

Tugas Besar 1 IF3270 Pembelajaran Mesin
Feedforward Neural Network



Disusun oleh Kelompok 47:

Filbert	13522021
Benardo	13522055
William Glory Henderson	13522113

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024

BAB I

DESKRIPSI MASALAH

Tugas Besar 1 IF3270 Pembelajaran Mesin bertujuan memberikan pemahaman mendalam mengenai implementasi *Feedforward Neural Network* (FFNN) secara menyeluruh dari *scratch* menggunakan bahasa *Python*. Dalam tugas ini, mahasiswa ditugaskan untuk mengembangkan sebuah modul FFNN yang fleksibel, yang mampu menerima konfigurasi jaringan dengan jumlah neuron dan layer yang bervariasi, serta dapat menggunakan berbagai fungsi aktivasi seperti *Linear*, *ReLU*, *Sigmoid*, *tanh*, dan *Softmax*. Selain itu, modul tersebut harus dapat menangani berbagai fungsi loss, seperti *Mean Squared Error*, *Binary Cross-Entropy*, dan *Categorical Cross-Entropy*, serta menerapkan metode inisialisasi bobot yang beragam seperti *zero initialization*, distribusi uniform, dan distribusi normal dengan parameter yang dapat diatur untuk *reproducibility*. Model yang dihasilkan juga diwajibkan untuk menyimpan bobot dan gradien tiap neuron, menyediakan visualisasi struktur jaringan dan distribusi bobot/gradien, serta menerapkan mekanisme update bobot menggunakan gradient descent. Eksperimen harus dilakukan untuk menganalisis pengaruh hyperparameter seperti *depth*, *width*, fungsi aktivasi, *learning rate*, metode inisialisasi bobot, dan penerapan regularisasi (L1 dan L2) serta normalisasi (misalnya *RMSNorm*) terhadap performa model, termasuk perbandingan hasil prediksi dengan implementasi dari library seperti *sklearn MLPClassifier*. Dengan demikian, tugas ini tidak hanya menuntut pemahaman teoritis mengenai jaringan syaraf tiruan, tetapi juga kemampuan praktis dalam mengimplementasikan dan menganalisis model FFNN secara komprehensif.

BAB II

PEMBAHASAN

2.1 Penjelasan Implementasi

2.1.1 Deskripsi kelas beserta deskripsi atribut dan methodnya

1. Kelas ffnn

Kelas ini berisi tentang struktur dan cara kerja algoritma *feed forward neural network* yang meliputi *forward propagation* dan *backward propagation*. Kelas ini dapat memakai berbagai fungsi aktivasi, metode inisialisasi bobot, regularisasi (L1 atau L2) serta optional penerapan *RMSNorm*. Berikut adalah rincian atribut dan method yang terdapat pada kelas ini:

Atribut:

- ***num_layers***: banyaknya layer
- ***layers***: Daftar yang menentukan jumlah neuron pada setiap layer (misalnya: [784, 128, 10]).
- ***activations***: Daftar fungsi aktivasi yang digunakan di tiap layer (kecuali input layer), seperti *sigmoid*, *relu*, dll.
- ***loss_func***: Fungsi *loss* yang digunakan untuk menghitung kesalahan prediksi.
- ***loss_grad***: Turunan dari fungsi *loss* untuk perhitungan gradien.
- ***init_method***: Fungsi yang digunakan untuk menginisialisasi bobot jaringan.
- ***init_params***: Parameter tambahan untuk metode inisialisasi bobot yang disediakan dalam bentuk *dictionary*.
- ***reg_type***: Jenis regularisasi yang diterapkan (misalnya 'L1' atau 'L2') atau *None* jika tidak ada.
- ***lambda_reg***: Nilai *hyperparameter* untuk regularisasi.
- ***use_rmsnorm***: *Boolean* yang menentukan apakah *RMSNorm* diterapkan pada layer tersembunyi.
- ***weights***: *List* yang menyimpan bobot untuk setiap koneksi antar neuron.

- ***biases***: *List* yang menyimpan bias untuk tiap *layer*.
- ***gradients***: *Dictionary* yang menyimpan gradien dari bobot dan bias untuk setiap *layer*.
- ***gamma* dan *gamma_grad* (opsional)**: Parameter dan gradien yang digunakan untuk *RMSNorm* pada layer tersembunyi.

Method:

- ***__init__(layers, activations, loss_func, loss_grad, init_method, init_params, reg_type, lambda_reg, use_rmsnorm)***: Konstruktor yang menginisialisasi semua *layer*, atribut, bobot, bias, dan (jika diaktifkan) parameter *RMSNorm* berdasarkan konfigurasi jaringan yang diberikan.
- ***forward(X)***: Melakukan *forward propagation* melalui jaringan dengan menghitung nilai *input* (z) dan *output* (a) untuk setiap layer, serta menerapkan *RMSNorm* jika diaktifkan.
- ***rms_norm(x, gamma, eps=1e-8)***: Menerapkan normalisasi RMS pada input yang diberikan dengan parameter gamma dan epsilon sebagai nilai penstabil.
- ***backward(X, y)***: Melakukan *backward propagation* untuk menghitung gradien *loss* terhadap bobot dan bias, termasuk propagasi melalui *RMSNorm* jika diperlukan.
- ***update_weights(learning_rate)***: Mengupdate bobot, bias, dan parameter gamma (jika *RMSNorm* aktif) berdasarkan gradien yang telah dihitung dan laju pembelajaran yang diberikan.
- ***calculate_loss(X, y)***: Menghitung nilai *loss* dari prediksi, serta menambahkan biaya regularisasi jika diterapkan.
- ***train(X_train, y_train, X_val, y_val, batch_size, epochs, learning_rate, verbose)***: Melatih jaringan dengan melakukan pembaruan bobot secara iteratif berdasarkan *batch* data, serta menyimpan *history loss* untuk data pelatihan dan validasi.

- ***display_model_graph(max_neurons_per_layer, node_size, weight_precision, min_display_value)***: Menampilkan visualisasi grafis dari struktur jaringan, termasuk representasi *neuron*, bobot, gradien, dan bias menggunakan *library networkx* dan *matplotlib*.
- ***plot_weight_and_gradient_distribution(layers_to_plot)***: Menampilkan histogram distribusi bobot dan gradien untuk *layer* yang dipilih, sehingga membantu dalam analisis pembelajaran jaringan.
- ***plot_loss_history(history)***: Menampilkan histogram distribusi *loss* pada *training* dan validasi untuk tiap *epoch*.
- ***save_model(filename)***: Menyimpan model dan seluruh *state* atributnya ke dalam file menggunakan modul *pickle*.
- ***load_model(filename)***: Memuat model yang telah disimpan dari *file* dan mengembalikan *state* atribut jaringan.

2. Fungsi aktivasi, fungsi *loss*, dan inisialisasi bobot dibuat tidak dalam bentuk kelas tetapi dalam bentuk method saja untuk setiap fungsinya. Untuk setiap algoritmanya disesuaikan dengan rumus aslinya.

```
def init_weights_zero(shape):
    return np.zeros(shape)

def init_weights_uniform(shape, lower_bound, upper_bound, seed=None):
    if seed is not None:
        np.random.seed(seed)
    return np.random.uniform(lower_bound, upper_bound, size=shape)

def init_weights_normal(shape, mean, variance, seed=None):
    if seed is not None:
        np.random.seed(seed)
    return np.random.normal(mean, np.sqrt(variance), size=shape)

def init_weights_xavier_uniform(shape, seed=None):
    if seed is not None:
        np.random.seed(seed)
    fan_in, fan_out = shape
    limit = np.sqrt(6 / (fan_in + fan_out))
    return np.random.uniform(-limit, limit, size=shape)

def init_weights_xavier_normal(shape, seed=None):
    if seed is not None:
        np.random.seed(seed)
    fan_in, fan_out = shape
    std = np.sqrt(2 / (fan_in + fan_out))
    return np.random.normal(0, std, size=shape)

def init_weights_he(shape, lower_bound, upper_bound, seed=None):
    if seed is not None:
        np.random.seed(seed)
    fan_in, _ = shape
    std = np.sqrt(2 / fan_in)
    return np.random.normal(lower_bound, std, size=shape)

def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def d_mse_loss(y_true, y_pred):
    N = y_true.size
    return 2 * (y_pred - y_true) / N

def binary_cross_entropy_loss(y_true, y_pred, eps=1e-15):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    loss = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
    return loss

def d_binary_cross_entropy_loss(y_true, y_pred, eps=1e-15):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    size = y_true.size
    return -(y_true / y_pred - (1 - y_true) / (1 - y_pred)) / size

def categorical_cross_entropy_loss(y_true, y_pred, eps=1e-15):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    loss = -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
    return loss

def d_categorical_cross_entropy_loss(y_true, y_pred, eps=1e-15):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    size = y_true.shape[0]
    return -(y_true - y_pred) / size
```

Gambar 2.1.1.2 Kode Fungsi *loss*

Gambar 2.1.1.1 Kode Fungsi Inisialisasi Bobot

```

def linear(x):
    return x

def d_linear(x):
    return np.ones_like(x)

def relu(x):
    return np.maximum(0, x)

def d_relu(x):
    return (x > 0).astype(float)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def d_sigmoid(x):
    s = sigmoid(x)
    return s * (1 - s)

def tanh(x):
    return np.tanh(x)

def d_tanh(x):
    return 1 - np.tanh(x)**2

```

```

28   def softmax(x):
29       exp_shifted = np.exp(x - np.max(x, axis=1, keepdims=True))
30       return exp_shifted / np.sum(exp_shifted, axis=1, keepdims=True)
31
32   def d_softmax(x):
33       s = softmax(x)
34       return s * (1 - s)
35
36   def leaky_relu(x, alpha=0.01):
37       return np.maximum(alpha * x, x)
38
39   def d_leaky_relu(x, alpha=0.01):
40       dx = np.ones_like(x)
41       dx[x < 0] = alpha
42       return dx
43
44   def elu(x, alpha=1.0):
45       return np.where(x > 0, x, alpha * (np.exp(x) - 1))
46
47   def d_elu(x, alpha=1.0):
48       return np.where(x > 0, 1, alpha * np.exp(x))
49

```

Gambar 2.1.1.3 Kode Fungsi Aktivasi

2.1.2 Penjelasan *Forward Propagation*

Forward propagation adalah proses mengalirkan atau meneruskan data input melalui setiap jaringan neural (*hidden layer*) untuk menghasilkan *output* akhir. Pada kelas FFNN, proses ini dimulai dengan memasukkan data *input* (X) sebagai aktivasi awal. Setiap *layer* kemudian menghitung nilai pre-aktivasi (z) dengan mengalikan aktivasi dari *layer* sebelumnya dengan bobot yang sesuai dan menambahkan bias. Berikut adalah poin-poin utama mengenai *forward propagation* dalam kelas ini:

- **Inisialisasi Input:** Proses dimulai dengan menyimpan *input* X sebagai aktivasi awal (*a_values*).
- **Perhitungan Nilai z:** Untuk setiap *layer*, nilai z dihitung sebagai perkalian *dot* antara aktivasi *layer* sebelumnya dengan bobot (W) ditambah bias (b), yaitu: $z = a \cdot W + b$.
- **Penerapan RMSNorm (opsional):** Jika opsi *RMSNorm* aktif dan *layer* tersebut merupakan *layer* tersembunyi, nilai z awal disimpan, kemudian dinormalisasi dengan menggunakan parameter gamma untuk menstabilisasi distribusi nilai.

- **Fungsi Aktivasi:** Setelah mendapatkan nilai z (atau nilai yang telah dinormalisasi), fungsi aktivasi yang telah didefinisikan (misalnya *sigmoid*, *relu*, dan lain-lain) diterapkan untuk menghasilkan *output* baru (nilai aktivasi) yang kemudian akan menjadi *input* bagi layer selanjutnya.
- **Penyimpanan Nilai:** Selama proses ini, nilai pre-aktivasi (z_values) dan nilai aktivasi (a_values) disimpan, yang kemudian dapat digunakan pada tahap *backward propagation* untuk perhitungan gradien.

Proses ini diulangi untuk setiap *layer* hingga mencapai *layer output*. Hasil akhir merupakan prediksi atau *output* dari jaringan. *Forward propagation* mengubah *input* menjadi *output* dengan melalui beberapa proses perhitungan dan setiap langkah membantu jaringan untuk belajar memodelkan hubungan antara data *input* dan *output* yang diinginkan.

2.1.3 Penjelasan *Backward Propagation*

Backward propagation atau *backpropagation* adalah proses membalikan alur data pada jaringan neural untuk mengetahui seberapa besar kontribusi setiap bobot dan bias dalam menghasilkan kesalahan (*loss*) pada output. Proses ini digunakan untuk memperbarui parameter-parameter jaringan agar hasil prediksi dapat menjadi lebih akurat. Berikut adalah poin-poin mengenai *backward propagation* dalam kelas ini:

- **Perhitungan Error Awal (Delta)**

Pertama, dilakukan *forward propagation* untuk mendapatkan prediksi *output* dari jaringan berdasarkan input. Kemudian, fungsi turunan *loss* digunakan untuk menghitung *error* awal, yang disimpan dalam variabel delta. *Error* ini mewakili perbedaan antara prediksi dan nilai sebenarnya (selisih antara output jaringan dan nilai yang seharusnya).

- **Iterasi dari Output ke Input (Belakang ke Depan)**

Proses *backward propagation* dilakukan dengan iterasi dari *layer* output ke *layer* input (dengan menggunakan loop terbalik). Di setiap layer, kita gunakan *error* (delta) yang sudah ada untuk menghitung seberapa banyak kontribusi tiap *neuron* terhadap kesalahan. Selain itu, kita juga menghitung gradien untuk bobot dan bias di layer tersebut.

- **Menggunakan Turunan Fungsi Aktivasi**

Pada setiap layer, fungsi aktivasi yang digunakan dapat berbeda-beda sesuai dengan *input* pengguna. Nilai delta dikalikan dengan turunan fungsi aktivasi yang digunakan di layer tersebut. Jika fitur *RMSNorm* diaktifkan, prosesnya akan menjadi sedikit berbeda. Turunan fungsi aktivasi dihitung berdasarkan nilai *z* yang sudah di *RMSNorm* dan gradien untuk parameter gamma dihitung serta *error* delta disesuaikan agar memperhitungkan efek normalisasi.

- **Menghitung Gradien untuk Bobot dan Bias**

Setelah mendapatkan turunan fungsi aktivasi, gradien untuk bobot dihitung dengan mengalikan transposisi dari aktivasi *layer* sebelumnya dengan nilai delta dan dibagi jumlah sampel. Gradien untuk bias dihitung dengan menjumlahkan semua nilai delta pada *layer* tersebut dan dibagi dengan jumlah sampel.

- **Penambahan Regularisasi (Opsiional) dan Perlakuan Khusus untuk *RMSNorm* (Opsiional)**

Jika regularisasi (L1 atau L2) digunakan, gradien bobot ditambahkan dengan nilai penalti yang sesuai untuk mengubah bobot yang terlalu besar sehingga membantu mencegah *overfitting*. Sementara itu, jika *RMSNorm* aktif di *hidden layer*, perhitungan turunannya menjadi sedikit berbeda. Selain menghitung gradien untuk bobot dan bias, jaringan juga menghitung gradien untuk parameter gamma yang digunakan dalam proses normalisasi.

- **Propagasi Error ke Layer Sebelumnya**

Setelah gradien untuk bobot dan bias dihitung di sebuah *layer*, *error delta* dihitung kembali untuk *layer* sebelumnya dengan mengalikan delta dengan transposisi bobot *layer* tersebut.

Proses ini terus berlanjut hingga mencapai *layer* input. *Backward propagation* membantu menentukan arah perbaikan pada setiap parameter dalam jaringan. Dengan mengetahui gradien dari setiap bobot dan bias, kita dapat mengupdate parameter tersebut secara bertahap untuk meminimalkan kesalahan prediksi jaringan.

2.2 Hasil Pengujian

2.2.1 Pengaruh Depth dan Width

Dalam konteks FFNN, *depth* dan *width* adalah dua aspek utama yang mendefinisikan arsitektur jaringan. *Depth* mengacu pada jumlah *layer* terutama jumlah *hidden layer* yang digunakan dalam model. Semakin banyak *hidden layer*, model dapat melakukan transformasi data secara bertahap dan membangun representasi hierarkis dari fitur, mulai dari fitur tingkat rendah hingga tingkat abstraksi yang lebih tinggi. Sementara itu, *width* mengacu pada jumlah neuron yang terdapat dalam satu *layer*. Lebar *layer* yang lebih besar memberikan kapasitas yang lebih tinggi untuk menangkap berbagai fitur dalam data mentah, karena lebih banyak *neuron* yang dapat memproses dan merepresentasikan variasi informasi secara paralel.

Untuk menguji pengaruh kedua *hyperparameter* tersebut, eksperimen dilakukan dengan dua pendekatan pengujian:

1. Variasi *Width* (dengan *Depth* tetap)

- Eksperimen dilakukan dengan menggunakan satu hidden layer (*depth* = 1) dengan jumlah *neuron* yang berbeda-beda di tiap pengujian, yakni 64, 128, dan 256.
- Pada tiap konfigurasi, model dilatih dengan parameter yang sama (misalnya *batch size*, *learning rate*, dan jumlah *epoch*) dan kemudian dievaluasi berdasarkan akurasi prediksi dan grafik *loss* pelatihan serta validasi.

2. Variasi *Depth* (dengan *Width* tetap)

- Di sini, eksperimen dilakukan dengan mempertahankan jumlah *neuron* per layer (misalnya *width* tetap 64) dan mengubah jumlah *hidden layer* menjadi 1, 2, dan 3 *layer*.
- Proses pelatihan dan evaluasi dilakukan serupa, sehingga perbandingan dapat dilakukan berdasarkan akurasi dan tren *loss* yang diperoleh.

Peningkatan *width* pada *hidden layer* menunjukkan dampak yang cukup jelas terhadap kemampuan representasi model. Dari hasil pengujian, ketika *width* ditetapkan sebesar 64, model memperoleh akurasi sekitar 90,77%. Dengan menaikkan *width* menjadi 128, akurasi meningkat menjadi 91,28%, dan selanjutnya pada *width* 256 akurasi mencapai 91,54%. Hal ini mengindikasikan bahwa penambahan neuron dalam satu *hidden layer* memungkinkan model untuk menangkap fitur-fitur yang lebih kompleks dan menyimpan informasi secara lebih rinci. Meskipun peningkatan akurasi tidak bersifat linear dan cenderung melambat setelah mencapai ambang tertentu, penambahan *width* tetap memberikan keuntungan (menambah tingkat akurasi) dalam hal kapasitas model, terutama dalam mengolah data mentah yang memiliki variabilitas tinggi.

Di sisi lain, peningkatan *depth* atau jumlah *hidden layer* juga memberikan kontribusi positif terhadap performa model, meskipun peningkatannya relatif lebih kecil jika dibandingkan dengan penambahan *width*. Hasil eksperimen menunjukkan bahwa model dengan 1 *hidden layer* mencapai akurasi sekitar 90,77%, yang meningkat menjadi 91,29% dengan 2 *hidden layer*, dan sedikit lagi naik menjadi 91,36% dengan 3 *hidden layer*. Dengan menambah *hidden layer*, model dapat membangun representasi hierarkis yang mengubah fitur-fitur dasar pada *layer* awal menjadi fitur yang lebih abstrak pada *layer* selanjutnya. Walaupun keuntungan yang diperoleh tidak sebesar peningkatan yang dihasilkan dari penambahan *width*, penambahan *depth* membantu model untuk mengorganisir dan memproses transformasi *non-linear* secara bertahap. Namun, perlu diingat bahwa penambahan *depth* juga menyebabkan tantangan seperti peningkatan kompleksitas komputasi dan potensi masalah *vanishing gradient*, sehingga perbaikan performa yang diperoleh cenderung lebih marginal.

Keuntungan penambahan *width* adalah peningkatan kapasitas representasi yang memungkinkan jaringan menangani data dengan variabilitas tinggi secara lebih efektif. Namun, perlu diperhatikan bahwa menambahkan *neuron* secara berlebihan dapat menyebabkan model menjadi terlalu besar, yang berpotensi meningkatkan risiko *overfitting* jika jumlah data pelatihan tidak memadai. Sementara itu, penambahan *depth* memberikan kemampuan untuk mengorganisasi informasi secara bertingkat dan memproses transformasi *non-linear* yang kompleks. Akan tetapi, peningkatan *depth* juga membawa tantangan seperti kompleksitas komputasi yang lebih tinggi dan potensi masalah *vanishing gradient*, yang dapat menghambat proses pembelajaran jika tidak ditangani dengan benar.

Dengan demikian, eksperimen yang telah dilakukan menunjukkan bahwa kedua aspek *width* dan *depth* memberikan kontribusi positif terhadap performa FFNN, namun dengan keuntungan yang berbeda. Peningkatan *width* cenderung memberikan peningkatan akurasi yang lebih signifikan karena kapasitas representasi yang lebih besar, sedangkan penambahan *depth* membantu model dalam mengabstraksi fitur secara bertahap dengan peningkatan performa yang lebih marginal. Pemilihan konfigurasi yang optimal sangat bergantung pada kompleksitas data, ukuran dataset, dan pertimbangan efisiensi komputasi, sehingga eksperimen lebih lanjut diperlukan untuk menemukan keseimbangan terbaik antara kedua *hyperparameter* tersebut.

Berikut adalah hasil perbandingan akurasi prediksi untuk pengujian variasi *width* :

```
Melatih model dengan hidden layer width: 64
Model dengan width 64 memperoleh akurasi: 0.9077
Melatih model dengan hidden layer width: 128
Model dengan width 128 memperoleh akurasi: 0.9128
Melatih model dengan hidden layer width: 256
Model dengan width 256 memperoleh akurasi: 0.9154
```

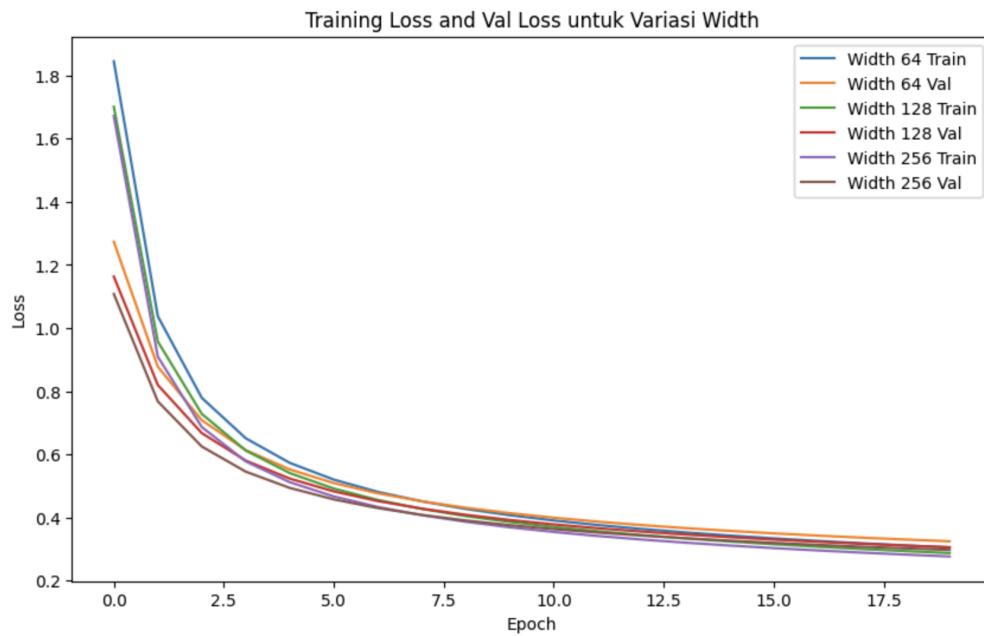
Gambar 2.2.1.1. Perbandingan Hasil Prediksi Variasi *Width*

Berikut adalah hasil perbandingan akurasi prediksi untuk pengujian variasi *depth* :

```
Melatih model dengan 1 hidden layer
Model dengan 1 hidden layer memperoleh akurasi: 0.9077
Melatih model dengan 2 hidden layer
Model dengan 2 hidden layer memperoleh akurasi: 0.9129
Melatih model dengan 3 hidden layer
Model dengan 3 hidden layer memperoleh akurasi: 0.9136
```

Gambar 2.2.1.2. Perbandingan Hasil Prediksi Variasi *Depth*

Berikut adalah hasil perbandingan *loss* untuk pengujian variasi *width* :



Gambar 2.2.1.3. Perbandingan *Loss* Pelatihan Variasi *Width*

Dari grafik perbandingan loss pada variasi width, kita dapat memperhatikan:

1. Penurunan Kurva *Loss*:

Semua kurva memiliki pola penurunan *loss* yang serupa. Pada *epoch*-*epoch* awal, *loss* turun dengan cepat seiring model mempelajari pola dasar pada data. Setelah beberapa *epoch*, laju penurunan mulai melambat, menandakan model semakin stabil dalam mempelajari parameter.

2. Dampak Perbedaan *Width*:

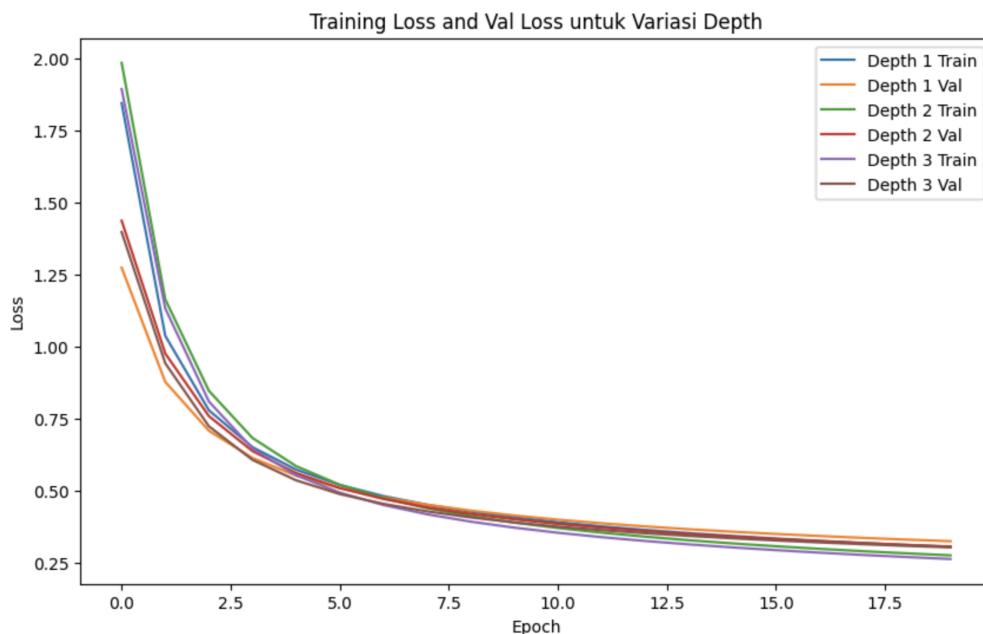
Model dengan *width* lebih besar (misalnya 256) umumnya mampu

mempelajari representasi yang lebih kompleks, sehingga cenderung memiliki kemampuan *fitting* lebih tinggi. Namun, jika data tidak terlalu kompleks atau jumlah data tidak memadai, peningkatan *width* yang terlalu besar bisa menyebabkan *overfitting* atau tidak memberi peningkatan signifikan (tergantung pada regularisasi yang digunakan). Dari plot, kita lihat bahwa semua kurva *train* dan *val loss* relatif mendekati satu sama lain di akhir pelatihan. Hal ini mengindikasikan bahwa pada rentang *width* 64, 128, hingga 256, perbedaannya tidak terlalu drastis.

3. ***Training vs Validation Loss:***

Pada sebagian besar *epoch*, *training loss* lebih tinggi dibandingkan *validation loss* pada *epoch* awal, tetapi pada *epoch* akhir lebih rendah *training loss*. Jika *validation loss* tetap selaras dan tidak meningkat tajam, berarti belum terjadi *overfitting* yang signifikan. Tidak ada perbedaan mencolok antara model dengan *width* 64, 128, maupun 256 di akhir *epoch*. Ini menunjukkan bahwa untuk dataset dan konfigurasi tertentu, peningkatan jumlah *neuron* tidak selalu memberikan peningkatan performa yang besar.

Berikut adalah hasil perbandingan loss untuk pengujian variasi depth :



Gambar 2.2.1.4. Perbandingan *Loss* Pelatihan variasi *depth*

Dari grafik perbandingan loss pada variasi depth, kita dapat memperhatikan:

1. Penurunan Kurva Loss:

Terlihat semua model (*depth* 1, 2, dan 3) menunjukkan penurunan *training* dan *validation loss* yang cukup serupa pada awal pelatihan, lalu semakin lambat mendekati *epoch* akhir. Model dengan *depth* lebih dalam mungkin lebih cepat atau lebih lambat menurun tergantung pada teknik inisialisasi dan optimisasi, namun cenderung memiliki potensi representasi lebih tinggi.

2. Dampak Penambahan Layer:

Model *Depth* 1 berarti hanya memiliki 1 *hidden layer*. Model ini cenderung lebih sederhana, sehingga kapasitas pemetaan fungsinya lebih terbatas. Model *depth* 2 atau 3 memiliki lebih banyak *layer*, sehingga dapat mempelajari representasi lebih dalam (*deep representation*) yang biasanya membantu menangkap pola yang lebih kompleks dalam data.

3. Training vs Validation Loss:

Pada sebagian besar *epoch*, *training loss* lebih tinggi dibandingkan *validation loss* pada *epoch* awal, tetapi pada *epoch* akhir lebih rendah *training loss*. Jika model *depth* 3 memiliki *training* dan *validation loss* yang lebih rendah di akhir pelatihan dibanding *depth* 1 atau 2, itu menandakan bahwa penambahan *layer* meningkatkan kapasitas model secara efektif, tanpa menimbulkan *overfitting* yang parah (selama *val loss* juga ikut turun). Sebaliknya, jika kurva *depth* 3 tidak jauh berbeda dari *depth* 2, berarti tambahan *layer* tidak memberikan keuntungan signifikan pada *dataset* tertentu.

2.2.2 Pengaruh Fungsi Aktivasi

Fungsi aktivasi dalam FFNN merupakan komponen penting yang mengubah *output* linear dari setiap *neuron* menjadi representasi non-linear. Dengan kata lain, fungsi aktivasi memperkenalkan non-linearitas ke dalam model sehingga jaringan dapat menangkap pola yang kompleks dalam data. Untuk menguji pengaruh fungsi aktivasi, dilakukan eksperimen dengan mengganti fungsi aktivasi pada *hidden layer* sambil mempertahankan parameter pelatihan lainnya (seperti *input_dim* = 784, *hidden_dim* = 64,

`output_dim = 10, epochs = 20, batch_size = 32, dan learning_rate = 0.01`). Pada pengujian tersebut, *hidden layer* menggunakan fungsi aktivasi yang berbeda, seperti *linear*, *ReLU*, *sigmoid*, *tanh*, *leaky_relu*, dan *ELU*, sedangkan *layer output* selalu menggunakan *softmax* untuk menghasilkan probabilitas kelas. Pengujian dilakukan dengan memantau akurasi prediksi pada data validasi, grafik *loss* pelatihan dan validasi, serta distribusi bobot dan gradien pada tiap *layer*.

Hasil eksperimen menunjukkan bahwa model dengan aktivasi *linear* memperoleh akurasi sekitar 90,36%, sedangkan model dengan aktivasi *ReLU* mencapai 90,77%, *leaky_relu* 90,75%, dan *ELU* 90,58%. Di sisi lain, model yang menggunakan fungsi *sigmoid* hanya mencapai akurasi sekitar 85,58%, dan *tanh* menghasilkan akurasi sekitar 89,67%. Di sisi lain, penggunaan fungsi *linear*, meskipun menghasilkan akurasi moderat sekitar 90,36%, tidak menambahkan non-linearitas pada jaringan karena jika semua *hidden layer* menggunakan fungsi linear, maka kombinasi *multi-layer* tersebut secara matematis ekuivalen dengan model *linear* tunggal (seperti regresi *linear multinomial* pada layer *softmax*), sehingga tidak mampu menangkap pola non-linear kompleks. Pada bagian *loss*, grafik yang diperoleh dari eksperimen memberikan gambaran yang jelas mengenai seberapa cepat dan stabil model melakukan konvergensi saat menggunakan fungsi aktivasi yang berbeda. Misalnya, model dengan aktivasi berbasis *ReLU* (termasuk varian *leaky_relu* dan *ELU*) menunjukkan tren penurunan *loss* yang konsisten dan stabil selama pelatihan, yang mengindikasikan bahwa aliran gradien tetap terjaga dengan baik. Hal ini disebabkan oleh fakta bahwa *ReLU* memberikan gradien konstan (nilai 1) pada input positif, sehingga tidak terjadi masalah *vanishing gradient*. Sebaliknya, model dengan fungsi *sigmoid* cenderung menunjukkan penurunan *loss* yang lebih lambat dan tidak konsisten. Hal ini dapat dijelaskan oleh sifat saturasi *sigmoid*, di mana output mendekati 0 atau 1 untuk nilai input yang ekstrim, sehingga gradien yang dihasilkan menjadi sangat kecil. Akibatnya, pembaruan bobot pun menjadi minimal, yang membuat proses konvergensi berjalan lebih lambat dan *loss* tidak turun secara signifikan.

Selain itu, distribusi bobot dan gradien pada *hidden layer* juga memberikan gambaran tambahan tentang efektivitas fungsi aktivasi. Model dengan aktivasi *ReLU* cenderung menghasilkan distribusi bobot yang seimbang dan gradien yang tersebar optimal, sehingga aliran gradien selama *backpropagation* tidak terhambat. Sebaliknya,

pada model dengan fungsi *sigmoid*, distribusi gradien lebih terkonsentrasi di sekitar nol, yang menunjukkan masalah saturasi di mana nilai output mendekati 0 atau 1 sehingga gradien menjadi sangat kecil. Hal ini mengakibatkan pembelajaran yang lambat dan *update* bobot yang minimal, yang sejalan dengan akurasi yang lebih rendah. Fungsi *tanh*, dengan rentang output antara -1 hingga 1 menunjukkan distribusi gradien yang sedikit lebih lebar dibandingkan *sigmoid*, namun masih belum mampu mengalirkan gradien seefisien keluarga *ReLU*.

Keunggulan fungsi aktivasi berbasis *ReLU* terutama terletak pada kemampuannya menghindari masalah *vanishing gradient*. Pada *ReLU*, gradien untuk input positif tetap konstan (yaitu 1), sehingga aliran gradien tidak terhambat dan pembelajaran dapat berlangsung lebih efisien. Varian seperti *leaky_relu* dan *ELU* juga mendukung aliran gradien dengan mengatasi potensi "*dying ReLU*" melalui pemberian gradien non-nol pada nilai negatif. Meskipun demikian, pemilihan fungsi aktivasi sangat bergantung pada karakteristik data dan masalah yang dihadapi. Misalnya, jika data memiliki distribusi yang simetris atau jika model diharapkan menangkap nilai negatif secara eksplisit, fungsi *tanh* mungkin dapat memberikan keuntungan tertentu. Namun, dalam konteks eksperimen yang dilakukan pada dataset *MNIST*, fungsi aktivasi berbasis *ReLU* terbukti memberikan kinerja terbaik karena kemampuannya untuk menjaga aliran gradien dan menghindari saturasi.

Secara keseluruhan, hasil pengujian tidak hanya menunjukkan perbedaan dalam akurasi prediksi dan grafik *loss*, tetapi juga perbedaan yang signifikan pada distribusi bobot dan gradien. Ini menegaskan bahwa fungsi aktivasi mempengaruhi seluruh aspek pembelajaran dalam FFNN, mulai dari kecepatan konvergensi hingga kestabilan pembaruan parameter. Oleh karena itu, pemilihan fungsi aktivasi harus didasarkan pada eksperimen yang cermat, dengan memperhatikan kondisi data serta kebutuhan spesifik model agar dapat menghasilkan performa yang optimal.

Berikut adalah Hasil Perbandingan Akurasi Hasil Prediksi :

```
Melatih model dengan fungsi aktivasi hidden: linear  
Model dengan aktivasi linear memperoleh akurasi: 0.9036

Melatih model dengan fungsi aktivasi hidden: relu  
Model dengan aktivasi relu memperoleh akurasi: 0.9077

Melatih model dengan fungsi aktivasi hidden: sigmoid  
Model dengan aktivasi sigmoid memperoleh akurasi: 0.8558

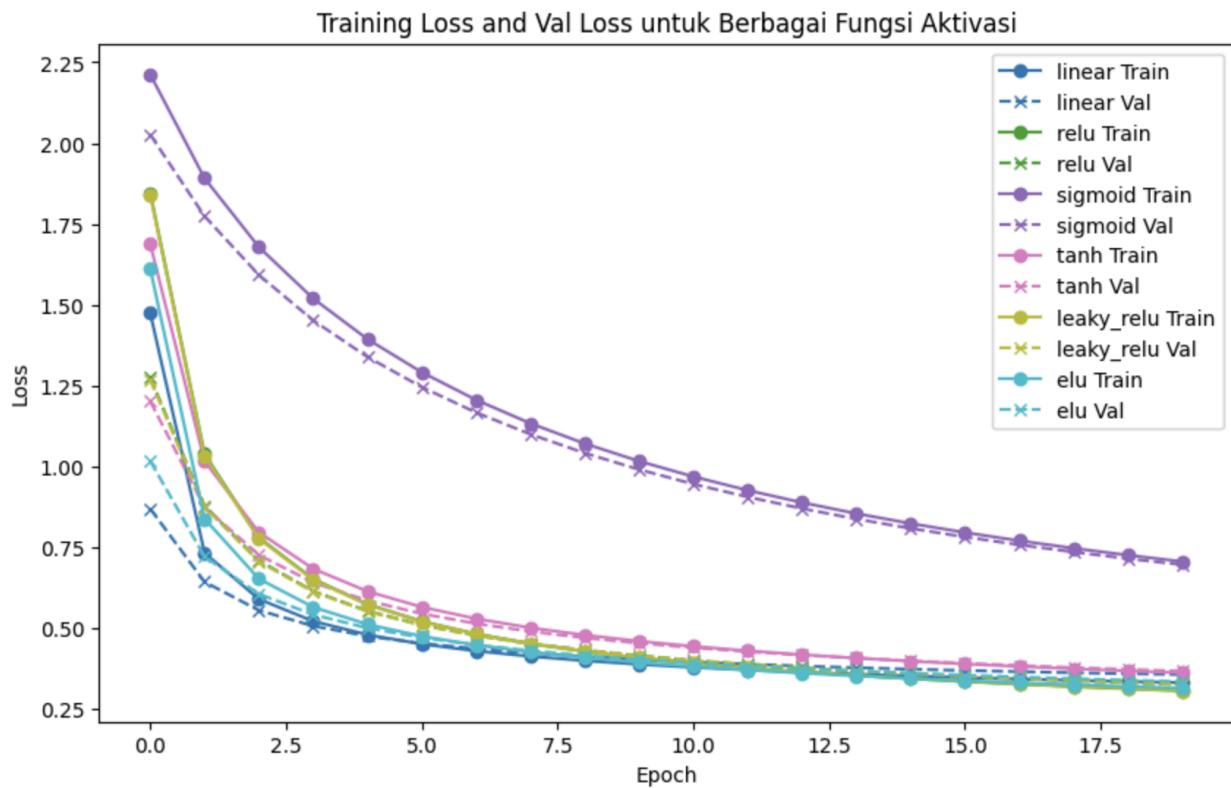
Melatih model dengan fungsi aktivasi hidden: tanh  
Model dengan aktivasi tanh memperoleh akurasi: 0.8967

Melatih model dengan fungsi aktivasi hidden: leaky_relu  
Model dengan aktivasi leaky_relu memperoleh akurasi: 0.9075

Melatih model dengan fungsi aktivasi hidden: elu  
Model dengan aktivasi elu memperoleh akurasi: 0.9058
```

Gambar 2.2.2.1. Perbandingan Hasil Prediksi Variasi Fungsi Aktivasi

Berikut adalah hasil perbandingan *loss* untuk pengujian variasi fungsi aktivasi :



Gambar 2.2.2.2. Perbandingan Loss Pelatihan

Dari grafik perbandingan *loss* pada berbagai fungsi aktivasi, kita dapat memperhatikan:

1. Penurunan Kurva *Loss*:

Dari grafik, tampak bahwa semua kurva baik untuk *linear*, *relu*, *sigmoid*, *tanh*, *leaky_relu*, maupun *elu* cenderung turun dengan laju yang cukup cepat pada awal pelatihan. Hal ini menunjukkan bahwa pada fase awal, model dengan fungsi aktivasi mana pun masih mempelajari pola dasar pada data. Mendekati *epoch* akhir, laju penurunan *loss* umumnya lebih landai (lambat). Ini menandakan model semakin menyesuaikan bobotnya untuk meminimalkan error lebih lanjut.

2. Dampak Fungsi Aktivasi:

- *Sigmoid*

Dari grafik, kurva *Sigmoid* (warna ungu) memiliki *loss* paling tinggi baik untuk training maupun validation, terutama di awal pelatihan dan juga pada epoch akhir.

- *Tanh*

Tanh (warna merah muda) umumnya dapat belajar sedikit lebih baik daripada *Sigmoid*, karena loss *Tanh* masih lebih rendah.

- *Linear*

Aktivasi linear (warna biru) menunjukkan bahwa *loss* untuk linear pada awalnya berada di paling rendah dibanding yang lain. Tetapi di akhir tidak sebaik keluarga *ReLU*.

- *ReLU*, *Leaky ReLU*, dan *ELU*

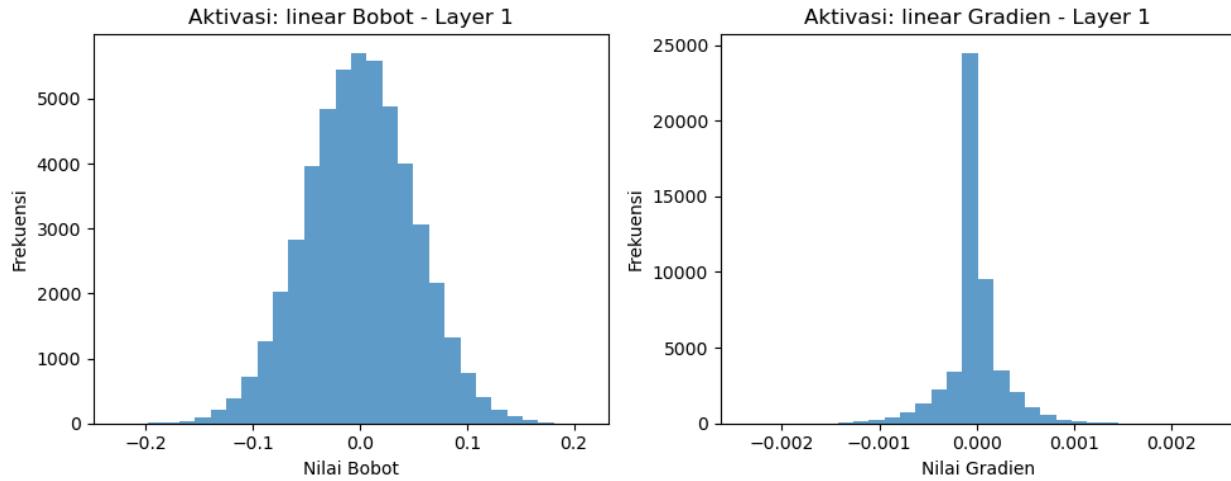
ReLU (warna hijau), *Leaky ReLU* (warna kuning), dan *ELU* (warna toska) bertahan di *loss* paling rendah hingga *epoch* akhir.

3. *Training vs Validation Loss*:

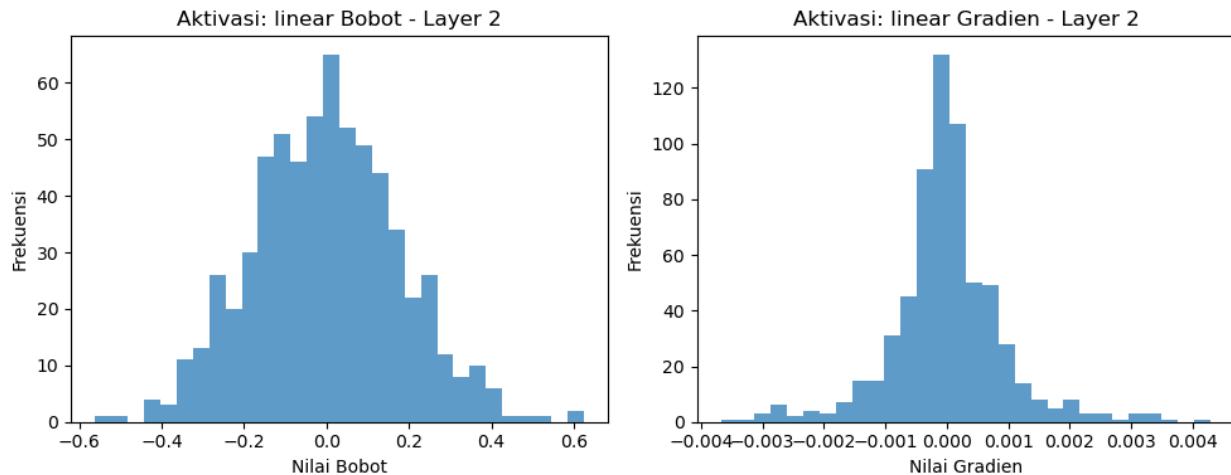
Pada sebagian besar fungsi aktivasi, kurva training *loss* dan validation *loss* relatif berdekatan dan menurun bersama. Ini menandakan bahwa model belajar dari data *training* tanpa mengalami *overfitting* yang parah, sebab validation *loss* juga ikut menurun dengan laju yang serupa. Untuk *Sigmoid*, perbedaan antara *training loss* dan *validation loss* bisa terlihat, namun keduanya tetap berada di skala yang lebih tinggi dibanding aktivasi lain. Bagi *ReLU-family*, *train* dan *val* *loss* sama-sama mencapai titik terendah dibandingkan aktivasi lain.

Pada *epoch-epoch* awal, *training loss* lebih tinggi dan seiring meningkatnya pemahaman model terhadap pola data, *training loss* turun lebih stabil dan berada sangat dekat dengan *validation loss* di *epoch-epoch* akhir.

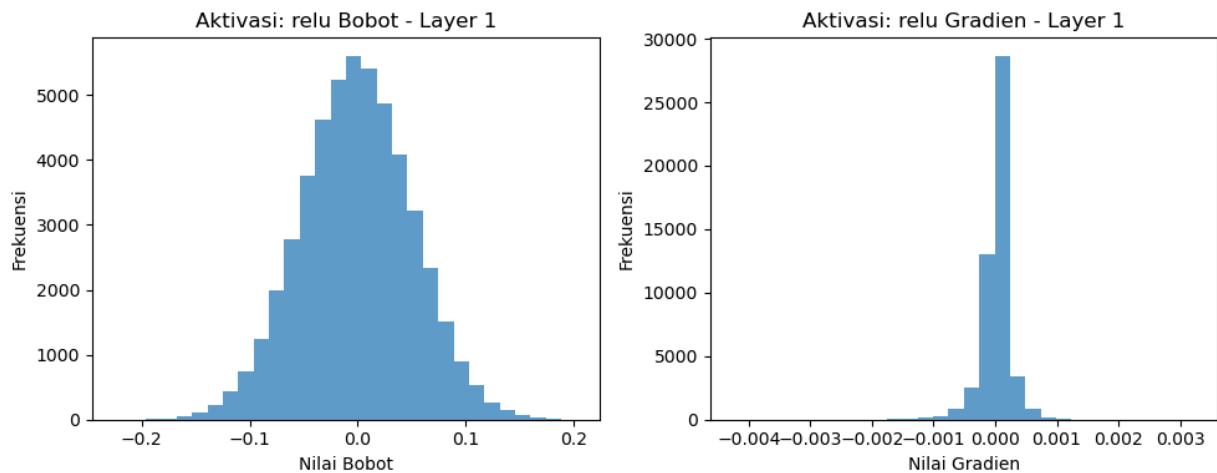
Berikut adalah Hasil Perbandingan Distribusi Bobot dan Gradien tiap *layer*:



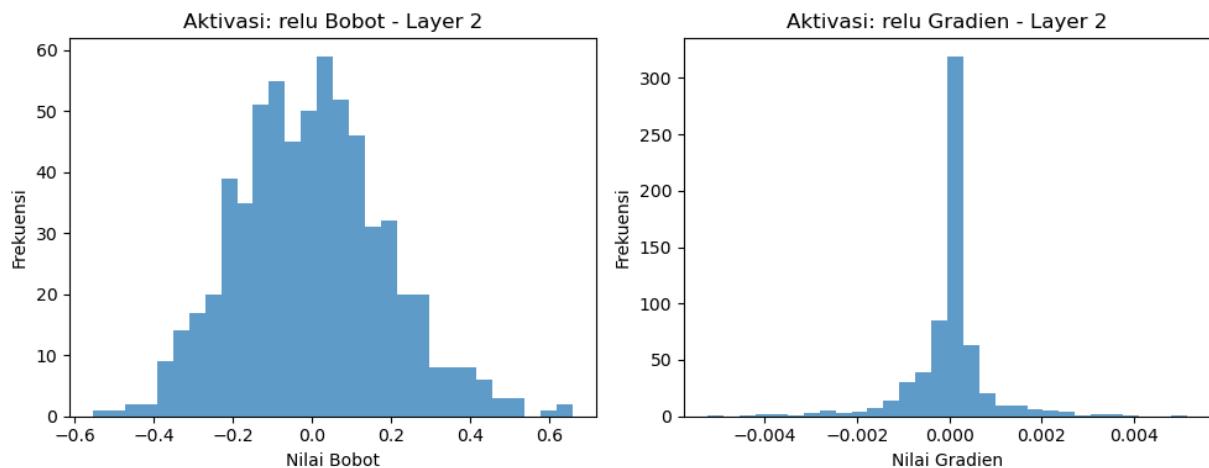
Gambar 2.2.2.3. Perbandingan Distribusi Bobot dan Gradien pada *Layer 1* (linear)



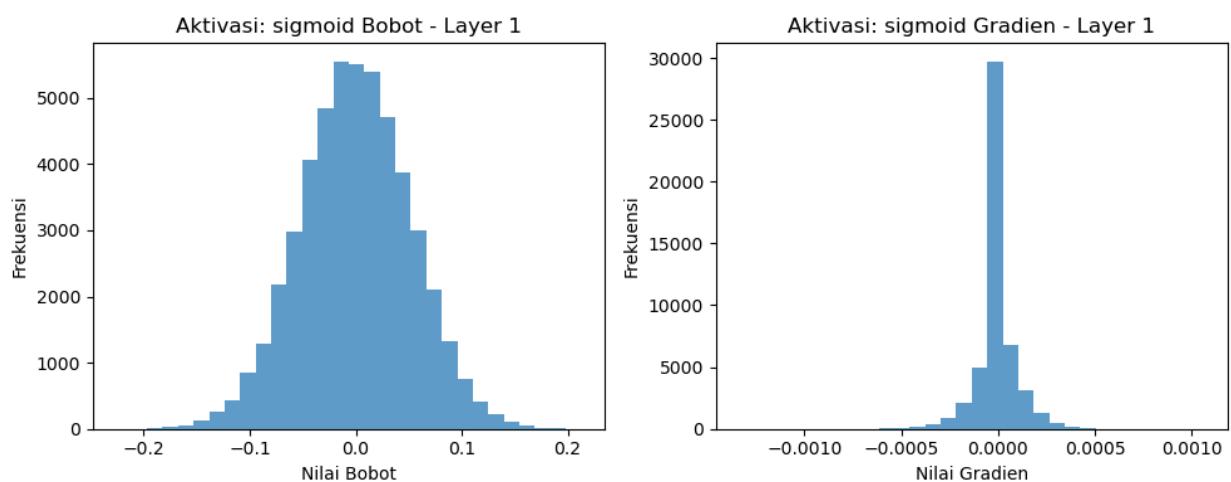
Gambar 2.2.2.4. Perbandingan Distribusi Bobot dan Gradien pada *Layer 2* (linear)



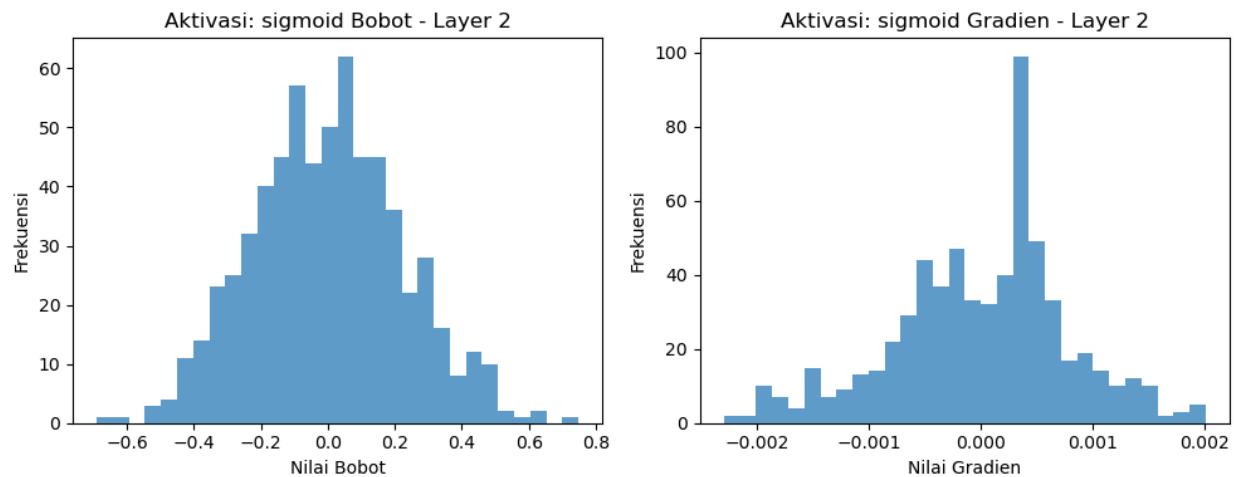
Gambar 2.2.2.5. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*relu*)



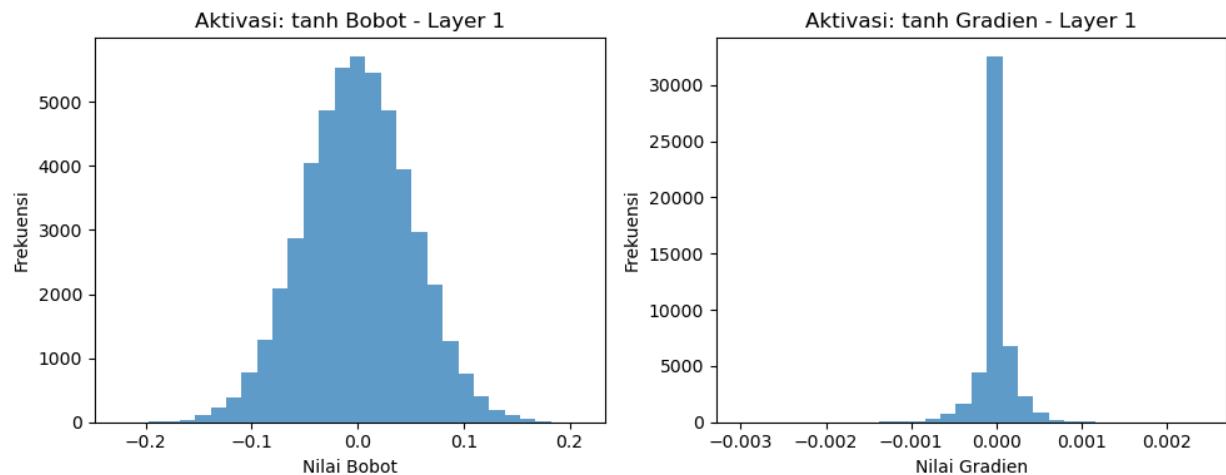
Gambar 2.2.2.6. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*relu*)



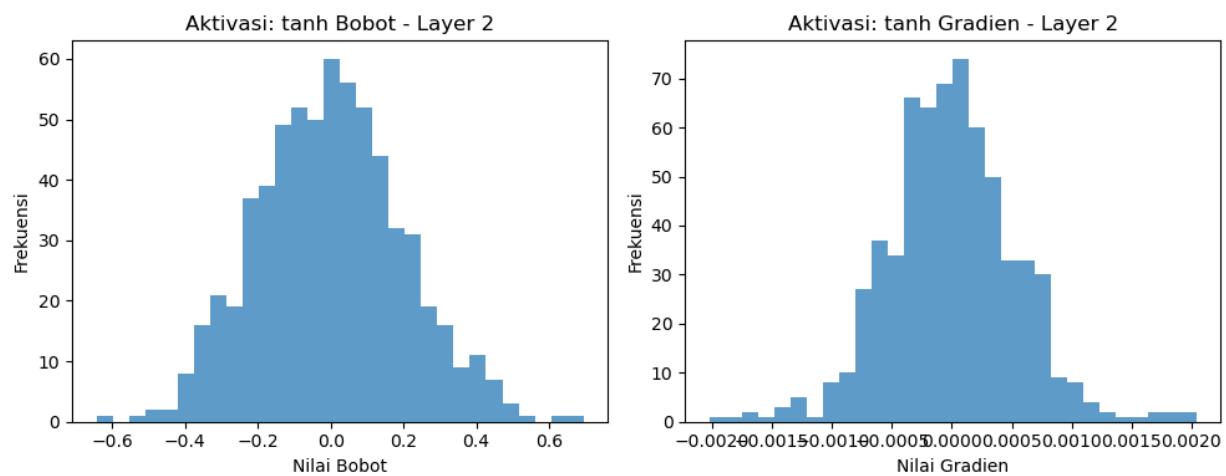
Gambar 2.2.2.7. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*sigmoid*)



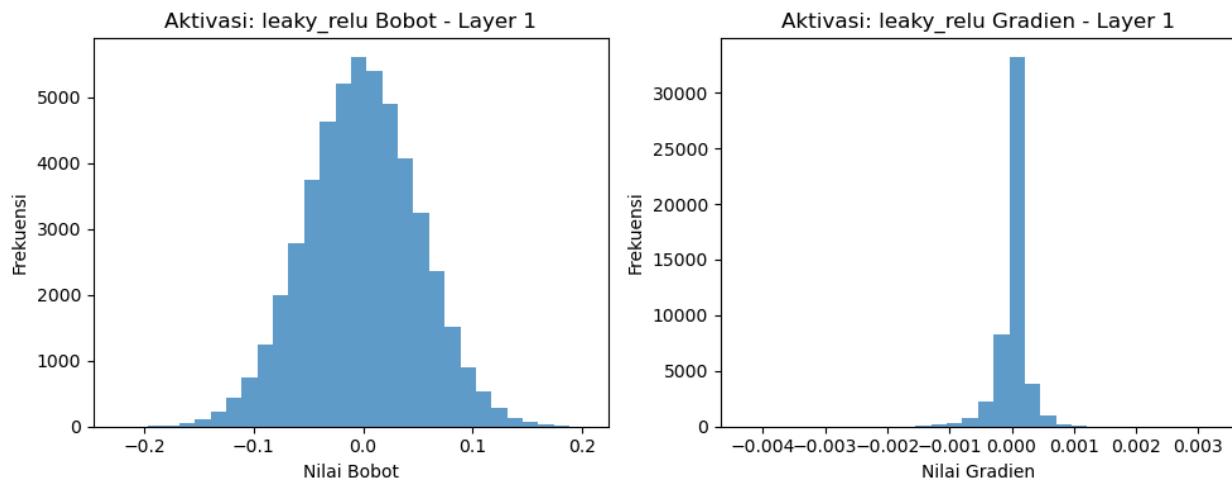
Gambar 2.2.2.8. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*sigmoid*)



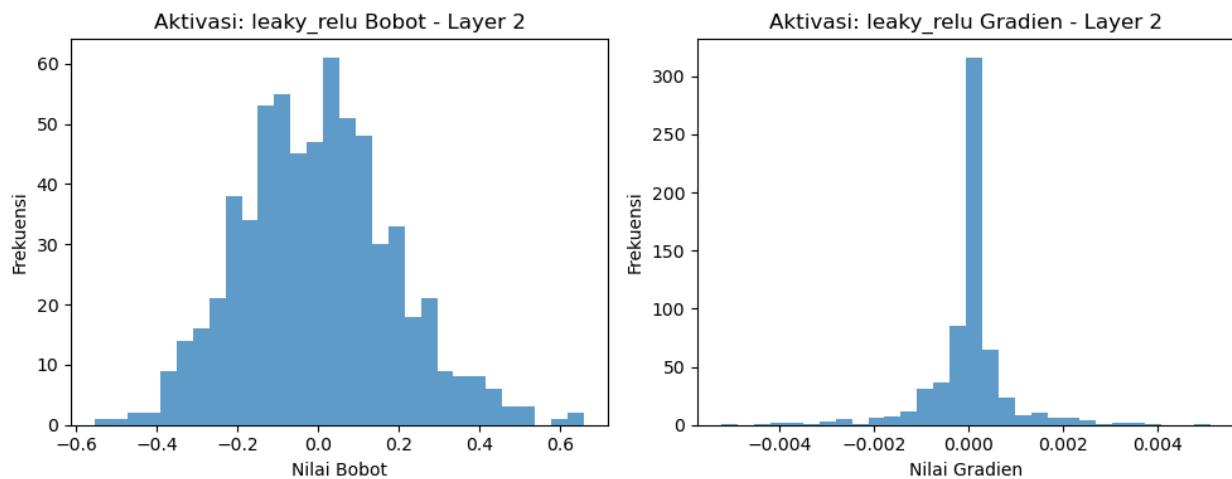
Gambar 2.2.2.9. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*tanh*)



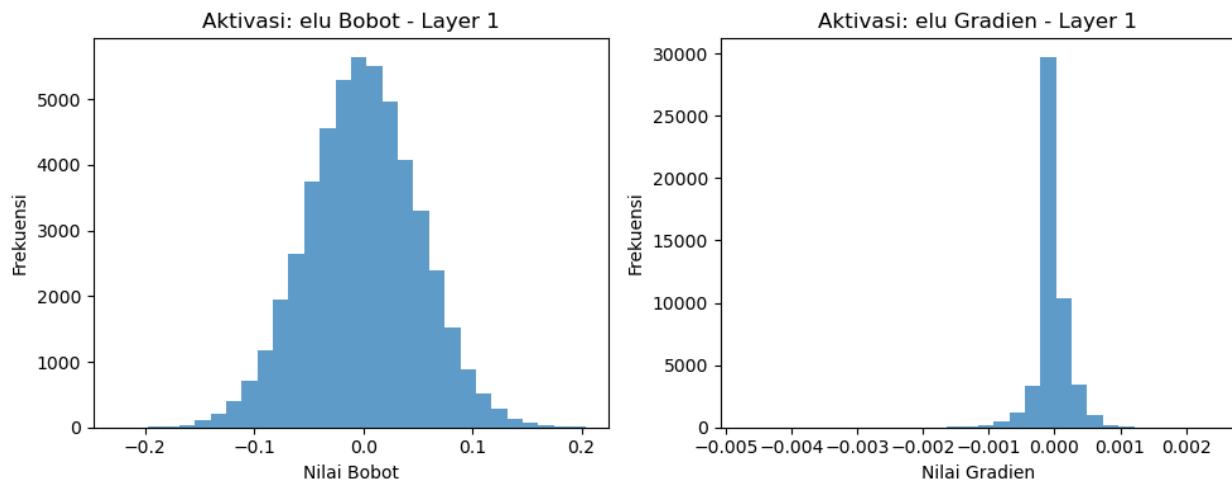
Gambar 2.2.2.10. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*tanh*)



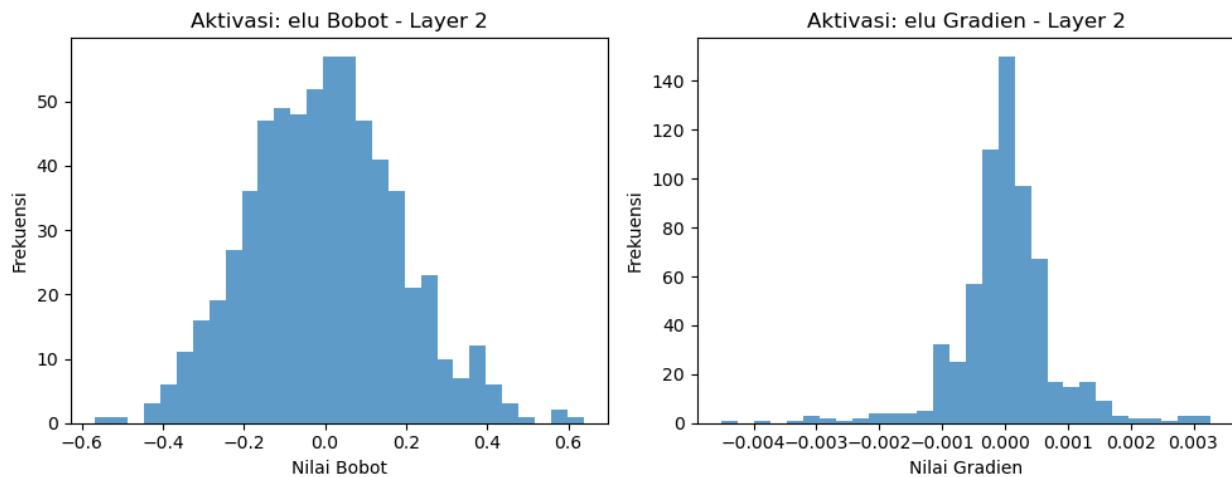
Gambar 2.2.2.11. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*leaky_relu*)



Gambar 2.2.2.12. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*leaky_relu*)



Gambar 2.2.2.13. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*elu*)



Gambar 2.2.2.14. Perbandingan Distribusi Bobot dan Gradien pada *Layer 2 (elu)*

2.2.3 Pengaruh *Learning Rate*

Learning rate adalah *hyperparameter* yang menentukan seberapa besar langkah pembaruan bobot (*weight update*) pada setiap iterasi proses *training*. Semakin tinggi nilai *learning rate*, semakin cepat model memperbarui bobotnya, namun risiko *overshoot* atau ketidakstabilan juga meningkat. Sebaliknya, jika *learning rate* terlalu kecil, pembaruan bobot berlangsung sangat lambat, sehingga proses konvergensi bisa memakan waktu lama atau bahkan terjebak di *local minimum*. Untuk menguji pengaruh *learning rate* pada FFNN, dilakukan eksperimen dengan tiga variasi nilai *Learning rate*: 0,001; 0,01; dan 0,1. Pada setiap konfigurasi, arsitektur jaringannya tetap sama yaitu $784 \rightarrow 64 \rightarrow 10$ dan parameter pelatihan lain (*batch size* = 32, *epochs* = 20) dipertahankan agar perbandingan menjadi adil.

Hasil pengujian menunjukkan bahwa model dengan *learning rate* 0,001 hanya mencapai akurasi sekitar 76,07%, sedangkan pada *learning rate* 0,01 akurasi meningkat signifikan menjadi 90,77%. Ketika *Learning rate* dinaikkan lebih jauh menjadi 0,1, model justru mencapai akurasi tertinggi, yaitu 95,30%. Meskipun pada umumnya *learning rate* yang terlalu besar berpotensi menyebabkan *loss* gagal konvergen, dalam kasus ini *learning rate* = 0,1 masih berada dalam rentang stabil untuk dataset *MNIST*, sehingga memberikan kecepatan konvergensi yang tinggi dan akurasi akhir yang terbaik. Dari grafik *loss* pelatihan dan validasi, terlihat bahwa *learning rate* yang terlalu kecil

(0,001) membuat penurunan *loss* berjalan lambat, sementara *learning rate* yang lebih besar (0,01 dan 0,1) menurunkan *loss* dengan lebih cepat dan stabil.

Selain melihat akurasi dan *loss*, analisis distribusi bobot dan gradien juga memperkuat temuan tersebut. Pada *learning rate* yang rendah (0,001), gradien cenderung kecil, sehingga *update* bobot pun minimal. Hal ini tercermin dari histogram gradien yang relatif terkonsentrasi di sekitar nol yang menandakan pembelajarannya lebih lambat. Ketika *learning rate* lebih besar, seperti 0,01 atau 0,1, distribusi gradien menjadi lebih luas, yang mengindikasikan *update* bobot lebih signifikan dan proses pembelajaran yang lebih agresif. Namun, jika *learning rate* terlalu besar, model bisa saja mengalami overshoot dan sulit mencapai konvergensi yang stabil. Untungnya, pada eksperimen ini *learning rate* = 0,1 masih cukup stabil sehingga menghasilkan akurasi yang lebih tinggi.

Secara keseluruhan, hasil eksperimen menunjukkan pentingnya pemilihan *learning rate* yang tepat. *Learning rate* terlalu kecil (0,001) membuat proses training berjalan sangat lambat dan menahan pencapaian akurasi yang optimal. *Learning rate* sedang (0,01) sudah cukup efektif untuk mencapai akurasi yang layak, sedangkan *learning rate* lebih besar (0,1) justru membawa model ke performa terbaik pada dataset *MNIST* ini. Meskipun demikian, pemilihan *learning rate* ideal harus tetap kontekstual dan sangat tergantung pada karakteristik data serta arsitektur model.

Berikut adalah Hasil Perbandingan Akurasi Hasil Prediksi :

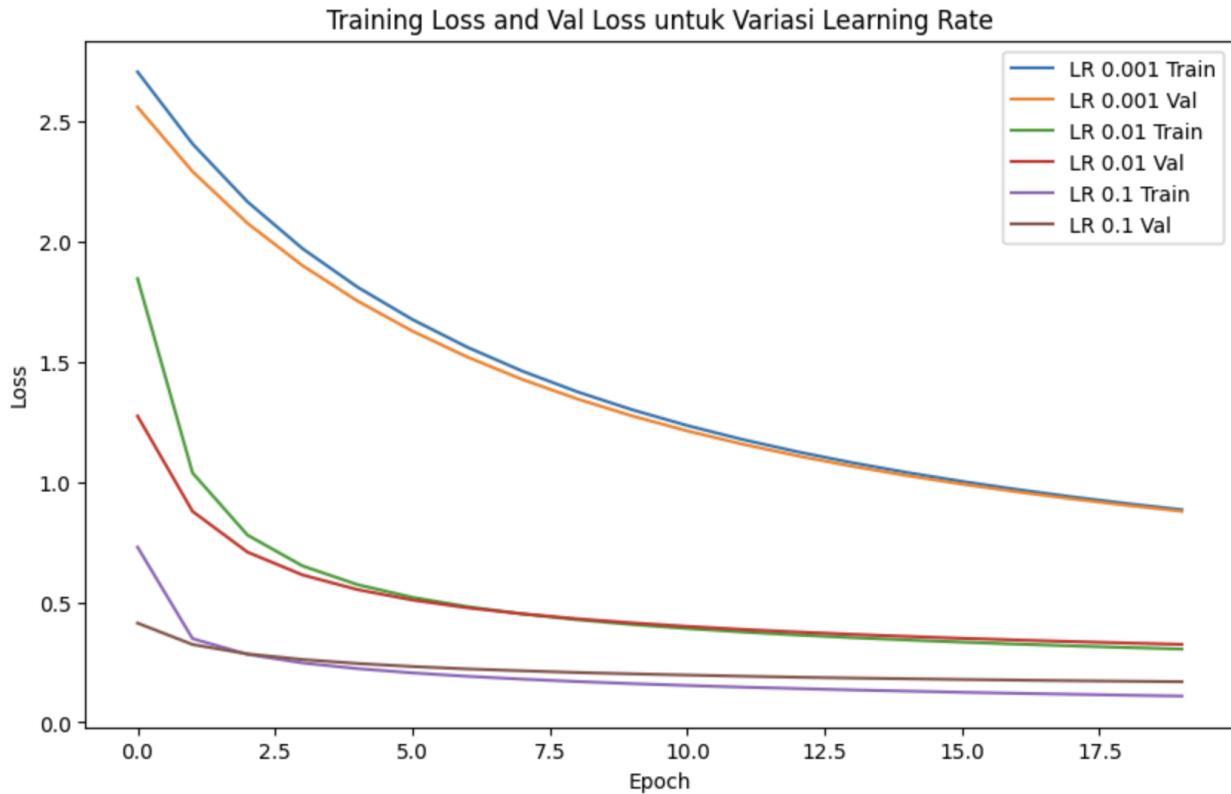
```
Melatih model dengan learning rate: 0.001
Model dengan learning rate 0.001 memperoleh akurasi: 0.7607

Melatih model dengan learning rate: 0.01
Model dengan learning rate 0.01 memperoleh akurasi: 0.9077

Melatih model dengan learning rate: 0.1
Model dengan learning rate 0.1 memperoleh akurasi: 0.9530
```

Gambar 2.2.3.1. Perbandingan Hasil Prediksi Variasi *Learning Rate*

Berikut adalah hasil perbandingan *loss* untuk pengujian variasi *learning rate* :



Gambar 2.2.3.2. Perbandingan Loss Pelatihan

Dari grafik perbandingan *loss* untuk variasi *learning rate*, kita dapat memperhatikan:

1. Penurunan Kurva *Loss*:

Terlepas dari perbedaan nilai *learning rate*, semua kurva menunjukkan tren penurunan *loss* seiring meningkatnya *epoch*. Ini menandakan bahwa model secara umum berhasil belajar dari data *training* dan meminimalkan error. LR=0.1 (garis ungu) tampak menurun lebih cepat di awal dan mencapai *loss* terendah dalam waktu singkat. LR=0.01 (garis hijau) juga menurun relatif cepat, tetapi tidak secepat LR=0.1. LR=0.001 (garis biru) memiliki laju penurunan paling lambat.

2. Dampak Variasi *Learning Rate*:

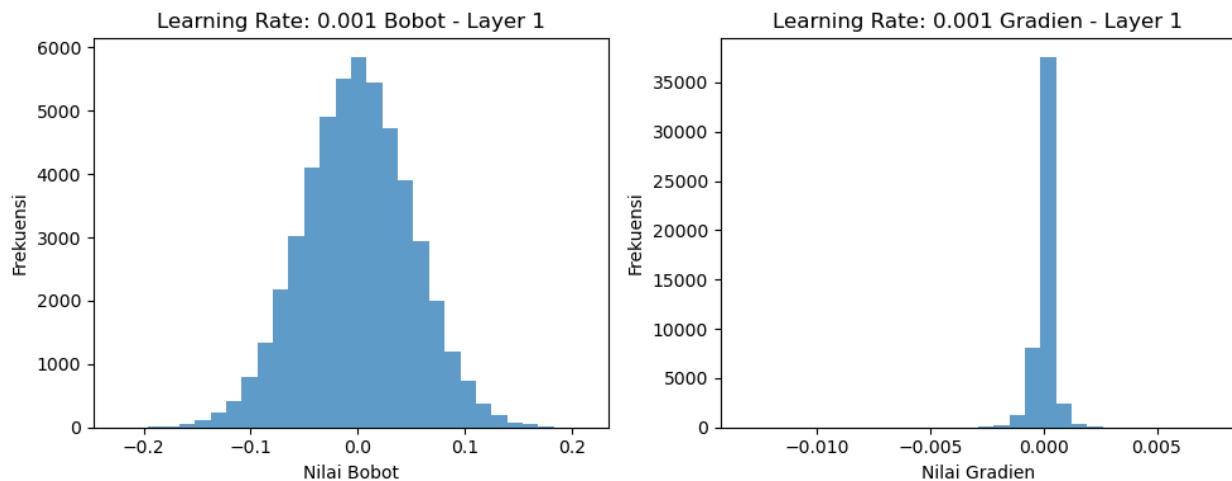
Learning rate kecil (0.001) memiliki risiko konvergensi yang sangat lambat. Model cenderung belajar stabil, tetapi butuh banyak *epoch* untuk mencapai *loss* yang rendah. *Learning rate* sedang (0.01) lebih cepat daripada

0.001 namun tetap relatif stabil. *Learning rate* besar (0.1) dapat mempercepat penurunan *loss* di awal dan mampu mencapai *loss* yang sangat rendah dengan cepat. Namun, terlalu besar *learning rate* juga bisa menyebabkan ketidakstabilan.

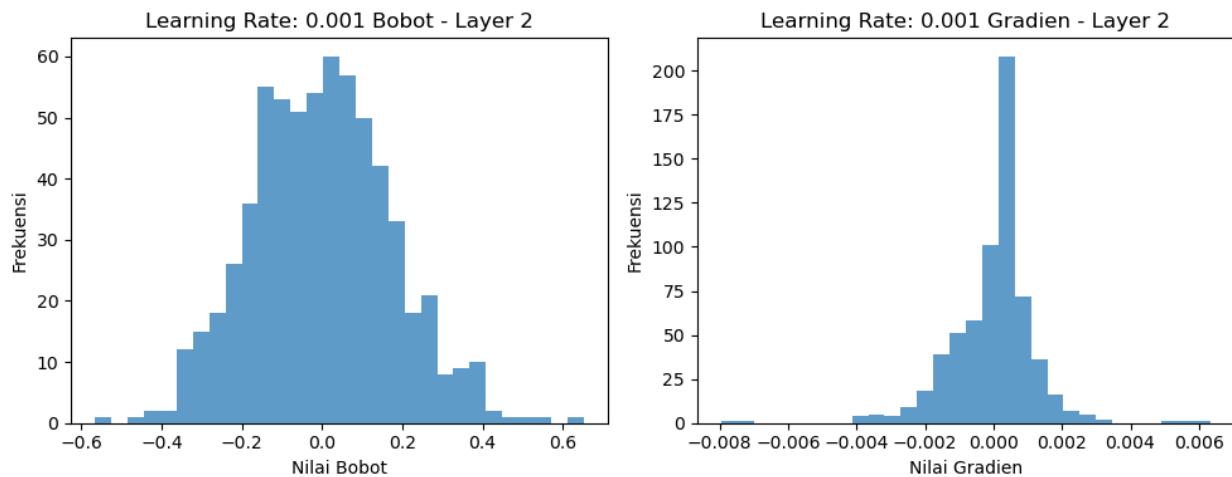
3. *Training vs Validation Loss:*

Pada masing-masing nilai LR, *training loss* dan *validation loss* turun bersama-sama dengan gap yang tidak terlalu besar, menandakan tidak ada *overfitting* yang sangat signifikan. LR=0.1 baik *training* maupun *validation loss* di LR=0.1 mencapai titik paling rendah dibandingkan LR lain. Hal ini menunjukkan bahwa meskipun LR=0.1 termasuk agresif, model tetap *generalize* dengan baik (*validation loss* juga menurun). Pada akhir *epoch*, *val loss* lebih rendah dibandingkan *training loss* kecuali untuk LR=0.001.

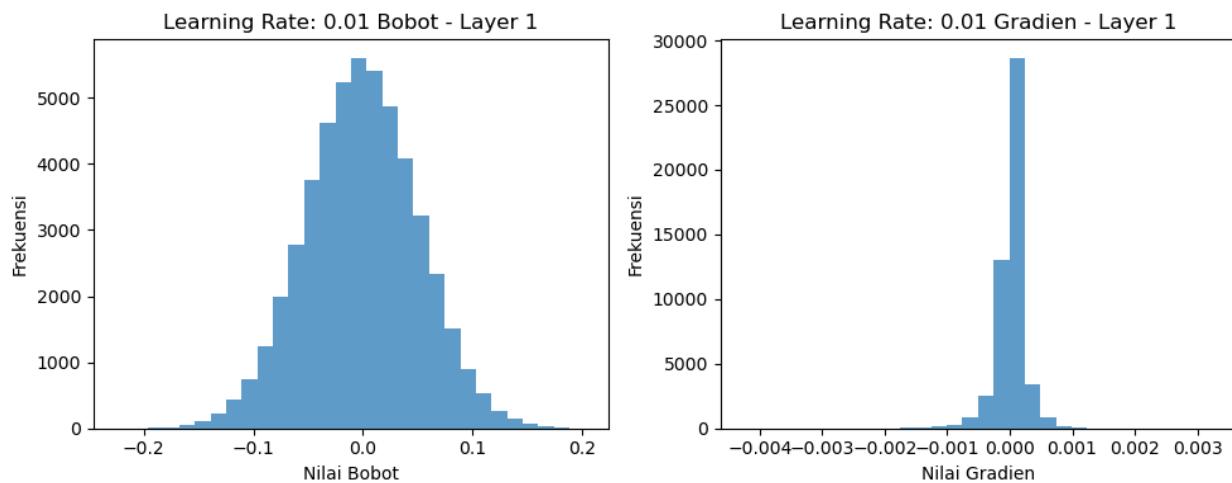
Berikut adalah Hasil Perbandingan Distribusi Bobot dan Gradien tiap layer:



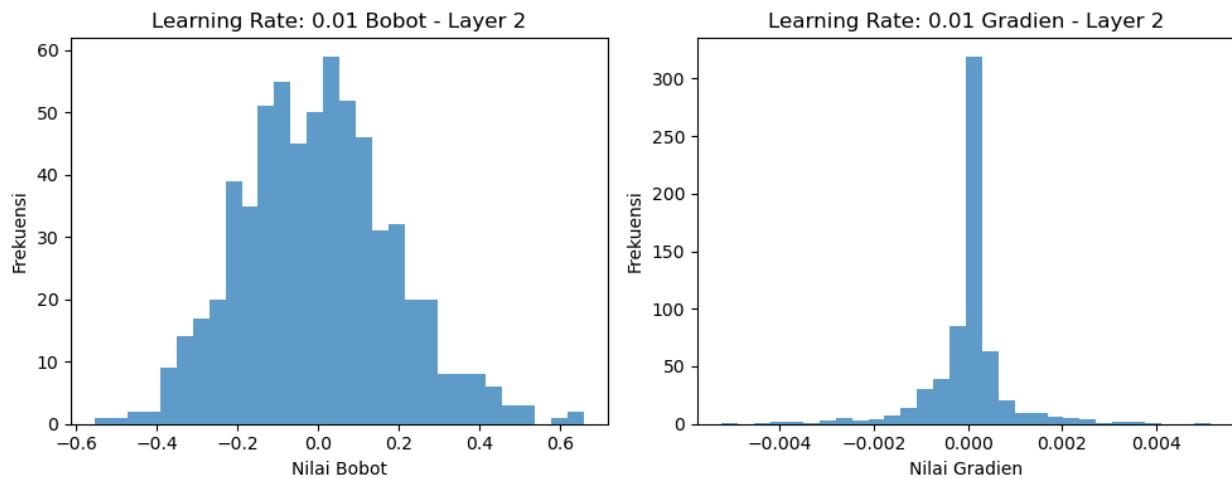
Gambar 2.2.3.3. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*Learning Rate* 0.001)



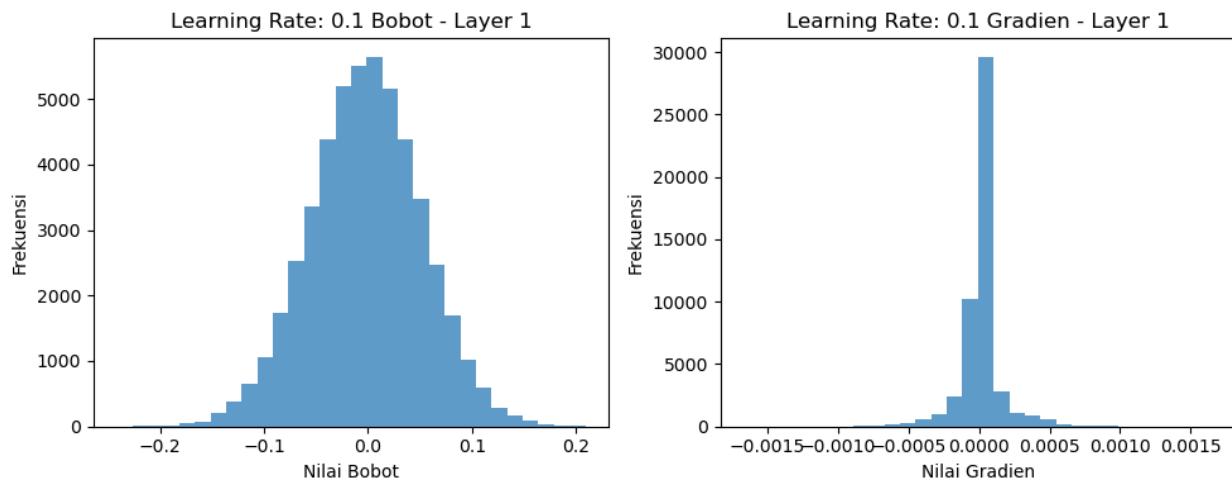
Gambar 2.2.3.4. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*Learning Rate* 0.001)



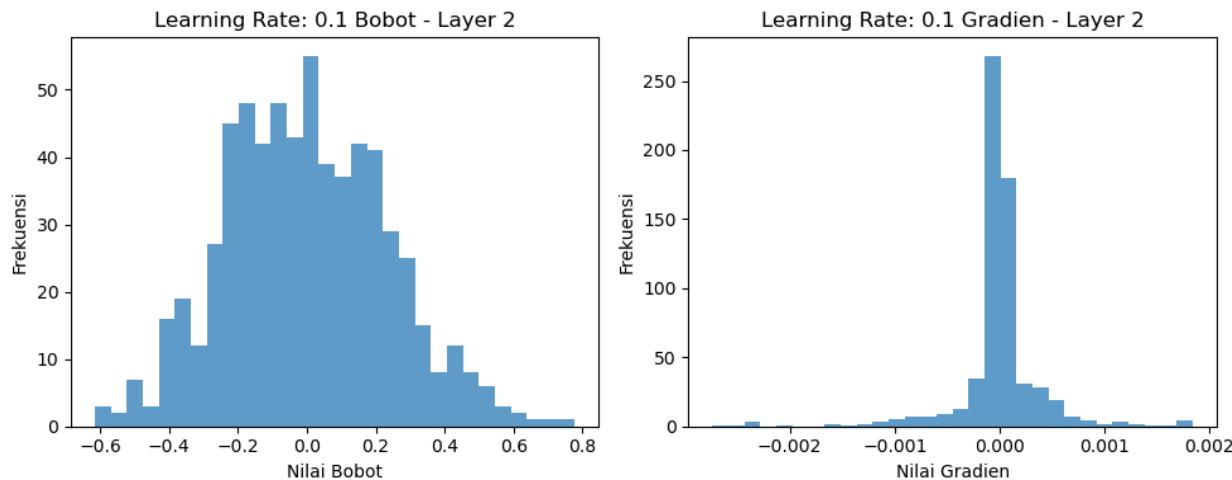
Gambar 2.2.3.5. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*Learning Rate* 0.01)



Gambar 2.2.3.6. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*Learning Rate* 0.01)



Gambar 2.2.3.7. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*Learning Rate* 0.1)



Gambar 2.2.3.8. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*Learning Rate* 0.01)

2.2.4 Pengaruh Inisialisasi Bobot

Inisialisasi bobot merupakan salah satu komponen dasar dalam FFNN. Secara umum, inisialisasi bobot bertujuan untuk menentukan nilai awal dari setiap bobot dalam jaringan sebelum proses *training* dimulai. Nilai awal tersebut sangat berpengaruh terhadap bagaimana sinyal dan gradien mengalir melalui jaringan, serta pada seberapa cepat dan stabil proses konvergensi selama pembelajaran. Jika inisialisasi dilakukan dengan cara yang tidak tepat misalnya, semua bobot diinisialisasi dengan nol maka jaringan tidak dapat memecah simetri antar neuron, sehingga setiap neuron menghasilkan *output* yang identik dan tidak ada variasi yang memungkinkan jaringan untuk belajar pola

yang berbeda dari data. Oleh karena itu, metode inisialisasi yang baik harus mampu memberikan sebaran nilai bobot yang optimal agar masing-masing *neuron* dapat bekerja secara independen dan mendukung aliran gradien yang stabil dan bisa menghindari masalah *vanishing gradient*.

Untuk menguji pengaruh metode inisialisasi bobot, dilakukan eksperimen pada arsitektur yang sama ($784 \rightarrow 64 \rightarrow 10$) dengan parameter pelatihan yang konstan (*epochs* = 20, *batch_size* = 32, *learning_rate* = 0.01) dan fungsi aktivasi *ReLU* di *hidden layer* serta *softmax* di *output layer*. Beberapa metode inisialisasi yang diuji meliputi:

- *Zero Initialization*: Menginisialisasi semua bobot dengan nol.
- *Uniform Initialization*: Mengambil bobot dari distribusi seragam (misalnya, dari interval [-0.5, 0.5]).
- *Normal Initialization*: Mengambil bobot dari distribusi normal dengan *mean* 0 dan *variance* tertentu (misalnya, 0.1).
- *He Initialization*: Menggunakan $N(0, \sqrt{2/fan_in})$, dirancang khusus untuk fungsi aktivasi *ReLU*.
- *Xavier Uniform/Normal Initialization*: Menghitung varians berdasarkan *fan_in* dan *fan_out* untuk menjaga kestabilan sinyal antar layer.

Hasil eksperimen menunjukkan perbandingan yang cukup mencolok. Model dengan *zero initialization* gagal memecah simetri antar *neuron*, menghasilkan akurasi hanya sekitar 11,43%. Sementara itu, model yang menggunakan *uniform* dan *normal initialization* memperoleh akurasi sekitar 83,30% dan 82,66% secara berturut-turut, yang menunjukkan kemampuan belajar yang jauh lebih baik daripada *zero initialization*, namun masih belum optimal. Di sisi lain, model dengan *He initialization* mencapai akurasi sekitar 90,77%, sedangkan model dengan *Xavier Uniform* dan *Xavier Normal* memperoleh akurasi masing-masing sekitar 91,09% dan 90,77%. Perbandingan hasil akhir prediksi ini menunjukkan bahwa metode inisialisasi yang didesain khusus untuk mempertahankan varians, seperti *He* dan *Xavier*, memberikan keuntungan dalam hal performa model. Hal ini terutama disebabkan oleh kesesuaian metode tersebut dengan fungsi aktivasi *ReLU* yang digunakan, di mana *He initialization* secara khusus dirancang untuk mengatasi potensi masalah *vanishing gradient* dengan mengatur varians bobot sebesar $2/fan_in$, sehingga menjaga aliran sinyal dan gradien tetap stabil pada setiap

layer. Sementara itu, *Xavier initialization* menjaga keseimbangan antara *fan_in* dan *fan_out* sehingga varians sinyal tetap konsisten dari input hingga output, yang juga sangat mendukung kinerja *ReLU* dalam menghindari saturasi yang berlebihan.

Selain akurasi akhir, grafik *loss* pelatihan dan validasi juga memberikan informasi penting. Model yang diinisialisasi dengan metode *He* dan *Xavier* menunjukkan penurunan *loss* yang lebih cepat dan stabil dibandingkan dengan model yang menggunakan *uniform* atau *normal initialization*. Hal ini mengindikasikan bahwa nilai awal bobot yang optimal membantu jaringan untuk melakukan pembaruan parameter secara efisien dan mencegah terjadinya fluktuasi *loss* yang besar selama proses training.

Lebih jauh lagi, analisis distribusi bobot dan gradien dari beberapa *layer* juga mengungkapkan perbedaan karakteristik antar metode inisialisasi. Pada *zero initialization*, histogram bobot dan gradien terkonsentrasi di angka nol, menandakan tidak adanya perbedaan nilai yang dapat memicu pembelajaran. Sementara itu, *uniform* dan *normal initialization* memberikan sebaran yang lebih lebar, namun distribusinya belum ideal untuk mendukung fungsi aktivasi *ReLU*, sehingga beberapa *neuron* mungkin beroperasi di daerah saturasi. Metode *He* dan *Xavier*, di sisi lain, menghasilkan distribusi bobot yang lebih terpusat namun dengan varians yang cukup untuk menjaga nilai aktivasi agar tidak terlalu kecil atau terlalu besar, sehingga mendukung aliran gradien yang optimal dan mencegah terjadinya *vanishing gradient*. Dengan demikian, distribusi bobot dan gradien yang optimal ini mendukung proses konvergensi model dan menghasilkan performa akhir yang lebih baik.

Secara keseluruhan, pengaruh inisialisasi bobot sangat besar terhadap performa FFNN. Metode inisialisasi yang tepat seperti *He* dan *Xavier* mampu memberikan nilai awal yang mendukung aliran sinyal dan gradien yang stabil, sehingga model dapat belajar dengan cepat dan efektif, menghasilkan akurasi yang tinggi serta *loss* yang menurun secara konsisten. Sebaliknya, metode yang tidak tepat, seperti *zero initialization*, akan menghambat pembelajaran secara drastis. Perbandingan hasil prediksi, grafik *loss*, dan distribusi bobot/gradien yang diperoleh dari eksperimen ini menegaskan bahwa pemilihan metode inisialisasi harus disesuaikan dengan karakteristik arsitektur dan fungsi aktivasi yang digunakan agar model dapat mencapai performa optimal.

Berikut adalah Hasil Perbandingan Akurasi Hasil Prediksi :

```
Melatih model dengan inisialisasi bobot: zero
Model dengan inisialisasi zero memperoleh akurasi: 0.1143

Melatih model dengan inisialisasi bobot: uniform
Model dengan inisialisasi uniform memperoleh akurasi: 0.8330

Melatih model dengan inisialisasi bobot: normal
Model dengan inisialisasi normal memperoleh akurasi: 0.8266

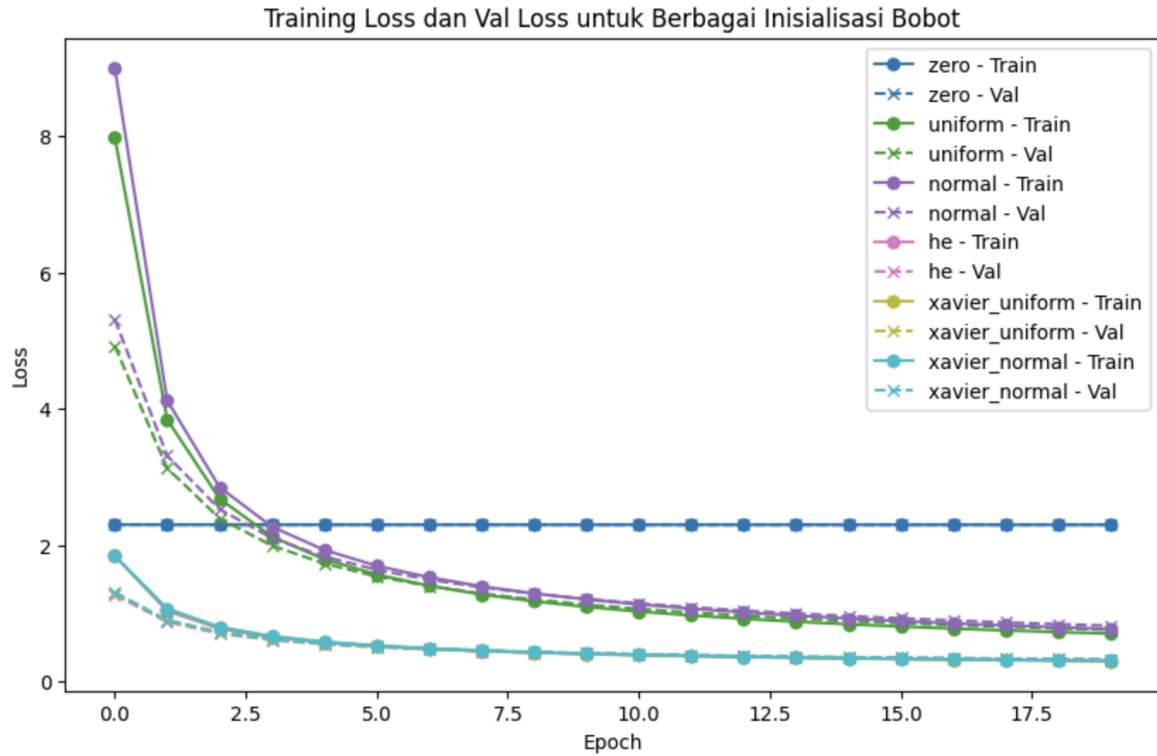
Melatih model dengan inisialisasi bobot: he
Model dengan inisialisasi he memperoleh akurasi: 0.9077

Melatih model dengan inisialisasi bobot: xavier_uniform
Model dengan inisialisasi xavier_uniform memperoleh akurasi: 0.9109

Melatih model dengan inisialisasi bobot: xavier_normal
Model dengan inisialisasi xavier_normal memperoleh akurasi: 0.9077
```

Gambar 2.2.4.1. Perbandingan Hasil Prediksi variasi inisialisasi bobot

Berikut adalah hasil perbandingan *loss* untuk pengujian variasi inisialisasi bobot :



Gambar 2.2.4.2. Perbandingan Loss Pelatihan

Dari grafik perbandingan *loss* untuk berbagai inisialisasi bobot, kita dapat memperhatikan:

1. Penurunan Kurva Loss:

Dari grafik, inisialisasi *zero* jelas memiliki kurva loss yang paling tinggi dan tidak menurun. Penyebabnya, jika bobot diinisialisasi dengan nol, semua *neuron* akan menerima gradien yang sama, sehingga pembelajaran tidak berlangsung efektif. Inisialisasi *Uniform & Normal*, meskipun keduanya random, skala yang dipilih dapat mempengaruhi seberapa cepat network dapat menurun di awal. Terkadang, *uniform* yang terlalu lebar atau *normal* dengan deviasi tertentu bisa menyebabkan *vanishing* atau *exploding gradients*, tergantung pada kedalaman jaringan dan fungsi aktivasi. Namun, keduanya masih menghasilkan penurunan loss yang jauh lebih baik dibanding *zero initialization*. *He Initialization* (umumnya disarankan untuk *ReLU* dan variannya) membantu menjaga variansi gradien tetap stabil sehingga loss dapat turun lebih cepat. *Xavier Initialization* (baik *uniform* maupun *normal*) didesain agar bobot terdistribusi dengan variansi tertentu, terutama cocok untuk aktivasi sigmoid/tanh, sehingga gradien tidak cepat lenyap atau meledak. Pada grafik, inisialisasi He, Xavier Uniform, dan Xavier Normal semuanya menunjukkan penurunan loss yang relatif lebih cepat dan stabil.

2. Dampak Inisialisasi Bobot:

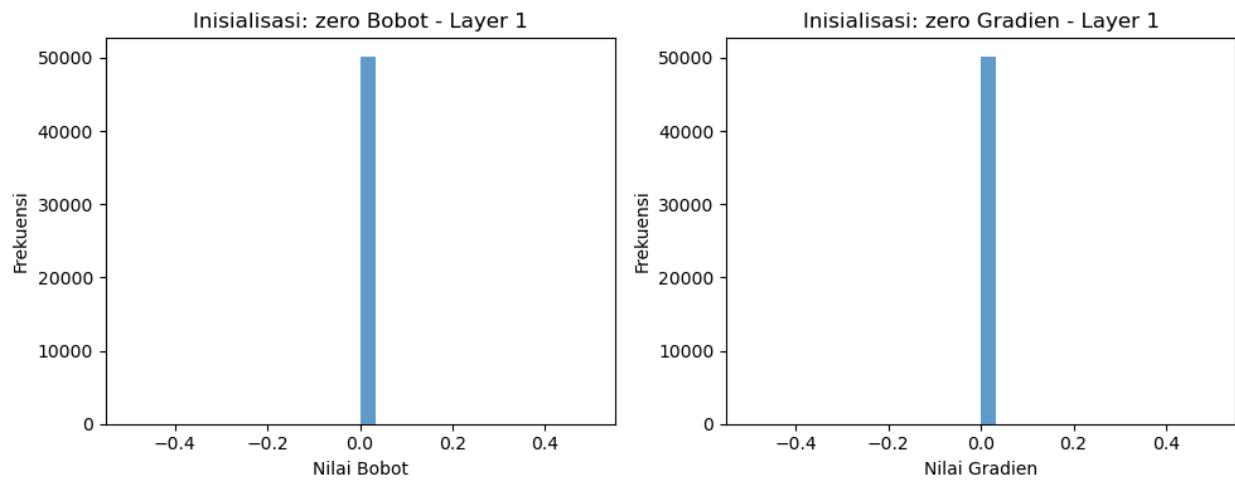
Inisialisasi *zero* membuat kemampuan jaringan untuk belajar menjadi mati karena gradien antar *neuron* menjadi identik. Terlihat pada grafik bahwa *loss* tidak menurun sama sekali. Random *Uniform* & Random *Normal* membuat model tetap bisa belajar, namun berpotensi menghadapi masalah jika distribusi bobot awal terlalu lebar (*exploding gradients*) atau terlalu sempit (*vanishing gradients*). Hasil akhirnya masih cukup baik, tetapi tidak seoptimal inisialisasi khusus seperti *He* atau *Xavier*. *He Initialization* dirancang untuk *ReLU-based networks*. Membantu menjaga skala output setiap *layer* agar gradien lebih stabil, terutama pada lapisan yang dalam. *Xavier (Uniform & Normal)* menyeimbangkan skala bobot agar tidak terjadi *vanishing* atau *exploding*. Metode ini cocok untuk *ReLU* juga.

3. Training vs Validation Loss:

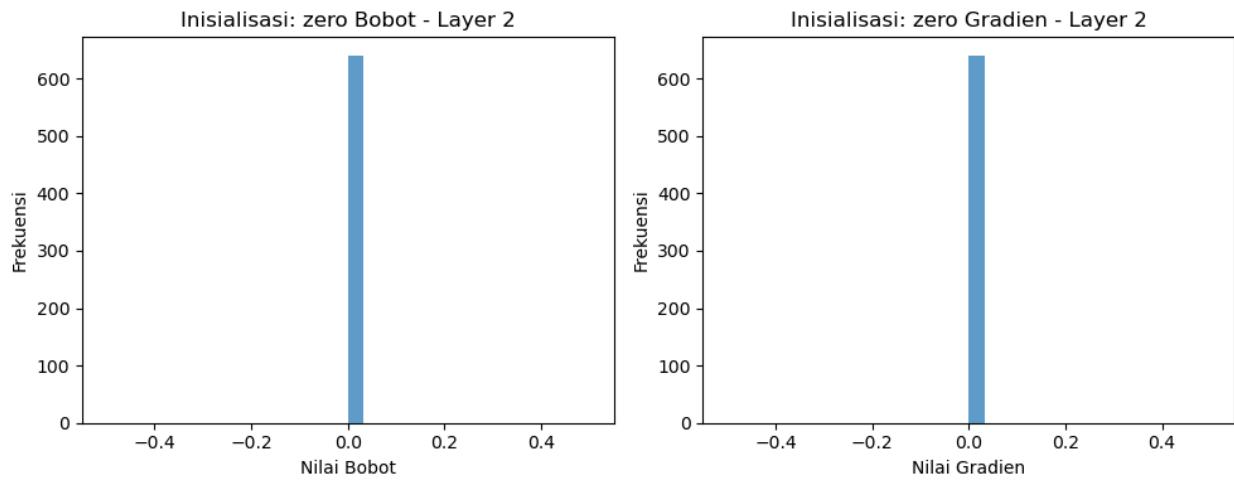
Pada inisialisasi yang baik (*He* dan *Xavier*), *training loss* dan *validation loss* tampak menurun bersama, dengan jarak yang tidak terlalu jauh. Hal ini

menandakan model tidak mengalami *overfitting* yang parah. Di *inisialisasi zero*, *training* maupun *validation loss* cenderung tinggi dan nyaris tidak turun. Ini membuktikan bahwa model gagal berlatih akibat bobot awal yang tidak ideal. Untuk *He* dan *Xavier*, *loss* semakin mendekati nilai yang rendah dan relatif stabil di *epoch* akhir, mengindikasikan bahwa inisialisasi awal memberi kondisi yang baik untuk konvergensi.

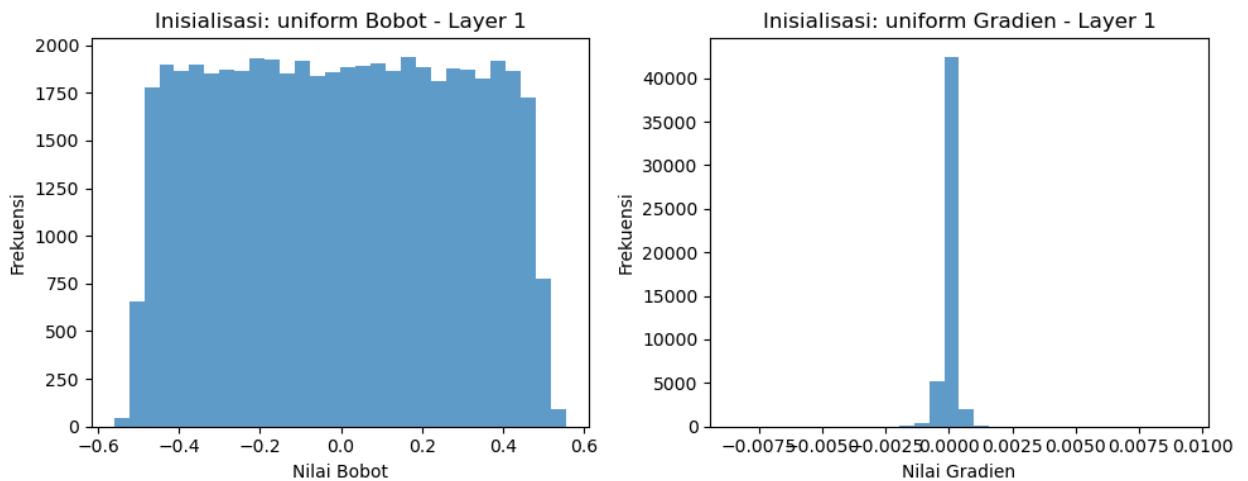
Berikut adalah Hasil Perbandingan Distribusi Bobot dan Gradien tiap layer:



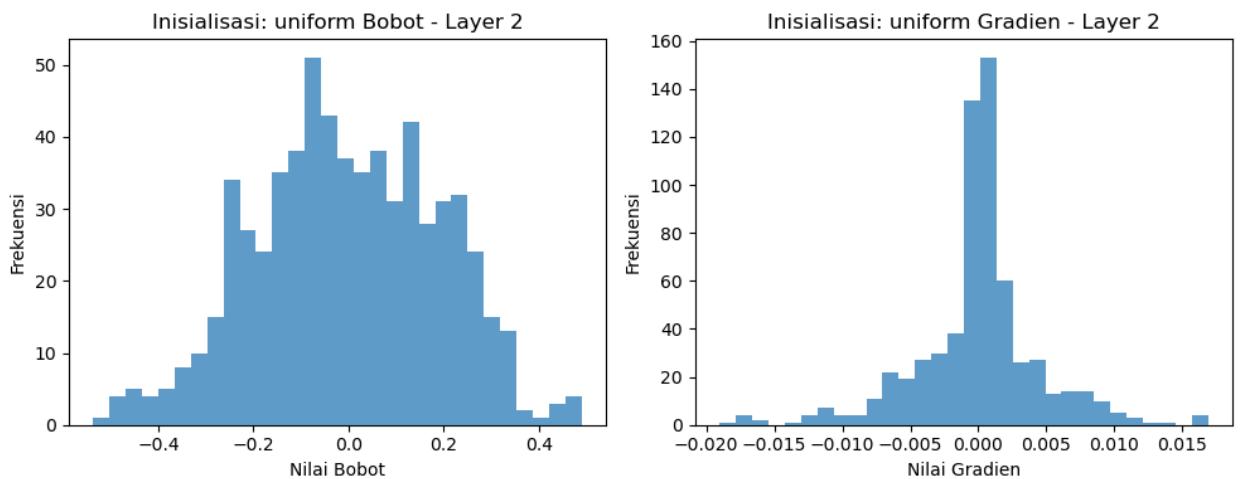
Gambar 2.2.4.3. Perbandingan Distribusi Bobot dan Gradien pada *Layer 1 (Zero)*



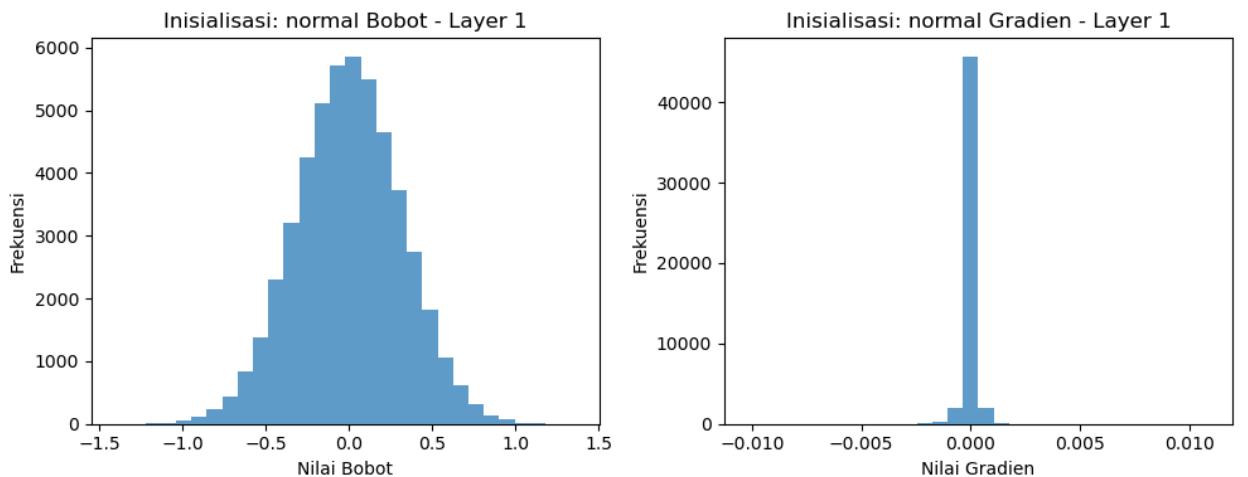
Gambar 2.2.4.4. Perbandingan Distribusi Bobot dan Gradien pada *Layer 2 (Zero)*



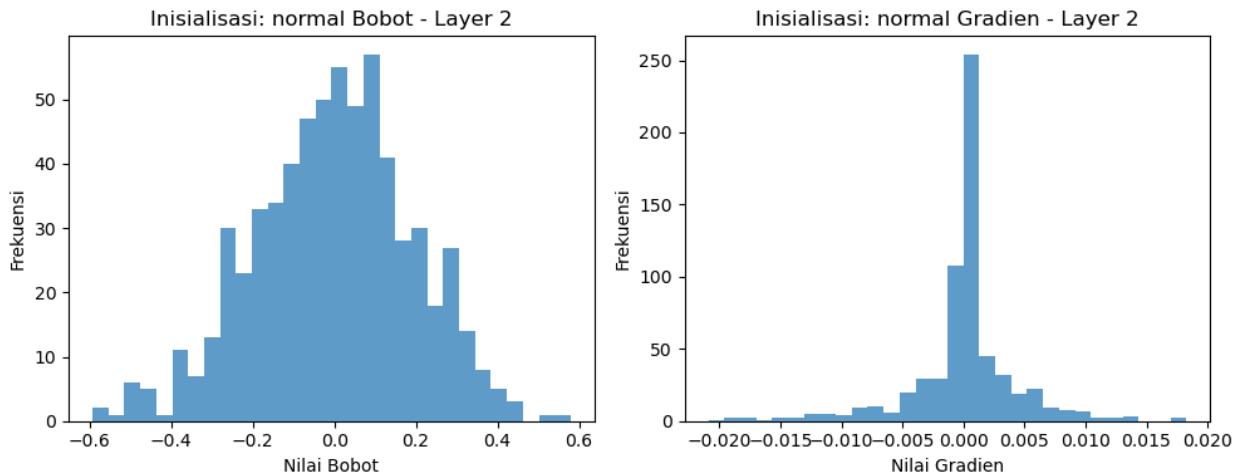
Gambar 2.2.4.5. Perbandingan Distribusi Bobot dan Gradien pada *Layer 1 (Uniform)*



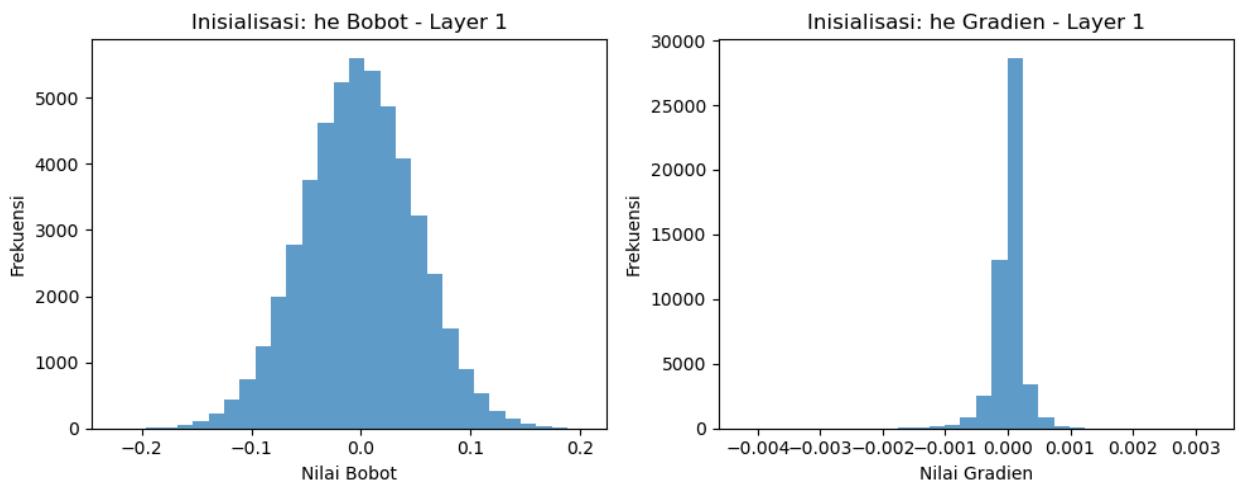
Gambar 2.2.4.6. Perbandingan Distribusi Bobot dan Gradien pada *Layer 2 (Uniform)*



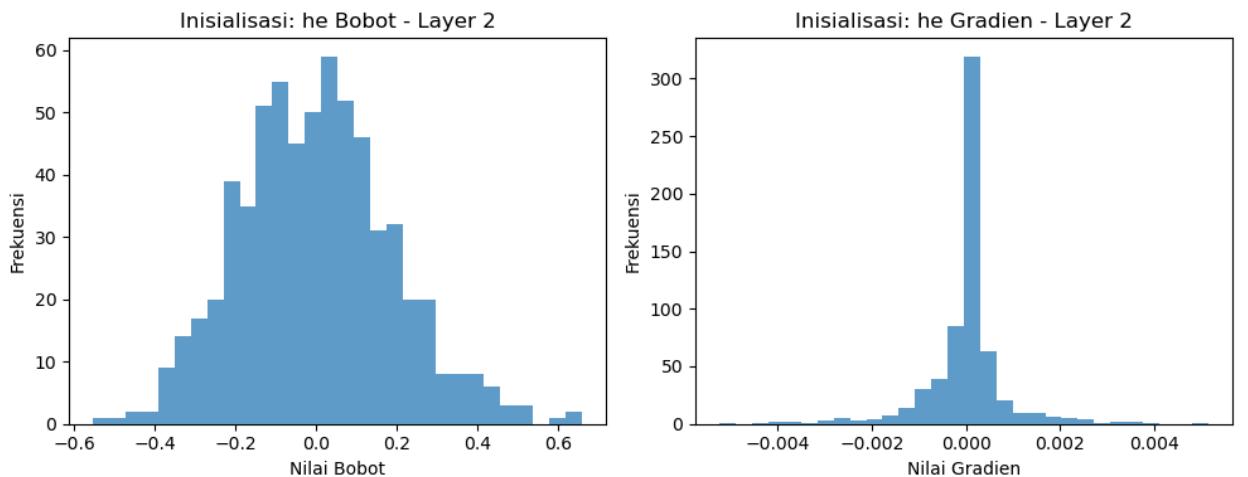
Gambar 2.2.4.7. Perbandingan Distribusi Bobot dan Gradien pada *Layer 1 (normal)*



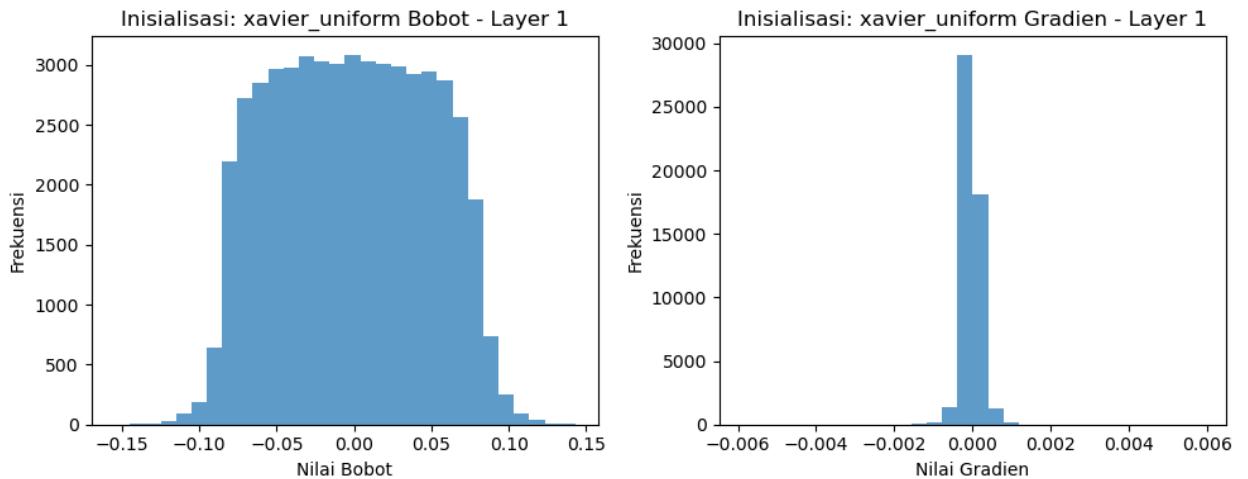
Gambar 2.2.4.8. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*normal*)



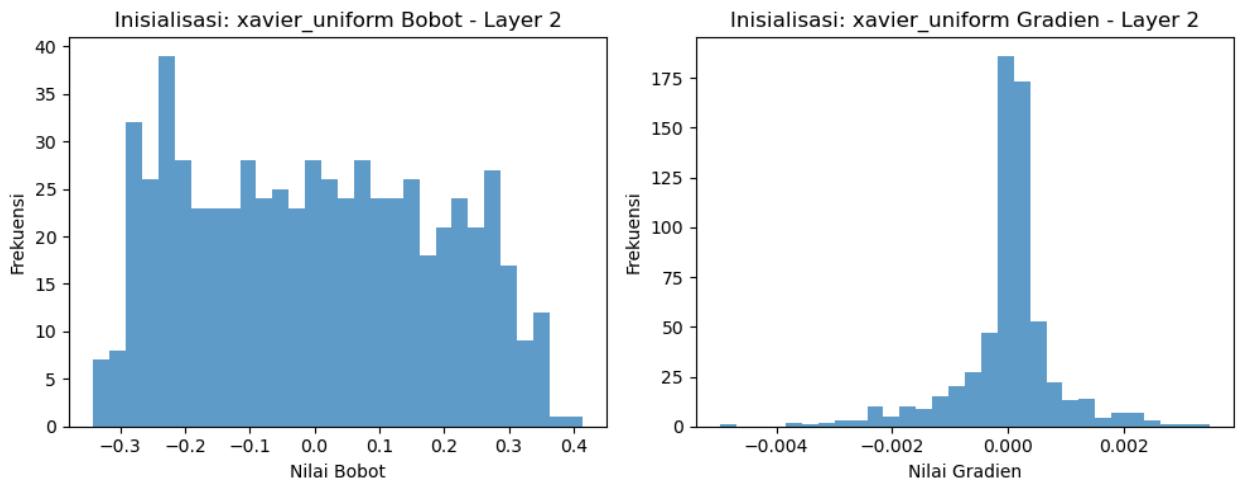
Gambar 2.2.4.9. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*he*)



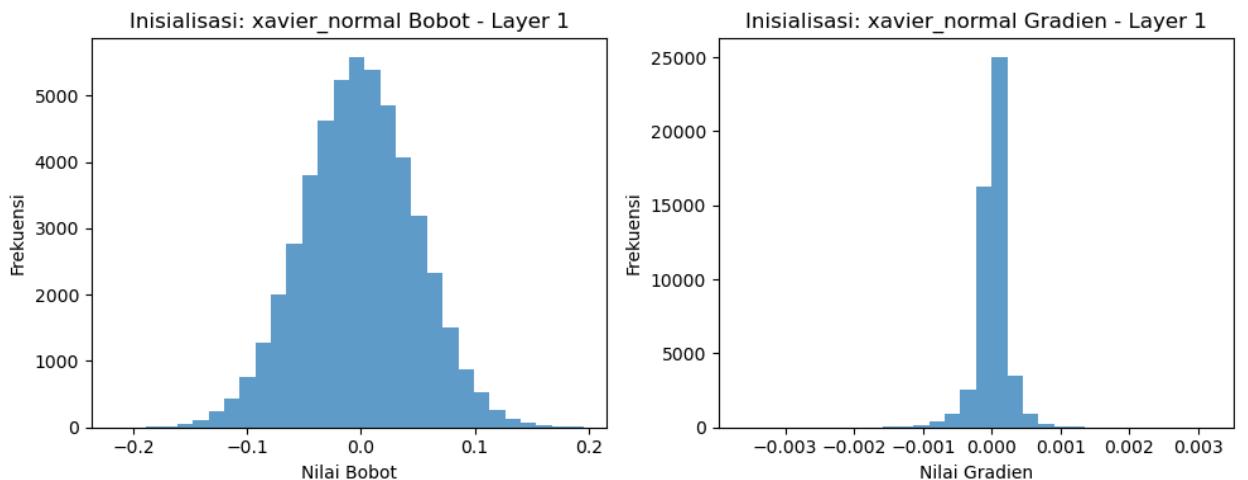
Gambar 2.2.4.10. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*he*)



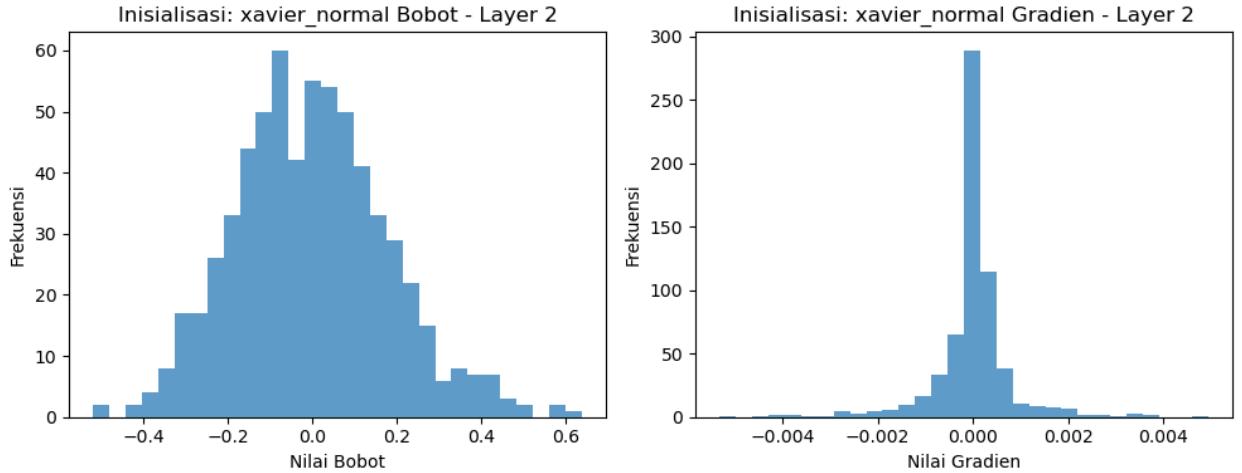
Gambar 2.2.4.11. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*Xavier Uniform*)



Gambar 2.2.4.12. Perbandingan Distribusi Bobot dan Gradien pada Layer 2 (*Xavier Uniform*)



Gambar 2.2.4.13. Perbandingan Distribusi Bobot dan Gradien pada Layer 1 (*Xavier Normal*)



Gambar 2.2.4.14. Perbandingan Distribusi Bobot dan Gradien pada *Layer 2 (Xavier Normal)*

2.2.5 Pengaruh Regularisasi

Dalam proses pelatihan jaringan saraf tiruan, sering terjadi *overfitting*, yaitu kondisi di mana model sangat baik dalam mempelajari data pelatihan tetapi kurang mampu menggeneralisasi ke data yang belum pernah dilihat sebelumnya. Salah satu penyebabnya adalah tingginya kompleksitas model serta nilai bobot yang besar. Untuk mengatasi masalah ini, teknik regularisasi sangat penting digunakan guna membatasi nilai bobot dan mencegah model menjadi terlalu rumit. Teknik *regularisation*, seperti L1 dan L2, menambahkan penalti pada fungsi *loss* berdasarkan besarnya bobot. Regularisasi L1 cenderung menghasilkan model dengan bobot yang lebih jarang (*sparse*), sedangkan regularisasi L2 membantu menjaga agar bobot tetap kecil tanpa memaksa banyak bobot menjadi nol. Dengan menerapkan teknik ini, diharapkan model tidak hanya mampu mempelajari pola dari data pelatihan, tetapi juga memiliki kemampuan generalisasi yang lebih baik pada data baru.

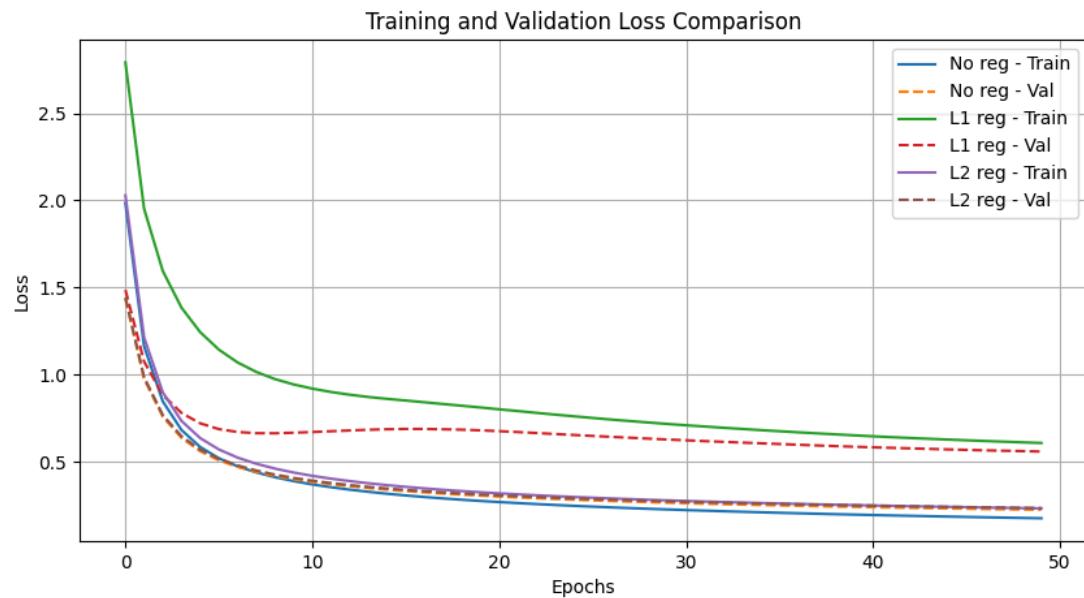
Implementasi regularisasi dalam model FFNN di atas dilakukan secara terintegrasi dalam proses *backward propagation*. Pada saat inisialisasi, model menerima parameter *reg_type* yang menentukan jenis regularisasi (L1 atau L2) serta *lambda_reg* sebagai nilai *hyperparameter* penalti. Selama proses perhitungan gradien, setelah gradien terhadap bobot dihitung, jika *reg_type* diatur ke L1, maka gradien tersebut ditambahkan dengan penalti berupa $(\lambda_{reg} \times sign(w)) / shape(x)$ untuk setiap bobot. Penalti ini

mendorong banyak bobot mendekati nol, sehingga menghasilkan model dengan bobot yang lebih jarang (*sparse*). Sebaliknya, jika *reg_type* diatur ke L2, gradien diperbarui dengan menambahkan $(\lambda \text{reg} \times w) / \text{shape}(x)$, yang berfungsi menjaga agar nilai bobot tidak terlalu besar tanpa secara eksplisit memaksa banyak bobot menjadi nol. Dengan cara ini, pembaruan bobot selama *training* tidak hanya dipengaruhi oleh *error* yang dihitung dengan fungsi *loss*, tetapi juga oleh penalti regularisasi, sehingga membantu mengurangi *overfitting* dan meningkatkan kemampuan generalisasi model terhadap data baru.

Untuk mengevaluasi pengaruh penggunaan regularisasi, dilakukan eksperimen dengan menggunakan 3 konfigurasi model yaitu Tanpa Regulasi, Regularisasi L1, Regularisasi L2. Metodologi eksperimen yang diterapkan meliputi:

- Dataset: Digunakan *dataset MNIST* yang berisi gambar tulisan tangan digit.
- Data Processing: Data fitur dinormalisasi menggunakan *StandardScaler* agar setiap fitur memiliki skala yang sama. Selanjutnya, label diubah ke format *one-hot encoding* dengan *OneHotEncoder*. Dataset dibagi dengan perbandingan 80% untuk data pelatihan dan 20% untuk data pengujian.
- Struktur Jaringan: Pada eksperimen ini, model FFNN dibangun dengan struktur [784, 64, 64, 10], di mana 784 adalah jumlah *neuron* pada *layer input*, 64 dan 64 adalah jumlah *neuron* pada *hidden layer* 1 dan 2, dan 10 adalah jumlah *neuron* pada *output layer*.
- Fungsi Aktivasi: Fungsi aktivasi yang digunakan adalah *ReLU* untuk *hidden layer* dan *softmax* untuk *output layer*.
- Inisiasi Bobot: Bobot diinisiasi menggunakan metode inisiasi bobot *He*.
- Perbandingan Model: Tiga konfigurasi model dibandingkan, yaitu model tanpa *Regularisasi* (*reg_type = None*, *lambda_reg = 0*), model dengan *Regularisasi L1* (*reg_type = L1*, *lambda_reg = 0.01*) dan model dengan *Regularisasi L2* (*reg_type = L2*, *lambda_reg = 0.01*).
- Parameter Pelatihan: Parameter pelatihan yang digunakan meliputi *batch size* 32, jumlah *epoch* 50, dan *learning rate* 0.01.
- Evaluasi Performa: Membandingkan grafik *loss* per *epoch*, akurasi prediksi, distribusi bobot dan gradien, serta visualisasi struktur jaringan.

Hasil Perbandingan *Loss* pada Pelatihan dan Validasi:



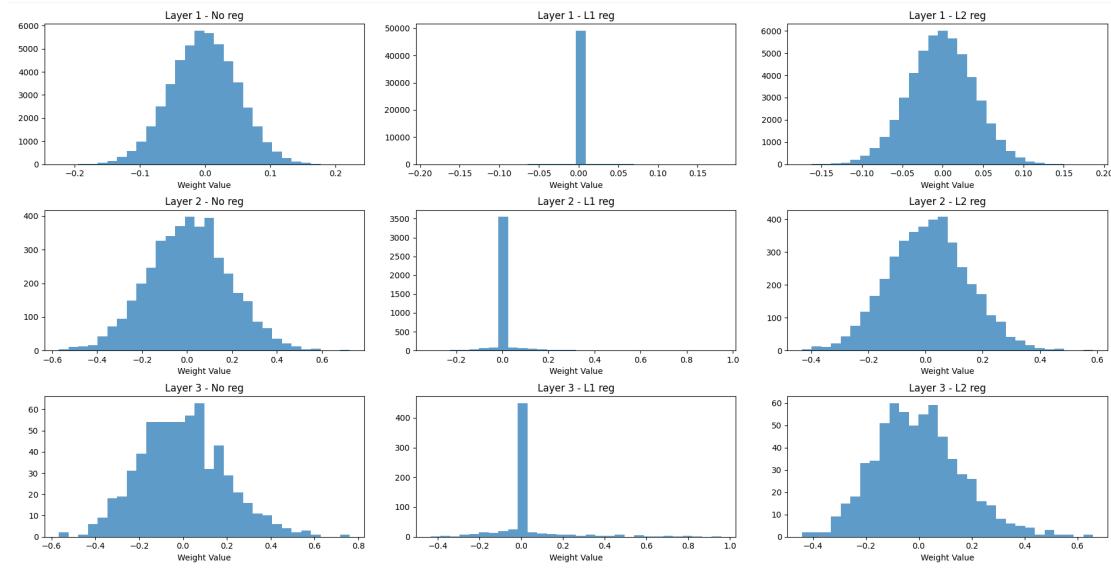
Gambar 2.2.5.1. Perbandingan Loss Pelatihan dan Validasi

Hasil Perbandingan Akurasi Hasil Prediksi:

```
Comparing prediction results:  
No regularization: Accuracy = 0.9367  
L1 regularization: Accuracy = 0.8714  
L2 regularization: Accuracy = 0.9349
```

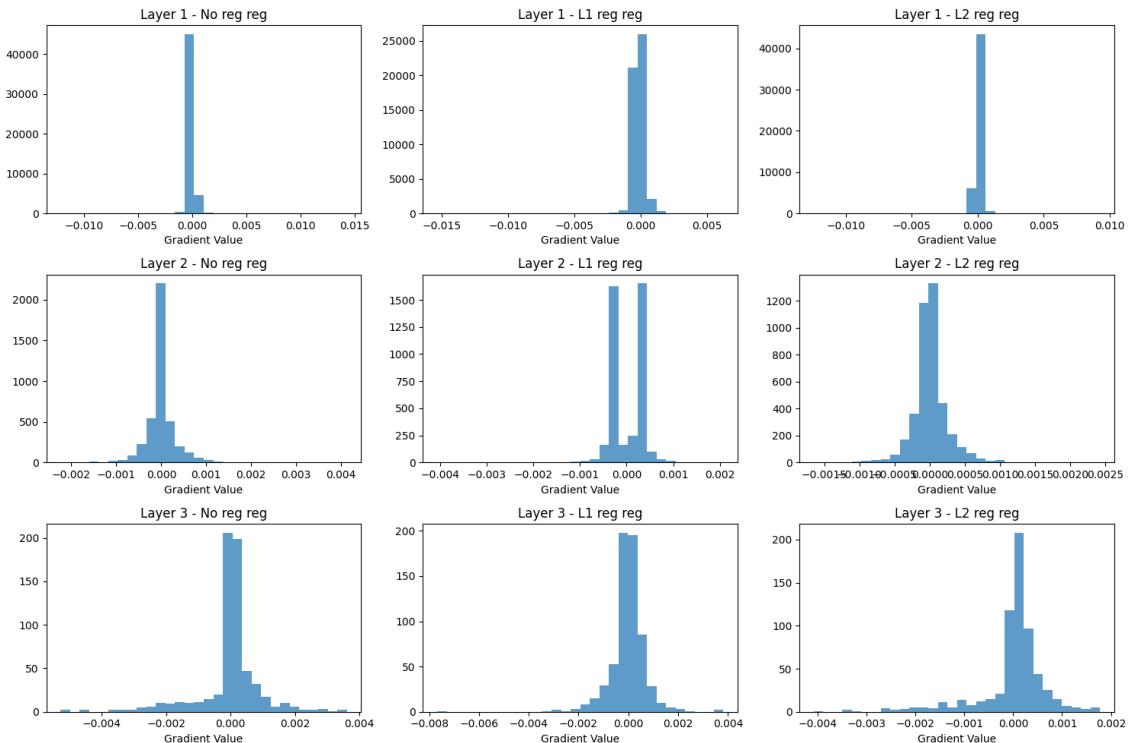
Gambar 2.2.5.2. Perbandingan Hasil Akurasi

Hasil Perbandingan bobot di setiap *layer* untuk ketiga model:



Gambar 2.2.5.3. Perbandingan Distribusi Bobot di Setiap *Layer*

Hasil Perbandingan Gradien di setiap *layer* untuk ketiga model:



Gambar 2.2.5.4. Perbandingan Distribusi Gradien di Setiap *Layer*

Pada grafik *loss* di atas (Gambar 2.2.5.1), model tanpa regularisasi (No reg) menurun dengan cepat pada data pelatihan (*train loss*) dan tetap cukup rendah pada data validasi (*val loss*), menandakan bahwa model mampu mempelajari pola dengan efektif. Sementara itu, model dengan L1 regularisasi menampilkan *train loss* dan *val loss* yang lebih tinggi, kemungkinan karena penalti L1 menekan bobot secara agresif sehingga menurunkan kapasitas model dan memicu *underfitting*. Di sisi lain, model dengan L2 regularisasi menunjukkan penurunan *train loss* yang stabil dan *val loss* yang mendekati model tanpa regularisasi, mengindikasikan bahwa penalti L2 dapat menjaga bobot tetap kecil tanpa terlalu mengurangi fleksibilitas model. Hal ini menegaskan bahwa pemilihan jenis dan besaran penalti regularisasi harus disesuaikan dengan karakteristik data serta tujuan pelatihan agar performa model dapat dioptimalkan.

Dari segi akurasi pada data uji (Gambar 2.2.5.2), model tanpa regularisasi berhasil meraih akurasi sekitar 0,9367, disusul oleh model dengan L2 regularisasi di kisaran 0,9349, sedangkan model L1 regularisasi hanya mencapai 0,8714. Perbedaan yang cukup besar pada model L1 menunjukkan bahwa penggunaan penalti L1 yang tidak diatur dengan tepat dapat membuat model kehilangan fleksibilitasnya dalam mengenali pola.

Berdasarkan histogram bobot di setiap *layer* (Gambar 2.2.5.3), model tanpa regularisasi (No reg) memiliki distribusi bobot yang lebih lebar di sekitar nol, menandakan ketiadaan penalti yang membatasi nilai bobot. Pada model L1, distribusi bobot membentuk puncak tajam di sekitar nol, terutama di *layer* kedua dan ketiga, yang menunjukkan banyak bobot ditekan mendekati nol (*sparse*). Sementara itu, model L2 menampilkan distribusi bobot yang lebih terpusat di sekitar nol, tetapi tidak setajam L1, menandakan bahwa penalti L2 membatasi nilai bobot agar tidak terlalu besar tanpa memaksa terlalu banyak bobot menjadi nol.

Beralih ke histogram gradien (Gambar 2.2.5.4), model tanpa regularisasi (No reg) menampilkan sebaran gradien yang relatif moderat dan simetris di sekitar nol. Pada model dengan L1 regularisasi, tampak puncak tajam di sekitar nol, menandakan penalti L1 mendorong banyak gradien menuju nol sehingga bobot cenderung diperbarui secara *sparse*. Sementara itu, model L2 menampilkan sebaran gradien yang lebih sempit namun tidak seekstrim L1, mencerminkan bahwa penalti L2 menjaga nilai gradien agar tetap kecil tanpa memaksa terlalu banyak gradien menjadi nol.

Secara keseluruhan, eksperimen ini menegaskan pentingnya regularisasi dalam mengendalikan kompleksitas model. Walaupun model tanpa regularisasi dapat meraih akurasi yang tinggi, model dengan L2 regularisasi memberikan performa hampir setara dengan kestabilan yang lebih baik. Model L1 regularisasi dapat menurunkan akurasi jika penalti (λ) terlalu besar, karena menekan terlalu banyak bobot menjadi nol. Regularisasi sangat cocok untuk situasi di mana data terbatas, risiko *overfitting* tinggi, atau ketika kita menginginkan model yang lebih sederhana dan mudah diinterpretasikan (khususnya L1). Kelebihan L2 adalah kemampuannya menjaga bobot tetap kecil tanpa memaksa terlalu banyak bobot menjadi nol, sementara L1 unggul dalam mendorong *sparsity*. Namun, pemilihan dan penyesuaian *hyperparameter* λ menjadi kunci untuk menghindari *underfitting* (pada L1) atau tetap berisiko *overfitting* (jika λ terlalu kecil). Dengan demikian, diperlukan penyesuaian yang tepat agar model dapat mencapai performa optimal.

2.2.6 Pengaruh Normalisasi *RMSNorm*

Normalisasi dalam jaringan saraf berperan penting dalam mengatasi masalah seperti gradien yang menghilang atau meledak selama proses pelatihan. *RMSNorm* (*Root Mean Square Normalization*) merupakan salah satu teknik normalisasi yang tidak bergantung pada statistik *batch*, sehingga cocok digunakan pada skenario dengan *batch size* kecil atau data *streaming*. Dengan menerapkan *RMSNorm*, nilai pre-aktivasi pada setiap *hidden layer* dinormalisasi berdasarkan nilai *RMS*, sehingga distribusi aktivasi menjadi lebih konsisten dan propagasi gradien pun menjadi lebih stabil. Dalam penelitian ini, *RMSNorm* diintegrasikan ke dalam model FFNN sebagai upaya untuk meningkatkan stabilitas dan kecepatan konvergensi selama training.

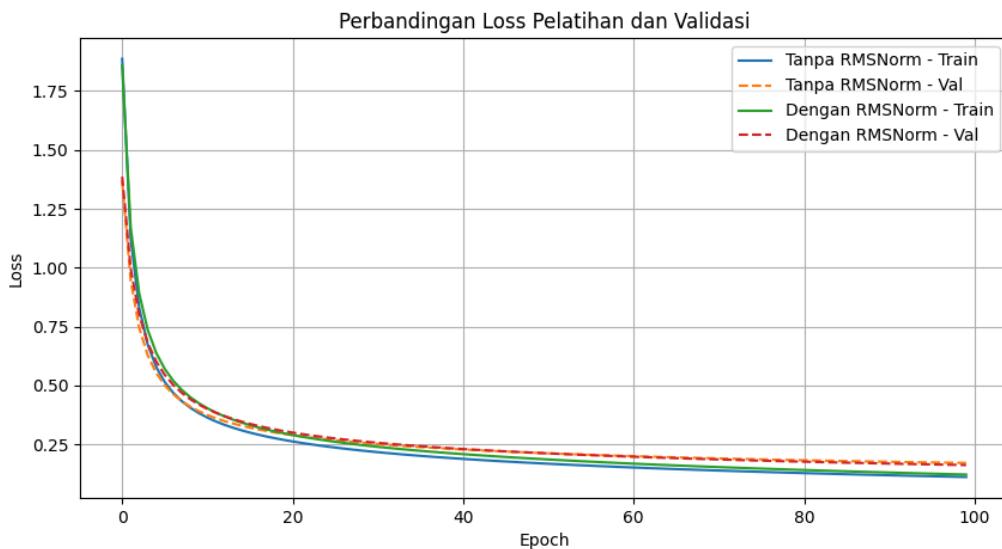
Dalam model FFNN yang dikembangkan, ketika opsi *RMSNorm* diaktifkan (`use_rmsnorm=True`), parameter *gamma* diinisialisasi untuk setiap hidden layer guna mengkalibrasi hasil normalisasi. Pada proses *forward pass*, nilai pre-aktivasi (*z*) yang dihasilkan dari operasi linier dinormalisasi dengan cara menghitung nilai akar rata-rata kuadrat (*RMS*) dari *z* dengan tambahan *epsilon* untuk menjaga stabilitas numerik,

kemudian membagi z dengan RMS tersebut, dan akhirnya mengalikan hasilnya dengan gamma agar skala nilai dapat disesuaikan. Selanjutnya, pada tahap backward pass, selain menghitung gradien terhadap bobot dan bias, model juga menghitung gradien untuk parameter gamma dengan memperhitungkan kontribusi nilai pre-aktivasi asli (z_{raw}) dan RMS . Dengan demikian, pembaruan parameter gamma dapat membantu menjaga kestabilan distribusi aktivasi dan gradien, yang pada akhirnya mengurangi risiko terjadinya gradien yang meledak atau menghilang, sehingga pelatihan model menjadi lebih stabil dan efisien.

Untuk mengevaluasi pengaruh penggunaan $RMSNorm$, dilakukan eksperimen dengan dua konfigurasi model, yaitu model tanpa $RMSNorm$ dan model dengan $RMSNorm$. Parameter eksperimen yang diterapkan meliputi:

- Dataset: Digunakan dataset *MNIST* yang berisi gambar tulisan tangan digit.
- Data Processing: Data fitur dinormalisasi menggunakan *StandardScaler* agar setiap fitur memiliki skala yang sama. Selanjutnya, label diubah ke format *one-hot encoding* dengan *OneHotEncoder*. Dataset kemudian dibagi dengan perbandingan 80% untuk data pelatihan dan 20% untuk data pengujian.
- Struktur Jaringan: Pada eksperimen ini, model FFNN dibangun dengan struktur [784, 128, 64, 10], di mana 784 merupakan jumlah *neuron* pada *layer input*, 128 dan 64 adalah jumlah *neuron* pada *hidden layer*, dan 10 adalah jumlah *neuron* pada *output layer*.
- Fungsi Aktivasi: Fungsi aktivasi yang digunakan adalah *ReLU* untuk *hidden layer* dan *softmax* untuk *output layer*.
- Inisiasi Bobot: Bobot diinisiasi menggunakan metode inisiasi bobot *He*.
- Perbandingan Model: Dua konfigurasi model dibandingkan, yaitu model tanpa $RMSNorm$ (*use_rmsnorm = False*) dan model dengan $RMSNorm$ (*use_rmsnorm = True*).
- Parameter Pelatihan: Parameter pelatihan yang digunakan meliputi batch size 32, jumlah epoch 100, dan learning rate 0.01.
- Evaluasi Performa: Evaluasi dilakukan dengan membandingkan grafik *loss* per *epoch*, akurasi prediksi, distribusi bobot dan gradien, serta visualisasi struktur jaringan.

Hasil Perbandingan Loss Pelatihan dan Validasi:



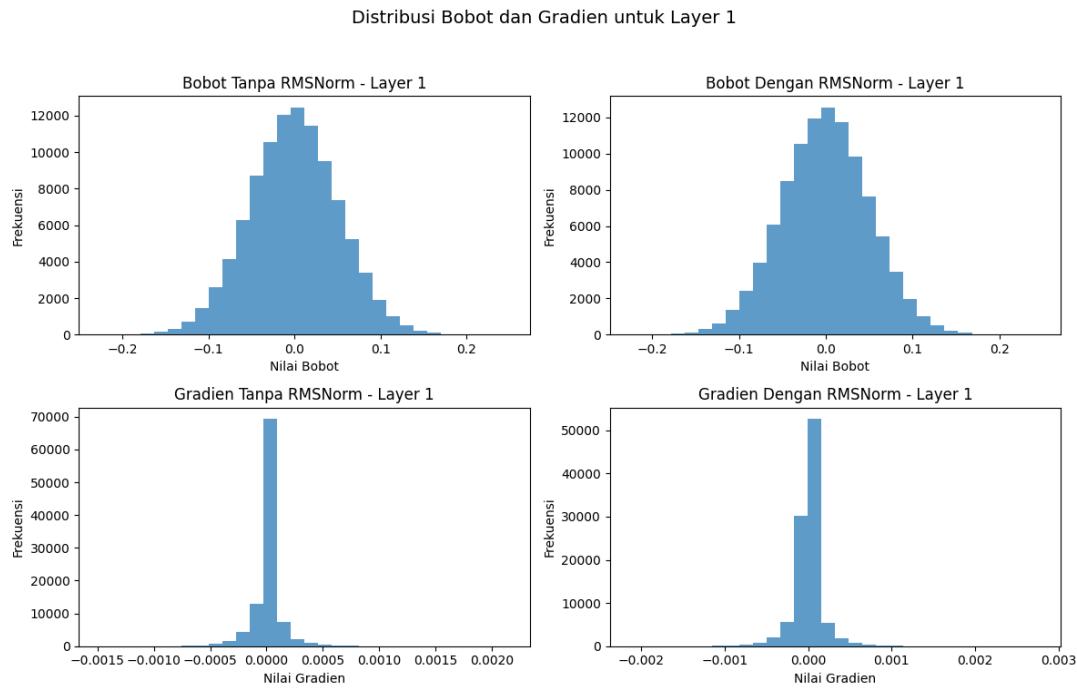
Gambar 2.2.6.1. Perbandingan Loss Pelatihan dan Validasi

Hasil Perbandingan Akurasi Hasil Prediksi:

```
Perbandingan hasil prediksi:  
Tanpa RMSNorm: Accuracy = 0.9534  
Dengan RMSNorm: Accuracy = 0.9529
```

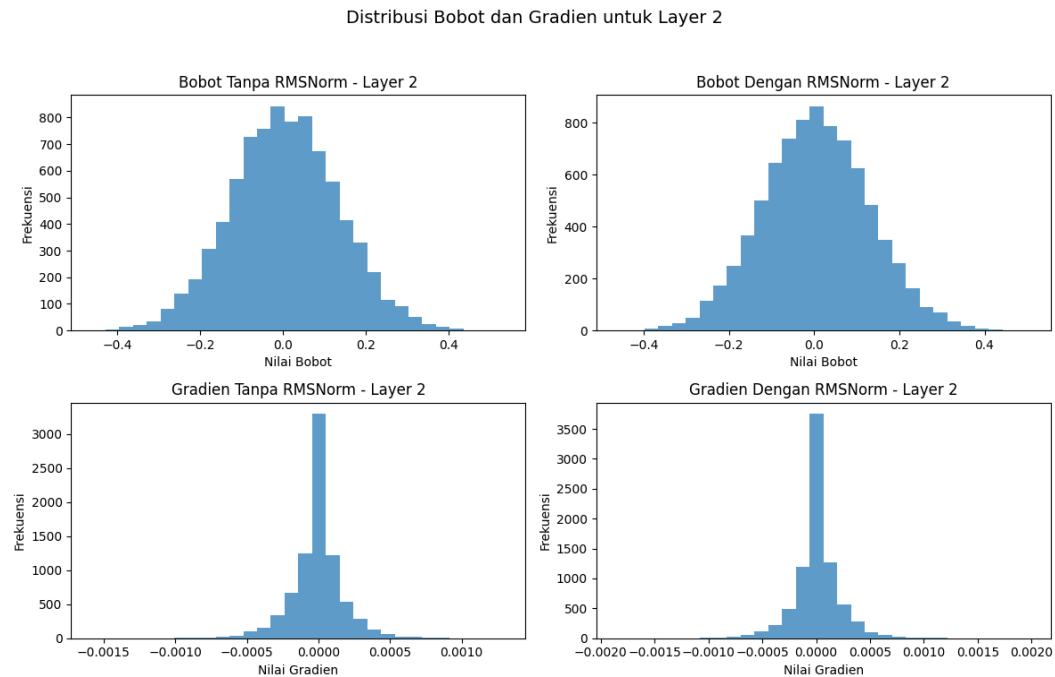
Gambar 2.2.6.2. Perbandingan Hasil Prediksi

Hasil Perbandingan Distribusi Bobot dan Gradien untuk *Layer 1*:



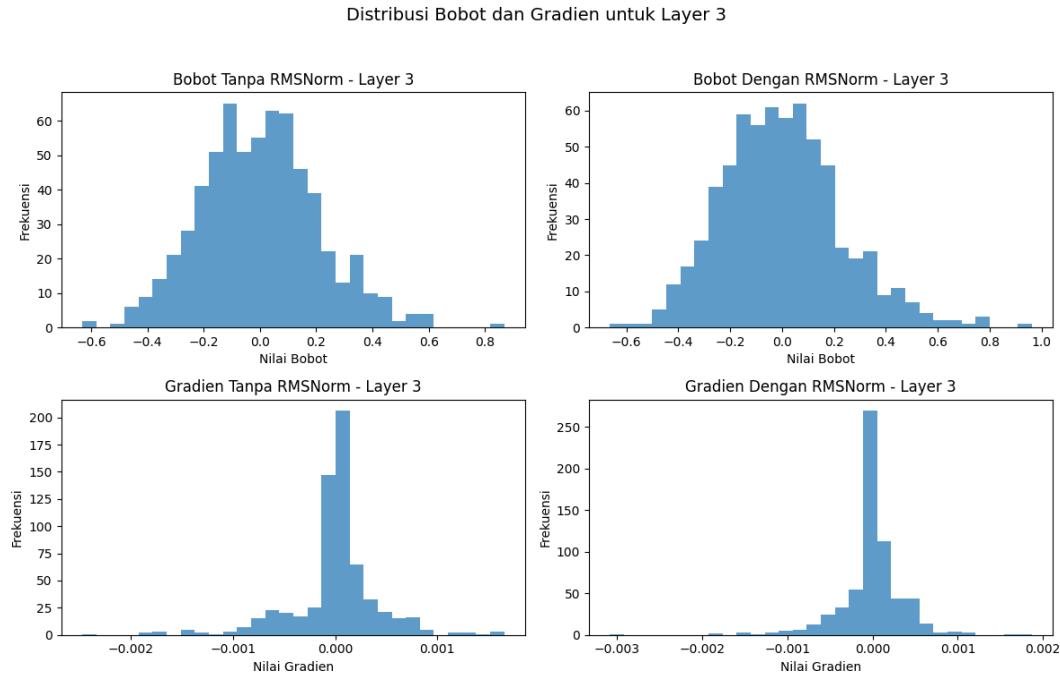
Gambar 2.2.6.3. Perbandingan Distribusi Bobot dan Gradien pada *Layer 1*

Hasil Perbandingan Distribusi Bobot dan Gradien untuk *Layer 2*:



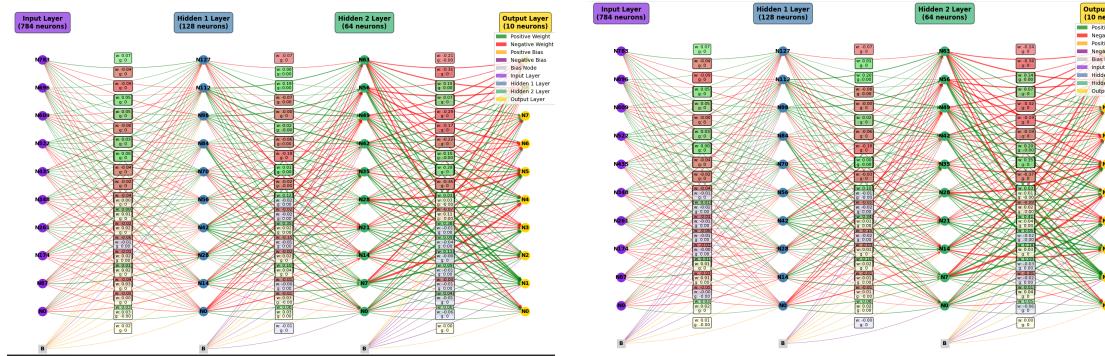
Gambar 2.2.6.4. Perbandingan Distribusi Bobot dan Gradien pada *Layer 2*

Hasil Perbandingan Distribusi Bobot dan Gradien untuk Layer 3:



Gambar 2.2.6.5. Perbandingan Distribusi Bobot dan Gradien pada Layer 3

Graf Model FFNN:



Gambar 2.2.6.6. Perbandingan Graf model FFNN tanpa RMSNorm dan dengan RMSNorm

Berdasarkan grafik *loss* pelatihan dan validasi (Gambar 2.2.6.1), kedua model baik tanpa *RMSNorm* maupun dengan *RMSNorm* menunjukkan pola penurunan *loss* yang serupa. Keduanya berhasil mencapai nilai *loss* yang relatif rendah pada akhir pelatihan. Hal ini menunjukkan bahwa penerapan *RMSNorm* tidak memberikan perbedaan signifikan dalam hal laju konvergensi maupun stabilitas *loss*, setidaknya pada konfigurasi

hyperparameter dan jumlah data pelatihan yang digunakan.

Dari segi akurasi pada data uji, model tanpa *RMSNorm* memperoleh akurasi sekitar 0,9534, sedangkan model dengan *RMSNorm* memperoleh akurasi 0,9529. Perbedaan yang sangat kecil ini mengindikasikan bahwa, untuk kasus MNIST dengan *batch size* 32, *RMSNorm* tidak memberikan peningkatan akurasi yang berarti. Meskipun demikian, hasil ini tidak serta-merta meniadakan manfaat *RMSNorm*; terdapat banyak studi yang menunjukkan bahwa *RMSNorm* dapat memberikan stabilitas pelatihan yang lebih baik, terutama pada skenario dengan *batch size* yang sangat kecil atau pada arsitektur model yang lebih dalam.

Dari distribusi bobot dan gradien (Gambar 2.2.6.3, 2.2.6.4, dan 2.2.6.5 untuk masing-masing layer), terlihat bahwa keduanya cenderung memiliki bentuk sebaran yang mirip, terutama berbentuk hampir simetris di sekitar nilai nol. *RMSNorm* memang mempengaruhi perhitungan *forward pass* dan *backward pass*, namun pada dataset MNIST yang relatif sederhana, efek normalisasi tidak terlalu kontras. Perbedaan yang lebih jelas mungkin akan terlihat pada tugas yang lebih kompleks atau ketika *batch size* sangat kecil, di mana stabilitas normalisasi akan lebih krusial.

Selain itu, visualisasi struktur jaringan (Gambar 2.2.6.6) tidak menunjukkan perbedaan yang kasatmata dari segi topologi, karena kedua model memiliki arsitektur yang sama. Perbedaan utama terletak pada parameter *gamma* yang digunakan *RMSNorm* untuk menyesuaikan skala aktivasi.

Secara umum, *RMSNorm* lebih relevan untuk membantu pelatihan tetap stabil pada arsitektur yang lebih dalam, skenario *batch size* kecil, atau data *streaming* yang tidak memungkinkan penggunaan Batch Normalization secara efektif. Dalam konteks MNIST dengan *batch size* menengah dan arsitektur yang tidak terlalu dalam, *RMSNorm* masih berfungsi dengan baik tetapi tidak memberikan peningkatan yang signifikan dibandingkan model tanpa normalisasi tersebut.

Dari hasil eksperimen ini, dapat disimpulkan bahwa:

1. Penerapan *RMSNorm* pada model FFNN dengan dataset MNIST dan *batch size* 32 tidak memberikan peningkatan akurasi yang berarti jika dibandingkan dengan model tanpa *RMSNorm*.

2. Distribusi bobot dan gradien pada kedua model cenderung mirip, mengindikasikan bahwa normalisasi *RMSNorm* tidak secara drastis mengubah sebaran parameter dalam kasus ini.
3. Walaupun pada eksperimen ini *RMSNorm* tidak menunjukkan perbedaan yang signifikan, metode ini tetap relevan dan dapat menjadi alternatif bagi *Batch Normalization*, terutama untuk kondisi *batch size* kecil atau arsitektur yang lebih dalam, di mana stabilitas dan kecepatan konvergensi menjadi lebih krusial.

2.2.7 Perbandingan dengan *Library Sklearn*

Untuk menilai performa model FFNN yang dikembangkan secara *custom*, dilakukan perbandingan dengan model *MLPClassifier* dari *library scikit-learn (sklearn)*. Keduanya sama-sama menggunakan parameter utama yang serupa, seperti jumlah neuron pada *hidden layer*, fungsi aktivasi *ReLU*, serta *learning rate* dan *batch size* yang sebanding. Langkah Eksperimen:

1. Inisialisasi Model Custom FFNN

Model FFNN didefinisikan dengan susunan layer [784,64,10], menggunakan fungsi aktivasi *ReLU* di *hidden layer* dan softmax di layer output. Bobot diinisialisasi dengan metode *He initialization*. Proses pelatihan dilakukan dengan *loss categorical_crossentropy_loss*, menggunakan *stochastic gradient descent* (SGD) sederhana selama 20 *epoch* dengan *learning rate* sebesar 0.01 dan *batch size* 32.

2. Inisialisasi Model *sklearn MLPClassifier*

Model *MLPClassifier* dari *sklearn* juga dikonfigurasi dengan satu *hidden layer* berukuran 64 *neuron*, fungsi aktivasi *ReLU*, *solver SGD*, *learning_rate_init* 0.01, dan *max_iter* 20. Hal ini dilakukan untuk menjaga kesetaraan kondisi pelatihan agar hasil perbandingan menjadi lebih adil.

3. Pelatihan dan Evaluasi

- Custom FFNN:

Model custom FFNN dilatih dengan data latih (*training set*), lalu

dilakukan evaluasi pada data validasi (*validation set*). Hasil pelatihan menampilkan akurasi akhir di sekitar 0,9077.

- **MLPClassifier:**

Model *sklearn MLPClassifier* dilatih dengan data yang sama dan dievaluasi pada data validasi. Akurasi yang diperoleh mencapai sekitar 0,9661.

```
Melatih model FFNN custom...
Akurasi FFNN custom: 0.9077
Melatih model sklearn MLPClassifier...
Akurasi sklearn MLPClassifier: 0.9661
Perbandingan prediksi (10 data pertama):
FFNN custom : [8 4 8 7 7 0 6 2 7 4]
sklearn MLP  : [8 4 8 7 7 0 6 2 7 4]
```

Gambar 2.2.7.1. Hasil Perbandingan Custom FFNN dan Library Sklearn

4. Perbandingan Hasil Prediksi

Untuk memberikan gambaran lebih jelas, ditampilkan juga hasil prediksi 10 data pertama dari set validasi. Dari hasil tersebut, dapat terlihat bahwa secara umum kedua model memberikan prediksi yang serupa. Tetapi jika ditampilkan keseluruhan data, maka akan ada perbedaan hasil prediksi.

Analisis Perbandingan

Hasil menunjukkan bahwa model *MLPClassifier* memiliki akurasi yang lebih tinggi (0,9661) dibandingkan dengan model custom FFNN (0,9077). Selisih akurasi tersebut kemungkinan besar disebabkan oleh beberapa faktor, di antaranya:

- Implementasi *Optimizer*:

MLPClassifier secara default menerapkan berbagai teknik optimasi dan *parameter tuning* (misalnya, *momentum*, *learning rate schedule*, dsb.) yang mungkin lebih canggih dibandingkan implementasi SGD sederhana pada model custom FFNN.

- Regularisation Otomatis:

MLPClassifier secara otomatis menerapkan beberapa bentuk regularisasi (misalnya, *L2 penalty*) yang dapat membantu model menghindari *overfitting* dan meningkatkan generalisasi. Jadi meskipun di percobaan kali ini untuk *reg_type* dimatikan, namun *MLPClassifier* menerapkan regularisasi secara otomatis.

- Hyperparameter Tuning:

Pada *MLPClassifier*, beberapa *hyperparameter* dapat diatur secara otomatis atau lebih mudah dikustomisasi. Sementara itu, pada model custom, proses tuning mungkin lebih terbatas sehingga mempengaruhi performa akhir. Sehingga meskipun pada percobaan ini sudah ditentukan parameter yang sama untuk kedua model, namun ada banyak *hyperparameter* lain yang bisa secara otomatis diatur oleh sklearn.

Perbandingan ini menunjukkan bahwa model FFNN custom memiliki performa yang kompetitif, namun masih di bawah sklearn *MLPClassifier* pada konfigurasi dasar. Meskipun demikian, hasil akurasi yang diperoleh model custom sudah tergolong tinggi dan cukup memadai untuk *training*.

BAB III

KESIMPULAN DAN SARAN

3.1 Kesimpulan

Dari hasil eksperimen dan analisis yang telah dilakukan, dapat disimpulkan bahwa implementasi FFNN secara from scratch berhasil menampilkan performa yang kompetitif pada dataset *MNIST*. Peningkatan *width* dan *depth* memberikan kontribusi positif terhadap akurasi, meskipun peningkatan *width* cenderung menghasilkan kenaikan yang lebih signifikan dibandingkan dengan penambahan *depth*, yang diiringi dengan tantangan peningkatan kompleksitas komputasi dan potensi *vanishing gradient*. Eksperimen fungsi aktivasi menunjukkan bahwa fungsi non-linear seperti *ReLU* dan variannya (*leaky_relu*, *ELU*) mengatasi masalah *vanishing gradient* dengan lebih efektif dibandingkan dengan *sigmoid* atau *tanh* atau linear, yang mengalami saturasi sehingga menghambat pembelajaran. Pengaturan *learning rate* juga memainkan peran penting, di mana nilai yang terlalu kecil menyebabkan konvergensi lambat, sedangkan nilai yang lebih besar (misalnya 0,1) dapat mempercepat pembelajaran jika masih dalam rentang stabil. Selain itu, metode inisialisasi bobot yang tepat seperti *He* dan *Xavier* terbukti mendukung aliran gradien dan mempercepat konvergensi, sedangkan inisialisasi *zero* tidak mampu memecah simetri neuron. Implementasi regularisasi, baik L1 maupun L2, menunjukkan bahwa penalti yang terlalu kuat dapat menyebabkan *underfitting*, namun regularisasi L2 cenderung memberikan keseimbangan antara kontrol *overfitting* dan fleksibilitas model. Penerapan *RMSNorm* pada eksperimen dengan batch size sedang dan jaringan dangkal tidak memberikan peningkatan yang signifikan, namun metode ini tetap relevan untuk skenario dengan *batch* kecil atau arsitektur yang lebih dalam. Secara keseluruhan, perbandingan dengan library `sklearn` *MLPClassifier* juga menegaskan bahwa meskipun model custom memiliki performa yang baik, optimasi tambahan yang diterapkan oleh *library* tersebut (seperti penggunaan optimizer dan momentum) dapat menghasilkan akurasi yang sedikit lebih tinggi. Model custom juga perlu disesuaikan hyperparameternya untuk dataset tertentu agar prediksi dapat menjadi lebih akurat.

3.2 Saran

Untuk perbaikan, disarankan agar eksperimen lebih lanjut dilakukan dengan menguji model pada dataset yang lebih kompleks dan arsitektur jaringan yang lebih dalam, sehingga manfaat teknik normalisasi seperti *RMSNorm* dan regularisasi lanjutan dapat lebih optimal diterapkan. Selain itu, penerapan metode optimasi yang lebih canggih, seperti *Adam Optimizer* atau penggunaan momentum yang lebih terintegrasi, perlu dieksplorasi untuk mempercepat konvergensi dan meningkatkan performa model. Pemanfaatan teknik *hyperparameter* tuning secara otomatis juga akan membantu menemukan konfigurasi optimal yang dapat mengatasi *trade-off* antara kompleksitas model dan risiko *overfitting*. Pendekatan-pendekatan ini diharapkan dapat memberikan kontribusi signifikan dalam pengembangan model FFNN yang lebih *robust* dan efisien di masa mendatang.

BAB IV

LAMPIRAN

4.1 Pembagian Tugas

Nama/NIM	Kontribusi
Filbert (13522021)	Laporan, L1 dan L2 Regularization, bonus activation function, eksperiment
Benardo (13522055)	Laporan, RMS Norm, bonus initialization method, eksperiment
William Glory Henderson (13522113)	Laporan, FFNN class, activation and loss function, initialization method, eksperiment

BAB VII

REFERENSI

<https://www.geeksforgeeks.org/what-is-forward-propagation-in-neural-networks/>

<https://www.jasonosajima.com/forwardprop>

<https://www.jasonosajima.com/backprop>

https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2022/tutorials/tut01.pdf

<https://www.geeksforgeeks.org/training-and-validation-loss-in-deep-learning/>