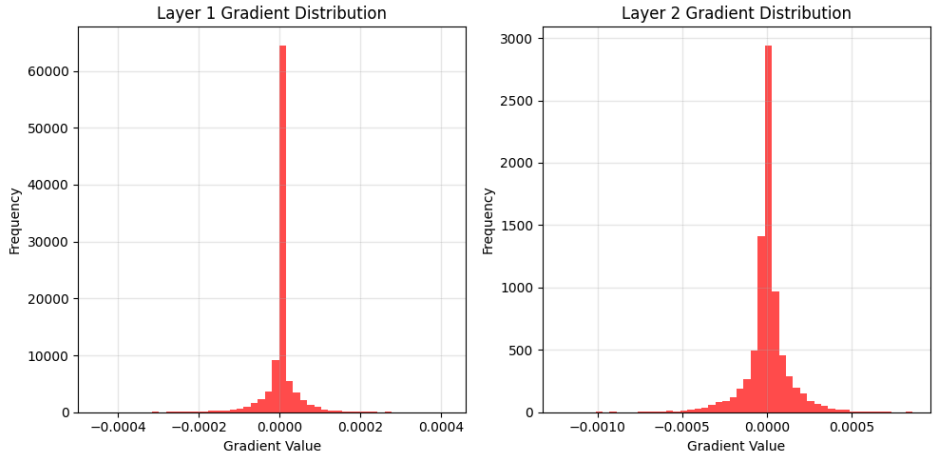
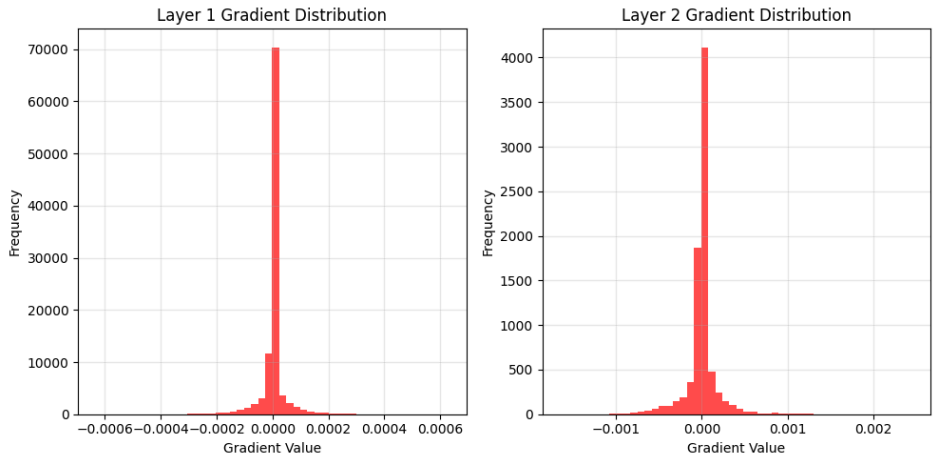
Inisialisasi Bobot	**Hasil**
Zero	

Uniform	
Normal	
Xavier	

He	

Distribusi Gradien Bobot:

Inisialisasi Bobot	**Hasil**
Zero	
Uniform	

Normal	
Xavier	
He	

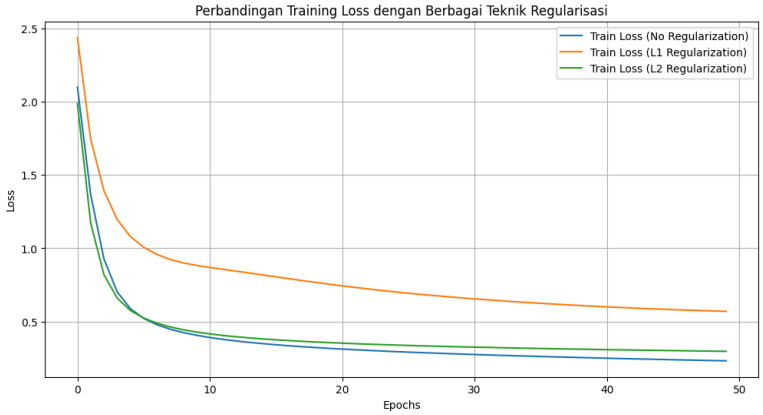
Zero Initialization gagal karena menghasilkan output yang identik, sehingga model tidak dapat belajar. **Uniform** dan **Normal Initialization** menunjukkan penurunan loss, tetapi lebih lambat dan kurang stabil. **Xavier** dan **He Initialization** memberikan hasil terbaik, dengan **He** sedikit lebih unggul sesuai dengan teori yang diajarkan.

2.2.5 Pengaruh regularisasi

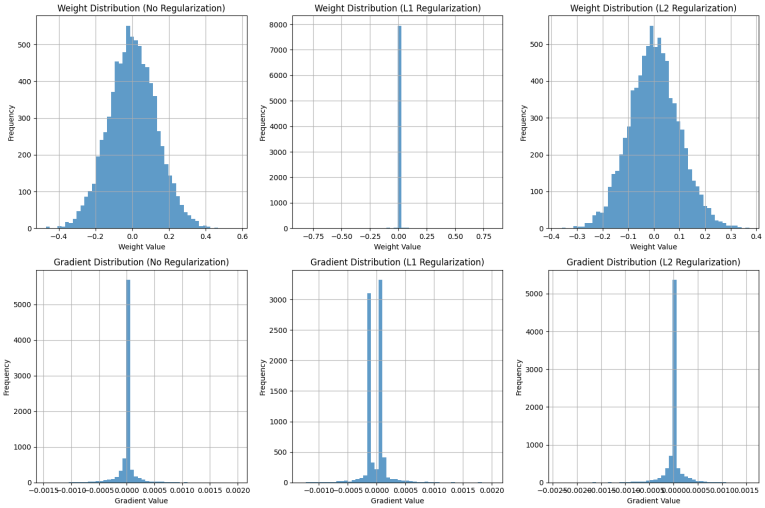
Regularisasi L1 dan L2 merupakan teknik yang digunakan untuk mencegah overfitting di FFNN dengan menambahkan sebuah penalty term kepada loss function. Pada eksperimen ini kita membandingkan model tanpa regularisasi dengan yang menggunakan regularisasi L1 maupun L2, hasil akurasi yang didapat yaitu:

```
--- Accuracy Summary ---
No Regularization: 0.9298
L1 Regularization: 0.8598
L2 Regularization: 0.9189
```

dari hasil di atas, data menunjukan kalau model tanpa regularisasi memberikan akurasi terbaik, diikuti oleh model dengan regularisasi L2. Regularisasi L1 tampaknya kurang efektif dalam kasus ini, ditunjukan juga pada grafik *training loss* nya



model dengan regularisasi L1, memiliki *loss* yang lebih tinggi dibandingkan dengan 2 model lainnya, ini menandakan bahwa model tersebut memiliki kesalahan prediksi yang lebih besar.



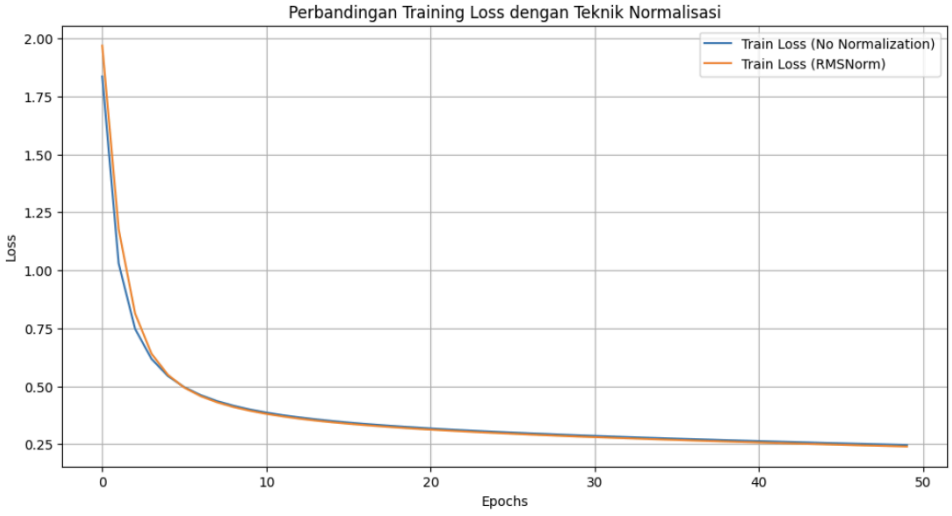
Kalau melihat dari grafik perbandingan bobot dan gradien, regularisasi L1 menghasilkan bobot yang jarang dan gradien yang lebih kecil, sedangkan L2 mengurangi besaran bobot dan gradien, tetapi tidak membuatnya menjadi nol.

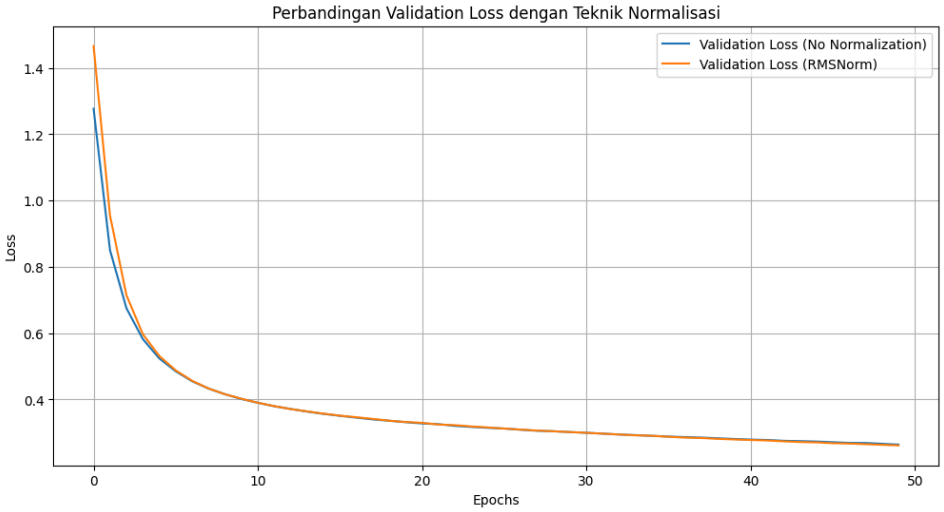
2.2.6 Pengaruh normalisasi RMSNorm

RMSNorm adalah metode normalisasi yang menstabilkan aktivasi neuron dengan membaginya dengan akar kuadrat rata-rata dari aktivasi tersebut. Pada eksperimen ini kita membandingkan training model yang tidak dinormalisasikan dengan RMSNorm dan yang dinormalisasikan dengan RMSNorm.

tidak dinormalisasikan dengan RMSNorm	`Test Accuracy (No Normalization): 0.9264`
dinormalisasikan dengan RMSNorm	`Test Accuracy (RMSNorm): 0.9279`

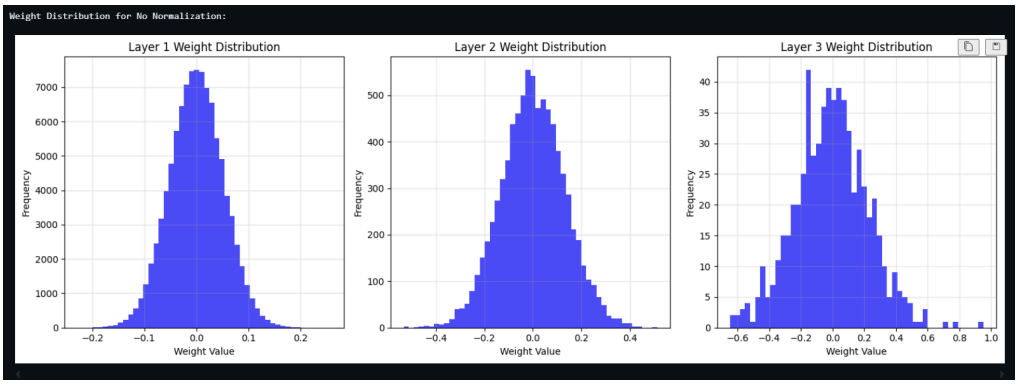
Penerapan RMSNorm memberikan sedikit peningkatan akurasi sebesar 0.0015 atau 0.15% dibandingkan model tanpa normalisasi. Meski perbedaannya kecil, hasil ini menunjukkan bahwa RMSNorm membantu stabilisasi pelatihan, meskipun dampaknya tidak terlalu besar dalam kasus ini.

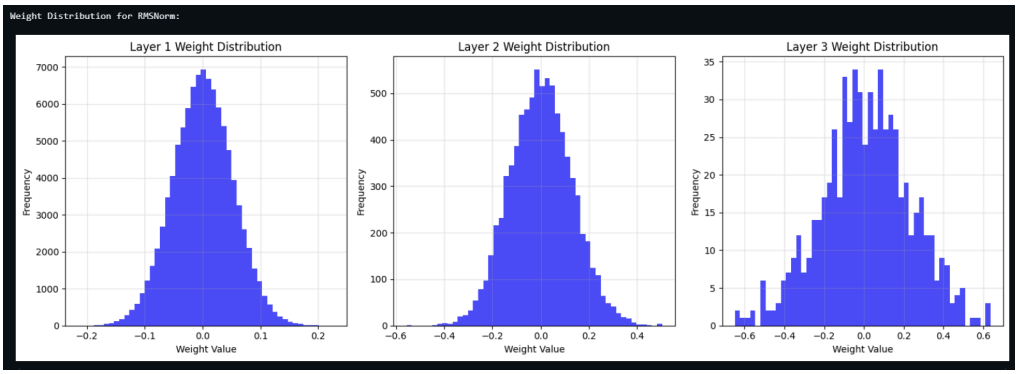


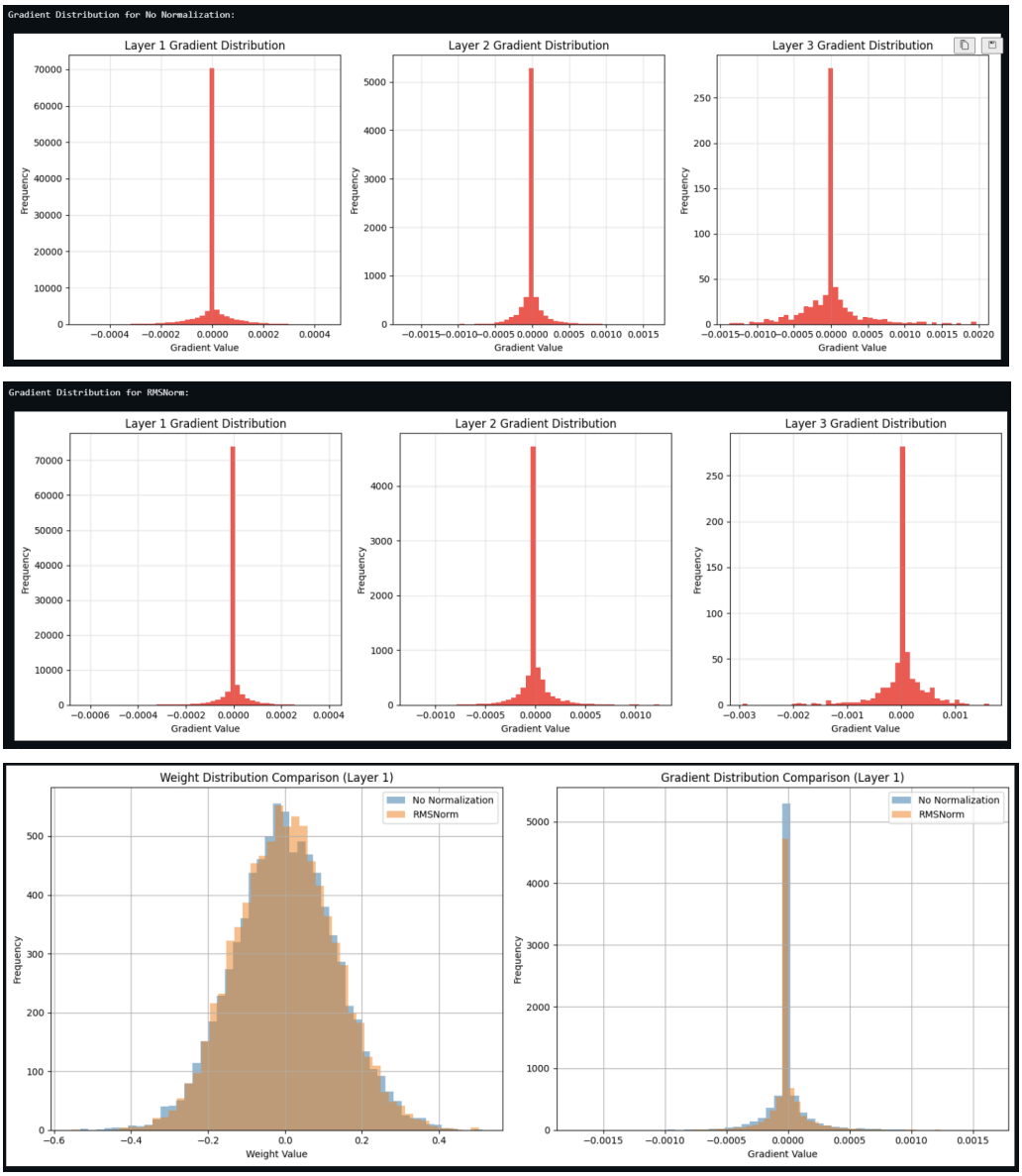


RMSNorm membantu mempertahankan penurunan loss yang lebih baik, baik pada data pelatihan maupun validasi. RMSNorm juga meningkatkan kemampuan generalisasi model, terlihat dari validation loss yang lebih rendah.

Distribusi Bobot







2.2.7 Perbandingan dengan library sklearn

Berdasarkan tes uji yang sudah dilakukan untuk perbandingan dengan library sklearn dibawah

```
--- Accuracy Summary ---
Custom FFNN Model: 0.9288
MLPClassifier: 0.9771
```

Berdasarkan tes uji yang sudah dilakukan untuk perbandingan dengan library sklearn hasil menunjukkan bahwa:

- Model **MLPClassifier** menghasilkan akurasi yang lebih tinggi dibanding model **Custom FFNN**.
- Selisih akurasi kurang lebih 4.83%, yang cukup signifikan, hal ini menunjukkan bahwa MLPClassifier memiliki keunggulan dalam menyelesaikan tugas yang diberikan dibandingkan model Custom FFNN yang belum dioptimasi lebih lanjut

Bab 3

Kesimpulan dan Saran

3.1 Kesimpulan

3.1.1 Pengaruh Depth dan Width

- Model dengan depth 3 hasilnya memiliki performa terbaik pada hal training loss dan validation loss. Hal ini menunjukan bahwa peningkatan kedalaman jaringan membantu model menangkap pola kompleks dengan lebih baik.
- Peningkatan width menunjukkan peningkatan performa, namun tidak signifikan dibandingkan dengan peningkatan depth yang mana dengan model width 3 menunjukkan hasil yang terbaik.
- Keseimbangan antara depth dan width sangat penting untuk mencapai model yang optimal tanpa overfitting.

3.1.2 Pengaruh Fungsi Aktivasi

- Fungsi aktivasi TanH menunjukkan performa yang terbaik dibandingkan fungsi aktivasi lainnya. Hal ini dibuktikan dengan akurasi tertinggi.
- ReLU juga menunjukkan performa yang baik dengan penurunan loss yang stabil selama proses pelatihan.
- Fungsi aktivasi memiliki pengaruh signifikan terhadap kemampuan model dalam menangkap pola kompleks dan generalisasi data.

3.1.3 Pengaruh Learning Rate

- Learning rate 0.1 menunjukkan hasil optimal dalam percobaan yang dilakukan, dengan performa yang lebih baik dibandingkan learning rate 0.01 dan 0.001.
- Learning rate yang terlalu kecil (0.001) menyebabkan konvergensi yang lambat, sementara learning rate yang tepat membantu model mencapai konvergensi yang lebih cepat.
- Pemilihan learning rate sangat mempengaruhi kecepatan pembelajaran dan kemampuan model untuk mencapai solusi optimal.

3.1.4 Pengaruh Inisialisasi Bobot

- Zero Initialization gagal karena menghasilkan output yang identik pada semua neuron dan mencegah model untuk belajar dengan efektif.

- He Initialization menunjukkan performa terbaik, diikuti oleh Xavier Initialization, sesuai dengan teori yang diajarkan.

3.1.5 Pengaruh Regularisasi

- Model tanpa regularisasi mendapatkan akurasi terbaik dalam kasus ini..
- Regularisasi L1 kurang efektif, bisa dilihat dengan loss yang lebih tinggi dibandingkan model lainnya.
- Regularisasi L1 menghasilkan bobot yang *sparse* dan gradien yang lebih kecil, sedangkan L2 mengurangi besaran bobot dan gradien tanpa membuatnya menjadi nol.

3.1.6 Pengaruh Normalisasi RMSNorm

- Penerapan RMSNorm tidak terlalu mendapatkan peningkatan akurasi, peningkatannya hanya sebesar 0.15% dibandingkan model tanpa normalisasi.
- RMSNorm membantu stabilitas pelatihan dan meningkatkan kemampuan generalisasi model, hal ini dapat terlihat dari validation loss yang lebih rendah.
- Meskipun tidak terlalu berpengaruh dalam kasus ini, RMSNorm tetap memberikan kontribusi yang positif pada kasus ini.

3.1.7 Perbandingan dengan Library Sklearn

- Model MLPClassifier dari sklearn menghasilkan akurasi yang lebih tinggi dibandingkan model bikinan kita.
- Hasil ini menunjukkan bahwa implementasi FFNN dari scratch masih memerlukan optimasi lebih lanjut untuk mencapai performa yang bisa bersaing dengan *library* standar.

3.2 Saran

- Eksplorasi arsitektur jaringan yang lebih kompleks dengan kombinasi depth dan width yang lebih optimal.
- Eksperimen dengan inisialisasi bobot yang lebih adaptif terhadap arsitektur jaringan dan fungsi aktivasi yang digunakan.
- Implementasikan teknik preprocessing data yang lebih robust seperti normalisasi dan standardisasi.
- Optimalkan implementasi forward dan backward propagation untuk meningkatkan efisiensi komputasi.
- Kembangkan interface yang lebih user-friendly untuk memudahkan penggunaan model oleh non-programmer.

Referensi

https://math.libretexts.org/Bookshelves/Calculus/Calculus_(OpenStax)/14%3A_Di-eren
https://numpy.org/doc/2.2/
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

Pembagian Tugas

NIM	Tugas
13522125	layers, normalization, activations(swish, gelu), notebook, laporan
13522128	weight_initializers, loss_function, laporan, readme
13522145	neural_network, activations, notebook, laporan