

LAPORAN TUGAS BESAR I
IF3270 Pembelajaran Mesin
“Feedforward Neural Network”



Dosen:

Dr. Fariska Zakhralativa Ruskanda, S.T., M.T.

Disusun Oleh:

13522126 Rizqika Mulia Pratama

13522139 Attara Majesta Ayub

13522147 Ikhwan Al Hakim

PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
SEMESTER II TAHUN 2024/2025

KATA PENGANTAR

Puji syukur kepada Tuhan Yang Maha Esa, penulis ucapkan atas kesempatan dan keberhasilan dalam menyelesaikan Tugas Besar 1 IF3270 Pembelajaran Mesin, Semester II tahun 2024/2025, yang berjudul "Feedforward Neural Network". Laporan ini merupakan dokumentasi dari proses implementasi *Feedforward Neural Network* (FFNN) yang dibuat dari nol (*from scratch*) sesuai dengan spesifikasi yang telah ditentukan.

Tugas ini memberikan pengalaman belajar yang sangat berharga dalam memahami konsep *deep learning*, mulai dari desain arsitektur model, pemilihan fungsi aktivasi, hingga proses *forward* dan *backward propagation*. Selain itu, berbagai eksperimen dilakukan untuk menganalisis pengaruh hyperparameter, metode inisialisasi bobot, serta perbandingan performa model dengan pustaka eksternal seperti scikit-learn.

Dengan bimbingan dari dosen, dukungan dari asisten pengajar, serta kerja sama dengan rekan satu tim, penulis dapat menyelesaikan tugas ini dengan baik. Akhir kata, penulis mengucapkan terima kasih kepada semua pihak yang telah berkontribusi dalam penggerjaan tugas ini. Semoga laporan ini dapat memberikan manfaat bagi pembaca serta menjadi referensi yang berguna dalam studi lebih lanjut mengenai pembelajaran mesin.

Bandung, 28 Februari 2025,
Kelompok 51

DAFTAR ISI

KATA PENGANTAR.....	1
DAFTAR ISI.....	2
DAFTAR GAMBAR DAN TABEL.....	4
BAB I	
DESKRIPSI PERSOALAN.....	5
1.1. Feedforward Neural Network.....	5
1.2. Backpropagation dengan Automatic Differentiation (AD).....	8
1.3. Fungsi Aktivasi.....	9
1.4. Fungsi Loss.....	10
BAB II	
PEMBAHASAN & IMPLEMENTASI.....	12
2.1 Repository GitHub.....	12
2.2 Deskripsi Kelas.....	12
2.2.1. Tensor.....	12
i. Atribut.....	12
ii. Metode.....	12
2.2.2. FFNN.....	13
i. Atribut.....	13
ii. Metode.....	14
2.2.3. Plotter.....	16
i. Atribut.....	16
ii. Metode.....	16
2.3 Algorithm.....	17
a) Forward Propagation.....	17
b) Backward Propagation dan Weight Update.....	18
c) Metode Inisialisasi.....	21
d) Metode Regularisasi.....	23
i) L1 (Lasso) Regularization.....	23
ii) L2 (Ridge) Regularization.....	23
e) Root Mean Square (RMS) Normalization.....	24
f) Root Mean Square (RMS) Propagation.....	25
BAB III	
HASIL PENGUJIAN.....	27
3.1 Pengaruh Depth dan Width.....	27
3.2 Pengaruh Fungsi Aktivasi Hidden Layer.....	32

3.3 Pengaruh Learning Rate.....	40
3.4 Pengaruh Inisialisasi Bobot.....	43
3.5 Pengaruh Regularisasi.....	50
3.6 Pengaruh Normalisasi RMSNorm.....	54
3.7 Perbandingan dengan Sklearn.....	57
BAB IV	
KESIMPULAN & SARAN.....	58
Kesimpulan.....	58
Saran.....	59
BAB V	
PEMBAGIAN TUGAS.....	60
REFERENSI.....	61

DAFTAR GAMBAR DAN TABEL

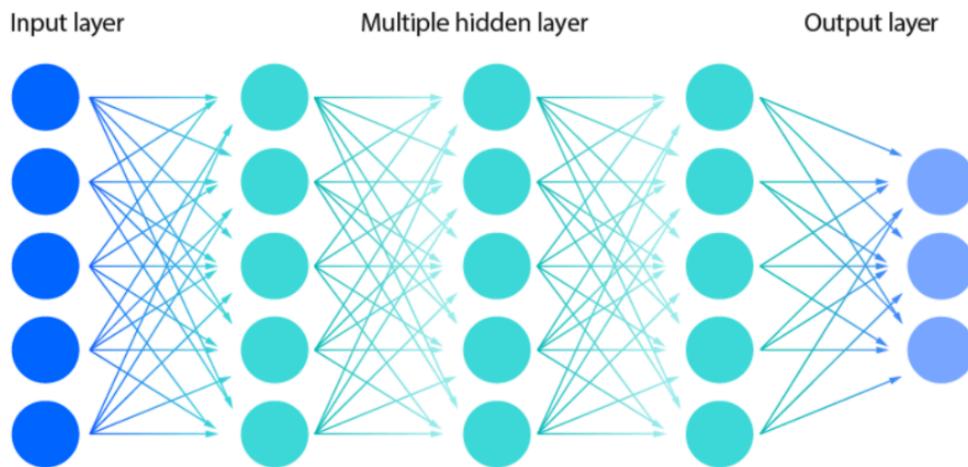
Gambar 1. Architecture of FFNN.....	5
Tabel 1. Fungsi Aktivasi (Sumber: Spesifikasi Tugas Besar 1 Pembelajaran Mesin).....	9
Tabel 2. Fungsi Loss (Sumber: Spesifikasi Tugas Besar 1 Pembelajaran Mesin).....	10
Tabel 3. Atribut Tensor.....	12
Tabel 4. Metode Tensor.....	12
Tabel 5. Atribut FFNN.....	13
Tabel 6. Metode FFNN.....	14
Tabel 7. Atribut Plotter.....	16
Tabel 8. Metode Plotter.....	16
Tabel 9. Project Assignments Table.....	58

BAB I

DESKRIPSI PERSOALAN

1.1. Feedforward Neural Network

Pada tugas ini, kami akan mengimplementasikan *Feedforward Neural Network (FFNN) from scratch*. FFNN adalah salah satu jenis *artificial neural network* yang terdiri dari beberapa lapisan neuron yang terhubung secara searah, tanpa adanya *feedback loop*. FFNN banyak digunakan dalam berbagai aplikasi *machine learning*, seperti *classification*, *regression*, dan *pattern recognition*. Model ini bekerja dengan memproses input secara bertahap melalui *hidden layers* sebelum menghasilkan output akhir.



Gambar 1. Architecture of FFNN

Struktur Feedforward Neural Network

FFNN terdiri dari tiga jenis lapisan utama:

1. Input Layer – Menerima data sebagai masukan.
2. Hidden Layers – Satu atau lebih lapisan yang melakukan transformasi non-linear menggunakan activation functions.

3. Output Layer – Menghasilkan prediksi berdasarkan hasil transformasi dari lapisan-lapisan sebelumnya.

Keunggulan Feedforward Neural Network

1. Sederhana, dapat diterapkan pada berbagai kasus tanpa kompleksitas arsitektur yang tinggi.
2. Dapat menangani berbagai jenis data, cocok untuk data numerik, gambar, maupun teks setelah proses transformasi yang sesuai.
3. Efektif dalam *supervised learning*, digunakan secara luas dalam masalah *classification* dan *regression* dengan data berlabel.

Kelemahan Feedforward Neural Network

1. Rentan terhadap overfitting, jika tidak diterapkan teknik *regularization* yang tepat, model dapat terlalu menyesuaikan diri terhadap data pelatihan.
2. Tidak optimal untuk data sekuensial karena FFNN tidak memiliki memori jangka panjang seperti Recurrent Neural Networks (RNN) yang lebih cocok untuk data berurutan.
3. Sangat bergantung pada pemilihan *hyperparameters*. Performa model ditentukan oleh faktor seperti jumlah *layers*, jumlah neuron, fungsi aktivasi, dan *learning rate*.

Model FFNN menyesuaikan *weights* dan *biases* menggunakan algoritma *backpropagation* dan teknik optimasi seperti *gradient descent*, sehingga model dapat mempelajari *pattern* dalam data secara bertahap. Selama proses ini, fungsi aktivasi memainkan peran penting dalam menentukan bagaimana sinyal diproses di setiap neuron, sementara fungsi loss mengukur seberapa jauh prediksi model dari nilai sebenarnya.

Fungsi aktivasi berfungsi untuk memperkenalkan non-linearitas dalam jaringan, memungkinkan FFNN untuk mempelajari pola yang lebih kompleks. Beberapa fungsi aktivasi yang umum digunakan meliputi *ReLU* (*Rectified Linear Unit*), yang mempertahankan nilai positif dan mengubah nilai negatif menjadi nol, serta *sigmoid* dan *tanh*, yang membatasi output dalam rentang tertentu untuk menjaga kestabilan nilai. Sementara itu, *softmax* sering digunakan di *output layer* untuk mengubah hasil keluaran menjadi probabilitas dalam kasus klasifikasi multi-kelas. Masing-masing fungsi aktivasi memiliki turunannya, yang digunakan dalam proses *backpropagation* untuk menghitung *gradients* dan memperbarui bobot.

Selain fungsi aktivasi, pemilihan fungsi loss juga sangat penting dalam FFNN. Untuk tugas regresi, *Mean Squared Error (MSE)* sering digunakan karena menghitung selisih kuadrat antara nilai prediksi dan nilai sebenarnya, memberikan penalti lebih besar untuk kesalahan yang lebih besar. Untuk klasifikasi biner, *Binary Cross-Entropy (BCE)* lebih efektif karena mengevaluasi probabilitas output terhadap label yang diharapkan, sedangkan untuk klasifikasi multi-kelas, *Categorical Cross-Entropy (CCE)* digunakan untuk mengukur seberapa baik model memprediksi kelas yang benar dalam format *one-hot encoding*.

Meskipun FFNN memiliki keunggulan dalam kesederhanaannya dan fleksibilitasnya dalam menangani berbagai jenis data, model ini juga memiliki beberapa kelemahan. Salah satunya adalah kecenderungan terhadap *overfitting*, terutama jika tidak diterapkan teknik *regularization* seperti *dropout* atau *L2 regularization*. Selain itu, FFNN tidak optimal untuk data sekuensial karena tidak memiliki memori jangka panjang seperti *Recurrent Neural Networks (RNN)*. Performa model juga sangat bergantung pada pemilihan *hyperparameters*, seperti jumlah *layers*, jumlah *neurons*, jenis

fungsi aktivasi, dan *learning rate*, yang harus disesuaikan dengan baik agar model dapat belajar secara optimal.

1.2. Backpropagation dengan Automatic Differentiation (AD)

Backpropagation dengan *automatic differentiation* (AD) adalah teknik yang digunakan untuk menghitung gradien secara efisien dalam pelatihan jaringan saraf buatan, seperti FFNN. Teknik ini memanfaatkan struktur grafik komputasi untuk secara otomatis menghitung turunan dari *loss function* terhadap setiap parameter (bobot dan bias) menggunakan aturan rantai, tanpa perlu menghitung turunan secara analitis untuk setiap operasi. Dalam konteks FFNN, *backpropagation* dengan AD memungkinkan model untuk menyesuaikan parameter secara iteratif berdasarkan data pelatihan, sehingga meningkatkan akurasi prediksi.

Proses Backpropagation dengan Automatic Differentiation

Proses *backpropagation* dengan AD terdiri dari beberapa langkah utama:

1. Forward Pass: Input proses melalui jaringan untuk menghasilkan prediksi dan menghitung nilai *loss* berdasarkan *loss function* yang dipilih.
2. Pembentukan Graf Komputasi: Setiap operasi (penjumlahan, perkalian, atau fungsi aktivasi) dicatat dalam grafik komputasi bersama gradien lokalnya.
3. Backward Pass: Gradien dari *loss function* disebarluaskan mundur melalui graf komputasi untuk menghitung gradien terhadap setiap parameter.

Keunggulan Backpropagation with Automatic Differentiation

1. Efisien dan akurat, AD menghitung gradien secara numerik stabil tanpa perlu derivasi manual yang rentan terhadap kesalahan.

2. Fleksibel karena dapat digunakan untuk berbagai operasi matematis dan fungsi aktivasi tanpa memerlukan penyesuaian khusus.
3. Mendukung kompleksitas model, cocok untuk jaringan dengan banyak lapisan dan operasi non-linear yang rumit.

Kelemahan Backpropagation with Automatic Differentiation

1. Konsumsi memori tinggi karena grafik komputasi membutuhkan penyimpanan tambahan untuk melacak semua operasi dan gradien lokal.
2. Meskipun otomatis, membangun sistem AD yang robust membutuhkan desain yang cermat.
3. Bergantung pada stabilitas numerik. Dalam kasus nilai ekstrim (misalnya, eksponensial besar pada sigmoid), AD dapat menghadapi masalah seperti *vanishing* atau *exploding gradient*.

1.3. Fungsi Aktivasi

Fungsi aktivasi memperkenalkan *non-linearity* ke dalam jaringan, memungkinkan FFNN untuk mempelajari pola yang lebih kompleks. Beberapa fungsi aktivasi yang umum digunakan meliputi:

Tabel 1. Fungsi Aktivasi (Sumber: [Spesifikasi Tugas Besar 1 Pembelajaran Mesin](#))

Nama Fungsi Aktivasi	Definisi Fungsi
Linear	$Linear(x) = x$
ReLU	$ReLU(x) = \max(0, x)$
Sigmoid	$\sigma(x) = \frac{1}{1 + e^{-x}}$

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

1.4. Fungsi Loss

Loss function mengukur sejauh mana prediksi model menyimpang dari nilai sebenarnya. Pemilihan fungsi *loss* bergantung pada jenis tugas yang diselesaikan:

Tabel 2. Fungsi Loss (Sumber: [Spesifikasi Tugas Besar 1 Pembelajaran Mesin](#))

Nama Fungsi Loss	Definisi Fungsi
MSE	$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
Binary Cross-Entropy	$\mathcal{L}_{BCE} = -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$ <p> y_i = Actual binary label (0 or 1) \hat{y}_i = Predicted value of y_i n = Batch size </p>

Categorical
Cross-Entropy

$$\mathcal{L}_{CCE} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^C (y_{ij} \log \hat{y}_{ij})$$

y_{ij} = Actual value of instance i for class j

\hat{y}_{ij} = Predicted value of y_{ij}

C = Number of classes

n = Batch size

BAB II

PEMBAHASAN & IMPLEMENTASI

2.1 Repository GitHub

Source code dari program ini dapat diakses melalui GitHub pada [laman ini](#).

2.2 Deskripsi Kelas

2.2.1. Tensor

i. Atribut

Tabel 3. Atribut Tensor

Atribut	Keterangan
value	Menyimpan nilai numerik dari node.
grad	Menyimpan gradien dari node saat proses backpropagation.
children	Menyimpan referensi ke node-node yang berkontribusi pada nilai node saat ini.
op	Menyimpan operasi yang menyebabkan terbentuknya node ini (jika ada).
requires_grad	Menentukan apakah node harus menghitung gradien atau tidak.
_backward	Fungsi yang digunakan untuk menghitung gradien selama backpropagation.

ii. Metode

Tabel 4. Metode Tensor

Metode	Keterangan
__init__()	Konstruktor untuk membuat node baru.

<code>__add__()</code>	Menjumlahkan node dengan node lain atau nilai konstan, membuat node hasil baru.
<code>__mul__()</code>	Mengalikan node dengan node lain atau nilai konstan, membuat node hasil baru.
<code>__neg__()</code>	Mengembalikan negatif dari nilai node.
<code>dot()</code>	Menghitung hasil kali dot antara dua node.
<code>relu()</code>	Menerapkan fungsi aktivasi ReLU (Rectified Linear Unit).
<code>sigmoid()</code>	Menerapkan fungsi aktivasi sigmoid.
<code>tanh()</code>	Menerapkan fungsi aktivasi hyperbolic tangent.
<code>leaky_relu()</code>	Menerapkan fungsi aktivasi Leaky ReLU dengan parameter alpha kustom.
<code>elu()</code>	Menerapkan fungsi aktivasi Exponential Linear Unit (ELU).
<code>softmax()</code>	Menerapkan fungsi aktivasi Softmax.
<code>log()</code>	Menghitung logaritma dari nilai node.
<code>mean()</code>	Menghitung rata-rata dari nilai node.
<code>backward()</code>	Metode rekursif untuk menghitung gradien mundur (backpropagation), menelusuri jejak komputasi dan memperbarui gradien di setiap node.

2.2.2. FFNN

i. Atribut

Tabel 5. Atribut FFNN

Atribut	Keterangan
<code>input_size</code>	Jumlah neuron pada lapisan input.

hidden_sizes	Daftar jumlah neuron di setiap lapisan tersembunyi.
num_hidden_layers	Jumlah lapisan tersembunyi.
output_size	Jumlah neuron pada lapisan output.
learning_rate	Tingkat pembelajaran untuk pembaruan bobot.
loss_function	Fungsi kerugian untuk mengukur kesalahan.
reg_type	Jenis regularisasi.
reg_lambda	Parameter regularisasi untuk mengendalikan kompleksitas model.
rms_norm	Flag untuk mengaktifkan normalisasi RMS (Root Mean Square).
rms_epsilon	Nilai epsilon kecil untuk menghindari pembagian dengan nol.
weights	Matriks bobot untuk setiap lapisan.
biases	Vektor bias untuk setiap lapisan.
hidden_activations	Fungsi aktivasi untuk lapisan tersembunyi.
output_activation	Fungsi aktivasi untuk lapisan output.
history	Riwayat kerugian pelatihan dan validasi.

ii. Metode

Tabel 6. Metode FFNN

Metode	Keterangan
__init__()	Menginisialisasi arsitektur jaringan

	saraf.
apply_activation()	Menerapkan fungsi aktivasi pada sebuah node.
apply_softmax()	Menerapkan fungsi aktivasi softmax pada kumpulan node.
feedforward()	Melakukan propagasi maju untuk menghasilkan prediksi.
cce()	Menghitung kerugian Cross-Categorical Entropy.
bce()	Menghitung kerugian Binary Cross-Entropy.
mse()	Menghitung kerugian Mean Squared Error.
compute_loss()	Memilih dan menghitung fungsi kerugian berdasarkan konfigurasi
compute_regularization_loss()	Menghitung kerugian regularisasi (L1 atau L2).
train()	Melatih jaringan saraf dengan data pelatihan dan validasi.
predict()	Membuat prediksi untuk input tertentu.
compare_lib()	Membandingkan akurasi dengan MLPClassifier dari scikit-learn.
plot_training_history()	Memvisualisasikan kerugian pelatihan dan validasi.
get_layer_weights()	Mendapatkan bobot untuk lapisan tertentu.
get_layer_name()	Mendapatkan nama lapisan berdasarkan

	indeks.
plot_weight_distributions()	Memvisualisasikan distribusi bobot.
plot_gradient_distributions()	Memvisualisasikan distribusi gradien.
visualize_network()	Membuat visualisasi struktur jaringan saraf.

2.2.3. Plotter

i. Atribut

Tabel 7. Atribut Plotter

Atribut	Keterangan
__init__()	Semua metode bekerja dengan objek feedforward neural network (ffnn) yang diberikan sebagai parameter.

ii. Metode

Tabel 8. Metode Plotter

Metode	Keterangan
visualize_network_1ight()	Memvisualisasikan jaringan saraf buatan dengan networkx dan matplotlib, membatasi jumlah neuron yang ditampilkan per lapisan agar tidak terlalu padat.
plot_training_history()	Menampilkan grafik riwayat pelatihan model berdasarkan training loss dan validation loss.
get_layer_weights()	Mengambil bobot (weight) dari lapisan tertentu dalam model jaringan saraf.
get_layer_gradients()	Mengambil nilai gradien dari lapisan tertentu dalam model jaringan saraf.

get_layer_name()	Mengembalikan nama deskriptif untuk lapisan tertentu dalam model jaringan saraf.
plot_weight_distributions()	Menampilkan histogram distribusi bobot untuk lapisan yang dipilih dalam model.
plot_gradient_distributions()	Menampilkan histogram distribusi gradien untuk lapisan yang dipilih dalam model.

2.3 Algorithm

a) Forward Propagation

Metode feedforward dalam FFNN meneruskan inputs melalui *network* hingga menghasilkan output. Inputs dikonversi menjadi Tensor dan disimpan dalam `layer_outputs`, yang merekam hasil komputasi setiap layer.

Perhitungan dilakukan secara berurutan dari *input layer* ke *hidden layers* hingga *output layer*. Setiap *hidden layer* menghitung nilai neuron dengan operasi dot product antara bobot dan output sebelumnya, ditambah bias. Hasilnya kemudian diterapkan activation function melalui `apply_activation`. Pada output layer, proses serupa dilakukan dengan *weights* dan bias terakhir sebelum *activation function* diterapkan. Seluruh hasil disimpan dalam `layer_outputs` dan dikembalikan sebagai keluaran metode ini.

```
def feedforward(self, inputs):
    inputs = Tensor(inputs, requires_grad=False)
    layer_outputs = [inputs]

    for layer_idx in range(self.num_hidden_layers):
        z = layer_outputs[-1].dot(self.weights[layer_idx]) +
        self.biases[layer_idx]
```

```

        a = self.apply_activation(z,
self.hidden_activations[layer_idx])
layer_outputs.append(a)

z = layer_outputs[-1].dot(self.weights[-1]) + self.biases[-1]
output = self.apply_activation(z, self.output_activation)
layer_outputs.append(output)

return layer_outputs

```

b) Backward Propagation dan Weight Update

Algoritma backpropagation melibatkan perhitungan gradien dari *loss function* terhadap parameter (bobot dan bias) dengan menggunakan aturan rantai yang kemudian digunakan untuk memperbarui parameter tersebut agar meminimalkan *loss*. Di class Tensor, operasi backward propagation dilakukan dalam metode `backward()`, yang mengimplementasikan autograd secara manual:

```

def backward(self):
    topo = []
    visited = set()
    def build_topo(v):
        if v not in visited:
            visited.add(v)
            for child in v._children:
                build_topo(child)
            topo.append(v)
    build_topo(self)

    self.grad = np.ones_like(self.data)
    for node in reversed(topo):
        node._backward()

```

Dalam kelas Tensor, metode `backward()` berfungsi untuk menyebarkan gradien menggunakan DAG (Directed Acyclic Graph) yang merepresentasikan dependensi antar operasi dalam jaringan. Proses ini

memastikan bahwa gradien dihitung dalam *reverse order propagation*, dari loss hingga ke layer pertama. Sebelum propagasi gradien, tensor-tensor diurutkan dalam *topological order* menggunakan traversal DAG. Ketika `backward()` dipanggil, gradien dihitung dan diteruskan ke *parent nodes* dengan mengalikan gradien global dengan gradien lokal dari operasi terkait, sesuai dengan *chain rule*.

Backpropagation dimulai setelah proses feedforward menghasilkan keluaran jaringan (`layer_outputs`) dari input tertentu. Dalam metode `train` di kelas FFNN, setelah menghitung keluaran untuk setiap sampel dalam batch menggunakan feedforward, *loss function* dihitung dengan menggunakan metode `compute_loss` berdasarkan `output` terakhir (`layer_outputs[-1]`) dan target. *Loss* ini bisa berupa MSE, BCE, atau CCE, tergantung pada konfigurasi di awal. Jika regularisasi (L1 atau L2) aktif, kerugian regularisasi ditambahkan melalui `compute_regularization_loss`.

```

for batch_idx in pbar:
    start_idx = batch_idx * batch_size
    end_idx = min((batch_idx + 1) * batch_size, len(training_data))
    batch_inputs = training_data[start_idx:end_idx]
    batch_targets = target_data[start_idx:end_idx]

    for w, b in zip(self.weights, self.biases):
        w.grad = np.zeros_like(w.data)
        b.grad = np.zeros_like(b.data)

    layer_outputs = self.feedforward(batch_inputs)
    outputs = layer_outputs[-1]

    if self.loss_function == 'mse':
        diff = outputs - Tensor(batch_targets, requires_grad=False)
        loss = (diff * diff).mean()
    elif self.loss_function == 'cce':

```

```

        targets = Tensor(batch_targets, requires_grad=False)
        loss = (-(targets * outputs.log())).mean()
    elif self.loss_function == 'bce':
        targets = Tensor(batch_targets, requires_grad=False)
        outputs_clipped = Tensor(np.clip(outputs.data, 1e-15, 1 - 1e-15))
        loss = (-(targets * outputs_clipped.log()) + (Tensor(1) - targets) * (Tensor(1) - outputs_clipped).log()).mean()

        batch_loss = loss.data
        if self.reg_type != 'none':
            reg_loss = self.compute_regularization_loss()
            batch_loss += reg_loss
        total_loss += batch_loss

    loss.backward()

```

Selanjutnya, metode backward dari kelas `Tensor` dipanggil pada `loss` (`loss.backward()`), yang memulai proses *backpropagation*. Pada implementasi ini, kami menggunakan *automatic differentiation* untuk menghitung gradien secara otomatis tanpa perlu menuliskan rumus turunan secara eksplisit.

Setelah gradien dihitung dan disebarluhan ke semua bobot dan bias (disimpan dalam atribut `out` pada objek `Tensor`), update bobot dilakukan dalam loop di metode `train`. Untuk setiap lapisan, bobot dan bias di-*update* berdasarkan gradiennya. Kemudian akan diperiksa apakah RMS normalization aktif. Jika tidak, *update* dilakukan dengan pengurangan sederhana `self.weights[layer_idx].data -= self.learning_rate * w_grad`, yang merupakan *update* gradien turun standar. Mengenai L1, L2, dan RMSNorm akan dirincikan pada poin-poin berikutnya.

```

for layer_idx in range(len(self.weights)):
    w_grad = self.weights[layer_idx].grad
    b_grad = self.biases[layer_idx].grad

    if self.reg_type == 'l1':
        # action
    elif self.reg_type == 'l2':
        # action

    if self.rms_norm:
        # action
    else:
        self.weights[layer_idx].data -= self.learning_rate * w_grad
        self.biases[layer_idx].data -= self.learning_rate * b_grad

```

Singkatnya, backpropagation dilakukan dengan menyebarkan gradien melalui grafik komputasi yang dibangun oleh kelas Tensor, menghitung turunan lokal setiap operasi, lalu memperbarui bobot menggunakan gradient descent atau RMS normalization, dengan opsi regularisasi untuk meningkatkan generalisasi model.

c) Metode Inisialisasi

Metode inisialisasi *weights* dan *biases* dalam FFNN dilakukan di dalam konstruktor (`__init__` method) saat objek pertama kali dibuat, sebelum network digunakan untuk *training* atau *inference*. Proses ini terjadi setelah parameter utama seperti ukuran layer, activation function, dan jenis inisialisasi diatur. Inisialisasi bertujuan untuk memberikan nilai awal pada parameter network sehingga dapat diproses selama *training*.

Beberapa variabel penting digunakan dalam proses inisialisasi ini, antara lain `input_size` yang menentukan jumlah neuron di *input layer*, `hidden_sizes` yang berisi list jumlah neuron di setiap *hidden layer*, serta `output_size` yang menunjukkan jumlah neuron di *output layer*. Selain itu, terdapat variabel `zero_init` yang menentukan apakah semua *weights* dan *biases* akan diinisialisasi ke nol. Jika `zero_init` tidak aktif, metode

inisialisasi yang digunakan akan bergantung pada nilai `init_type`, yang dapat berupa `"uniform"`, `"normal"`, `"xavier_uniform"`, `"xavier_normal"`, atau `"he"`. Rentang nilai distribusi ditentukan oleh `lower_bound` dan `upper_bound` untuk inisialisasi uniform, serta `mean` dan `variance` untuk inisialisasi normal. Untuk memastikan hasil yang dapat direproduksi, sebuah `seed` juga dapat digunakan.

Proses inisialisasi dimulai dengan menentukan ukuran setiap layer, yang kemudian disusun dalam list `layer_sizes` yang berisi jumlah neuron pada setiap layer dari *input* hingga *output*. Jika `rms_norm` diaktifkan, maka array kosong `rms_weights_cache` dan `rms_biases_cache` akan disiapkan untuk menyimpan informasi normalisasi *weights* dan *biases*. Selanjutnya, iterasi dilakukan untuk setiap pasangan layer dalam `layer_sizes`. Jika `zero_init` diaktifkan, semua *weights* dan *biases* akan diatur ke nol. Namun, jika tidak, metode inisialisasi diterapkan sesuai dengan `init_type`.

```

for i in range(len(layer_sizes) - 1):
    if not zero_init:
        if init_type == "uniform":
            w = np.random.uniform(lower_bound, upper_bound,
layer_sizes[i], layer_sizes[i+1])
            b = np.random.uniform(lower_bound, upper_bound,
layer_sizes[i+1])
        elif init_type == "normal":
            w = np.random.normal(mean, np.sqrt(variance),
layer_sizes[i], layer_sizes[i+1])
            b = np.random.normal(mean, np.sqrt(variance),
layer_sizes[i+1])
        elif init_type == "xavier":
            limit = np.sqrt(6 / (layer_sizes[i] + layer_sizes[i+1]))
            w = np.random.uniform(-limit, limit, (layer_sizes[i],
layer_sizes[i+1]))
            b = np.zeros(layer_sizes[i+1])
        elif init_type == "he":
            w = np.random.normal(0, np.sqrt(2 / layer_sizes[i]),
layer_sizes[i], layer_sizes[i+1])
            b = np.zeros(layer_sizes[i+1])

```

```

        else:
            raise ValueError("init_type should be 'uniform',
'normal', 'xavier', or 'he'")
        else:
            w = np.zeros((layer_sizes[i], layer_sizes[i+1]))
            b = np.zeros(layer_sizes[i+1])

```

d) Metode Regularisasi

i) L1 (Lasso) Regularization

L1 regularization, juga dikenal sebagai Lasso regularization, menambahkan penalti yang sebanding dengan nilai absolut dan bobot model yaitu (rumus), di mana lambda adalah koefisien regulasi (reg_lambda) dan w adalah bobot.

Dalam implementasi, saat `reg_type = 'l1'`, penalti dihitung dengan menjumlahkan nilai absolut semua bobot dan dikalikan dengan (lambda). Selama proses pembelajaran, gradien bobot akan ditambah dengan lambda (jika bobot positif) atau dikurangi (lambda) (jika bobot negatif), mendorong beberapa bobot menjadi nol. Hal ini membuat L1 regularization efektif untuk seleksi fitur, karena bobot yang tidak relevan cenderung dihilangkan sepenuhnya.

ii) L2 (Ridge) Regularization

L2 regularization, yang disebut juga Ridge regularization, menambahkan penalti berdasarkan kuadrat dari bobot model, yaitu (rumus). Saat `reg_type='l2'`, Penalti dihitung sebagai jumlah kuadrat bobot dikalikan dengan (lambda/2), dan gradien bobot ditambah dengan (lambda kali w). Berbeda dengan L1, L2 tidak mendorong bobot menjadi nol, tetapi membuat bobot-bobot menjadi kecil dan merata. Ini membantu mengurangi sensitivitas model terhadap outlier dan menghasilkan distribusi bobot yang lebih halus, sehingga meningkatkan stabilitas model. Dalam praktik, L2 sering lebih disukai karena cenderung memberikan hasil yang lebih baik pada dataset dengan banyak fitur yang saling berkorelasi.

```

if self.reg_type == 'l1':

```

```

        w_grad += self.reg_lambda *
np.sign(self.weights[layer_idx].data)
elif self.reg_type == 'l2':
    w_grad += self.reg_lambda * self.weights[layer_idx].data

```

e) Root Mean Square (RMS) Normalization

RMS *Normalization* digunakan untuk mempercepat dan menstabilkan pembelajaran dengan menormalkan aktivasi pada setiap lapisan jaringan saraf secara adaptif. RMS *normalization* bekerja dengan cara membagi aktivasi oleh akar rata-rata kuadrat (RMS) dari nilainya, sehingga skala aktivasi tetap konsisten selama pelatihan. Teknik ini membantu mengatasi masalah ketidakstabilan gradien, seperti vanishing atau exploding gradients, yang sering terjadi pada jaringan saraf dalam atau saat menangani data berurutan.

Implementasi RMS *Normalization* yang dilakukan adalah dengan menerapkannya pada aktivasi lapisan tersembunyi dalam FFNN. Setelah aktivasi dihitung menggunakan fungsi aktivasi (misalnya, sigmoid atau ReLU), RMS *Normalization* diterapkan jika flag `rms_norm` diatur ke True. Normalisasi ini dihitung menggunakan formula berikut:

$$RMSNorm(x) = \frac{x}{\sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2 + \epsilon}}$$

di mana:

- x adalah vektor aktivasi,
- n adalah jumlah fitur dalam vektor aktivasi,
- ϵ (`rms_epsilon`, default `1e-8`) adalah konstanta kecil untuk mencegah pembagian dengan nol.

Formula ini memastikan bahwa aktivasi dinormalisasi berdasarkan estimasi akar rata-rata kuadrat (RMS) dari nilainya, yang kemudian digunakan untuk menjaga stabilitas selama pelatihan.

Keunggulan RMS *Normalization* terletak pada kesederhanaan dan efisiensinya dalam menyesuaikan aktivasi secara individual untuk setiap

sampel, tanpa bergantung pada statistik batch seperti *Batch Normalization*. Jika aktivasi memiliki variasi yang besar, RMS *Normalization* akan menormalkannya sehingga gradien dapat mengalir lebih baik selama *backpropagation*, mencegah osilasi atau divergensi. Sebaliknya, jika aktivasi kecil, normalisasi ini memastikan pembelajaran tetap stabil dan cepat.

f) Root Mean Square (RMS) Propagation

RMS *propagation* digunakan untuk mempercepat dan menstabilkan pembelajaran dengan menyesuaikan *learning rate* secara adaptif. RMS *normalization* bekerja dengan cara melacak rata-rata bergerak dari kuadrat gradien untuk setiap bobot dan bias. Teknik ini membantu mengatasi masalah ketidakstabilan gradien yang sering terjadi pada metode *gradient descent* standar, terutama pada topologi jaringan yang dalam atau dataset kompleks.

Implementasi RMS *propagation* yang dilakukan adalah dengan menyimpan cache untuk bobot dan bias dalam `rms_weights_cache` dan `rms_biases_cache`. Selama pelatihan, cache ini diperbarui menggunakan formula eksponensial moving average: $\text{cache} = 0.9 * \text{cache} + 0.1 * (\text{gradient}^2)$. Nilai 0.9 dan 0.1 adalah koefisien dekay dan kontribusi gradien baru, yang dapat dianggap sebagai hyperparameter untuk mengontrol seberapa cepat cache “melupakan” informasi gradien sebelumnya. Cache ini merepresentasikan estimasi akar rata-rata kuadrat (RMS) dari gradien, yang kemudian digunakan untuk menormalkan pembaruan bobot. Pembaruan bobot dihitung dengan (rumus), di mana ϵ (`rms_epsilon`, default 1e-8) adalah konstanta kecil untuk mencegah pembagian dengan nol.

Keunggulan RMS normalization terletak pada kemampuannya untuk menyesuaikan laju pembelajaran secara individual untuk setiap parameter berdasarkan besarnya gradien historis. Jika gradien suatu bobot secara konsisten besar, cache akan meningkat, sehingga langkah pembaruannya menjadi lebih kecil, mencegah osilasi atau divergensi. Sebaliknya, jika gradien kecil, cache menurun, memungkinkan langkah yang lebih besar untuk mempercepat konvergensi.

```
if self.rms_prop:
```

```
        self.rms_weights_cache[layer_idx] = 0.9 *
self.rms_weights_cache[layer_idx] + 0.1 * (w_grad ** 2)
adjusted_lr = self.learning_rate /
(np.sqrt(self.rms_weights_cache[layer_idx] + self.rms_epsilon))
self.weights[layer_idx].data -= adjusted_lr * w_grad

        self.rms_biases_cache[layer_idx] = 0.9 *
self.rms_biases_cache[layer_idx] + 0.1 * (b_grad ** 2)
adjusted_lr = self.learning_rate /
(np.sqrt(self.rms_biases_cache[layer_idx] + self.rms_epsilon))
self.biases[layer_idx].data -= adjusted_lr * b_grad
else:
    self.weights[layer_idx].data -= self.learning_rate * w_grad
    self.biases[layer_idx].data -= self.learning_rate * b_grad
```

BAB III

HASIL PENGUJIAN

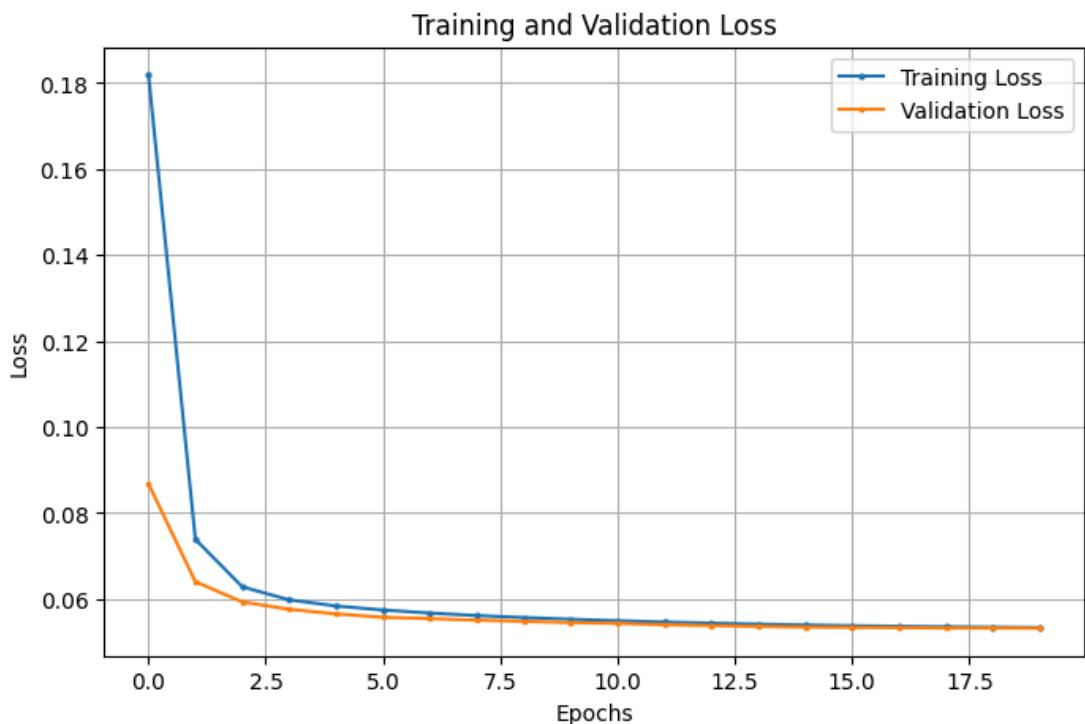
3.1 Pengaruh Depth dan Width

Kami menggunakan FFNN yang diinstansiasikan dengan parameter berikut

Parameter	Nilai
Input nodes	784
Output nodes	10
Learning rate	0.001
Hidden activation	ReLU
Output activation	Softmax
Loss function	Categorical Cross-Entropy (CCE)
Random seed	69420
Epochs	20

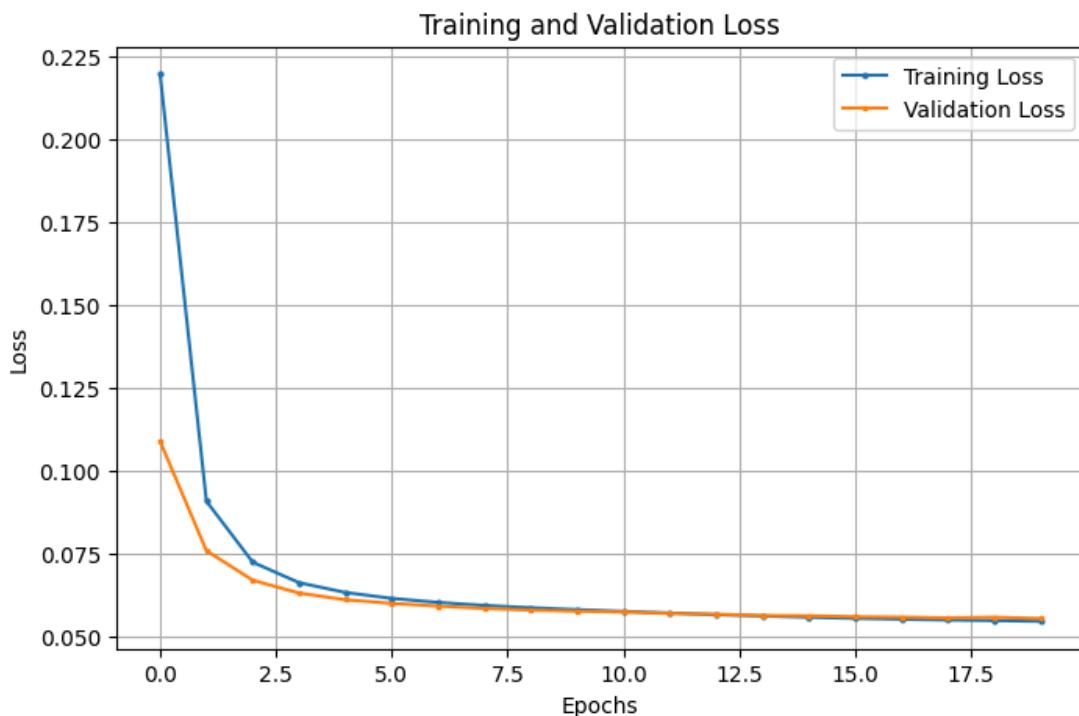
- Pengaturan Depth dan Width awal

Parameter	Nilai
Depth	3
Susunan Neuron	256, 128
Akurasi	93.87%



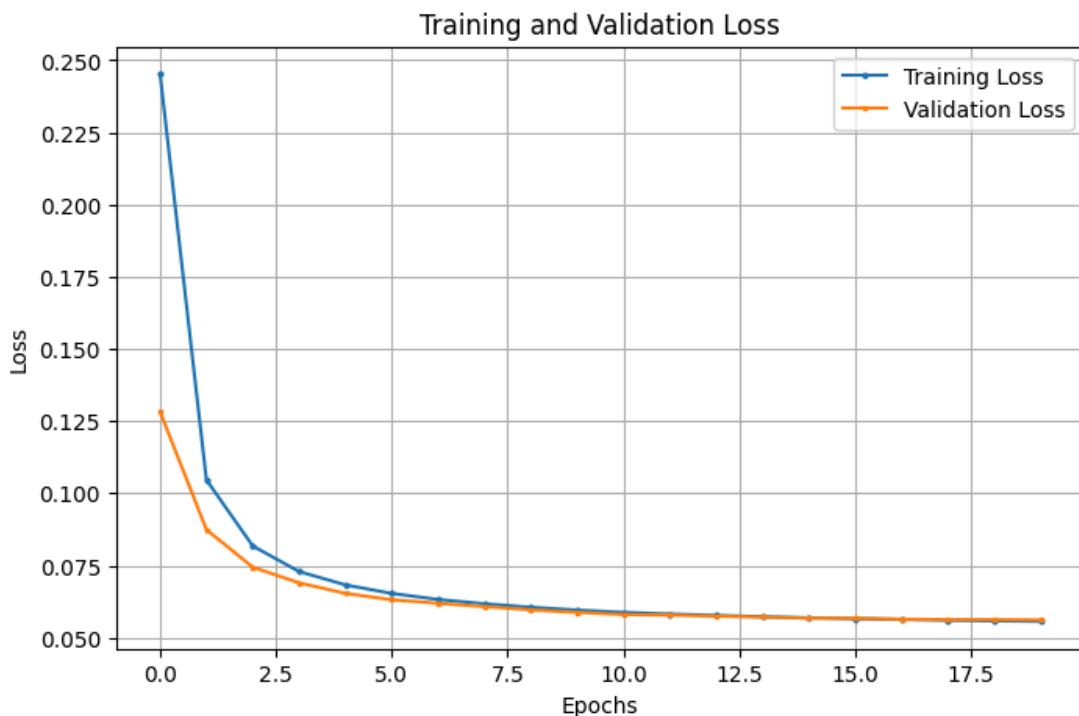
- Variasi Depth 1

Parameter	Nilai
Depth	4
Susunan Neuron	256, 128, 64
Akurasi	94.33%



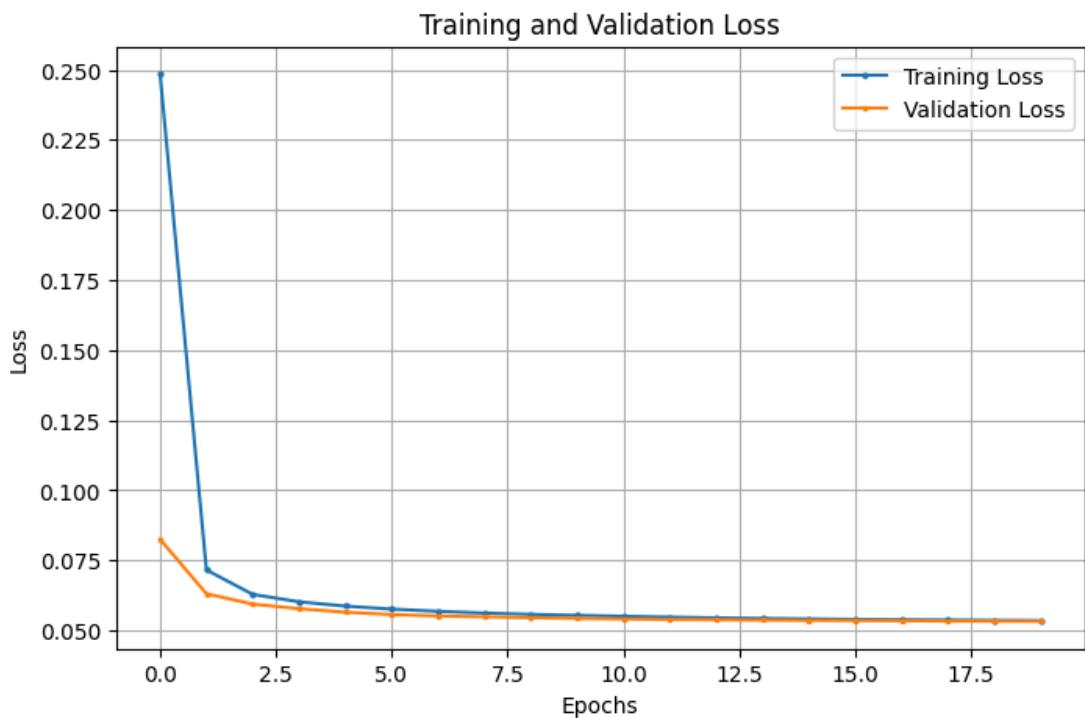
- Variasi Depth 2

Parameter	Nilai
Depth	5
Susunan Neuron	256, 128, 64, 32
Akurasi	94.33%



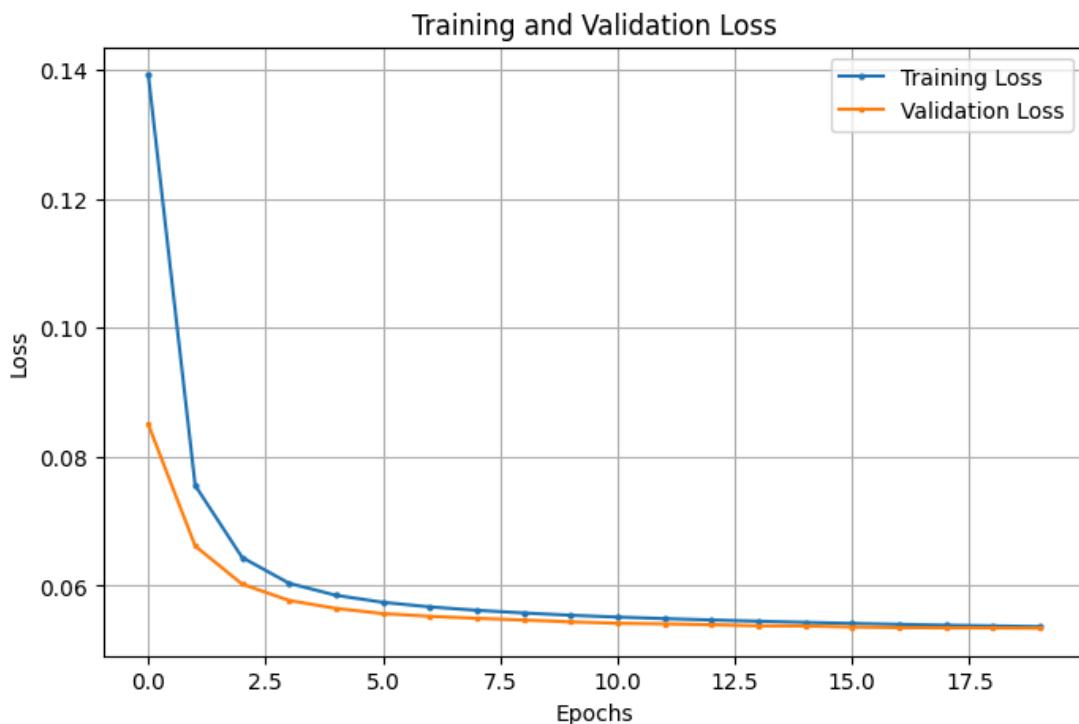
- Variasi Width 1

Parameter	Nilai
Depth	3
Susunan Neuron	512, 256
Akurasi	93.80%



- Variasi Width 2

Parameter	Nilai
Depth	3
Susunan Neuron	128, 64
Akurasi	93.83%



Peningkatan *width*, yaitu menambah jumlah neuron dalam satu lapisan, memperluas kapasitas model untuk menangkap lebih banyak fitur atau pola dalam data pada tingkat yang sama, sehingga cocok untuk masalah dengan variasi data yang luas tetapi tidak terlalu kompleks secara hierarkis. Sebaliknya, peningkatan *depth*, yaitu menambah jumlah lapisan, memungkinkan model untuk mempelajari representasi yang lebih abstrak dan hierarkis melalui pemrosesan bertahap, yang sangat berguna untuk tugas-tugas kompleks seperti pengenalan gambar atau bahasa alami. Namun, menambah *depth* dapat menyebabkan masalah seperti *vanishing gradient* dan membutuhkan teknik seperti normalisasi atau optimasi khusus, sementara menambah *width* cenderung lebih stabil tetapi bisa meningkatkan kebutuhan komputasi tanpa menjamin peningkatan performa jika data tidak cukup kaya.

3.2 Pengaruh Fungsi Aktivasi Hidden Layer

Kami menggunakan FFNN yang diinstansiasikan dengan parameter berikut

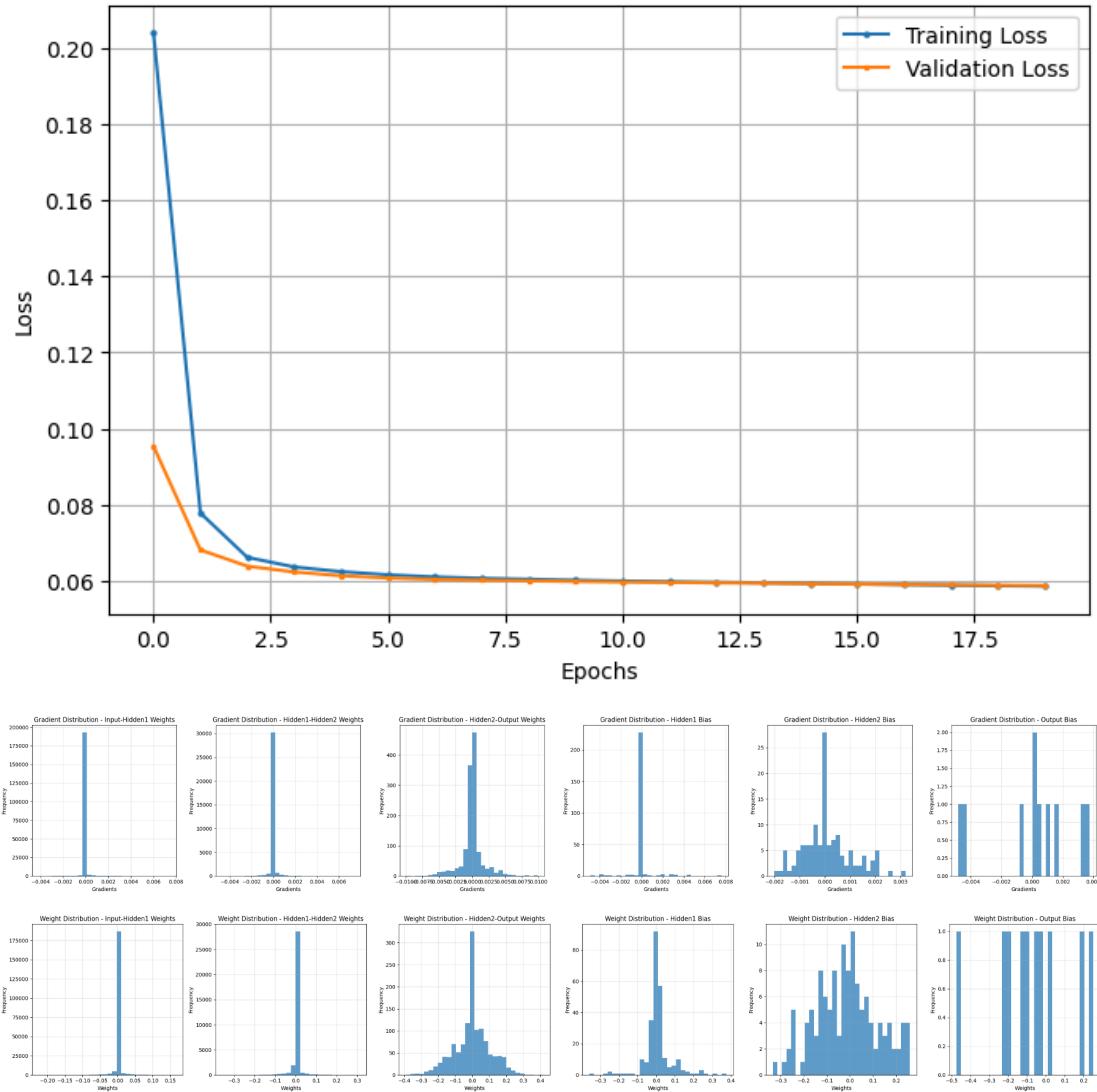
Parameter	Nilai
-----------	-------

Input nodes	784
Susunan Neuron	256, 128
Output nodes	10
Learning rate	0.001
Output activation	Softmax
Loss function	Categorical Cross-Entropy (CCE)
Random seed	69420
Epochs	20

- Fungsi Aktivasi Linear

Parameter	Nilai
Akurasi	89.67%

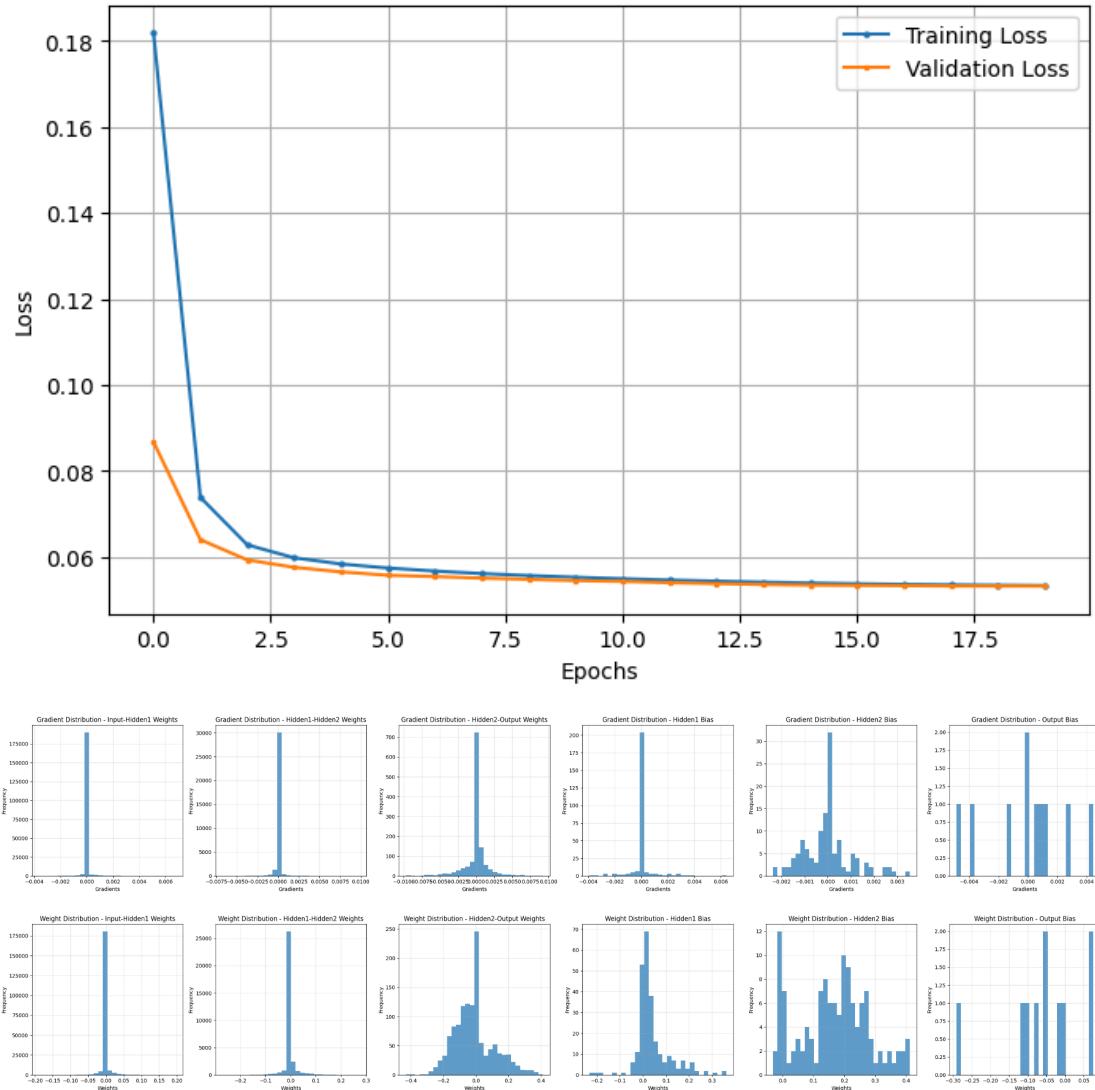
Training and Validation Loss



- Fungsi Aktivasi ReLU

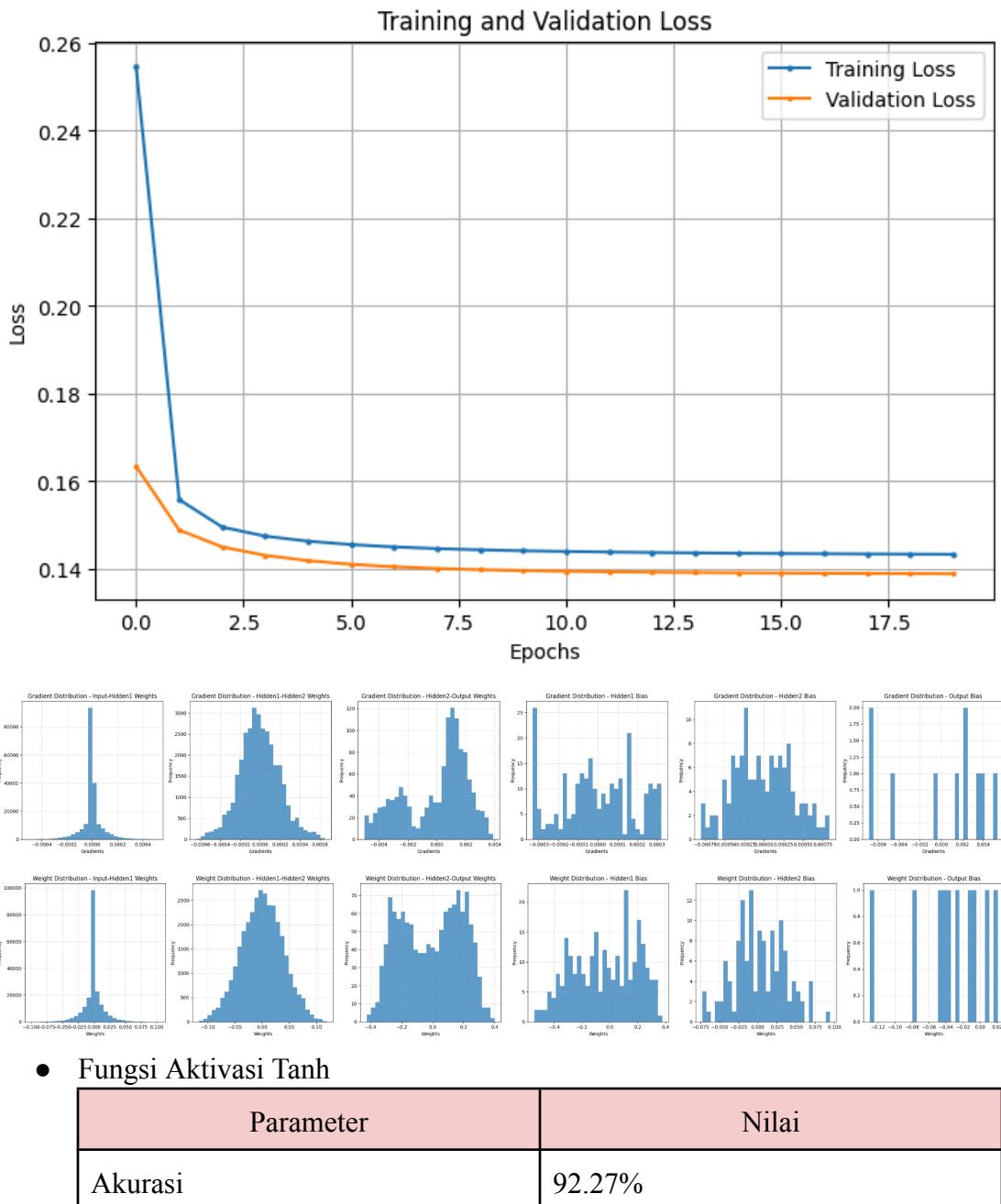
Parameter	Nilai
Akurasi	93.87%

Training and Validation Loss



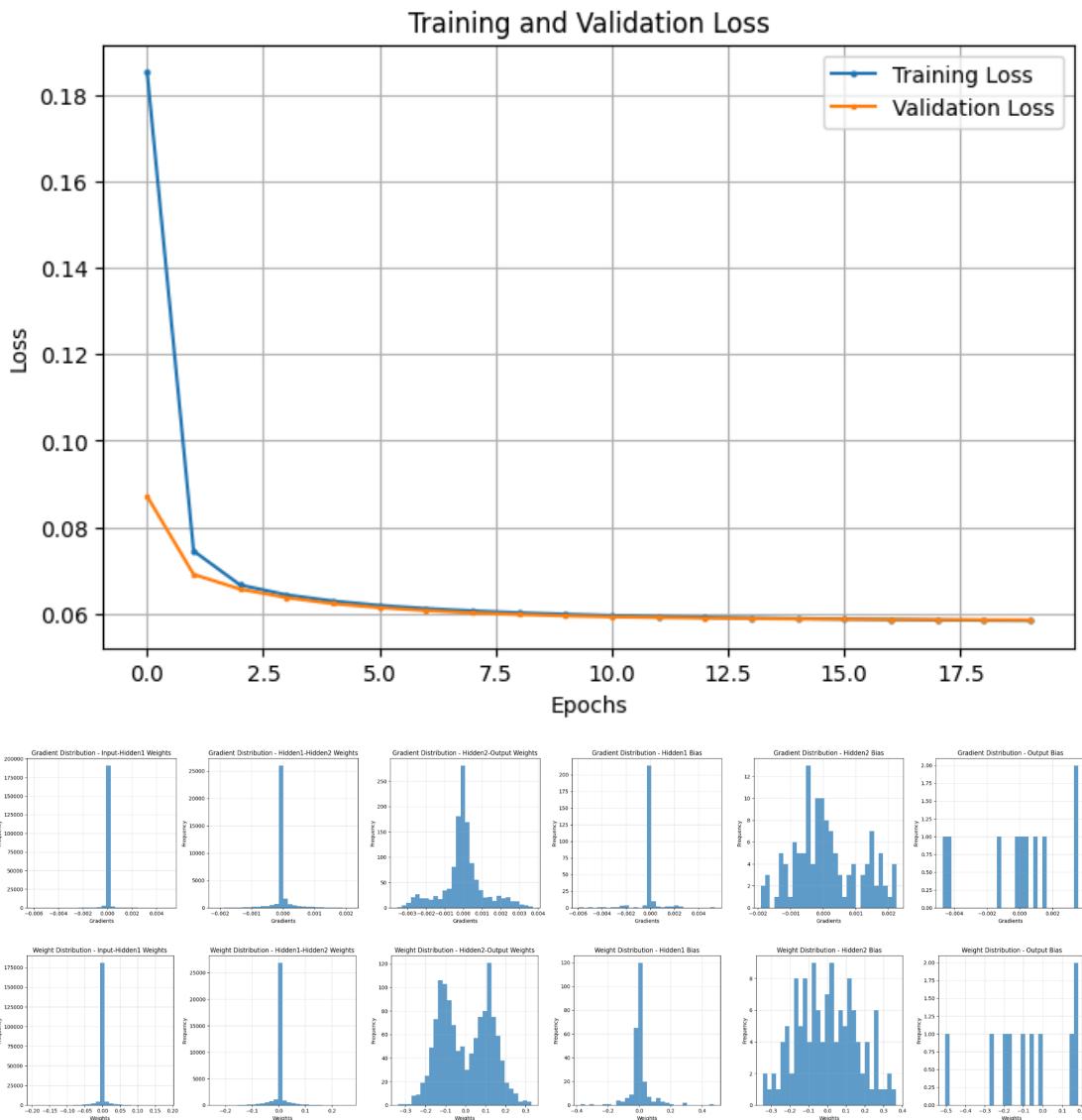
- Fungsi Aktivasi Sigmoid

Parameter	Nilai
Akurasi	80.07%



- Fungsi Aktivasi Tanh

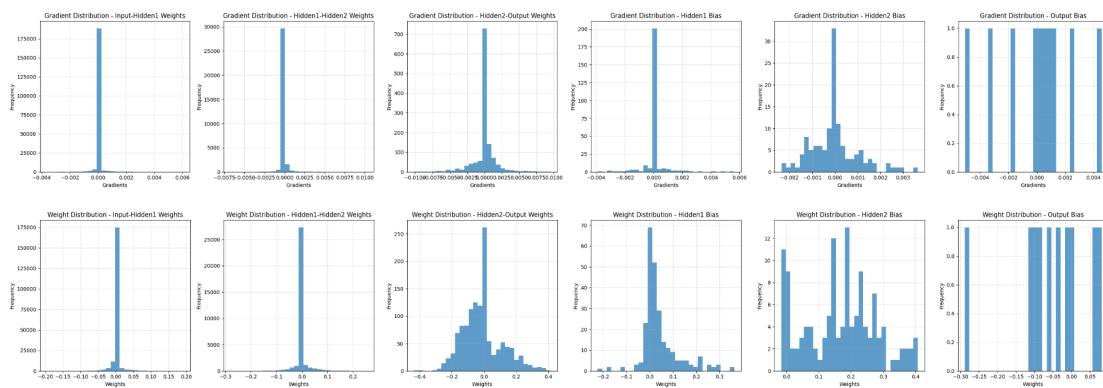
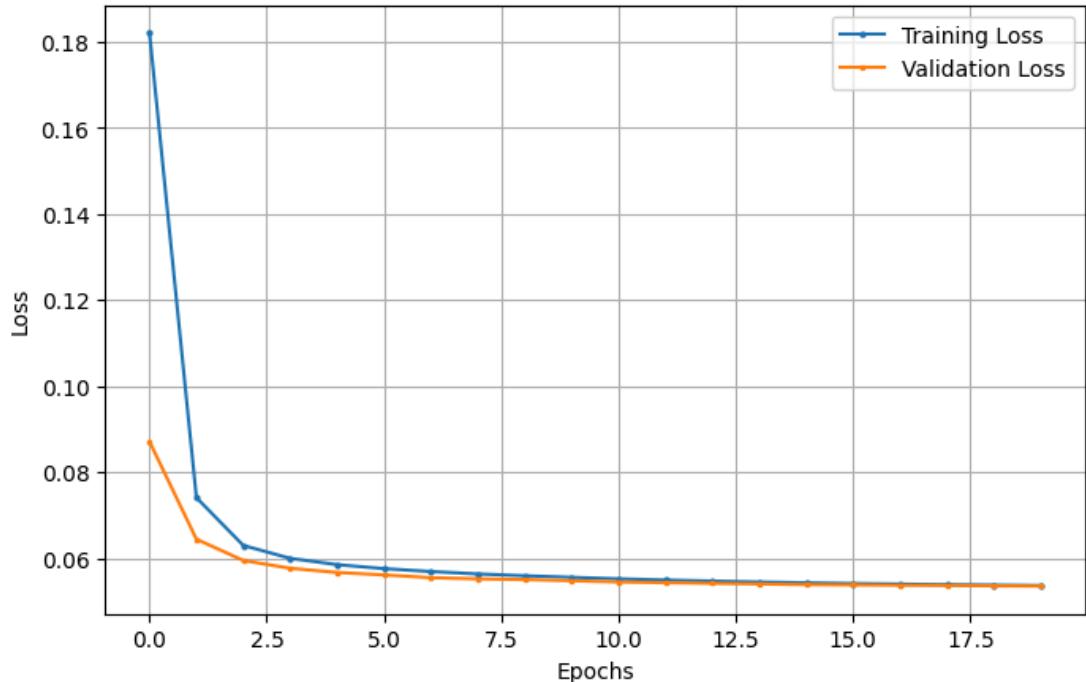
Parameter	Nilai
Akurasi	92.27%



- Fungsi Aktivasi Leaky ReLU

Parameter	Nilai
Akurasi	93.70%

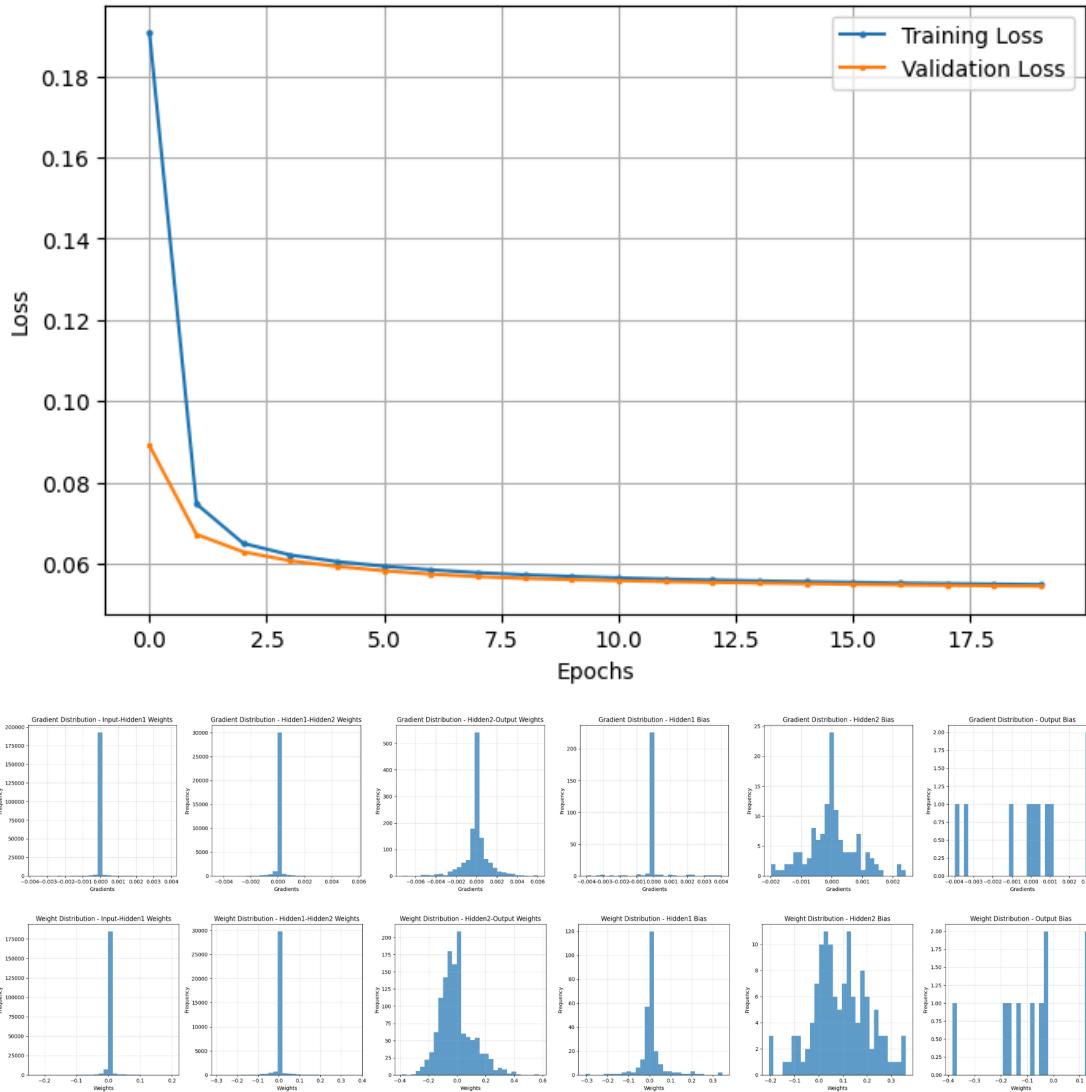
Training and Validation Loss



- Fungsi Aktivasi eLU

Parameter	Nilai
Akurasi	92.63%

Training and Validation Loss



Fungsi linear menghasilkan transformasi sederhana tanpa ambang batas, sehingga kurang mampu menangkap pola non-linear kompleks. ReLU (Rectified Linear Unit) efektif mengatasi masalah vanishing gradient, mengizinkan jaringan belajar dengan cepat dan mencegah saturasi, namun dapat mengalami "mati" pada neuron yang selalu menghasilkan output nol. Sigmoid menghasilkan output antara 0-1, berguna untuk klasifikasi biner, tetapi rentan terhadap vanishing gradient pada jaringan dalam. Fungsi tanh serupa dengan sigmoid namun memiliki rentang output -1 hingga 1, memungkinkan konvergensi yang lebih cepat. Leaky ReLU mengatasi

masalah "mati" pada ReLU dengan mengizinkan gradient kecil untuk input negatif, sementara Exponential Linear Unit (ELU) memberikan keuntungan ReLU dengan menambahkan kemampuan untuk menghasilkan output negatif, yang dapat meningkatkan akurasi dan konvergensi model dalam berbagai pembelajaran mesin.

3.3 Pengaruh Learning Rate

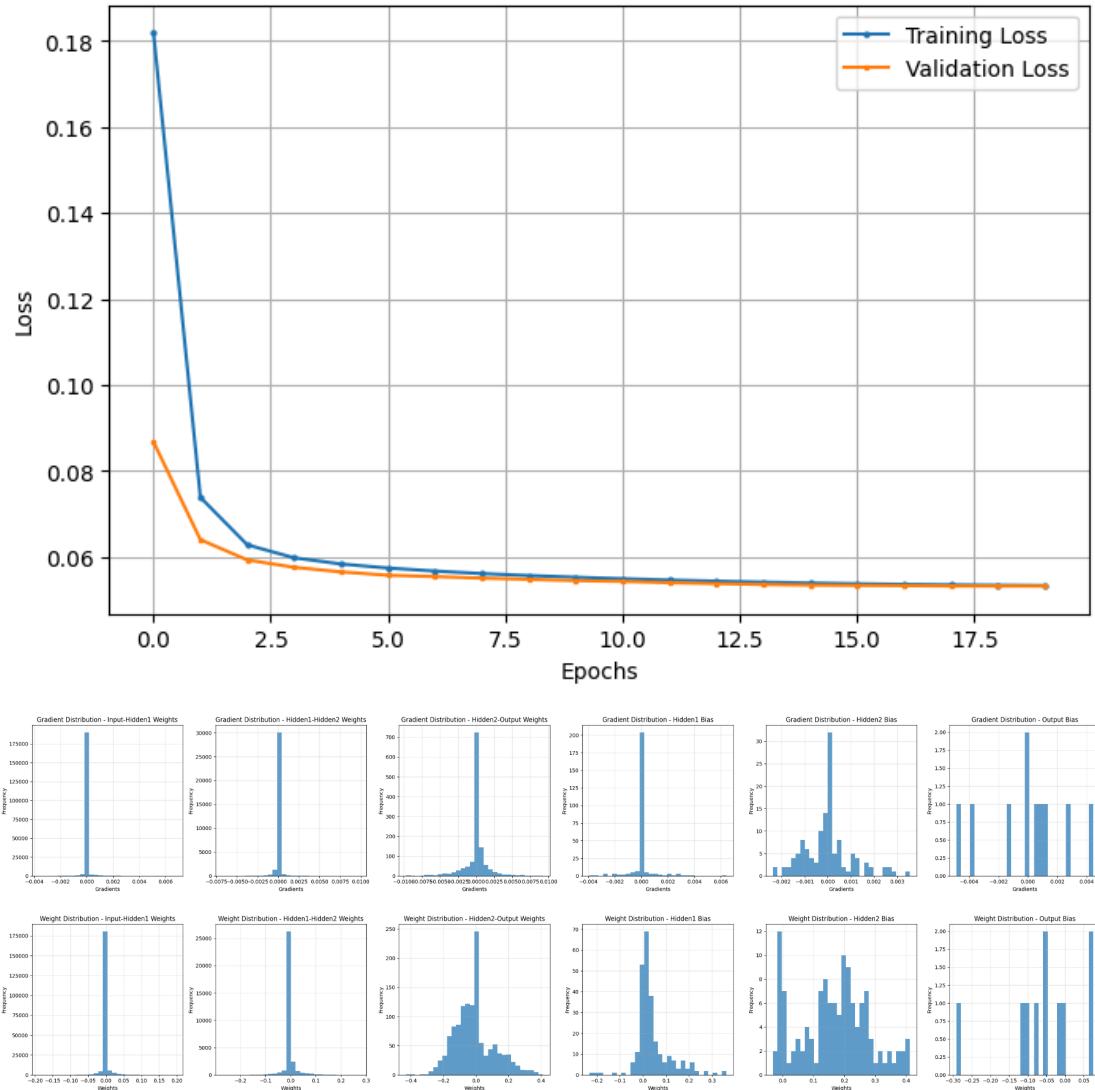
Kami menggunakan FFNN yang diinstansiasikan dengan parameter berikut

Parameter	Nilai
Input nodes	784
Susunan Neuron	256, 128
Output nodes	10
Hidden activation	ReLU
Output activation	Softmax
Loss function	Categorical Cross-Entropy (CCE)
Random seed	69420
Epochs	20

- Learning Rate 0.001

Parameter	Nilai
Akurasi	93.87%

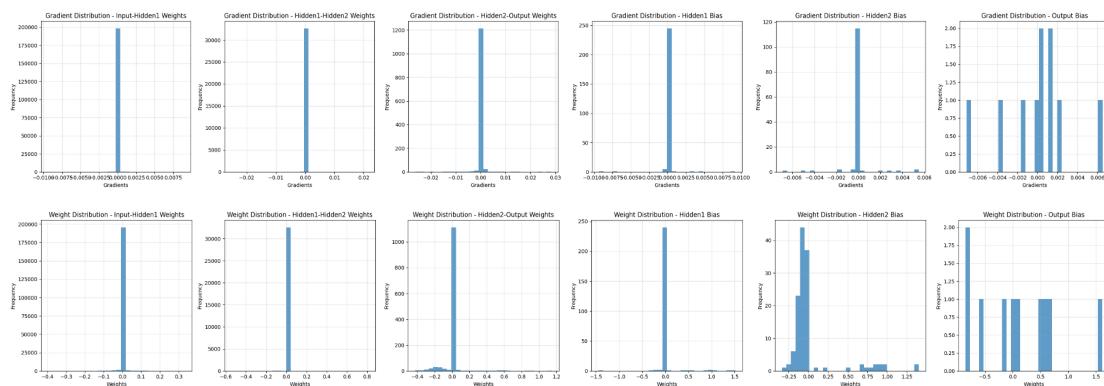
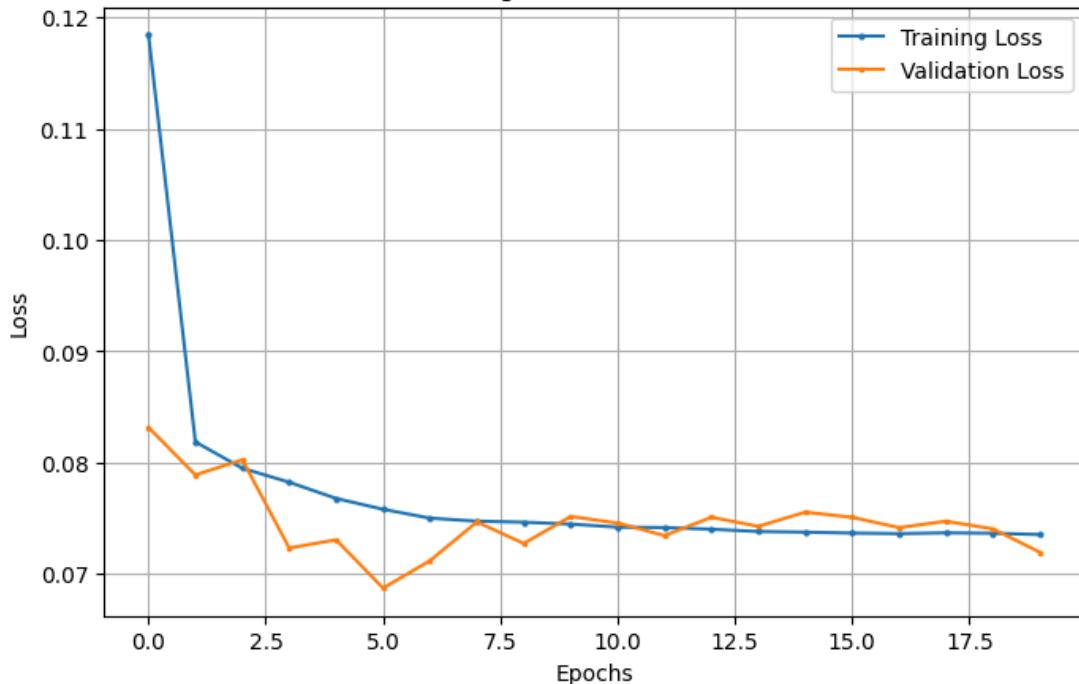
Training and Validation Loss



- Learning Rate 0.01

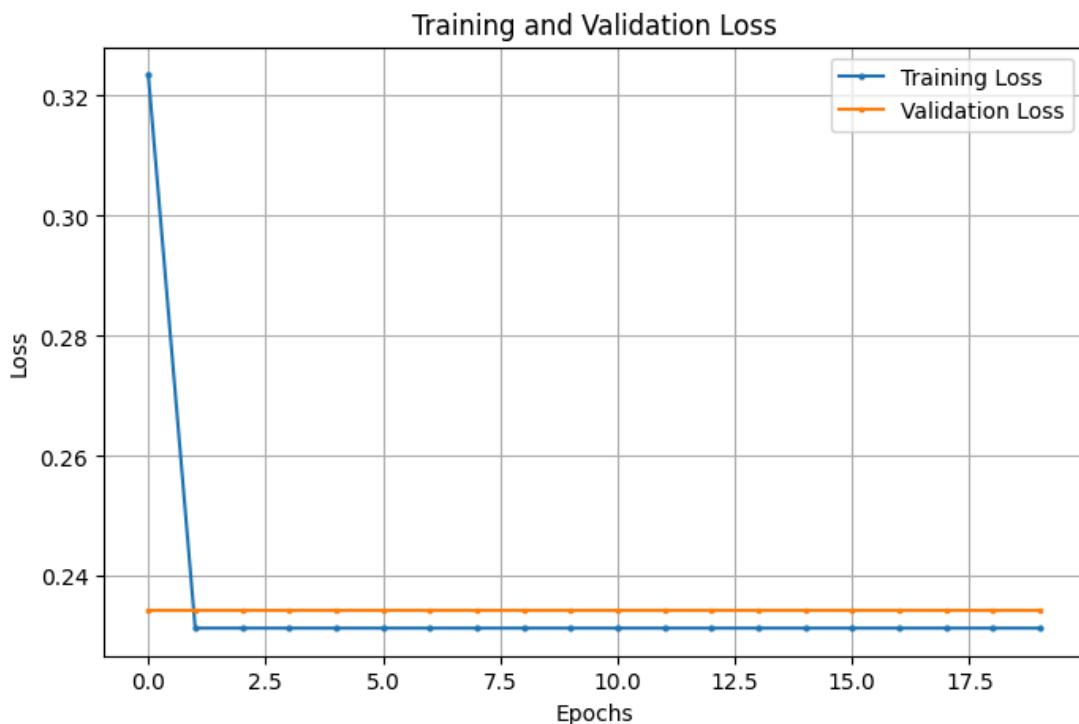
Parameter	Nilai
Akurasi	87.67%

Training and Validation Loss



- Learning Rate 0.1

Parameter	Nilai
Akurasi	8.90%



Distribusi bobot dan gradien bobot tidak bisa ditampilkan karena akurasi yang sangat kecil.

Learning rate 0.001 (learning rate rendah) menghasilkan pembaruan bobot yang sangat kecil, menyebabkan konvergensi lambat namun memungkinkan penyesuaian halus dan lebih stabil, cocok untuk dataset kompleks atau arsitektur jaringan yang rumit. Learning rate 0.01 (learning rate sedang) memberikan keseimbangan antara kecepatan konvergensi dan stabilitas, memungkinkan model belajar dengan lebih responsif dibandingkan 0.001 sambil mengurangi risiko osilasi atau divergensi yang signifikan. Learning rate .0.1 (learning rate tinggi) menghasilkan pembaruan bobot besar yang dapat mempercepat konvergensi awal, namun berisiko melewati titik optimal, menyebabkan osilasi di sekitar minimum global, ketidakstabilan proses pelatihan, atau bahkan divergensi total, sehingga memerlukan pemantauan ketat dan mungkin tidak cocok untuk sebagian besar arsitektur jaringan saraf.

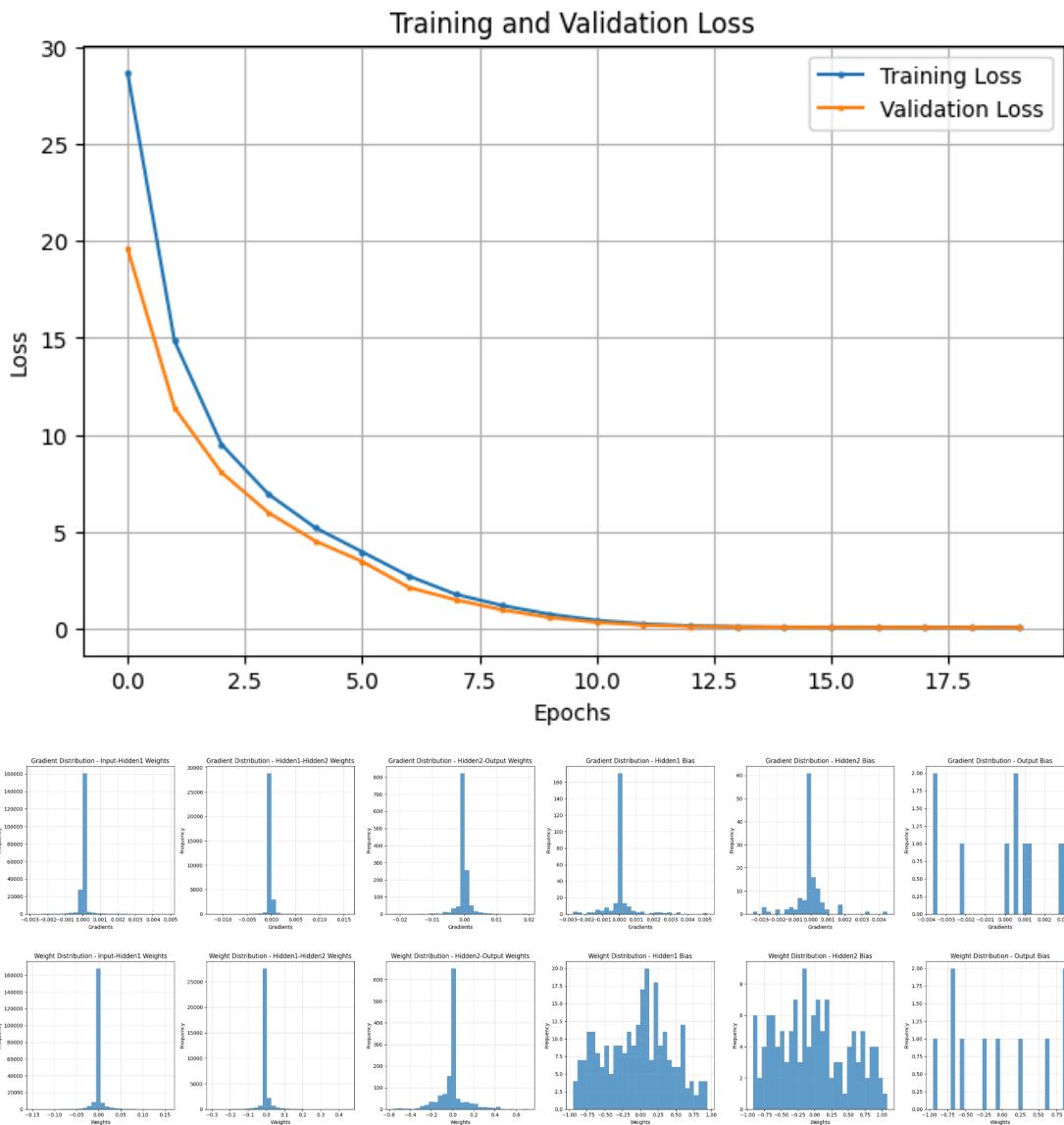
3.4 Pengaruh Inisialisasi Bobot

Kami menggunakan FFNN yang diinstansiasikan dengan parameter berikut

Parameter	Nilai
Input nodes	784
Susunan Neuron	256, 128
Output nodes	10
Learning Rate	0.001
Hidden activation	ReLU
Output activation	Softmax
Loss function	Categorical Cross-Entropy (CCE)
Random seed	69420
Epochs	20

- Inisialisasi Uniform

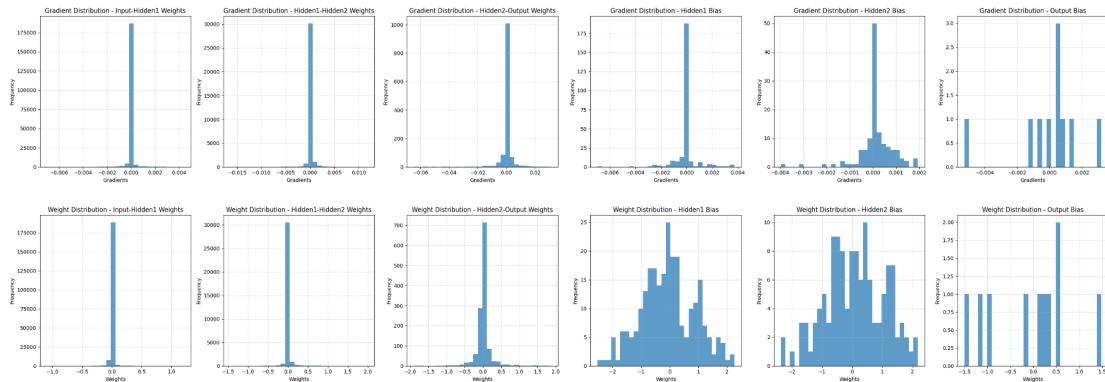
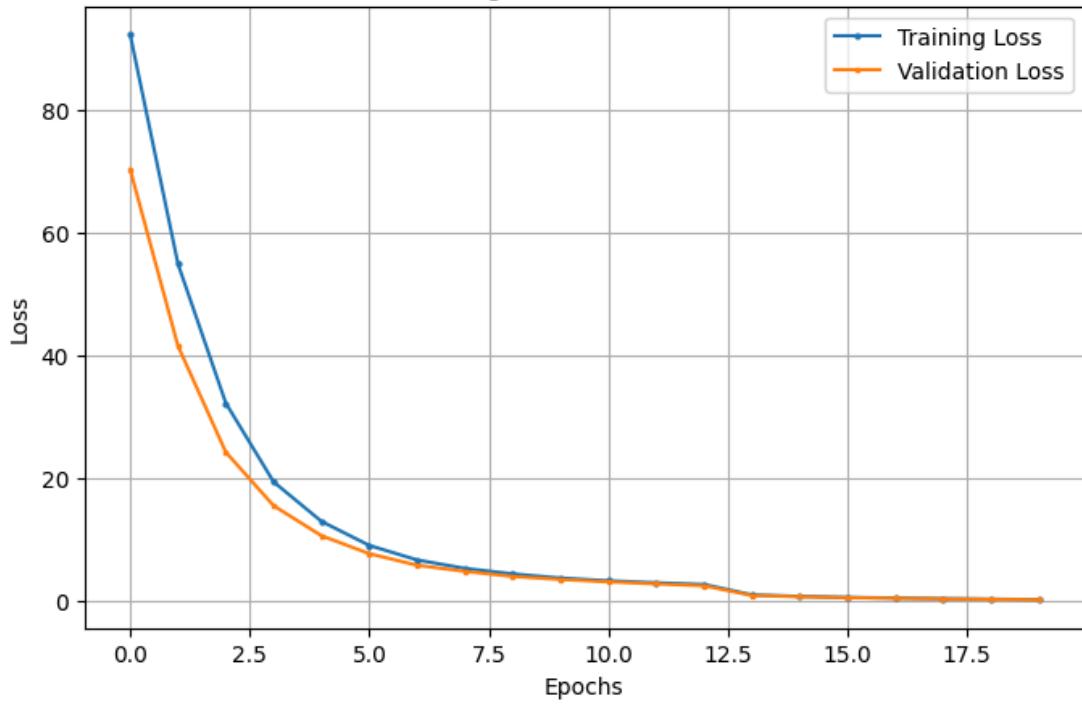
Parameter	Nilai
Akurasi	93.53%



- Inisialisasi Normal

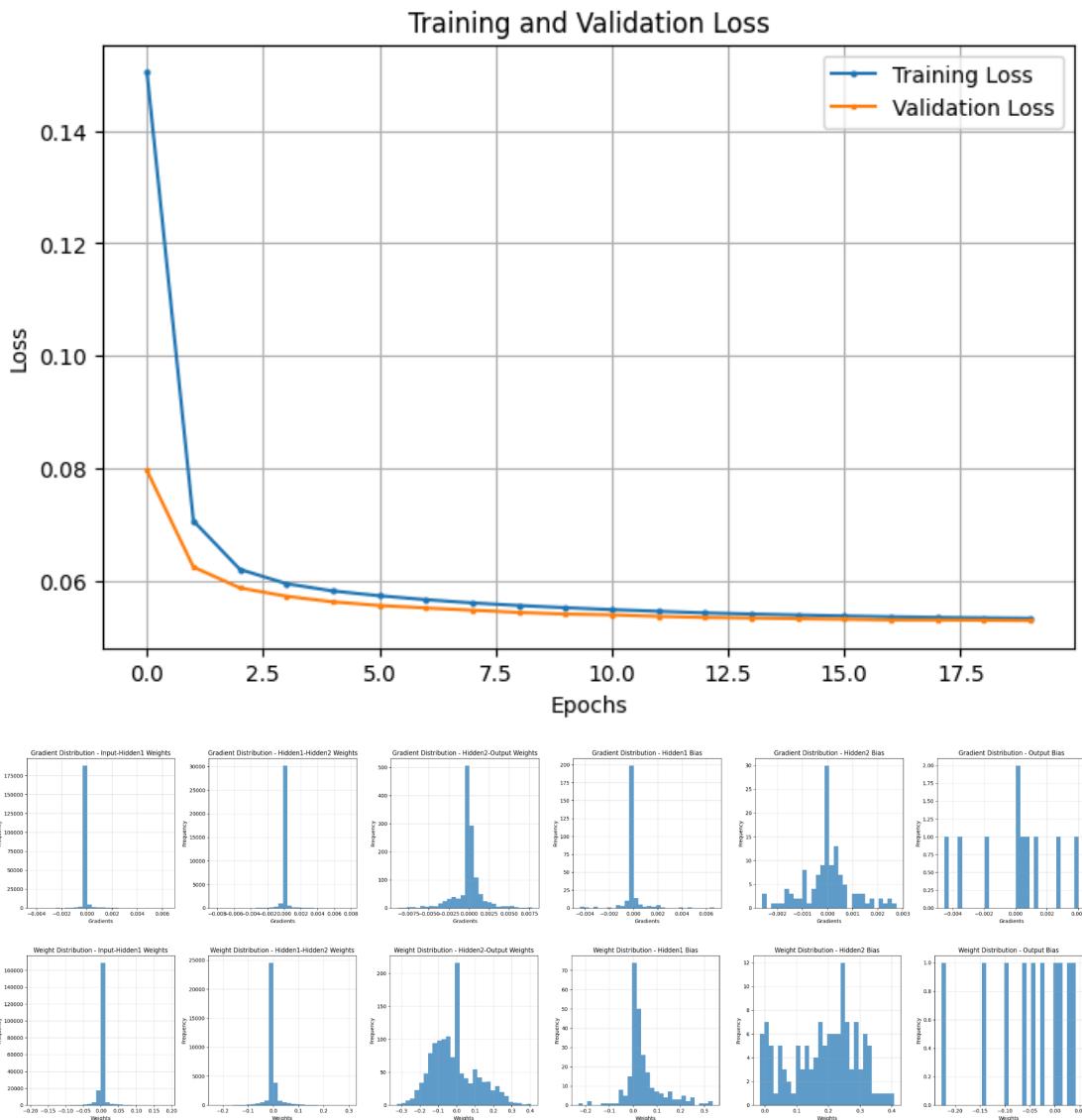
Parameter	Nilai
Akurasi	90.20%

Training and Validation Loss



- Inisialisasi Xavier

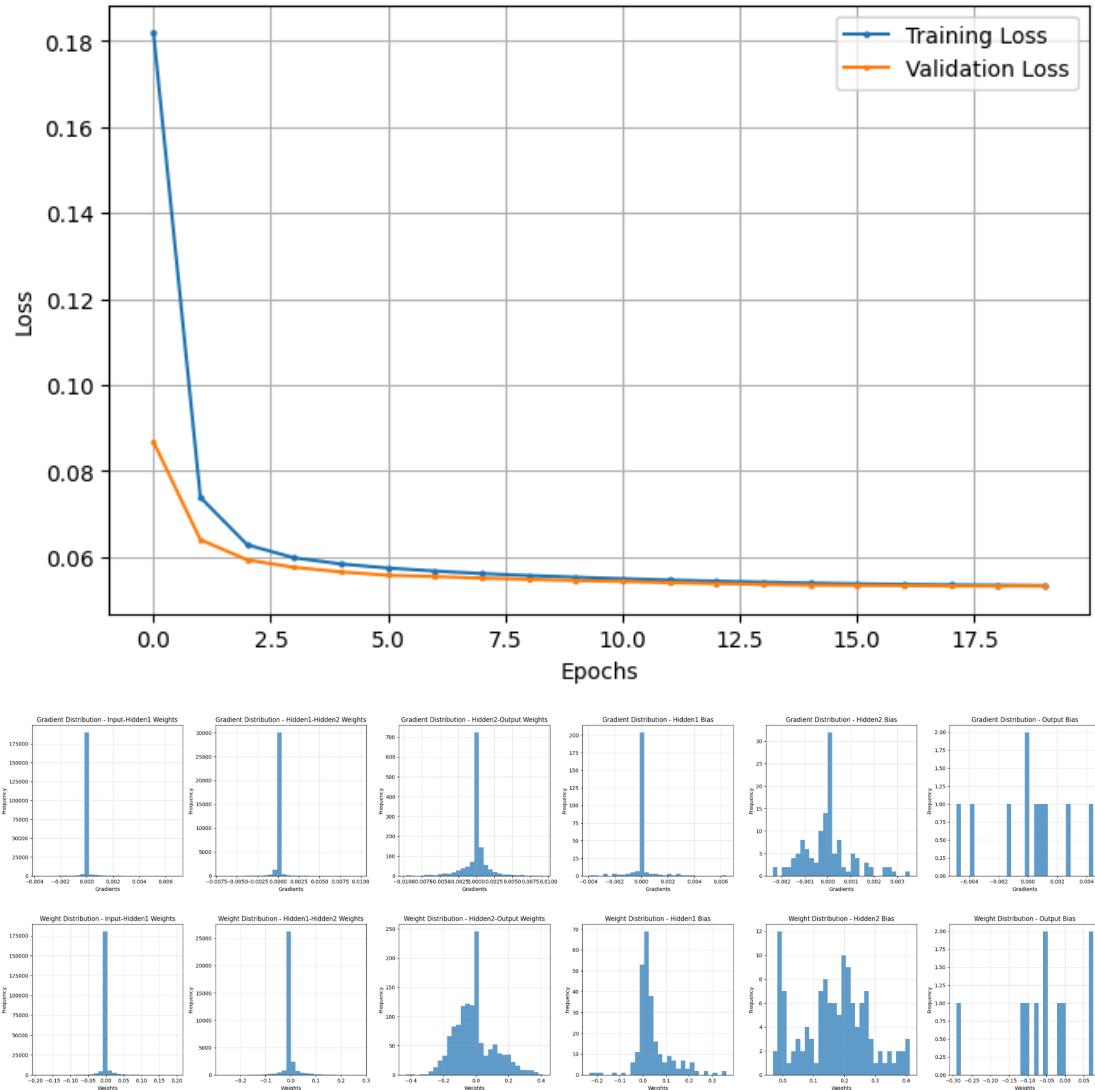
Parameter	Nilai
Akurasi	94.00%



- Inisialisasi He

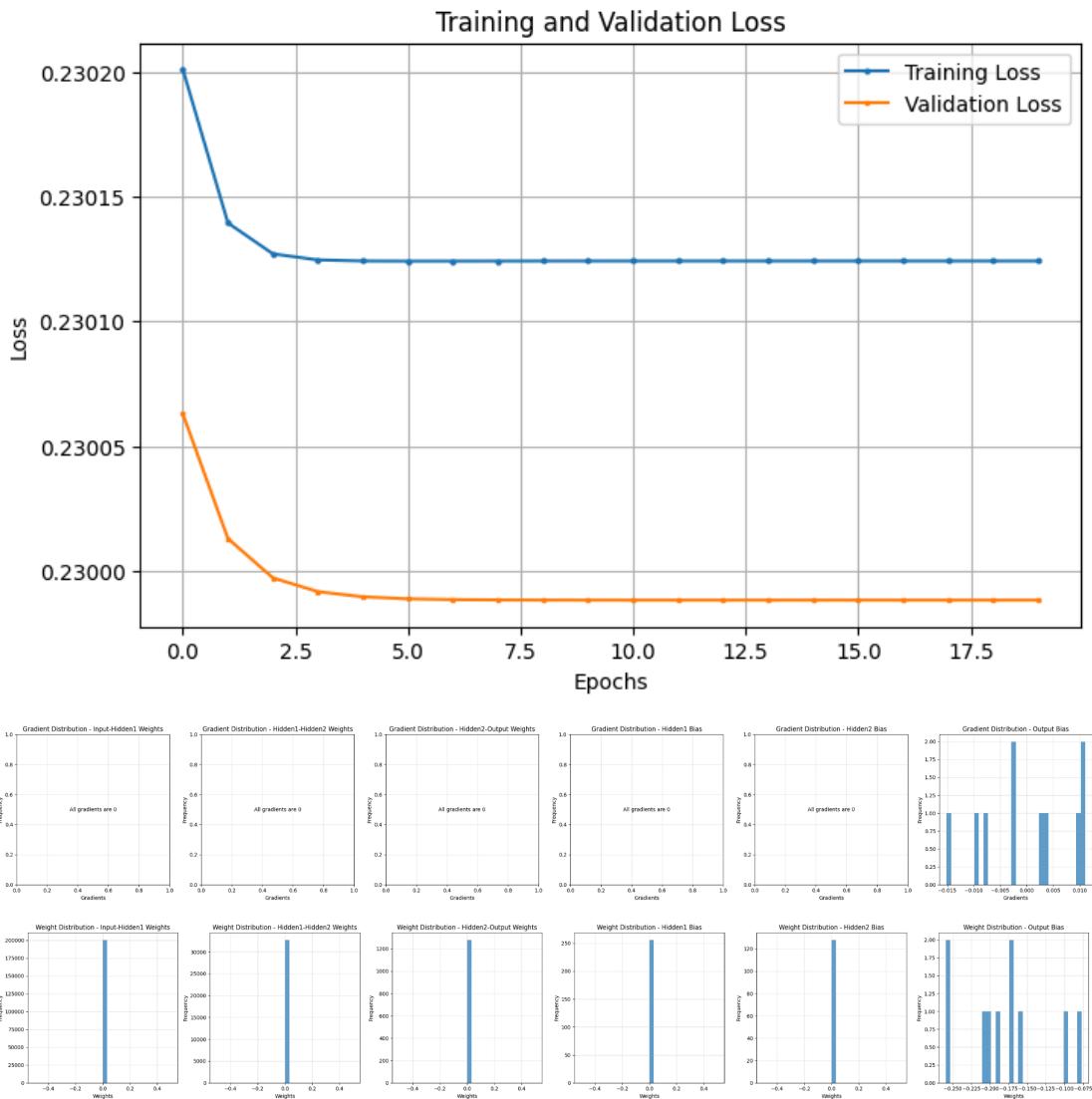
Parameter	Nilai
Akurasi	93.87%

Training and Validation Loss



- Inisialisasi Nol

Parameter	Nilai
Akurasi	10.47%



Inisiasi uniform mendistribusikan bobot secara merata dalam rentang tertentu, memberikan kesempatan awal yang sama untuk setiap koneksi, namun dapat kurang efektif untuk jaringan yang sangat dalam. Inisiasi normal menggunakan distribusi Gaussian dengan rata-rata nol dan deviasi standar tertentu, membantu mengurangi risiko saturasi dan memberikan variasi bobot yang lebih realistik. Inisiasi Xavier (Glorot) dirancang khusus untuk menjaga varians sinyal tetap konstan antara layer, sangat efektif untuk jaringan dengan fungsi aktivasi sigmoid atau tanh, membantu mencegah masalah vanishing atau exploding gradient. Inisiasi He, dikembangkan untuk jaringan dengan fungsi aktivasi ReLU, mempertimbangkan jumlah input pada neuron,

menghasilkan bobot awal yang lebih besar untuk mencegah saturasi pada jaringan dengan ReLU. Inisiasi zero, di mana semua bobot diatur ke nol, umumnya tidak direkomendasikan karena menyebabkan semua neuron berperilaku identik, menghambat kemampuan jaringan untuk belajar dan membedakan fitur.

3.5 Pengaruh Regularisasi

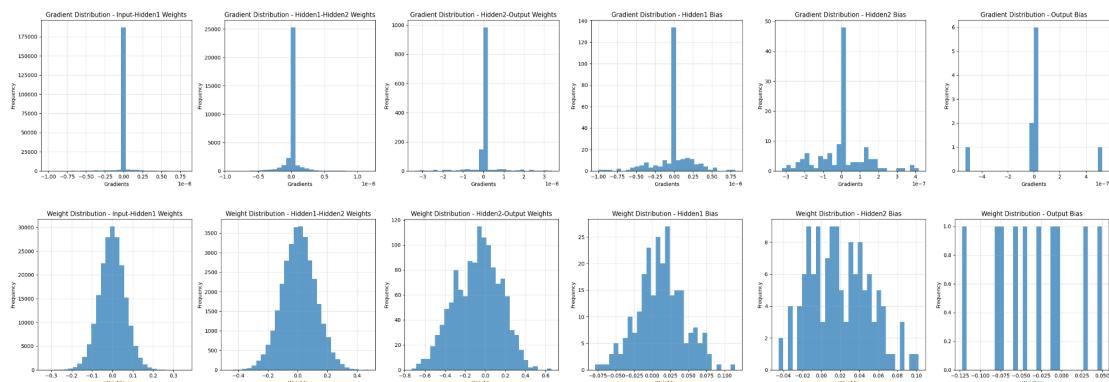
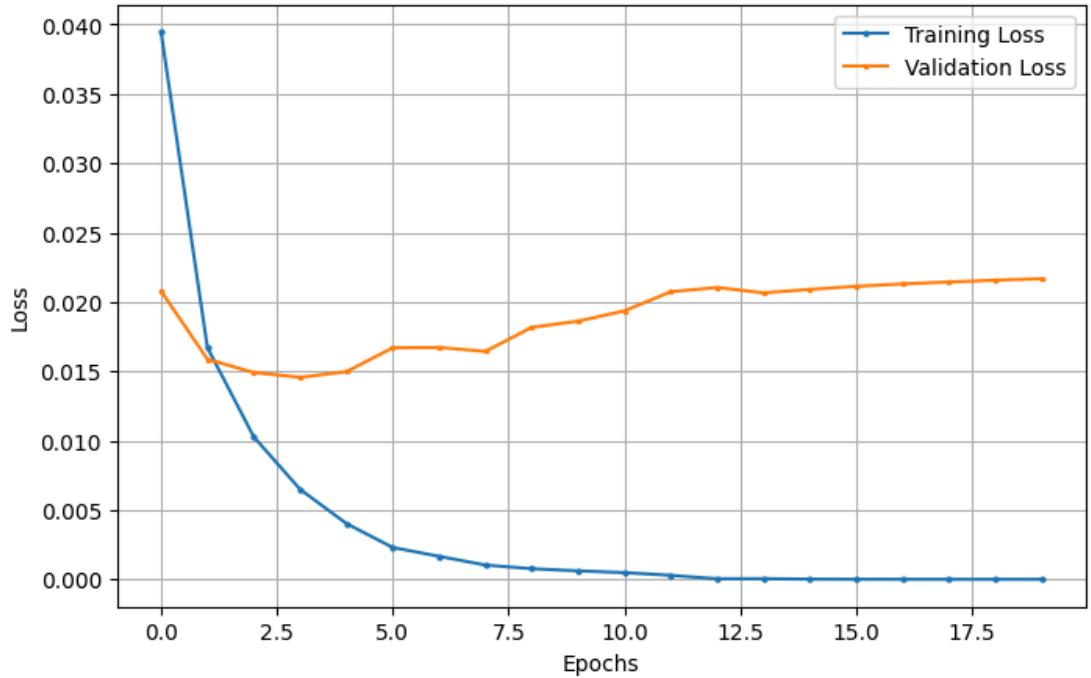
Kami menggunakan FFNN yang diinstansiasikan dengan parameter berikut

Parameter	Nilai
Input nodes	784
Susunan Neuron	256, 128
Output nodes	10
Learning Rate	0.001
Hidden activation	ReLU
Output activation	Softmax
Loss function	Categorical Cross-Entropy (CCE)
Random seed	69420
Epochs	20

- Tanpa Regularisasi

Parameter	Nilai
Akurasi	96.47%

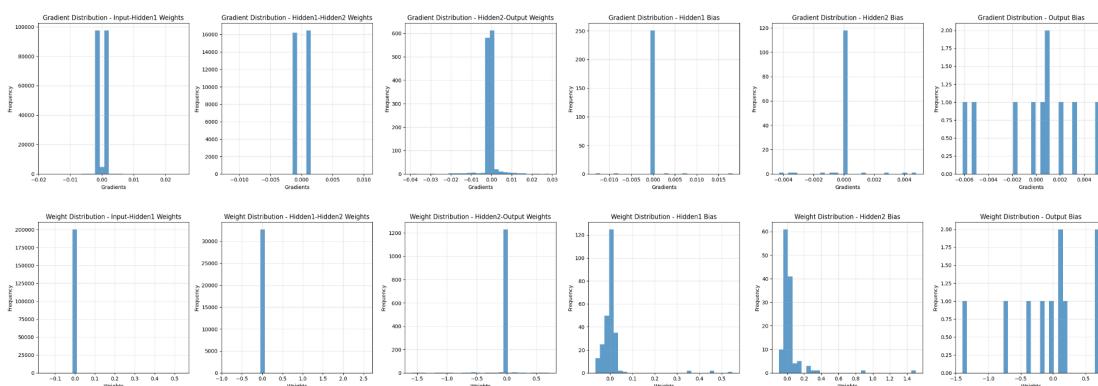
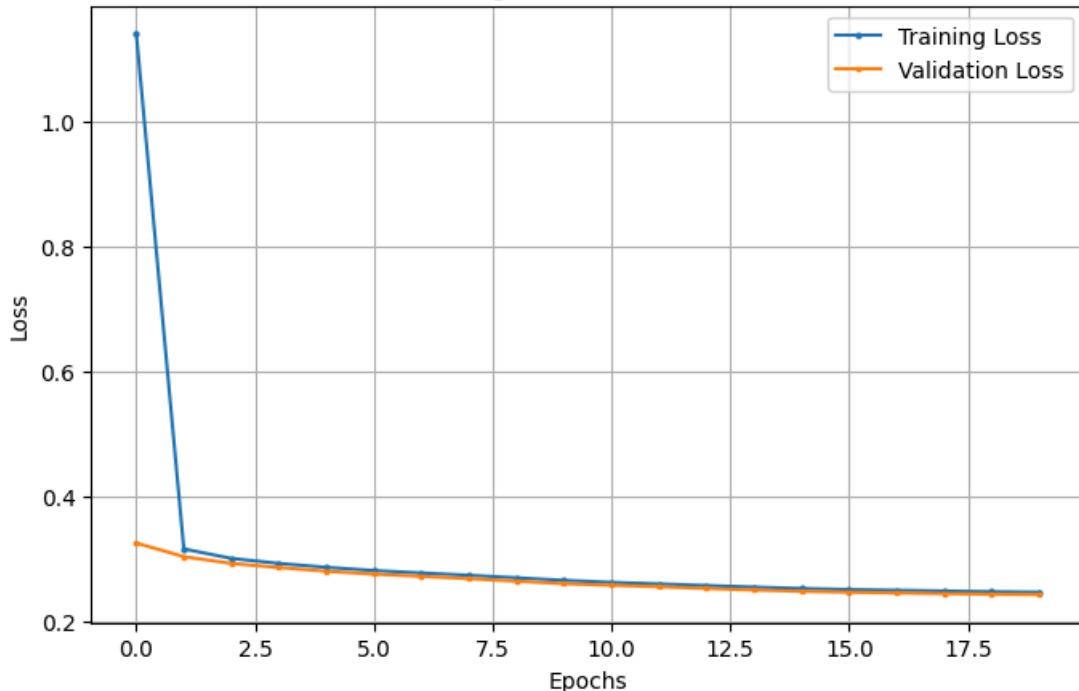
Training and Validation Loss



- Regularisasi L1

Parameter	Nilai
Akurasi	82.53%

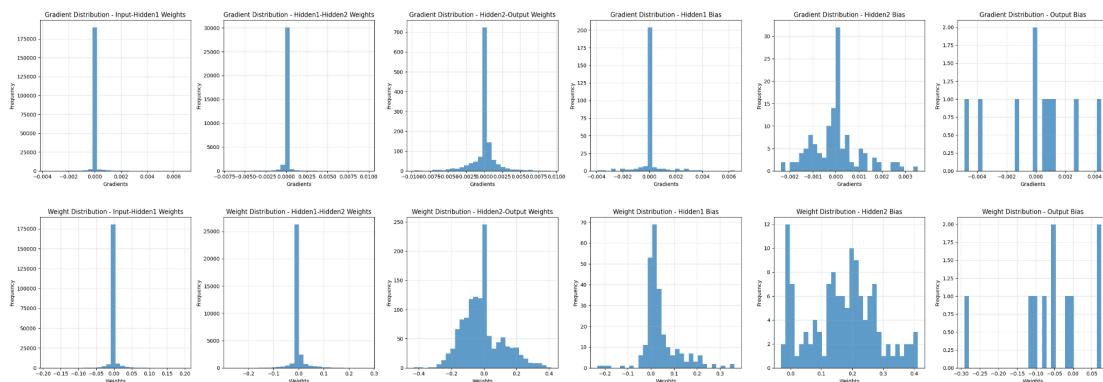
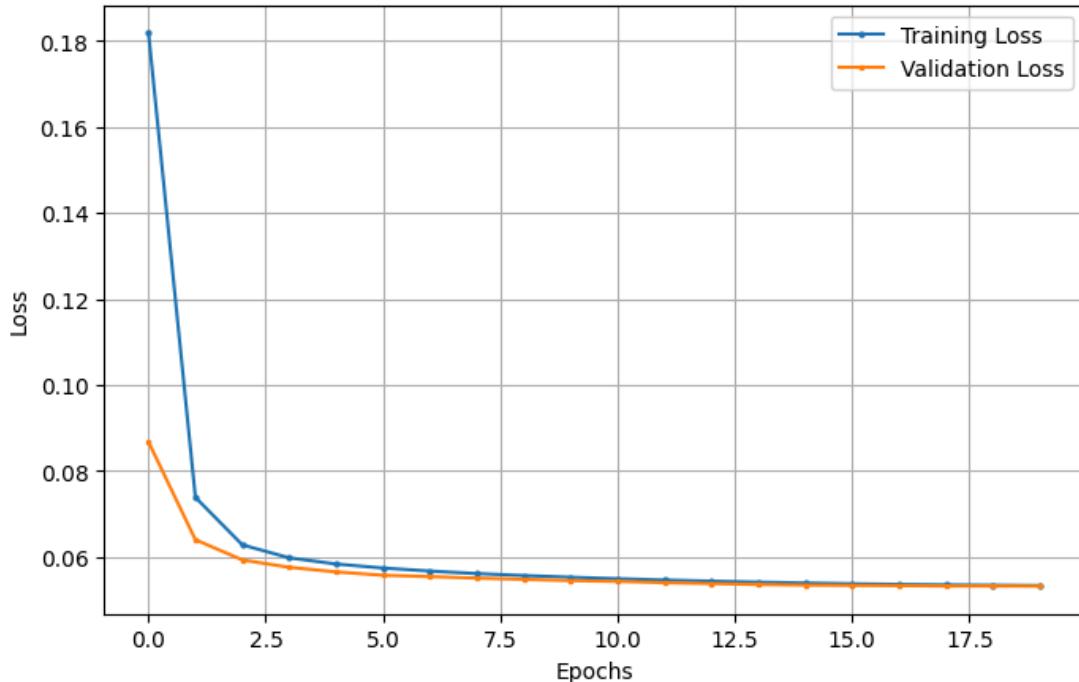
Training and Validation Loss



- Regularisasi L2

Parameter	Nilai
Akurasi	93.87%

Training and Validation Loss



Tanpa regularisasi, jaringan rentan terhadap overfitting, di mana model dapat mempelajari detail tidak penting atau noise dalam data pelatihan, menghasilkan performa buruk pada data baru. Regularisasi L1 (Lasso) menambahkan penalti berdasarkan nilai absolut bobot, mendorong model ke arah sparse solution dengan beberapa bobot tepat menjadi nol, efektif untuk seleksi fitur dan mengurangi kompleksitas model. Regularisasi L2 (Ridge) menambahkan penalti berdasarkan kuadrat bobot, mendistribusikan bobot secara merata di seluruh fitur, mencegah overfitting dengan membatasi kompleksitas model tanpa menghapus bobot secara total, sehingga lebih cocok

untuk situasi di mana semua fitur dianggap penting dan mempertahankan struktur model yang lebih halus.

3.6 Pengaruh Normalisasi RMSNorm

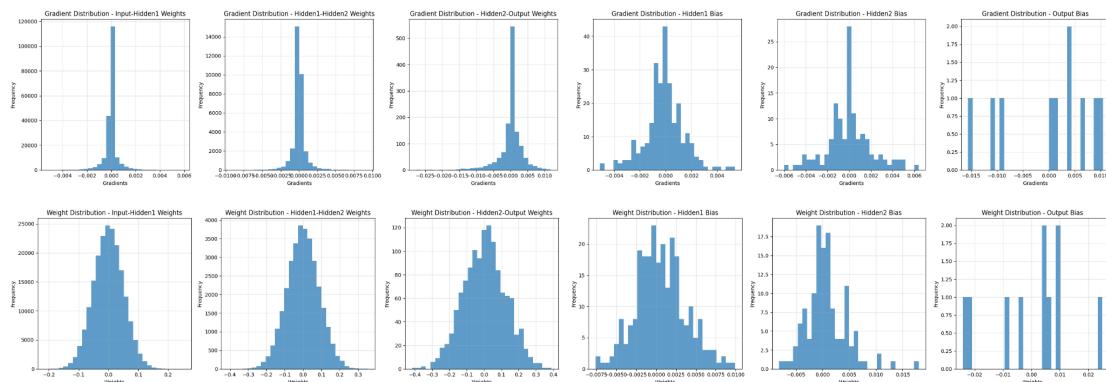
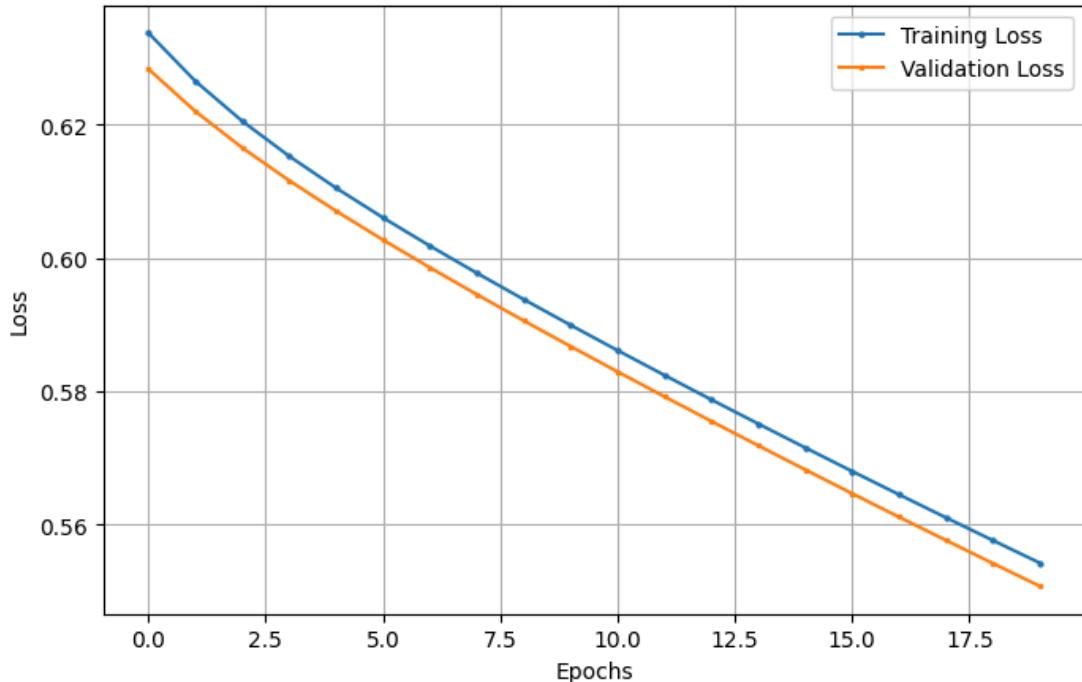
Kami menggunakan FFNN yang diinstansiasikan dengan parameter berikut

Parameter	Nilai
Input nodes	784
Susunan Neuron	256, 128
Output nodes	10
Learning Rate	0.001
Hidden activation	ReLU
Output activation	Softmax
Loss function	Categorical Cross-Entropy (CCE)
Random seed	69420
Epochs	20

- Tanpa Normalisasi

Parameter	Nilai
Akurasi	64.80%

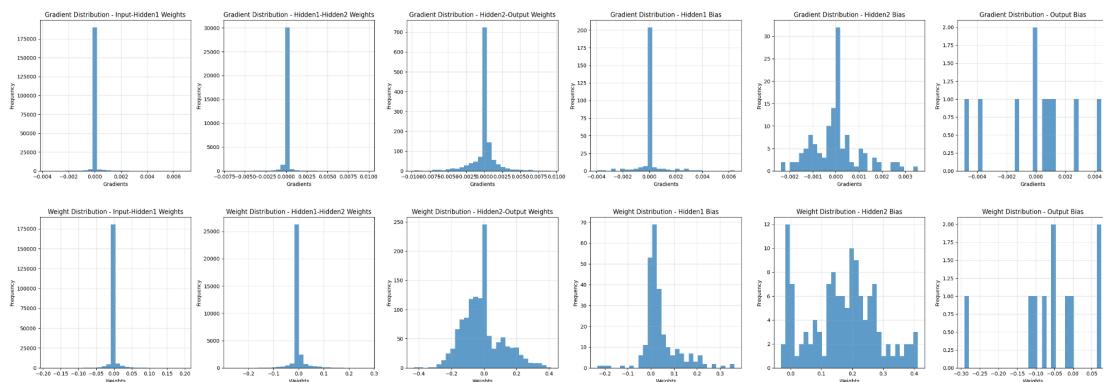
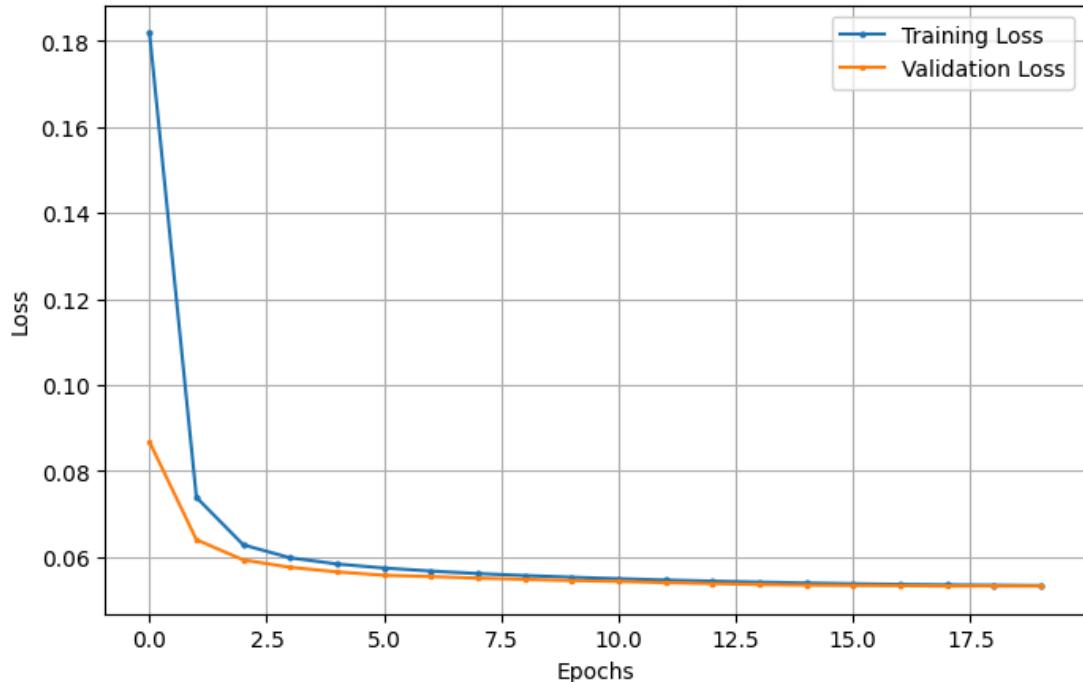
Training and Validation Loss



- Dengan Normalisasi

Parameter	Nilai
Akurasi	93.87%

Training and Validation Loss



Tanpa normalisasi, fitur dengan skala yang berbeda dapat mendominasi proses optimasi, menyebabkan fitur dengan rentang nilai besar memiliki pengaruh berlebihan terhadap bobot jaringan, mengakibatkan konvergensi lambat, sensitivitas tinggi terhadap outlier, dan potensi bias dalam representasi data. Sebaliknya, dengan normalisasi (seperti min-max scaling atau standarisasi), semua fitur dibawa ke rentang yang serupa (umumnya antara 0-1 atau memiliki mean 0 dan standar deviasi 1), yang memungkinkan algoritma optimasi bekerja lebih efisien, mempercepat konvergensi, mengurangi pengaruh outlier, meningkatkan stabilitas proses pelatihan, dan

memungkinkan jaringan untuk fokus pada pola intrinsik dalam data tanpa terganggu oleh perbedaan skala fitur yang ekstrem.

3.7 Perbandingan dengan Sklearn

Kami menggunakan FFNN yang diinstansiasikan dengan parameter berikut

Parameter	Nilai
Input nodes	784
Susunan Neuron	256, 128
Output nodes	10
Learning rate	0.001
Hidden activation	ReLU
Output activation	Softmax
Loss function	Categorical Cross-Entropy (CCE)
Random seed	69420
Epochs	20

- FFNN Buatan

Parameter	Nilai
Akurasi	93.87%

- Library scikit-learn

Parameter	Nilai
Akurasi	93.97%

BAB IV

KESIMPULAN & SARAN

Kesimpulan

Tugas Besar 1 IF3270 Pembelajaran Mesin ini berhasil mengimplementasikan Feed Forward Neural Network (FFNN) dari nol (*from scratch*) sesuai dengan spesifikasi yang diberikan. Implementasikan mencakup fungsi-fungsi utama seperti *forward propagation*, *backward propagation* dengan *automatic differentiation*, pembaruan bobot menggunakan *gradient descent*, serta berbagai metode inisialisasi bobot, fungsi aktivasi, dan fungsi loss. Pengujian dilakukan menggunakan dataset MNIST 784 untuk menganalisis pengaruh berbagai hyperparameter terhadap performa model, dengan hasil sebagai berikut:

1. **Pengaruh Depth dan Width:** Peningkatan depth (jumlah lapisan) dari 3 ke 5 meningkatkan akurasi dari 93.87% menjadi 94.33%, menunjukkan kemampuan model untuk menangkap representasi yang lebih hierarkis. Sebaliknya, variasi width (jumlah neuron per lapisan) tidak memberikan peningkatan signifikan (93.80% untuk 512-256 dan 93.83% untuk 128-64), mengindikasikan bahwa penambahan neuron lebih cocok untuk data dengan variasi fitur luas namun kurang efektif untuk kompleksitas hierarkis.
2. **Pengaruh Fungsi Aktivasi Hidden Layer:** Fungsi aktivasi ReLU mencapai akurasi tertinggi (93.87%) berkat kemampuannya mengatasi vanishing gradient dan mempercepat konvergensi. Leaky ReLU (93.70%) dan ELU (92.63%) juga menunjukkan performa baik dengan keunggulan tambahan pada stabilitas neuron. Sigmoid (80.07%) memiliki akurasi terendah karena rentan terhadap vanishing gradient, sedangkan Linear (89.67%) kurang efektif untuk pola non-linear kompleks.
3. **Pengaruh Learning Rate:** Learning rate 0.001 menghasilkan akurasi optimal (93.87%) dengan konvergensi stabil, sedangkan learning rate 0.01 (87.67%) menunjukkan penurunan akurasi akibat osilasi, dan 0.1 (8.90%) menyebabkan divergensi total karena pembaruan bobot yang terlalu besar.
4. **Pengaruh Inisialisasi Bobot:** Inisialisasi Xavier menghasilkan akurasi tertinggi (94.00%) berkat kemampuannya menjaga varians sinyal, diikuti oleh He (93.87%) yang cocok untuk ReLU. Uniform (93.53%) dan Normal (90.20%) memberikan hasil yang baik namun kurang optimal untuk jaringan dalam, sementara inisialisasi nol (10.47%) gagal karena menghambat pembelajaran.

5. **Pengaruh Regularisasi:** Tanpa regularisasi, akurasi mencapai 96.47%, tetapi rentan overfitting. L2 regularization (93.87%) efektif menyeimbangkan kompleksitas dan generalisasi, sedangkan L1 (82.53%) kurang optimal karena mendorong sparsity berlebihan.
6. **Pengaruh Normalisasi RMSNorm:** Dengan RMSNorm, akurasi meningkat signifikan dari 64.80% (tanpa normalisasi) menjadi 93.87%, menunjukkan peran penting normalisasi dalam mempercepat konvergensi dan menstabilkan pelatihan.
7. **Perbandingan dengan Scikit-learn:** FFNN buatan mencapai akurasi 93.87%, sangat mendekati MLPClassifier scikit-learn (93.97%), membuktikan bahwa implementasi dari nol ini memiliki performa yang kompetitif dengan library standar.

Secara keseluruhan, implementasi FFNN ini berhasil memenuhi spesifikasi tugas, memberikan wawasan mendalam tentang dinamika deep learning, dan menunjukkan bahwa pemilihan hyperparameter yang tepat sangat menentukan performa model. Penggunaan automatic differentiation, regularisasi, dan normalisasi RMSNorm turut meningkatkan efisiensi dan stabilitas model, menjadikannya solusi robust untuk tugas klasifikasi seperti MNIST.

Saran

Untuk pengembangan lebih lanjut, disarankan untuk mengintegrasikan teknik optimasi lanjutan seperti Adam atau RMSProp, menambah fungsi aktivasi seperti Swish, serta melakukan pengujian pada dataset yang lebih beragam untuk menguji skalabilitas model. Selain itu, optimalisasi konsumsi memori pada automatic differentiation dapat meningkatkan efisiensi komputasi pada jaringan yang lebih besar.

BAB V

PEMBAGIAN TUGAS

Tabel 9. Project Assignments Table

Project Assignments		
👤 Name	NIM	ΤΤ Contributions
Rizqika Mulia Pratama	13522126...	<ul style="list-style-type: none">- Backpropagation with Automatic Differentiation- Perubahan parameter dan implementasinya (batch size, learning rate, jumlah epoch, verbose, fungsi loss)- Menampilkan model berupa struktur jaringan beserta bobot dan gradien dalam bentuk graf- Implementasi weight update dengan menggunakan gradient descent- Fungsi Aktivasi: Linear, ReLU, Sigmoid, Tanh- Implementasi metode regulasi L1 dan L2- Implementasi metode normalisasi RMSNorm
Attara Majesta Ayub	13522139...	<ul style="list-style-type: none">- Implementasi validation phase- Menampilkan visualisasi history dari proses pelatihan (training loss & validation loss per epoch)- Perbandingan dengan library scikit-learn MLP- Fungsi Loss: Binary Cross-Entropy (BCE), Categorical Cross-Entropy (CCE)- Implementasi metode inisialisasi tambahan: Xavier initialization (Uniform dan Normal) dan He initialization
Ikhwan Al Hakim	13522147...	<ul style="list-style-type: none">- Inisiasi bobot tiap neuron dengan zero initiation- Menampilkan distribusi bobot dari setiap layer- Menampilkan distribusi gradien bobot dari tiap layer- Fungsi Aktivasi: Softmax- Fungsi Loss: Mean Squared Error (MSE)- Implementasi 2 fungsi aktivasi tambahan

REFERENSI

- <https://www.jasonosajima.com/forwardprop>
- <https://www.jasonosajima.com/backprop>
- <https://numpy.org/doc/2.2/>
- https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
- [https://math.libretexts.org/Bookshelves/Calculus/Calculus_\(OpenStax\)/14%3A_Differentiation_of_Functions_of_Several_Variables/14.05%3A_The_Chain_Rule_for_Multivariable_Functions](https://math.libretexts.org/Bookshelves/Calculus/Calculus_(OpenStax)/14%3A_Differentiation_of_Functions_of_Several_Variables/14.05%3A_The_Chain_Rule_for_Multivariable_Functions)
- <https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>
- <https://douglasorr.github.io/2021-11-autodiff/article.html>
- https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2022/tutorials/tut01.pdf