



Laporan Tugas Besar #1

Feedforward Neural Network

Disusun Oleh:
Kelompok 1

Anggota:

1. Marzuli Suhada M - 13522070
2. Ahmad Mudabbir Arif - 13522072
3. Naufal Adnan - 13522116

DAFTAR ISI

DAFTAR ISI.....	2
DAFTAR GAMBAR.....	3
DAFTAR TABEL.....	4
I. DESKRIPSI PERSOALAN.....	5
II. PEMBAHASAN.....	6
1. Penjelasan Implementasi.....	6
A. Deskripsi Kelas, Atribut, dan Method.....	6
B. Forward Propagation.....	12
C. Backward Propagation dan Weight Update.....	13
2. Hasil Pengujian.....	15
A. Pengaruh Depth dan Width.....	15
A.1. Pengaruh Width.....	15
A.2. Pengaruh Depth.....	16
A.3. Analisis Keseluruhan.....	16
B. Pengaruh Fungsi Aktivasi.....	19
C. Pengaruh Learning Rate.....	26
D. Pengaruh Inisialisasi Bobot.....	30
E. Pengaruh Regularisasi.....	36
F. Pengaruh Normalisasi RMSNorm.....	40
G. Perbandingan dengan Library Sklearn.....	44
H. Visualisasi Arsitektur.....	49
III. KESIMPULAN DAN SARAN.....	50
Kesimpulan.....	50
Saran.....	51
IV. PEMBAGIAN TUGAS.....	53
V. REFERENSI.....	56

DAFTAR GAMBAR

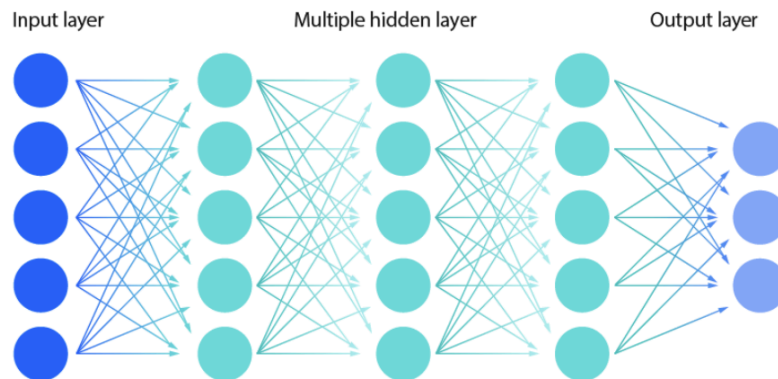
Gambar 2.1.1. Learning Curves Width Variation (Kiri) dan Depth Variation (Kanan).....	19
Gambar 2.2.1. Learning Curves Activation Functions.....	22
Gambar 2.3.1. Learning Curves Variasi Nilai Learning Rate.....	28
Gambar 2.4.1. Learning Curves - Initialization Methods.....	33
Gambar 2.5.1. Learning Curves - Regularization Effects.....	38
Gambar 2.6.1. Learning Curves - Normalization RMSNorm.....	42
Gambar 2.7.1. Perbandingan Distribusi Kelas Prediksi dari Kedua Model.....	46
Gambar 2.7.2. Learning Curve Berupa Training Loss dari Kedua Model.....	47
Gambar 2.8.1. Visualisasi Arsitektur FFNN.....	49

DAFTAR TABEL

Tabel 1.1. Atribut Kelas FFNN.....	7
Tabel 1.2. Method Kelas FFNN.....	8
Tabel 1.3. Method Kelas Activation.....	9
Tabel 1.4. Method Kelas Loss.....	10
Tabel 1.5. Method Kelas Initializer.....	11
Tabel 2.1.1. Perbandingan Loss Tiap Arsitektur.....	16
Tabel 2.2.1. Perbandingan Accuracy dan Loss Setiap Fungsi Aktivasi.....	20
Tabel 2.2.2. Perbandingan Weight dan Gradient Setiap Fungsi Aktivasi.....	23
Tabel 2.3.1. Perbandingan Accuracy dan Loss Setiap Learning Rate.....	27
Tabel 2.3.2. Perbandingan Weight dan Gradient Setiap Variasi Nilai Learning Rate.....	29
Tabel 2.4.1 Perbandingan Accuracy dan Loss Setiap Learning Rate.....	31
Tabel 2.4.2. Perbandingan Weight dan Gradient Setiap Variasi Initialization Methods.....	34
Tabel 2.5.1. Perbandingan Accuracy dan Loss Setiap Konfigurasi Regularisasi.....	37
Tabel 2.5.2. Perbandingan Weight dan Gradient Setiap Metode Regularisasi.....	39
Tabel 2.6.1. Perbandingan Accuracy dan Loss oleh Normalisasi RMSNorm.....	41
Tabel 2.6.2. Perbandingan Weight dan Gradient Terhadap Normalisasi.....	43
Tabel 2.7.1. Perbandingan Konfigurasi Model.....	45
Tabel 2.7.2. Perbandingan Accuracy Model.....	46

I. DESKRIPSI PERSOALAN

Feedforward Neural Network (FFNN) merupakan salah satu arsitektur fundamental dalam bidang *machine learning* dan *deep learning*. Dalam rangka memahami konsep dasar *machine learning* secara mendalam, mahasiswa ditugaskan untuk mengimplementasikan *Feedforward Neural Network* (FFNN) dari awal (*from scratch*) tanpa menggunakan *library* yang siap pakai.



Gambar 1.1 Arsitektur Feedforward Neural Network (FFNN)

Kami diminta untuk membuat implementasi FFNN yang mampu:

1. Menerima variasi jumlah *neuron* di setiap *layer*
2. Mendukung berbagai fungsi aktivasi (*Linear*, *ReLU*, *Sigmoid*, *Tanh*, *Softmax*)
3. Menggunakan fungsi *loss* berbeda (*MSE*, *Binary Cross-Entropy*, *Categorical Cross-Entropy*)
4. Menerapkan metode inisialisasi bobot (*Zero*, *Random Uniform*, *Random Normal*)

Model harus memiliki kemampuan:

1. Menyimpan bobot dan gradien setiap *neuron*
2. Menampilkan struktur jaringan dalam bentuk graf
3. Memvisualisasikan distribusi bobot dan gradien per *layer*
4. Melakukan *save* dan *load* model

Proses pelatihan harus mendukung:

1. *Input batch*
2. Propagasi maju dan balik
3. Perhitungan gradien menggunakan *chain rule*
4. Pembaruan bobot dengan *gradient descent*
5. Konfigurasi *hyperparameter*: *batch size*, *learning rate*, *epoch*, dan *verbose*

Untuk mengevaluasi kinerja model, dilakukan pengujian yang meliputi:

1. Analisis pengaruh *depth* dan *width* model
2. Evaluasi berbagai fungsi aktivasi
3. Studi dampak *learning rate*
4. Perbandingan metode inisialisasi bobot
5. Validasi dengan *library* scikit-learn MLP

Pengujian akan menggunakan dataset MNIST (*mnist_784*) yang akan dimuat menggunakan *method* `fetch_openml` dari scikit-learn.

Terdapat beberapa fitur bonus yang dapat diimplementasikan:

1. *Automatic Differentiation*
2. Implementasi fungsi aktivasi tambahan
3. Metode inisialisasi bobot lanjutan (Xavier, He)
4. Regularisasi L1 dan L2
5. Normalisasi layer dengan *Root Mean Square* (RMS)

II. PEMBAHASAN

1. Penjelasan Implementasi

A. Deskripsi Kelas, Atribut, dan Method

Implementasi program ini merupakan implementasi *Feedforward Neural Network* (FFNN) *from scratch*. Struktur implementasi ini dibagi menjadi empat kelas utama yang saling terintegrasi yaitu *class* **FFNN** sebagai kelas utama yang menangani jaringan saraf (*Neural Network*) secara keseluruhan, *class* **Activation** yang mengelola berbagai fungsi aktivasi, *class* **Loss** yang menyediakan fungsi-fungsi loss, dan *class* **Initializer** yang menangani inisialisasi bobot. Struktur modular ini memungkinkan kustomisasi yang fleksibel, kemudahan pengembangan, serta ekspansi fitur di masa depan.

Pada implementasi ini terdapat fitur tambahan seperti regularisasi L1/L2, normalisasi RMSNorm, dan *gradient clipping*, yang meningkatkan performa dan stabilitas model. Visualisasi jaringan, distribusi bobot, dan performa model juga diintegrasikan untuk memudahkan analisis dan pemahaman. Selain itu, model juga dapat disimpan dan dimuat kembali. Berikut adalah penjelasan detail untuk masing-masing kelas:

a. Kelas FFNN

Kelas FFNN adalah komponen utama yang mengimplementasikan arsitektur dan fungsi jaringan saraf *feedforward*. Kelas ini mengelola keseluruhan alur kerja *neural network*, mulai dari inisialisasi parameter hingga pelatihan dan prediksi. Implementasi FFNN dapat menangani struktur yang fleksibel dengan mendukung variasi jumlah *layer*, jumlah *neuron* per *layer*, dan fungsi aktivasi yang berbeda. Kelas ini menggunakan *library NumPy* untuk operasi matriks dan untuk pelatihan menggunakan *mini-batch gradient descent*. Fitur regularisasi L1 dan L2 diimplementasikan untuk mengurangi *overfitting*, sementara normalisasi *RMSNorm* dapat diaktifkan untuk stabilitas pelatihan. Implementasi juga mencakup visualisasi struktur jaringan dan distribusi parameter bobot dan gradien untuk membantu analisis model.

Tabel 1.1. Atribut Kelas FFNN

No	Atribut	Deskripsi
1.	<code>layers</code>	Daftar jumlah <i>neuron</i> pada setiap <i>layer</i> (termasuk <i>input</i> , <i>hidden</i> , dan <i>output</i>)
2.	<code>activation</code>	Daftar fungsi aktivasi untuk setiap layer
3.	<code>d_activation</code>	Daftar turunan fungsi aktivasi untuk setiap layer
4.	<code>loss</code>	Fungsi <i>loss</i> yang digunakan
5.	<code>d_loss</code>	Turunan dari fungsi <i>loss</i>
6.	<code>weights</code>	Daftar matriks bobot antar <i>layer</i>
7.	<code>biases</code>	Daftar bias untuk setiap <i>layer</i>
8.	<code>d_weights</code>	Daftar gradien bobot
9.	<code>d_biases</code>	Daftar gradien bias
10.	<code>l1_lambda</code>	Parameter regularisasi L1
11.	<code>l2_lambda</code>	Parameter regularisasi L2
12.	<code>rms_norm</code>	<i>Flag</i> untuk mengaktifkan normalisasi <i>RMSNorm</i>

13.	<code>gamma</code>	Parameter skala untuk <i>RMSNorm</i>
14.	<code>epsilon</code>	Konstanta untuk menghampiri nilai 0
15.	<code>a</code>	Menyimpan hasil setelah fungsi aktivasi diterapkan
16.	<code>z</code>	Menyimpan nilai <i>net</i> sebelum aktivasi dalam <i>neural network</i> .

Tabel 1.2. Method Kelas FFNN

No	Method	Deskripsi
1.	<code>def __init__(self, layers, activations, loss, init_method, l1_lambda, l2_lambda, rms_norm, gamma, **init_params)</code>	Inisialisasi model dengan parameter yang ditentukan, dengan <i>**init_params</i> merupakan argumen tambahan untuk <i>initialization method</i> tertentu
2.	<code>rms_norm_layer(x)</code>	Implementasi normalisasi <i>RMS Normalization</i> untuk <i>forward propagation</i>
3.	<code>forward(X)</code>	Implementasi <i>forward propagation</i>
4.	<code>backward(X, y)</code>	Implementasi <i>backward propagation</i> , di dalamnya mencakup implementasi <i>RMS Norm</i> serta regularisasi menggunakan l1 dan l2
5.	<code>update_weights(lr)</code>	Pembaruan bobot menggunakan <i>gradient descent</i> , di dalamnya mencakup <i>clipping gradient</i> untuk mencegah terjadinya <i>exploding</i> pada gradien
6.	<code>train(X_train, y_train, epochs, lr, batch_size, verbose)</code>	Pelatihan model, mencakup penanganan <i>input</i> yang diperlukan
7.	<code>predict(X)</code>	Prediksi label untuk data baru

8.	<code>predict_proba(X)</code>	Prediksi probabilitas untuk data baru
9.	<code>visualize_network()</code>	Visualisasi struktur jaringan dengan bobot dan bias
10.	<code>plot_weight_distribution(layers)</code>	Visualisasi distribusi bobot pada layer tertentu
11.	<code>plot_gradient_distribution(layers)</code>	Visualisasi distribusi gradien pada layer tertentu
12.	<code>save_model(filename)</code>	Menyimpan model ke file
13.	<code>load_model(filename)</code>	Memuat model dari file (<i>static method</i>)

b. Kelas Activation

Kelas Activation berperan sebagai repositori fungsi-fungsi aktivasi yang dapat digunakan dalam jaringan saraf. Kelas ini mendukung fungsi aktivasi standar dan beberapa fungsi tambahan. Implementasi Activation terdiri dari berbagai fungsi aktivasi dan turunannya sebagai method statis. Setiap fungsi aktivasi diimplementasikan menggunakan *NumPy* untuk operasi vektor yang efisien. Penanganan khusus diberikan untuk fungsi *softmax*, dimana turunannya diimplementasikan sebagai matriks *Jacobian* untuk menangani kasus *multivariate* dengan benar. Selain itu, juga terdapat fungsi aktivasi tambahan seperti *Leaky ReLU*, *ELU*, dan *Swish* sebagai bonus.

Tabel 1.3. Method Kelas Activation

No	Method	Deskripsi
1.	<code>linear(x)</code>	Fungsi aktivasi linear (identitas)
2.	<code>d_linear(x)</code>	Turunan fungsi aktivasi linear
3.	<code>relu(x)</code>	Fungsi aktivasi <i>ReLU</i>
4.	<code>d_relu(x)</code>	Turunan fungsi aktivasi <i>ReLU</i>
5.	<code>sigmoid(x)</code>	Fungsi aktivasi <i>sigmoid</i>
6.	<code>d_sigmoid(x)</code>	Turunan fungsi aktivasi <i>sigmoid</i>

7.	<code>tanh(x)</code>	Fungsi aktivasi <i>tanh</i>
8.	<code>d_tanh(x)</code>	Turunan fungsi aktivasi <i>tanh</i>
9.	<code>softmax(x)</code>	Fungsi aktivasi <i>softmax</i>
10.	<code>d_softmax(x)</code>	Jacobian dari fungsi aktivasi <i>softmax</i>
11.	<code>leaky_relu(x, alpha)</code>	Fungsi aktivasi <i>Leaky ReLU</i> (bonus)
12.	<code>d_leaky_relu(x, alpha)</code>	Turunan fungsi aktivasi <i>Leaky ReLU</i> (bonus)
13.	<code>elu(x, alpha)</code>	Fungsi aktivasi <i>ELU</i> (bonus)
14.	<code>d_elu(x, alpha)</code>	Turunan fungsi aktivasi <i>ELU</i> (bonus)
15.	<code>swish(x, beta)</code>	Fungsi aktivasi <i>Swish</i> (bonus)
16.	<code>d_swish(x, beta)</code>	Turunan fungsi aktivasi <i>Swish</i> (bonus)

c. Kelas Loss

Kelas Loss menyediakan implementasi berbagai fungsi loss dan turunannya untuk proses pelatihan jaringan saraf. Implementasi Loss terdiri dari tiga fungsi loss standar dan turunannya yaitu *Mean Squared Error* (MSE), *Binary Cross-Entropy*, dan *Categorical Cross-Entropy*. Setiap fungsi diimplementasikan dengan penanganan kasus khusus untuk mencegah masalah numerik seperti pembagian dengan nol atau logaritma dari nol. Fungsi *Binary Cross-Entropy* diimplementasikan sebagai kasus khusus dari *Categorical Cross-Entropy* dengan dua kelas. Epsilon kecil ditambahkan pada beberapa perhitungan untuk mencegah ketidakstabilan numerik.

Tabel 1.4. Method Kelas Loss

No	Method	Deskripsi
1.	<code>mse(y_true, y_pred)</code>	<i>Mean Squared Error</i>
2.	<code>d_mse(y_true, y_pred)</code>	Turunan <i>Mean Squared Error</i>
3.	<code>binary_cross_entropy(y_true,</code>	<i>Binary Cross-Entropy Loss</i>

	<code>y_pred, epsilon)</code>	
4.	<code>d_binary_cross_entropy(y_true, y_pred, epsilon)</code>	Turunan <i>Binary Cross-Entropy Loss</i>
5.	<code>categorical_cross_entropy(y_true, y_pred, epsilon)</code>	<i>Categorical Cross-Entropy Loss</i>
6.	<code>d_categorical_cross_entropy(y_true, y_pred, epsilon)</code>	Turunan <i>Categorical Cross-Entropy Loss</i>

d. Kelas Initializer

Kelas Initializer memiliki berbagai metode untuk inisialisasi bobot jaringan saraf yang diimplementasikan sebagai *static method*. Metode inisialisasi bobotnya yaitu inisialisasi *Zero*, *Uniform*, dan *Normal*. Untuk meningkatkan kinerja model dengan jaringan dalam, kelas ini juga mengimplementasikan inisialisasi *Xavier* dan *He* sebagai bonus. Setiap metode inisialisasi mendukung parameter *seed* untuk *reproducibility*. Metode *Xavier* dan *He* dirancang khusus untuk mengatasi masalah *vanishing* dan *exploding gradient* pada jaringan yang dalam dengan menginisialisasi bobot berdasarkan jumlah *neuron input* dan *output*.

Tabel 1.5. Method Kelas Initializer

No	Method	Deskripsi
1.	<code>zero(shape)</code>	Inisialisasi semua bobot dengan 0
2.	<code>uniform(shape, lower, upper, seed)</code>	Inisialisasi bobot dengan distribusi uniform
3.	<code>normal(shape, mean, variance, seed)</code>	Inisialisasi bobot dengan distribusi normal
4.	<code>xavier(shape, seed)</code>	Inisialisasi bobot dengan metode <i>Xavier</i> (bonus)
5.	<code>he(shape, seed)</code>	Inisialisasi bobot dengan metode <i>He</i> (bonus)

B. Forward Propagation

Forward propagation adalah algoritma dalam ANN yang menentukan bagaimana *input* diproses melalui jaringan untuk menghasilkan *output*. Implementasi *forward propagation* dalam kelas FFNN menyediakan beberapa konfigurasi dengan jumlah *layer* dan fungsi aktivasi yang berbeda. Implementasi ini juga mempertimbangkan komputasi *batch* untuk mempercepat pelatihan dan mengintegrasikan fitur normalisasi *RMSNorm* untuk meningkatkan stabilitas pelatihan pada jaringan yang dalam.

Forward propagation diimplementasikan dalam method `forward(X)` dengan parameter input `x` yang dapat berupa *single sample* atau *batch*. *Method* ini berfungsi untuk melakukan operasi linear dan nonlinear pada setiap *layer* jaringan, dan menyimpan nilai-nilai *intermediate* yang akan digunakan kembali selama *backward propagation*.

```
def forward(self, X):
    self.a = [X]
    self.z = []
    self.norms = []

    for i in range(len(self.weights)):
        z = self.a[-1] @ self.weights[i] + self.biases[i]

        if self.rms_norm:
            z, norm = self.rms_norm_layer(z)
            self.norms.append(norm)

        a = self.activations[i](z)
        self.z.append(z)
        self.a.append(a)

    return self.a[-1]
```

Forward propagation dimulai dengan menyimpan input `x` sebagai aktivasi *layer* input dalam daftar `self.a`. Kemudian, untuk setiap *layer* dalam jaringan, dilakukan operasi linear dengan mengalikan aktivasi *layer* sebelumnya dengan matriks bobot dan menambahkan bias. Jika normalisasi *RMSNorm* diaktifkan, *output* linear dinormalisasi berdasarkan akar kuadrat rata-rata dan ditransformasi dengan parameter *gamma*.

Setelah operasi linear dan normalisasi (jika ada), fungsi aktivasi yang sesuai diterapkan pada *output* untuk menghasilkan aktivasi *layer* saat ini. Aktivasi dan

pre-aktivasi (nilai sebelum fungsi aktivasi diterapkan) disimpan dalam daftar `self.a` dan `self.z` masing-masing, yang akan digunakan selama *backward propagation*.

Operasi ini diulang untuk setiap *layer* hingga *layer output*. *Output* aktivasi *layer* terakhir dikembalikan sebagai hasil *forward propagation*, yang dapat digunakan untuk prediksi atau perhitungan *loss*.

C. *Backward Propagation* dan *Weight Update*

Backward propagation adalah algoritma dalam ANN yang memungkinkan model untuk belajar dari data dengan menyesuaikan parameter (bobot dan bias) berdasarkan *error output*. Implementasi *backward propagation* dalam kelas FFNN ini untuk menghitung *gradient loss* terhadap semua parameter jaringan secara efisien menggunakan *chain rule* dari kalkulus.

Implementasi ini mempertimbangkan berbagai kasus khusus, termasuk kombinasi fungsi aktivasi *softmax* dengan *categorical cross-entropy* yang memiliki perhitungan gradien sederhana, serta normalisasi *RMSNorm* dan regularisasi L1/L2. Hasil dari *backward propagation* ini adalah gradien bobot yang digunakan untuk update bobot dan bias pada model. Sebelum melakukan update, *gradient clipping* juga diterapkan untuk mencegah masalah *exploding gradient* yang sering terjadi pada jaringan yang dalam.

Backward propagation diimplementasikan dalam method `backward(X, y)` yang menerima *input* data `x` dan target `y`, dan memperbarui gradien dalam atribut `self.d_weights` dan `self.d_biases`.

```
def backward(self, X, y):
    m = y.shape[0]

    if self.activations[-1] == Activation.softmax:
        if self.loss.__name__ == "categorical_cross_entropy":
            dz = self.a[-1] - y # Softmax + CCE
        else:
            dL_ds = self.d_loss(y, self.a[-1])
            J = Activation.d_softmax(self.z[-1]) # Jacobian softmax

            dz = np.zeros_like(dL_ds)
            for b in range(m):
                dz[b] = J[b] @ dL_ds[b]
    else:
        dz = self.d_loss(y, self.a[-1]) *
        self.d_activations[-1](self.z[-1])

    for i in reversed(range(len(self.weights))):
```

```

        # backprop RMSNorm
        if self.rms_norm and i < len(self.norms):
            norm = self.norms[i]
            mean_square = np.mean(self.z[i]**2, axis=-1,
keepdims=True)
            d_norm = dz * self.gamma
            d_mean_square = np.mean(d_norm * norm, axis=-1,
keepdims=True)

            dz = d_norm / np.sqrt(mean_square + self.epsilon) -
(self.z[i] * d_mean_square) / ((mean_square + self.epsilon) *
np.sqrt(mean_square + self.epsilon))

            self.d_weights[i] = self.a[i].T @ dz / m
            self.d_biases[i] = np.sum(np.array(dz), axis=0,
keepdims=True) / m

            # regularisasi
            self.d_weights[i] += self.l1_lambda *
np.sign(self.weights[i]) + self.l2_lambda * self.weights[i]

            if i > 0:
                dz = dz @ self.weights[i].T *
self.d_activations[i-1] (self.z[i-1])

```

Backward propagation dimulai dengan menghitung gradien pada layer output, yang bergantung pada kombinasi fungsi *loss* dan fungsi aktivasi. Untuk kombinasi *softmax* dan *categorical cross-entropy*, gradien dapat dihitung langsung sebagai selisih antara *output* prediksi dan target. Untuk kombinasi lain, gradien dihitung menggunakan turunan *loss* dan turunan aktivasi dengan aturan rantai.

Setelah gradien awal dihitung, algoritma melakukan iterasi mundur melalui jaringan, menghitung gradien untuk setiap layer. Jika *RMSNorm* diaktifkan, gradien diteruskan melalui operasi normalisasi menggunakan aturan rantai. Gradien terhadap bobot dan bias dihitung berdasarkan gradien layer dan aktivasi layer sebelumnya. Regularisasi L1 dan L2 ditambahkan ke gradien bobot jika diaktifkan.

Untuk *Weight Update*, setelah gradien dihitung, parameter jaringan diperbarui menggunakan algoritma *gradient descent* dalam method `update_weights(lr)`. Pada saat melakukan update bobot dan bias, untuk mencegah *exploding gradient*, maka *gradient clipping* diterapkan dengan membatasi norma gradien ke nilai maksimum tertentu. Setelah gradien bobot dan bias dihitung, gradien diteruskan ke layer sebelumnya untuk digunakan dalam iterasi berikutnya. *Learning rate* `lr` mengatur seberapa besar perubahan parameter pada setiap iterasi.

```
def update_weights(self, lr):
    max_norm = 1.0
    for i in range(len(self.weights)):
        w_norm = np.linalg.norm(self.d_weights[i])
        if w_norm > max_norm:
            self.d_weights[i] *= max_norm / w_norm

        b_norm = np.linalg.norm(self.d_biases[i])
        if b_norm > max_norm:
            self.d_biases[i] *= max_norm / b_norm
        self.weights[i] -= lr * self.d_weights[i]
        self.biases[i] -= lr * self.d_biases[i]
```

Pembaruan bobot dan bias dilakukan dengan mengurangkan gradien dikalikan dengan *learning rate* dari nilai parameter saat ini. Proses ini mengimplementasikan langkah dalam arah negatif gradien untuk meminimalkan fungsi *loss*.

2. Hasil Pengujian

A. Pengaruh *Depth* dan *Width*

A.1. Pengaruh *Width*

Pengujian dilakukan untuk mengetahui bagaimana variasi jumlah *neuron* dalam satu *hidden layer* (*width*) mempengaruhi kinerja model *Feedforward Neural Network* (FFNN). Hasil pengujian menunjukkan bahwa semakin banyak *neuron* dalam satu *hidden layer*, semakin baik akurasi yang diperoleh.

- Model dengan **16 *neuron*** dalam satu *hidden layer* mencapai akurasi **93.83%**.
- Model dengan **64 *neuron*** dalam satu *hidden layer* mencapai akurasi **95.31%**.
- Model dengan **128 *neuron*** dalam satu *hidden layer* mencapai akurasi **95.62%**.

Berdasarkan hasil tersebut, terlihat bahwa menambahkan jumlah *neuron* dalam *hidden layer* dapat meningkatkan akurasi model. Namun, perbedaannya semakin kecil ketika jumlah *neuron* bertambah, menunjukkan adanya titik batas di mana peningkatan *width* tidak memberikan peningkatan performa yang signifikan.

A.2. Pengaruh Depth

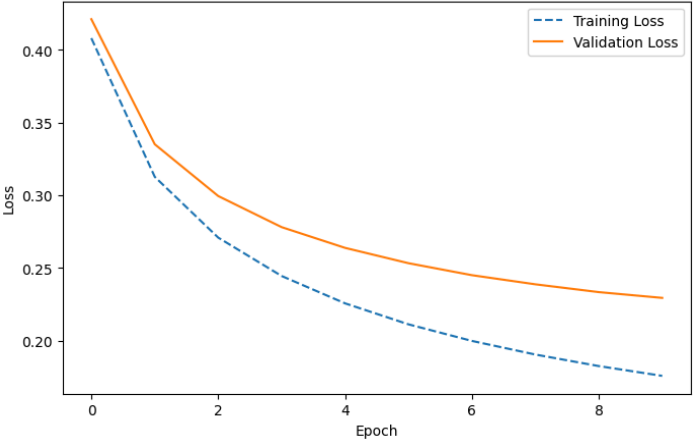
Pengujian juga dilakukan untuk mengetahui bagaimana variasi jumlah *hidden layer* (*depth*) mempengaruhi kinerja model. Hasil menunjukkan bahwa menambah kedalaman model dapat meningkatkan akurasi, tetapi dengan dampak yang lebih kecil dibandingkan variasi *width*.

- Model dengan **1 hidden layer (32 neuron)** mencapai akurasi **94.55%**.
- Model dengan **2 hidden layer (32-32 neuron)** mencapai akurasi **94.68%**.
- Model dengan **3 hidden layer (32-32-32 neuron)** mencapai akurasi **94.95%**.

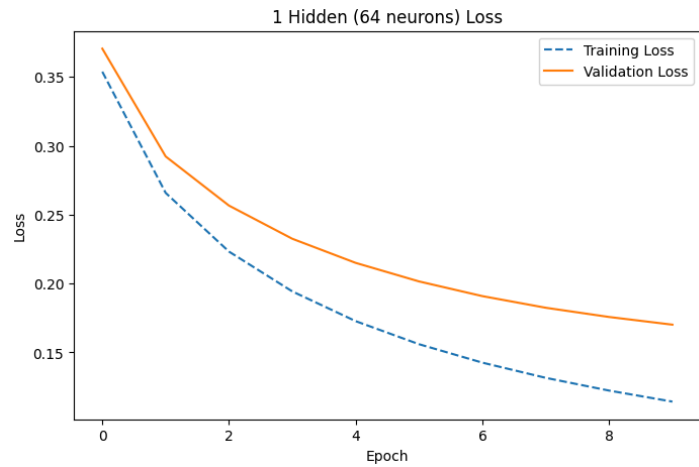
Dari hasil ini, terlihat bahwa menambah *hidden layer* memberikan sedikit peningkatan akurasi, tetapi tidak sebesar peningkatan akibat penambahan jumlah *neuron* dalam satu *hidden layer*. Hal ini menunjukkan bahwa dalam konteks dataset dan parameter yang digunakan, memperlebar *hidden layer* lebih efektif dibandingkan memperdalam model.

A.3. Analisis Keseluruhan

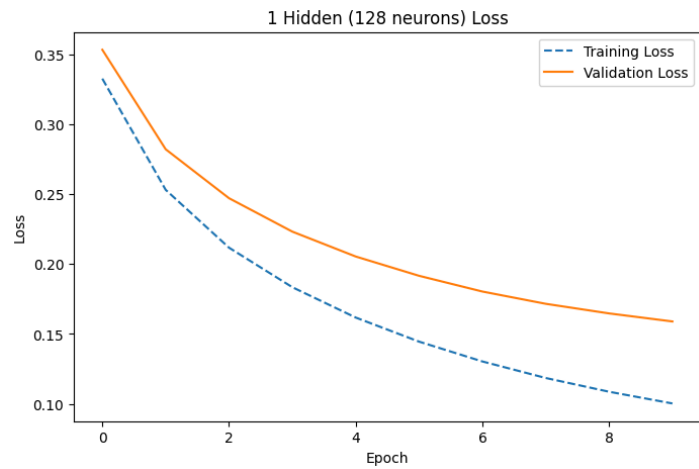
Tabel 2.1.1. Perbandingan Loss Tiap Arsitektur

Arsitektur	Training - Validation Loss																																				
1 Hidden (16 neurons)	<div><div>1 Hidden (16 neurons) Loss</div><table><caption>Estimated data for 1 Hidden (16 neurons) Loss</caption><thead><tr><th>Epoch</th><th>Training Loss</th><th>Validation Loss</th></tr></thead><tbody><tr><td>0</td><td>0.41</td><td>0.42</td></tr><tr><td>1</td><td>0.31</td><td>0.33</td></tr><tr><td>2</td><td>0.27</td><td>0.30</td></tr><tr><td>3</td><td>0.24</td><td>0.28</td></tr><tr><td>4</td><td>0.22</td><td>0.26</td></tr><tr><td>5</td><td>0.20</td><td>0.25</td></tr><tr><td>6</td><td>0.19</td><td>0.24</td></tr><tr><td>7</td><td>0.18</td><td>0.23</td></tr><tr><td>8</td><td>0.17</td><td>0.23</td></tr><tr><td>9</td><td>0.16</td><td>0.23</td></tr><tr><td>10</td><td>0.15</td><td>0.23</td></tr></tbody></table></div>	Epoch	Training Loss	Validation Loss	0	0.41	0.42	1	0.31	0.33	2	0.27	0.30	3	0.24	0.28	4	0.22	0.26	5	0.20	0.25	6	0.19	0.24	7	0.18	0.23	8	0.17	0.23	9	0.16	0.23	10	0.15	0.23
Epoch	Training Loss	Validation Loss																																			
0	0.41	0.42																																			
1	0.31	0.33																																			
2	0.27	0.30																																			
3	0.24	0.28																																			
4	0.22	0.26																																			
5	0.20	0.25																																			
6	0.19	0.24																																			
7	0.18	0.23																																			
8	0.17	0.23																																			
9	0.16	0.23																																			
10	0.15	0.23																																			

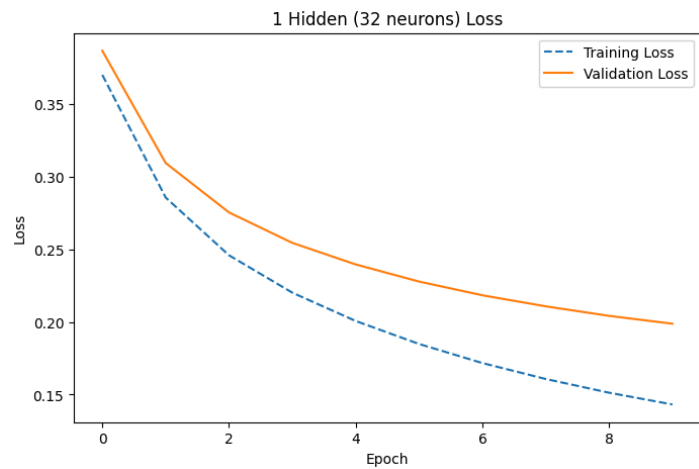
1 Hidden (64 neurons)



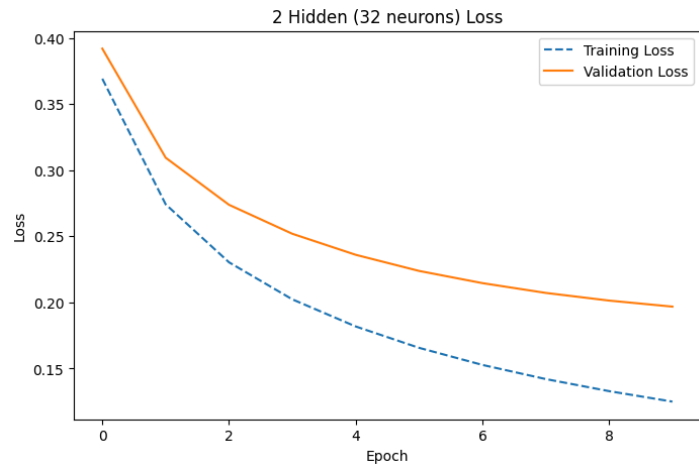
1 Hidden (128 neurons)



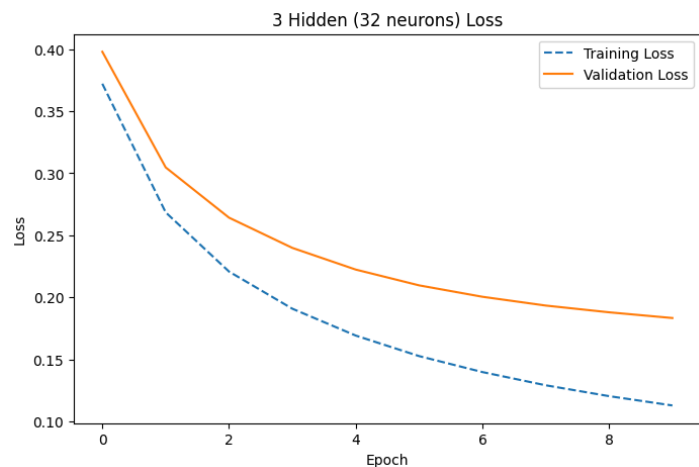
1 Hidden (32 neurons)



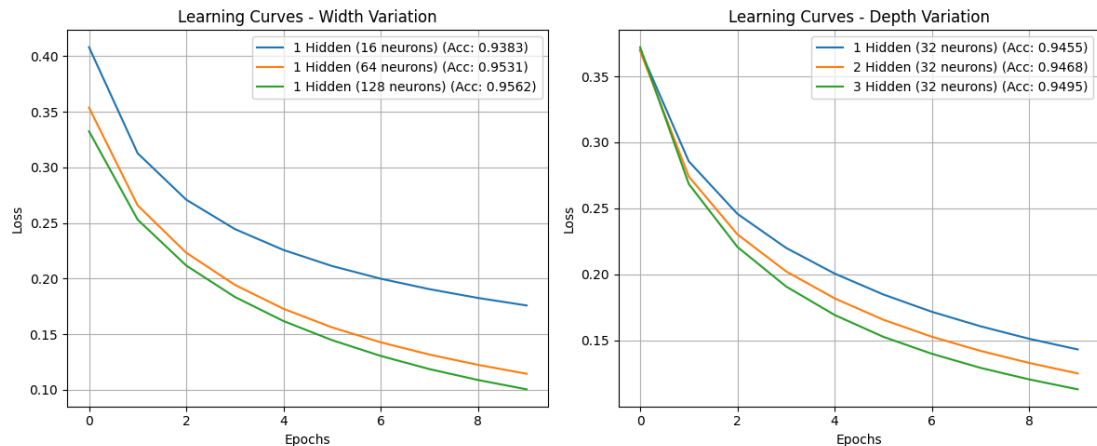
2 Hidden (32 neurons)



3 Hidden (32 neurons)



Tren penurunan *Training Loss* dan *Validation Loss* hampir sama untuk keseluruhan model. Seiring bertambahnya epoch, *Training Loss* masih menunjukkan penurunan sementara *Validation Loss* sudah hampir stagnan. Hal ini menandakan model sudah akan memasuki *overfitting*, di mana model terlalu menyesuaikan data pelatihan hingga kehilangan kemampuan untuk melakukan generalisasi pada data baru.



Gambar 2.1.1. Learning Curves Width Variation (Kiri) dan Depth Variation (Kanan)

Secara keseluruhan, hasil pengujian menunjukkan bahwa:

- Penambahan jumlah *neuron* dalam *hidden layer (width)* lebih berdampak pada peningkatan akurasi dibandingkan penambahan jumlah *hidden layer (depth)*.
- Model dengan lebih banyak *neuron* dalam satu *hidden layer* cenderung lebih baik dalam menangkap pola dalam data.
- Model yang lebih dalam tetap menunjukkan peningkatan akurasi, tetapi peningkatannya tidak sebesar penambahan *width*.
- Penggunaan jumlah *hidden layer* dan *neuron* yang lebih besar perlu diimbangi dengan pertimbangan *computational cost* dan risiko *overfitting*.

B. Pengaruh Fungsi Aktivasi

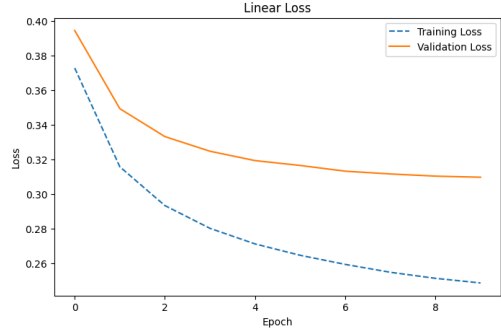
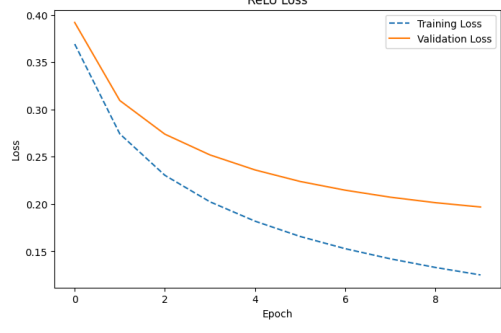
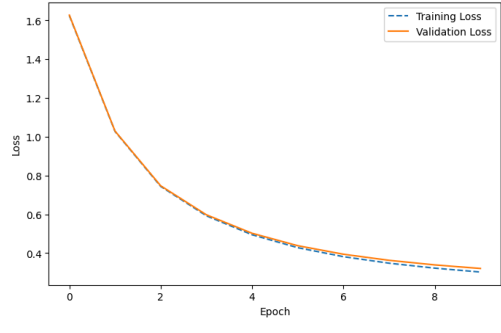
Pengujian ini bertujuan untuk mengevaluasi kinerja berbagai fungsi aktivasi, yaitu *Linear*, *ReLU*, *Sigmoid*, *Tanh*, *Leaky ReLU*, *ELU*, dan *Swish*, dalam konteks *Feedforward Neural Network (FFNN)*. Model yang digunakan memiliki arsitektur dengan *layer*: 784 *input neurons*, dua *hidden layers* masing-masing berisi 32 *neurons*, dan 10 *output neurons* yang menggunakan aktivasi *softmax*. Parameter lainnya yang digunakan adalah:

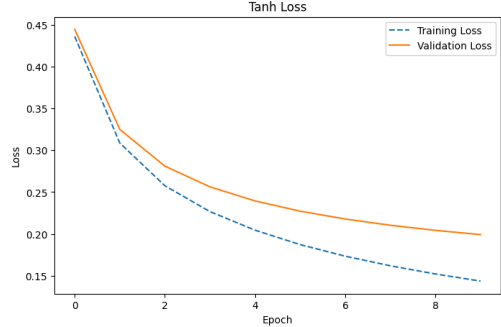
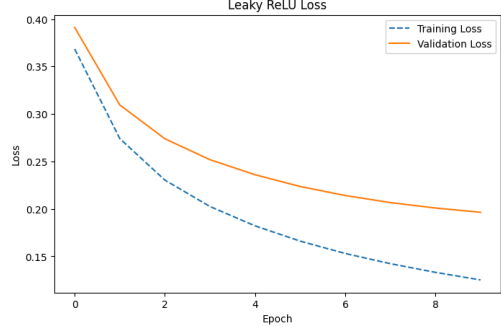
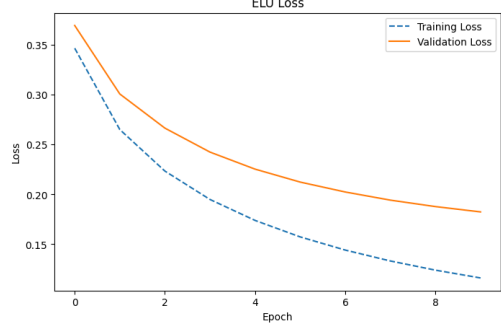
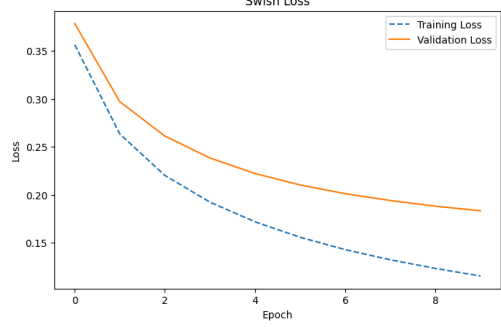
- **Loss function:** *Categorical Cross-Entropy*
- **Metode inisialisasi bobot:** *He Initialization*
- **Jumlah Epochs:** 10

- **Learning Rate:** 0.01
- **Batch Size:** 32

Hasil pengujian menunjukkan bahwa fungsi aktivasi memiliki dampak yang signifikan terhadap kinerja model. Berikut adalah ringkasan akurasi dari setiap fungsi aktivasi:

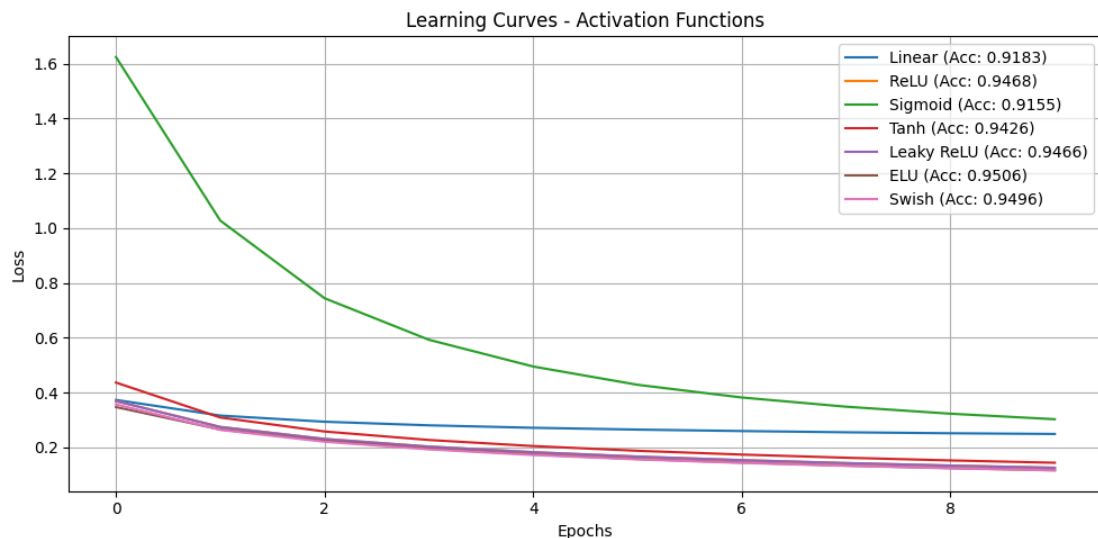
Tabel 2.2.1. Perbandingan Accuracy dan Loss Setiap Fungsi Aktivasi

Fungsi Aktivasi	Accuracy	Training - Validation Loss
Linear	0.9183	
ReLU	0.9468	
Sigmoid	0.9155	

Tanh	0.9426	
Leaky ReLU	0.9466	
ELU	0.9506	
Swish	0.9496	

Dari hasil di atas, **ELU** memberikan akurasi tertinggi (95.06%), diikuti oleh **Swish** (94.96%) dan **ReLU** (94.68%). Sementara itu, fungsi aktivasi **Linear** dan **Sigmoid** menghasilkan performa yang lebih rendah, yaitu sekitar 91%. Tren penurunan

Training Loss dan *Validation Loss* hampir sama untuk keseluruhan model, kecuali yang menggunakan fungsi aktivasi *Sigmoid*. Pada fungsi aktivasi *Sigmoid*, *Training Loss* dan *Validation Loss* menurun dengan pola yang serupa dan tetap rendah. Hal ini menunjukkan bahwa model belajar secara efektif tanpa kehilangan kemampuan generalisasi.

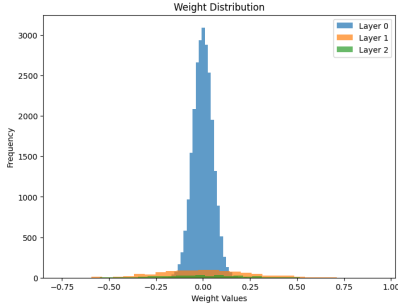
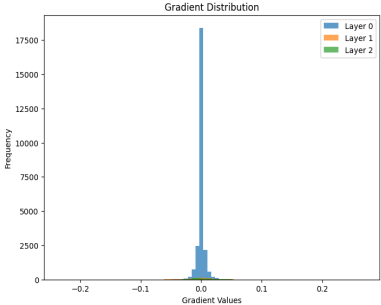
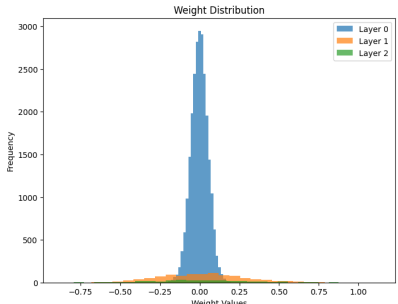
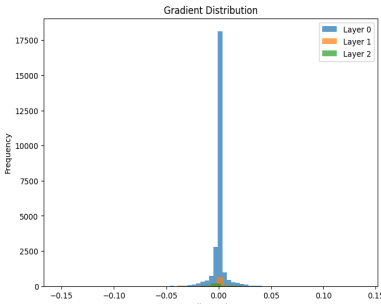


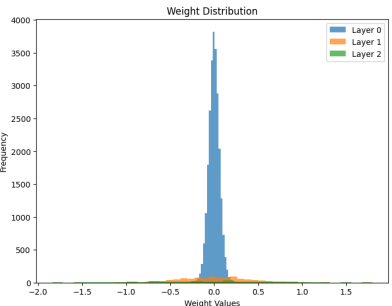
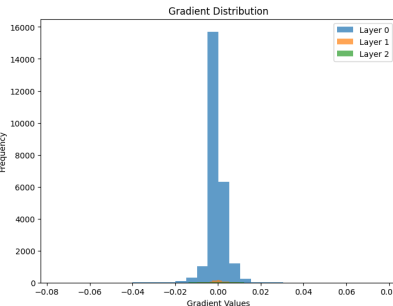
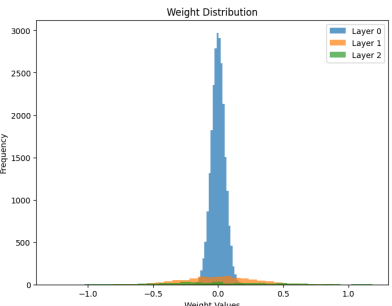
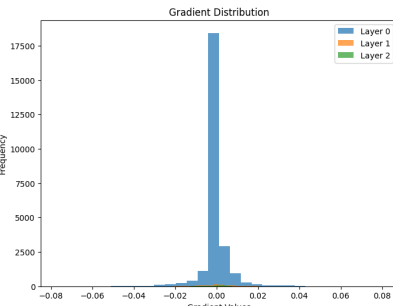
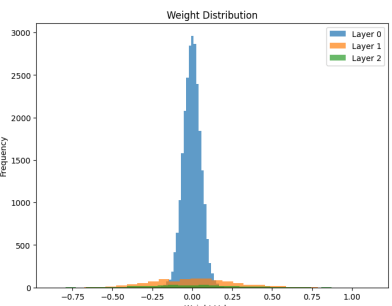
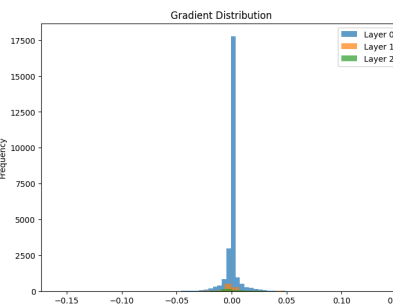
Gambar 2.2.1. Learning Curves Activation Functions

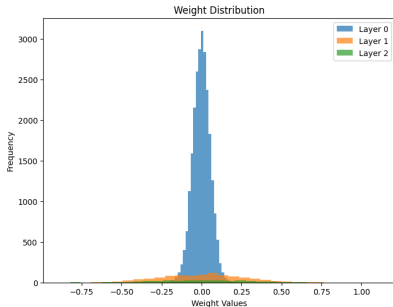
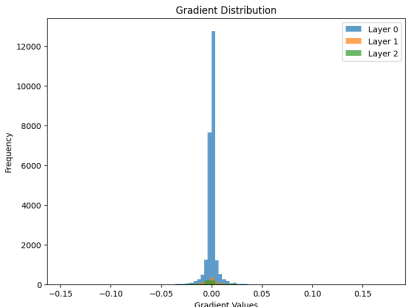
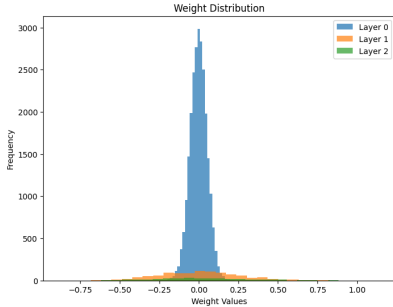
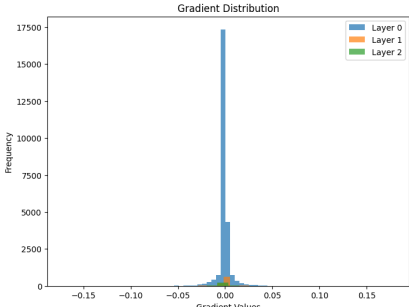
Dari grafik *loss* selama pelatihan, terlihat bahwa fungsi aktivasi yang berbasis **ReLU**, **Leaky ReLU**, **ELU**, dan **Swish** memiliki konvergensi *loss* yang lebih cepat dibandingkan dengan **Sigmoid** dan **Linear**. Hal ini menunjukkan bahwa fungsi aktivasi yang dapat menangani masalah **vanishing gradient** (seperti **ReLU** dan turunannya) lebih efektif dalam mempercepat proses pembelajaran.

Sebaliknya, fungsi **Sigmoid** cenderung mengalami **Saturasi** pada nilai ekstremnya, menyebabkan perbedaan gradien yang kecil dan memperlambat pembelajaran. Hal yang serupa terjadi pada **Tanh**, meskipun performanya lebih baik dibandingkan **Sigmoid**.

Tabel 2.2.2. Perbandingan Weight dan Gradient Setiap Fungsi Aktivasi

Fungsi Aktivasi	Distribusi Weight	Distribusi Gradient
Linear	 <p>Bobot terdistribusi secara simetris di sekitar nol, dengan sebagian besar nilai terkonsentrasi dalam rentang kecil.</p>	 <p>Gradien sangat terkonsentrasi di sekitar nol, menunjukkan minimnya perubahan bobot saat backpropagation.</p>
ReLU	 <p>Bobot memiliki distribusi yang hampir serupa dengan Linear tetapi sedikit lebih menyebar.</p>	 <p>Gradien menunjukkan banyak nilai nol, yang dapat mengindikasikan <i>dying ReLU problem</i>.</p>

<p>Sigmoid</p>	 <p>Weight Distribution</p> <p>Frequency</p> <p>Weight Values</p> <p>Layer 0 Layer 1 Layer 2</p> <p>Distribusi bobot lebih luas dibandingkan Linear dan ReLU, tetapi tetap simetris.</p>	 <p>Gradient Distribution</p> <p>Frequency</p> <p>Gradient Values</p> <p>Layer 0 Layer 1 Layer 2</p> <p>Gradien sangat kecil untuk sebagian besar bobot, menunjukkan potensi masalah <i>vanishing gradient</i>.</p>
<p>Tanh</p>	 <p>Weight Distribution</p> <p>Frequency</p> <p>Weight Values</p> <p>Layer 0 Layer 1 Layer 2</p> <p>Distribusi bobot sedikit lebih lebar dibandingkan Sigmoid, tetap simetris.</p>	 <p>Gradient Distribution</p> <p>Frequency</p> <p>Gradient Values</p> <p>Layer 0 Layer 1 Layer 2</p> <p>Gradien lebih besar dibandingkan Sigmoid, tetapi masih menunjukkan pola <i>vanishing gradient</i>.</p>
<p>Leaky ReLU</p>	 <p>Weight Distribution</p> <p>Frequency</p> <p>Weight Values</p> <p>Layer 0 Layer 1 Layer 2</p> <p>Distribusi bobot hampir serupa dengan ReLU tetapi memiliki ekor lebih panjang di sisi</p>	 <p>Gradient Distribution</p> <p>Frequency</p> <p>Gradient Values</p> <p>Layer 0 Layer 1 Layer 2</p> <p>Gradien lebih menyebar dibandingkan ReLU, dengan lebih sedikit nilai nol,</p>

	negatif.	mengurangi kemungkinan neuron mati.
ELU	 <p>Bobot terdistribusi secara simetris dengan ekor panjang di sisi negatif.</p>	 <p>Gradien lebih tersebar dibandingkan ReLU dan Leaky ReLU, membantu dalam pembelajaran stabil.</p>
Swish	 <p>Distribusi bobot mirip ELU tetapi sedikit lebih menyebar.</p>	 <p>Gradien memiliki distribusi lebih lebar dibandingkan aktivasi lainnya, menunjukkan pembelajaran yang lebih stabil.</p>

Dari hasil analisis distribusi bobot dan gradien, dapat disimpulkan bahwa pemilihan fungsi aktivasi sangat mempengaruhi stabilitas dan efektivitas pembelajaran dalam sebuah model *neural network*. Fungsi aktivasi **Linear** tidak memperkenalkan non-linearitas, sehingga kurang efektif dalam menangkap pola kompleks dan terbatas dalam menyelesaikan masalah non-linear. Fungsi aktivasi seperti **Sigmoid** dan **Tanh** cenderung mengalami masalah *vanishing gradient*, yang menghambat pembaruan bobot terutama pada lapisan-lapisan awal model. **ReLU**, meskipun populer karena kesederhanaannya, memiliki risiko *dying neurons*, yang terlihat dari

distribusi gradien yang memiliki banyak nilai nol. **Leaky ReLU dan ELU** menunjukkan peningkatan dengan distribusi bobot dan gradien yang lebih seimbang, sehingga mengurangi kemungkinan neuron menjadi tidak aktif selama training. **Swish**, sebagai salah satu fungsi aktivasi terbaru, menunjukkan distribusi gradien yang lebih luas dan stabil, menjadikannya pilihan yang baik untuk model dengan *deep network*.

Secara keseluruhan, distribusi bobot pada model cenderung tetap simetris di sekitar nol, tetapi distribusi gradien sangat bergantung pada fungsi aktivasi yang digunakan, di mana fungsi aktivasi yang lebih adaptif seperti ELU dan Swish mampu mempertahankan gradien yang lebih sehat untuk pembelajaran optimal di semua lapisan model.

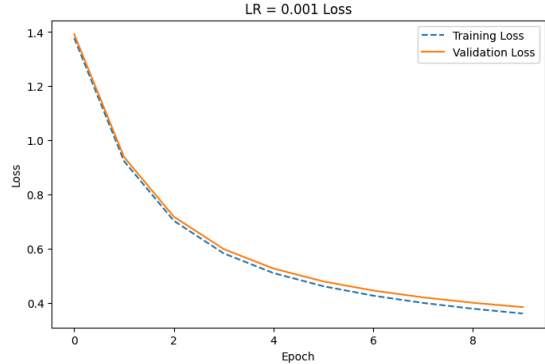
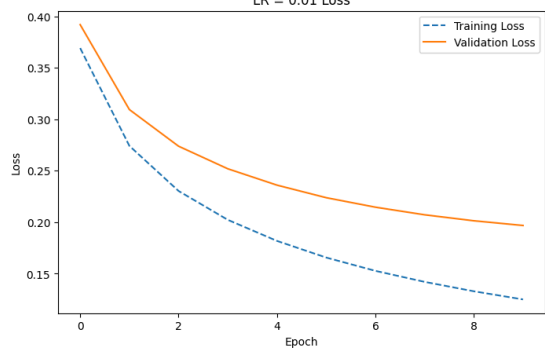
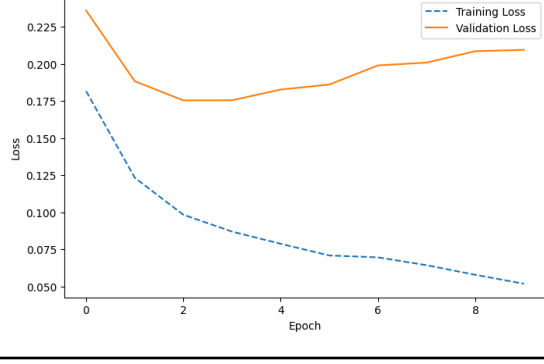
C. Pengaruh *Learning Rate*

Pengujian ini bertujuan untuk mengevaluasi dampak nilai yang diberikan pada parameter *learning rate* dalam konteks *Feedforward Neural Network* (FFNN). Model yang digunakan memiliki arsitektur dengan *layer*: 784 *input neurons*, dua *hidden layers* masing-masing berisi 32 *neurons*, dan 10 *output neurons* yang menggunakan aktivasi *softmax*. Parameter lainnya yang digunakan adalah:

- **Loss function:** *Categorical Cross-Entropy*
- **Activation function:** ["*relu*", "*relu*", "*softmax*"]
- **Metode inisialisasi bobot:** *He Initialization*
- **Jumlah Epochs:** 10
- **Batch Size:** 32

Berikut adalah ringkasan akurasi dari setiap variasi nilai *learning rate* yang diberikan:

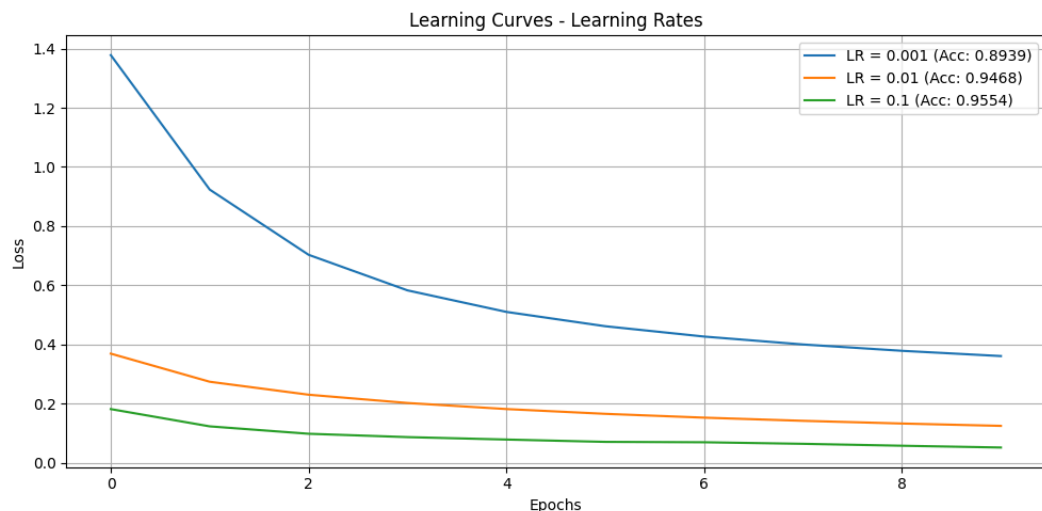
Tabel 2.3.1. Perbandingan Accuracy dan Loss Setiap Learning Rate

Learning Rate	Accuracy	Training - Validation Loss
0.001	0.8939	
0.01	0.9468	
0.1	0.9554	

Perbedaan akurasi yang dihasilkan dari variasi nilai *learning rate* menunjukkan dampak yang signifikan terhadap proses pelatihan model dalam FFNN. Pada nilai *learning rate* yang kecil (0.001), pembaruan bobot dalam setiap iterasi berlangsung lebih lambat. Hal ini menyebabkan model memerlukan lebih banyak *epoch* untuk belajar secara efektif sehingga model tidak bisa mencapai konvergen dengan cepat

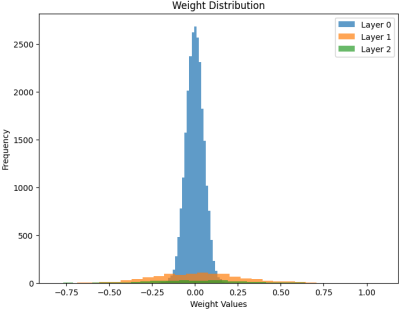
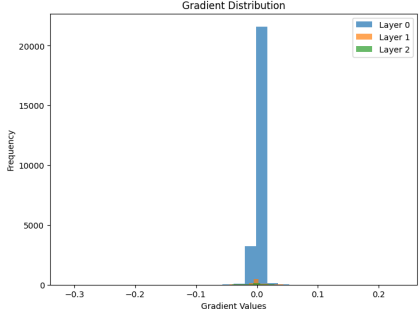
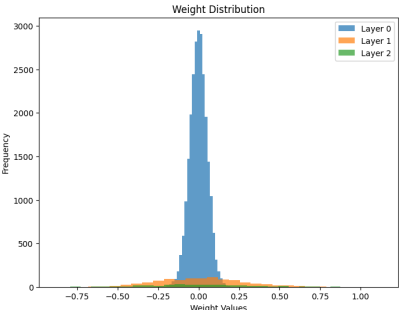
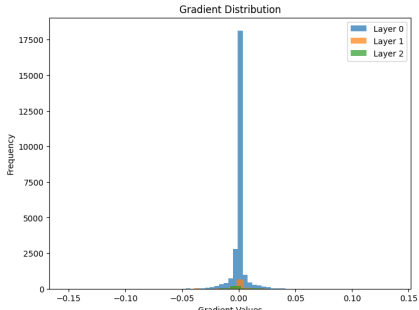
dalam 10 *epoch* yang terbatas. Nilai *learning rate* 0.01 memberikan keseimbangan yang baik antara kecepatan dan kestabilan tanpa sehingga menghasilkan akurasi yang lebih tinggi dibandingkan dengan 0.001. Sementara itu, dengan *learning rate* 0.1 pembaruan bobot lebih cepat sehingga memungkinkan model untuk mencapai hasil yang mendekati konvergen dalam 10 *epoch*. Namun, *learning rate* yang terlalu besar juga bisa menyebabkan model melewati titik minimum optimal dan berisiko mengarah pada ketidakstabilan sehingga gagal mencapai konvergen.

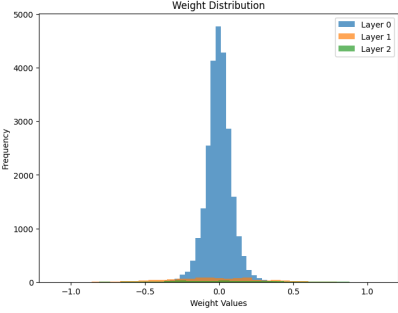
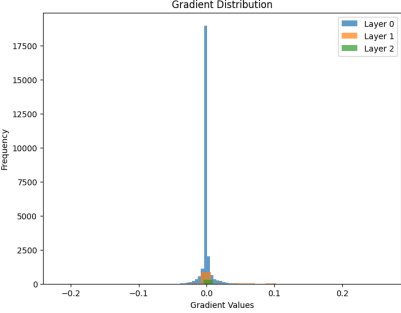
Tren perubahan *Training Loss* dan *Validation Loss* menunjukkan bahwa semakin besar *Learning Rate* (LR=0.1), *Training Loss* terus menurun tetapi *Validation Loss* mulai meningkat. Hal ini menunjukkan overfitting, di mana model terlalu menyesuaikan data pelatihan hingga kehilangan kemampuan untuk melakukan generalisasi pada data baru. Sebaliknya *Learning Rate* yang kecil (LR=0.001) memiliki *Training Loss* dan *Validation Loss* menurun dengan pola yang serupa dan tetap rendah. Hal ini menunjukkan bahwa model belajar secara efektif tanpa kehilangan kemampuan generalisasi. Secara keseluruhan, nilai yang moderat cenderung lebih efektif untuk mendapatkan konvergensi yang stabil dan optimal.



Gambar 2.3.1. Learning Curves Variasi Nilai Learning Rate

Tabel 2.3.2. Perbandingan Weight dan Gradient Setiap Variasi Nilai Learning Rate

Learning Rate	Distribusi Weight	Distribusi Gradient
0.001	 <p>Distribusi bobot terpusat di sekitar nol dengan varian kecil. Sebagian besar bobot tetap berada dalam kisaran nilai yang sangat kecil, menunjukkan perubahan bobot yang lambat selama pembelajaran.</p>	 <p>Gradien cenderung sangat kecil, yang dapat menyebabkan pembaruan bobot yang lambat. Ini berisiko membuat model belajar lebih stabil tetapi membutuhkan waktu lebih lama untuk konvergen.</p>
0.01	 <p>Bobot masih terpusat di sekitar nol, tetapi memiliki distribusi yang lebih melebar dibandingkan dengan <i>learning rate</i> 0.001. Ini menunjukkan adanya pembaruan bobot yang lebih signifikan selama pelatihan.</p>	 <p>Gradien lebih tersebar dibandingkan <i>learning rate</i> 0.001, yang berarti pembaruan bobot lebih aktif terjadi. Namun, tetap dalam batas stabil dan tidak menunjukkan lonjakan ekstrem.</p>

<p>0.1</p>	 <p>Distribusi bobot menjadi lebih melebar, menunjukkan pembaruan bobot yang agresif. Namun, jika terlalu ekstrem, ini dapat menyebabkan bobot tidak stabil dan sulit untuk konvergen.</p>	 <p>Gradien menunjukkan penyebaran yang lebih luas, dengan kemungkinan fluktuasi yang lebih besar. Ini berisiko menyebabkan pelatihan yang tidak stabil, terutama jika <i>learning rate</i> terlalu tinggi.</p>
------------	---	--

Berdasarkan hasil analisis distribusi bobot dan gradien, *learning rate* yang lebih kecil (0.001) menghasilkan pembaruan bobot yang lambat dan stabil, tetapi dapat menyebabkan pelatihan berlangsung lebih lama. *Learning rate* yang sedang (0.01) memberikan keseimbangan antara stabilitas dan kecepatan konvergensi, dengan distribusi bobot dan gradien yang lebih luas tetapi tetap terkendali. Sementara itu, *learning rate* yang lebih besar (0.1) menyebabkan perubahan bobot yang lebih agresif, yang dapat mempercepat pelatihan tetapi juga berisiko menyebabkan ketidakstabilan jika terlalu besar. Oleh karena itu, pemilihan *learning rate* yang optimal sangat penting untuk memastikan model dapat belajar secara efektif tanpa mengalami masalah konvergensi atau divergensi.

D. Pengaruh Inisialisasi Bobot

Pada pengujian ini, dilakukan eksperimen terhadap lima metode inisialisasi bobot:

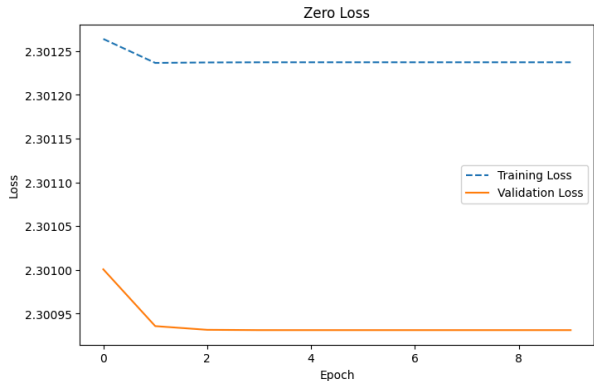
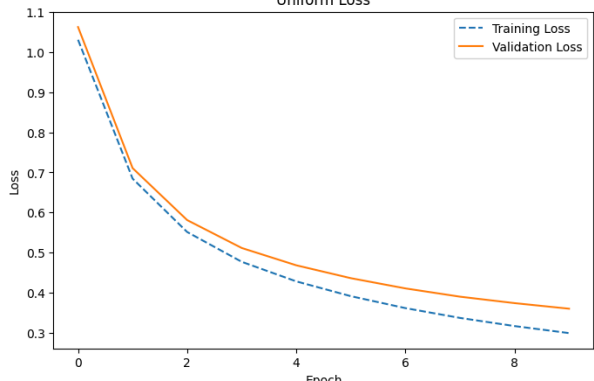
- *Zero Initialization*
- *Uniform Initialization*
- *Normal Initialization*
- *He Initialization*
- *Xavier Initialization*

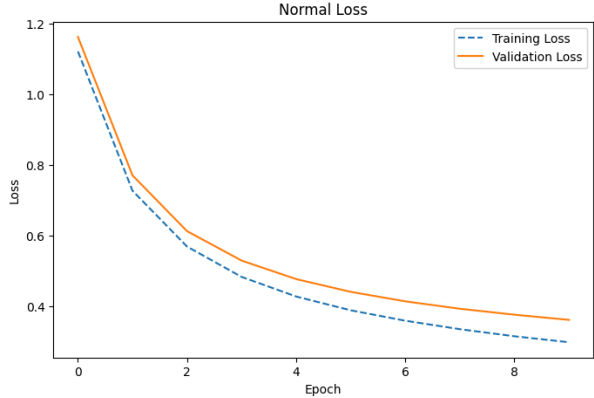
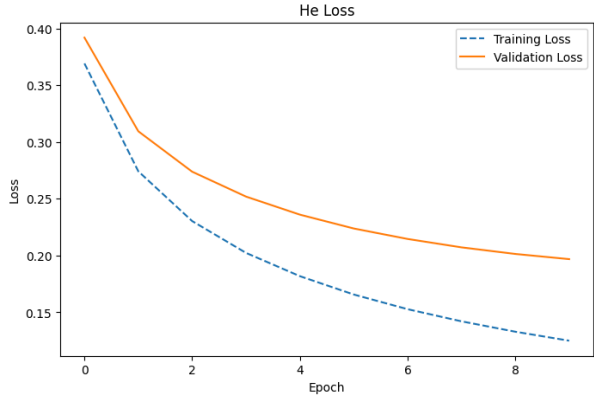
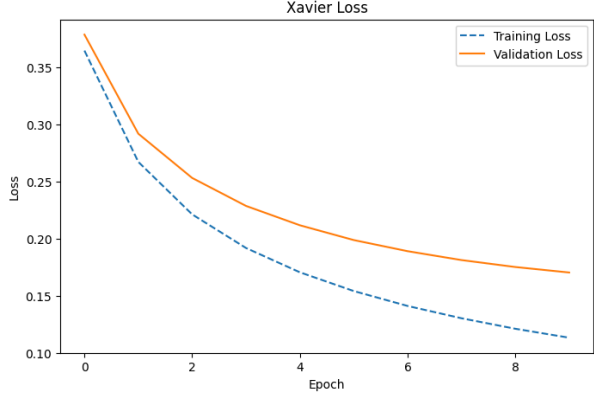
Parameter yang digunakan adalah sebagai berikut:

- **Layer input:** 784 neuron
- **Hidden layer 1:** 32 neuron, aktivasi *ReLU*
- **Hidden layer 2:** 32 neuron, aktivasi *ReLU*
- **Layer output:** 10 neuron, aktivasi *Softmax*
- **Loss function:** *Categorical Cross-Entropy*
- **Learning rate:** 0.01
- **Batch size:** 32
- **Epochs:** 10

Berikut adalah akurasi yang diperoleh dari masing-masing metode inisialisasi:

Tabel 2.4.1 Perbandingan Accuracy dan Loss Setiap Learning Rate

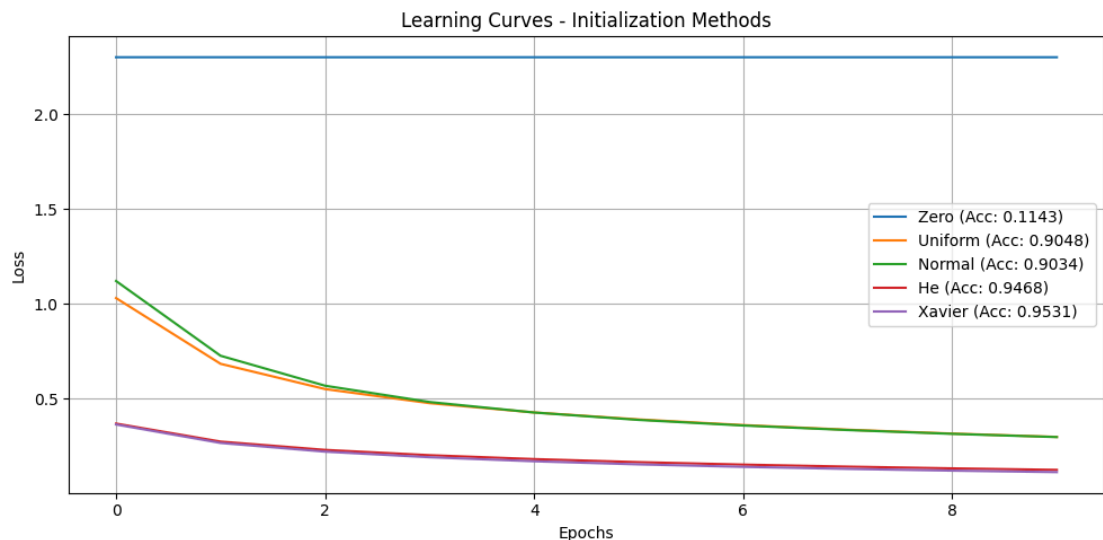
Initialization	Accuracy	Training - Validation Loss
Zero	0.1143	
Uniform	0.9048	

<i>Normal</i>	0.9034	 <p>Normal Loss</p>
<i>He</i>	0.9468	 <p>He Loss</p>
<i>Xavier</i>	0.9531	 <p>Xavier Loss</p>

Dari hasil di atas, terlihat bahwa inisialisasi dengan metode **Zero** menghasilkan akurasi yang sangat rendah (~11.43%), yang mengindikasikan bahwa model gagal belajar. Hal ini disebabkan oleh fakta bahwa semua bobot awalnya nol, sehingga setiap *neuron* menerima gradien yang sama dan tidak dapat melakukan pembelajaran.

Sebaliknya, metode **He** dan **Xavier** menunjukkan akurasi yang lebih tinggi dibandingkan metode lainnya, dengan **Xavier** mencapai akurasi tertinggi (95.31%). Kedua metode ini dirancang untuk mempertahankan skala varians bobot tetap stabil selama propagasi maju dan balik, sehingga memungkinkan pembelajaran lebih efektif.

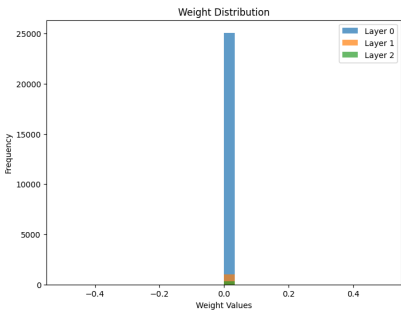
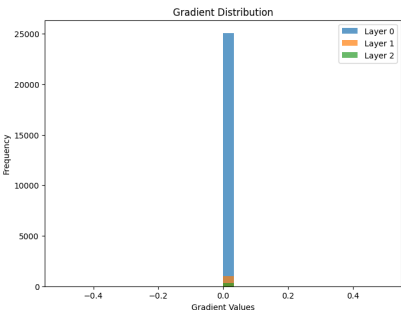
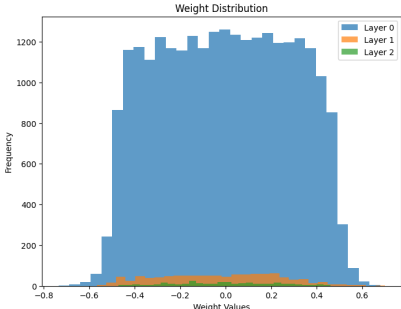
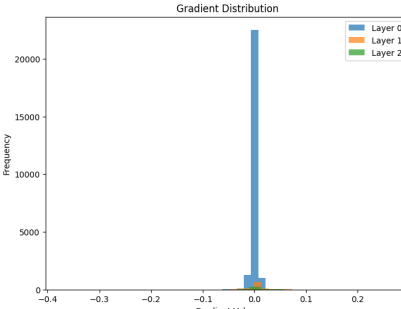
Tren penurunan *Training Loss* dan *Validation Loss* untuk metode **He** dan **Xavier** hampir sama. Pada metode **Uniform** dan **Normal**, *Training Loss* dan *Validation Loss* menurun dengan pola yang serupa dan tetap rendah dibanding inisialisasi lainnya. Hal ini menunjukkan bahwa model belajar secara efektif tanpa kehilangan kemampuan generalisasi. Sementara metode **zero** gagal dalam melakukan pembelajaran.

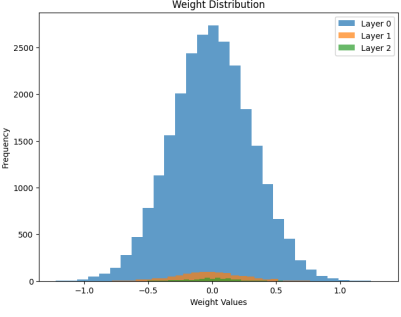
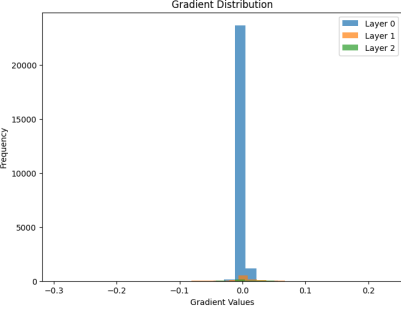
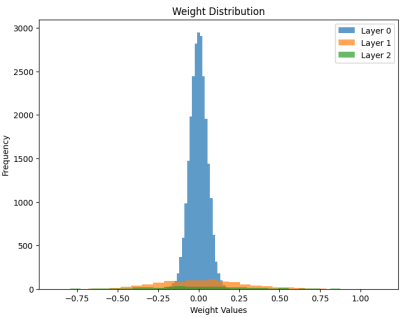
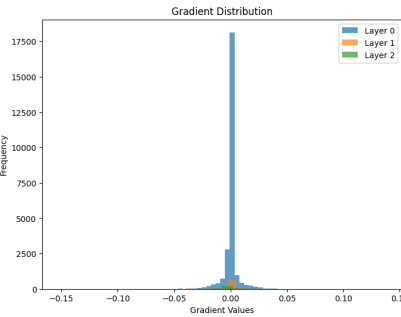
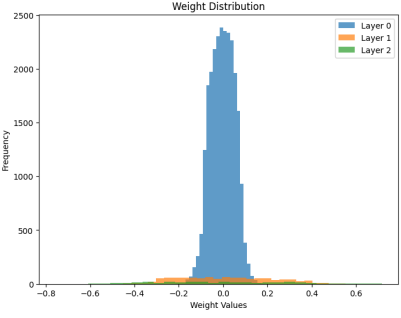
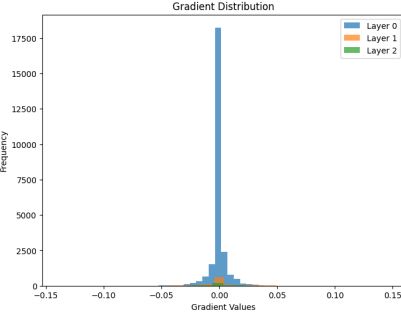


Gambar 2.4.1. Learning Curves - Initialization Methods

Gambar diatas menunjukkan perbedaan signifikan antara metode inisialisasi. Metode **Zero** memiliki *loss* yang stagnan, menunjukkan bahwa model tidak belajar. Sementara itu, metode **Uniform**, **Normal**, **He**, dan **Xavier** mengalami penurunan *loss* yang lebih signifikan seiring bertambahnya *epoch*, dengan **Xavier** dan **He** menunjukkan laju konvergensi yang lebih cepat.

Tabel 2.4.2. Perbandingan Weight dan Gradient Setiap Variasi Initialization Methods

Initialization Method	Distribusi Weight	Distribusi Gradient
Zero	 <p>Semua bobot bernilai nol, menyebabkan distribusi bobot terpusat pada nol tanpa variasi.</p>	 <p>Gradien juga bernilai nol di seluruh layer, menyebabkan tidak adanya pembaruan bobot selama pelatihan.</p>
Uniform	 <p>Distribusi bobot lebih menyebar dengan rentang tertentu, menunjukkan bahwa bobot memiliki nilai awal yang bervariasi.</p>	 <p>Gradien masih memiliki distribusi yang sempit tetapi sedikit lebih tersebar dibandingkan metode nol.</p>

<p><i>Normal</i></p>	 <p>Bobot terdistribusi mengikuti distribusi normal dengan <i>mean</i> sekitar nol dan varians tertentu.</p>	 <p>Gradien lebih bervariasi dibandingkan dengan metode uniform, tetapi masih cukup terkonsentrasi di sekitar nol.</p>
<p><i>He</i></p>	 <p>Distribusi bobot lebih terkonsentrasi tetapi memiliki penyebaran lebih luas dibandingkan normal, mendukung aktivasi ReLU.</p>	 <p>Gradien lebih stabil dibandingkan metode normal, tidak terlalu kecil maupun besar.</p>
<p><i>Xavier</i></p>	 <p>Distribusi bobot cukup simetris dan lebih tersebar</p>	 <p>Gradien juga lebih stabil dibandingkan metode lainnya,</p>

	dibandingkan He, dengan mean nol dan varians seimbang.	menunjukkan keseimbangan yang baik antara penyebaran <i>signal</i> dan gradien.
--	--	---

Berdasarkan hasil analisis distribusi bobot dan gradien, metode inisialisasi bobot memiliki pengaruh signifikan terhadap stabilitas dan efektivitas pembelajaran model. Inisialisasi nol menyebabkan masalah *symmetry breaking*, yang menghambat pembelajaran. Metode uniform dan normal lebih baik tetapi memerlukan skala yang tepat untuk menghindari masalah *vanishing* atau *exploding gradients*. Inisialisasi He menunjukkan performa terbaik untuk aktivasi ReLU karena menjaga signal propagation tetap stabil. Sementara itu, inisialisasi Xavier optimal untuk aktivasi sigmoid dan tanh karena mempertahankan keseimbangan antara distribusi bobot dan gradien. Oleh karena itu, pemilihan metode inisialisasi harus disesuaikan dengan jenis fungsi aktivasi yang digunakan dalam model.

E. Pengaruh Regularisasi

Dalam eksperimen ini, diuji tiga konfigurasi regularisasi pada model *Feedforward Neural Network* (FFNN):

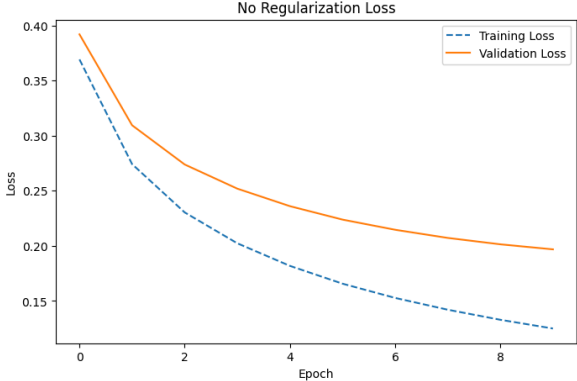
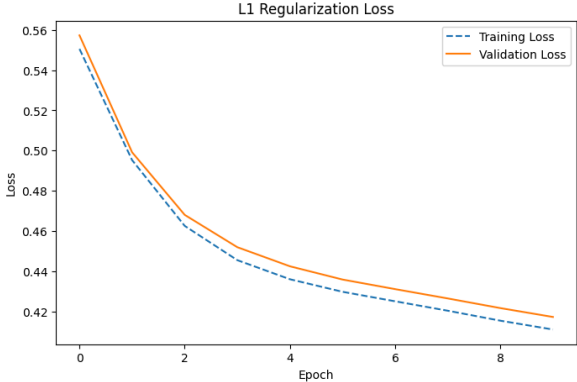
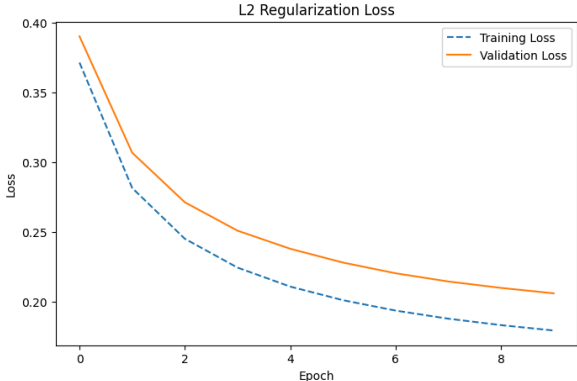
- Tanpa Regularisasi
- Regularisasi L1 (*Lasso*)
- Regularisasi L2 (*Ridge*)

Parameter yang digunakan adalah sebagai berikut:

- **Layer input:** 784 neuron
- **Hidden layer 1:** 32 neuron, aktivasi *ReLU*
- **Hidden layer 2:** 32 neuron, aktivasi *ReLU*
- **Layer output:** 10 neuron, aktivasi *Softmax*
- **Loss function:** *Categorical Cross-Entropy*
- **Initialization Method:** *He Initialization*
- **Batch size:** 32
- **Epochs:** 10

Berikut adalah ringkasan akurasi dari setiap konfigurasi regularisasi:

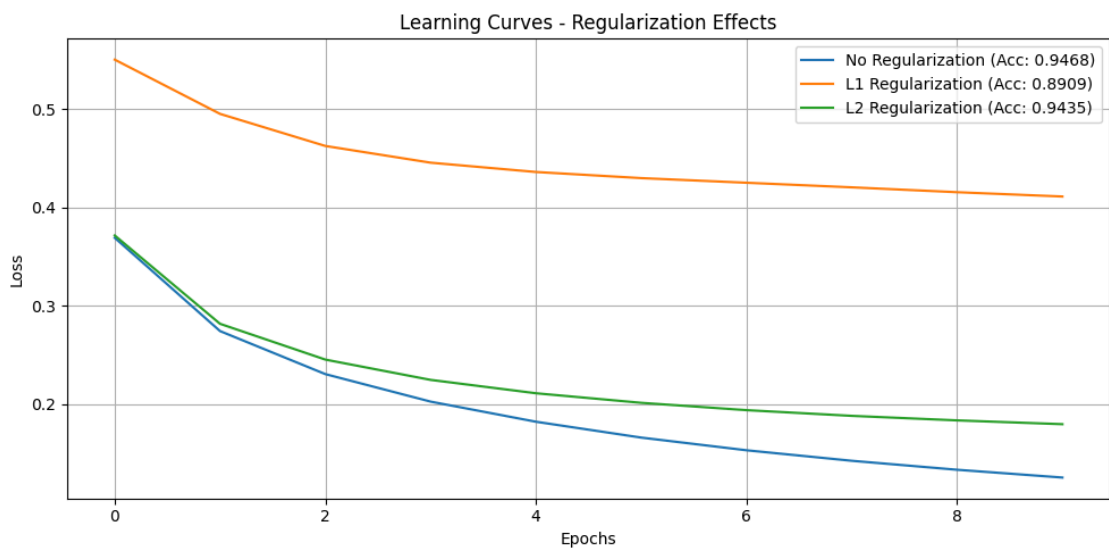
Tabel 2.5.1. Perbandingan Accuracy dan Loss Setiap Konfigurasi Regularisasi

Konfigurasi	Accuracy	Training - Validation Loss
Tanpa Regularisasi	0.9468	
Regularisasi L1	0.8909	
Regularisasi L2	0.9435	

Diperoleh informasi bahwa konfigurasi **tanpa regularisasi** memberikan akurasi tertinggi (94.68%) namun lebih rentan terhadap *overfitting*. **L1 regularization** menghasilkan model yang lebih sederhana tetapi mengalami penurunan akurasi cukup signifikan (89.09%). **L2 regularization** memberikan hasil yang hampir setara

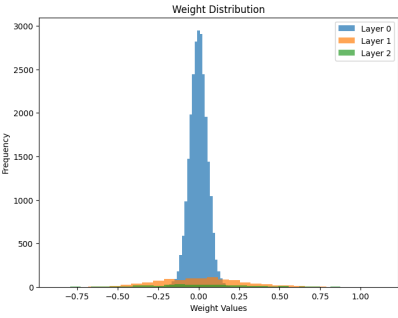
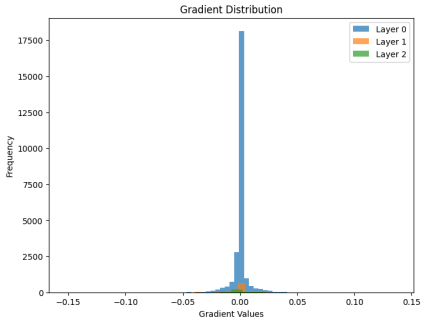
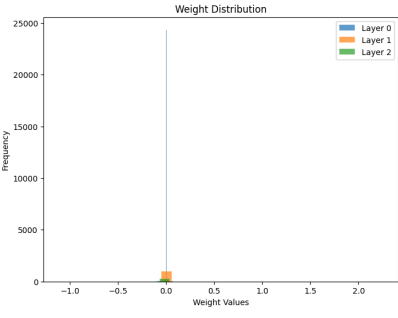
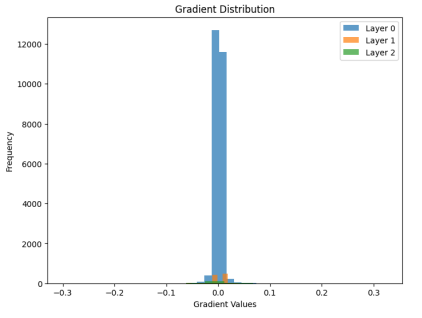
dengan model tanpa regularisasi (94.35%) namun dengan bobot yang lebih terkendali, mengurangi kemungkinan *overfitting*.

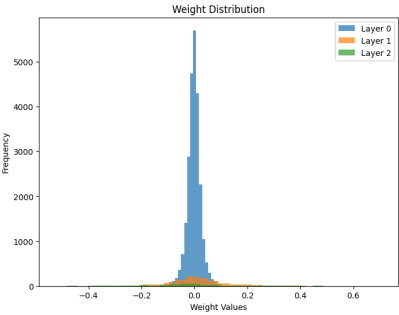
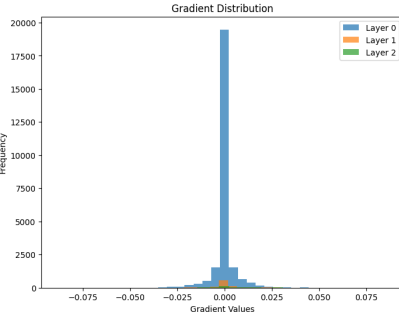
Pada model tanpa regularisasi, terlihat bahwa *Training Loss* menurun lebih cepat dibandingkan *Validation Loss*, yang bisa menjadi indikasi *overfitting* karena model terlalu menyesuaikan data latih. Pada model dengan *L1 regularization*, *loss* awalnya lebih tinggi dibandingkan model lainnya karena penalti L1 cenderung menyebabkan *sparseness* pada bobot, sehingga model lebih simpel dan kurang kompleks. Namun, gap antara *training loss* dan *validation loss* lebih kecil, yang menunjukkan efek regularisasi dalam mengurangi *overfitting*. Pada model dengan *L2 regularization*, pola *loss* mirip dengan tanpa regularisasi, tetapi dengan gap yang lebih kecil antara *training* dan *validation loss*, menunjukkan bahwa model tetap belajar dengan baik tetapi dengan regularisasi yang lebih halus dibandingkan L1.



Gambar 2.5.1. Learning Curves - Regularization Effects

Tabel 2.5.2. Perbandingan Weight dan Gradient Setiap Metode Regularisasi

Konfigurasi	Distribusi Weight	Distribusi Gradient
Tanpa Regularisasi	 <p>Bobot terdistribusi cukup simetris dengan mayoritas nilai berada di sekitar nol dan memiliki penyebaran yang lebih luas.</p>	 <p>Gradien memiliki distribusi yang cukup terkonsentrasi di sekitar nol tetapi masih terdapat variasi yang cukup baik di seluruh layer.</p>
Regularisasi L1	 <p>Banyak bobot memiliki nilai nol, menyebabkan distribusi bobot menjadi sangat terpusat di nol. Penyebaran bobot menjadi sangat sempit dengan hanya beberapa bobot yang memiliki nilai signifikan.</p>	 <p>Gradien juga mengalami pemusatan di sekitar nol, dengan sebagian besar nilai gradien menjadi nol atau sangat kecil.</p>

<p>Regularisasi L2</p>	 <p>Distribusi bobot lebih mirip dengan tanpa regularisasi tetapi dengan penyebaran yang lebih terkendali, tidak terlalu ekstrem. Nilai bobot lebih kecil dibandingkan tanpa regularisasi.</p>	 <p>Gradien tetap terkonsentrasi di sekitar nol tetapi lebih terdistribusi secara proporsional, sehingga masih ada perbedaan nilai antar layer.</p>
------------------------	---	--

Berdasarkan hasil analisis distribusi bobot dan gradien, metode regularisasi mempengaruhi penyebaran nilai bobot dan stabilitas gradien selama pelatihan model. Tanpa regularisasi, bobot memiliki distribusi yang lebih luas, yang dapat menyebabkan *overfitting* karena bobot bisa memiliki nilai besar yang terlalu spesifik terhadap data pelatihan. Regularisasi L1 membuat banyak bobot menjadi nol, menyebabkan model menjadi lebih jarang (*sparse*), yang membantu dalam seleksi fitur tetapi juga dapat menurunkan performa jika terlalu banyak bobot dihilangkan. Regularisasi L2, di sisi lain, mempertahankan distribusi bobot yang lebih stabil dengan mencegah nilai bobot menjadi terlalu besar, sehingga membantu mengurangi *overfitting* tanpa kehilangan terlalu banyak informasi. Oleh karena itu, pemilihan metode regularisasi harus mempertimbangkan *trade-off* antara kompleksitas model dan kemampuan generalisasi terhadap data baru.

F. Pengaruh Normalisasi RMSNorm

Pengujian ini bertujuan untuk mengevaluasi dampak penggunaan *RMS_Norm* terhadap performa model *Feedforward Neural Network* (FFNN) dalam klasifikasi data. Pengujian dilakukan dengan melatih dua model FFNN dengan arsitektur yang sama, tetapi dengan konfigurasi yang berbeda:

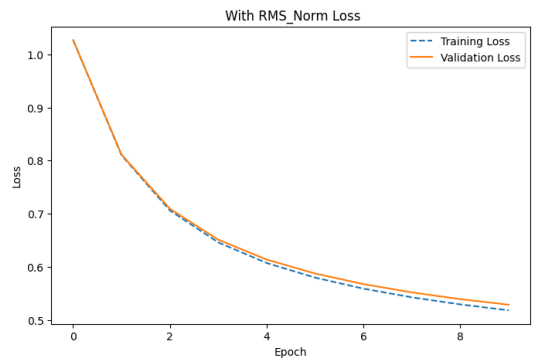
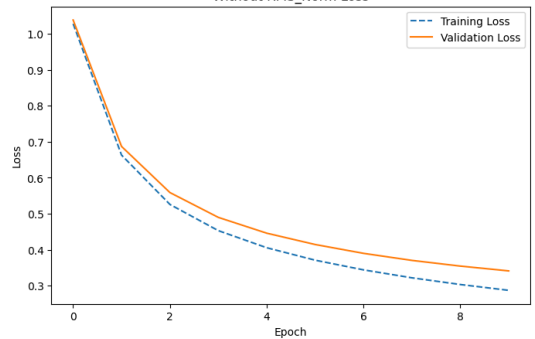
- **Model dengan RMS_Norm:** Menggunakan *RMS_Norm* dengan parameter $\gamma = 1$.
- **Model tanpa RMS_Norm:** Tidak menggunakan *RMS_Norm* ($\gamma = 0$).

Parameter yang digunakan adalah sebagai berikut:

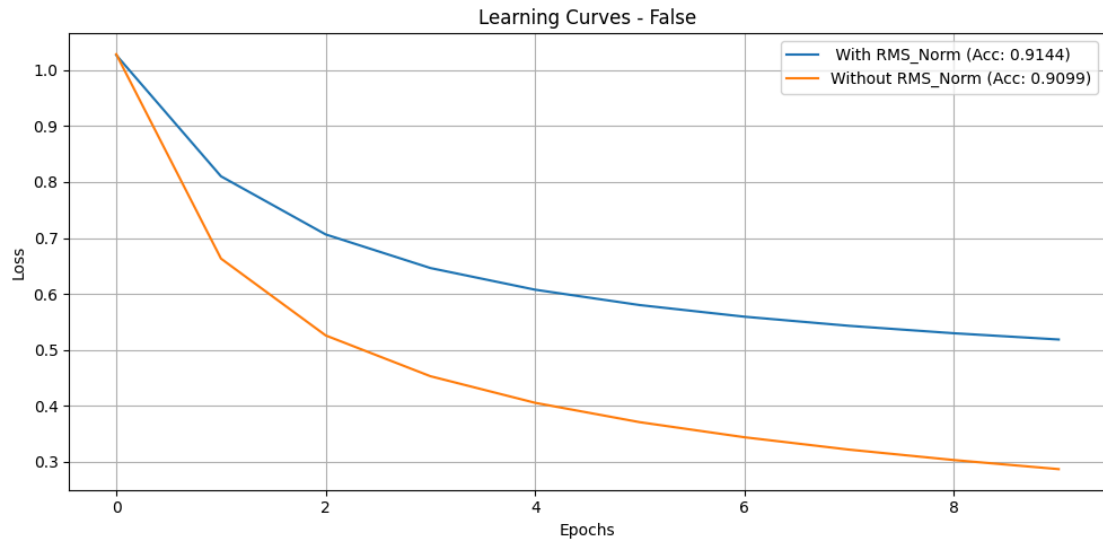
- **Layer input:** 784 neuron
- **Hidden layer 1:** 32 neuron, aktivasi *ReLU*
- **Hidden layer 2:** 32 neuron, aktivasi *ReLU*
- **Layer output:** 10 neuron, aktivasi *Softmax*
- **Loss function:** *Categorical Cross-Entropy*
- **Initialization Method:** *Uniform*
- **Batch size:** 32
- **Epochs:** 10

Berikut adalah ringkasan akurasi dari pengaruh ada tidaknya normalisasi *RMSNorm*:

Tabel 2.6.1. Perbandingan Accuracy dan Loss oleh Normalisasi *RMSNorm*

Konfigurasi	Accuracy	Training - Validation Loss
Menggunakan RMSNorm	0.9144	
Tidak Menggunakan RMSNorm	0.9099	

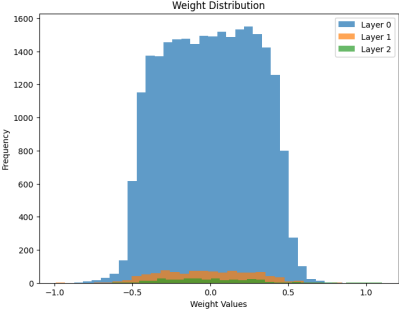
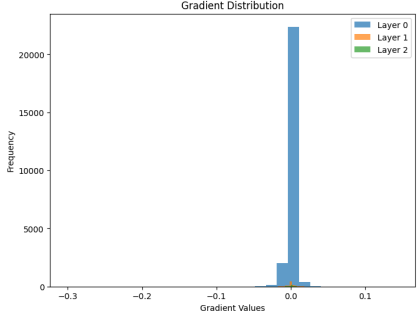
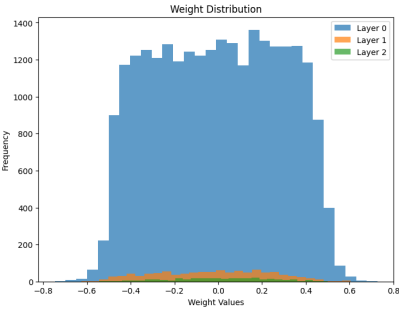
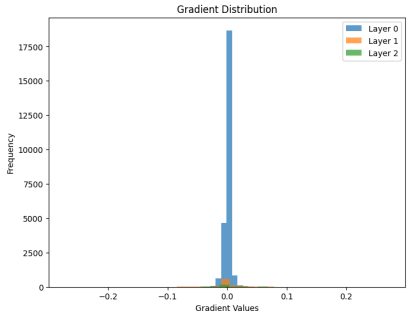
Model tanpa *RMS_Norm* menunjukkan *loss* yang lebih kecil pada setiap *epoch* dibandingkan model dengan *RMS_Norm*. Hal ini menunjukkan bahwa model tanpa *RMS_Norm* mengalami konvergensi lebih cepat dalam 10 *epoch* pertama.



Gambar 2.6.1. Learning Curves - Normalization RMSNorm

Model dengan *RMS_Norm* mencapai akurasi **91.44%**, sedikit lebih tinggi dibandingkan model tanpa *RMS_Norm* yang mencapai **90.99%**. Ini menunjukkan bahwa *RMS_Norm* membantu meningkatkan generalisasi model pada data uji meskipun perbedaannya tidak signifikan. Pada model dengan *RMS_Norm*, *Training Loss* dan *Validation Loss* lebih dekat satu sama lain sepanjang pelatihan, menunjukkan bahwa normalisasi ini membantu stabilisasi pembelajaran dan mengurangi *overfitting*. Sebaliknya, pada model tanpa *RMS_Norm*, terjadi gap yang lebih besar antara *Training Loss* dan *Validation Loss* di akhir pelatihan, yang menunjukkan kecenderungan model untuk lebih menyesuaikan diri terhadap data latih dibandingkan data validasi. Selain itu, *loss* pada model dengan *RMS_Norm* lebih lambat turun dibandingkan model tanpa normalisasi, yang mengindikasikan bahwa *RMS_Norm* mungkin membatasi pembelajaran terlalu agresif, tetapi dengan manfaat generalisasi yang lebih baik.

Tabel 2.6.2. Perbandingan Weight dan Gradient Terhadap Normalisasi

Normalisasi	Weight	Gradient
Menggunakan RMSNorm	 <p>Distribusi bobot tampak lebih terkonsentrasi di sekitar nol dengan rentang yang lebih luas dibandingkan model tanpa RMSNorm. Layer 0 memiliki distribusi bobot yang lebih besar dibandingkan layer lain, sedangkan Layer 1 dan Layer 2 memiliki bobot yang jauh lebih kecil.</p>	 <p>Distribusi gradien sangat terkonsentrasi di sekitar nol dengan nilai yang sangat kecil untuk sebagian besar bobot, menunjukkan bahwa gradien lebih terkontrol dan stabil selama pelatihan.</p>
Tidak Menggunakan RMSNorm	 <p>Distribusi bobot juga terkonsentrasi di sekitar nol, tetapi lebih merata dibandingkan model dengan RMSNorm. Layer 0 masih memiliki bobot dominan, tetapi perbedaan bobot antara layer</p>	 <p>Distribusi gradien masih terkonsentrasi di sekitar nol, tetapi dengan rentang yang sedikit lebih besar dibandingkan model dengan RMSNorm. Hal ini</p>

	lebih kecil dibandingkan model dengan RMSNorm.	menunjukkan bahwa gradien bisa mengalami fluktuasi lebih besar selama pelatihan.
--	--	--

Berdasarkan distribusi bobot, model dengan RMSNorm menunjukkan bobot yang lebih terkonsentrasi di sekitar nol dengan rentang yang lebih luas dibandingkan model tanpa RMSNorm. Sebaliknya, model tanpa RMSNorm memiliki distribusi bobot yang lebih merata, yang dapat membantu eksplorasi parameter yang lebih luas. Dalam kedua model, Layer 0 mendominasi distribusi bobot, sedangkan Layer 1 dan Layer 2 memiliki bobot yang lebih kecil.

Dari sisi distribusi gradien, model dengan RMSNorm memiliki gradien yang lebih terkonsentrasi di sekitar nol, menunjukkan stabilitas dalam pembaruan bobot selama pelatihan. Sementara itu, model tanpa RMSNorm memiliki distribusi gradien yang lebih menyebar, yang dapat menyebabkan fluktuasi lebih besar dalam pembaruan bobot dan memungkinkan konvergensi lebih cepat.

Oleh karena itu, RMSNorm memberikan stabilisasi dalam pembaruan bobot dan gradien, yang bermanfaat untuk meningkatkan generalisasi model. Namun, model tanpa RMSNorm lebih cepat mencapai konvergensi dalam jumlah epoch yang terbatas. Pemilihan penggunaan RMSNorm bergantung pada kebutuhan, apakah lebih mengutamakan stabilitas dan generalisasi atau kecepatan konvergensi.

G. Perbandingan dengan *Library Sklearn*

Pengujian ini bertujuan untuk mengevaluasi keakuratan model *Feedforward Neural Network* (FFNN) yang diimplementasikan *from scratch*, dengan membandingkannya terhadap model *neural network* dari *library sklearn*. Kedua model yang digunakan memiliki *hyperparameter* yang disamakan.

Berikut perbandingan konfigurasi masing-masing model.

Tabel 2.7.1. Perbandingan Konfigurasi Model

<i>Parameter</i>	<i>MLP Classifier</i>	<i>FFNN From Scratch</i>
Layer	[784, 128, 64, 10] (784 input, 2 hidden layer, 10 output)	(128, 64) (2 hidden layers, 10 output secara implisit)
Activation Function	ReLU (untuk hidden layers), softmax (untuk output)	ReLU (untuk hidden layers), softmax (untuk output)
Loss Function	Categorical Cross Entropy	Default (Sama, Categorical Cross-Entropy)
Optimisasi	Implementasi manual, menggunakan gradient descent	Stochastic Gradient Descent
Learning Rate	0.01	0.01 (Constant Learning Rate)
Batch Size	128	128
Inisialisasi Bobot	Xavier	Default (Glorot/Xavier)
Regularization L2	$l2_lambda=0.0001$	Default ($\alpha=0.0001$)
Seed/Random State	42	42
Verbose Output	1	1

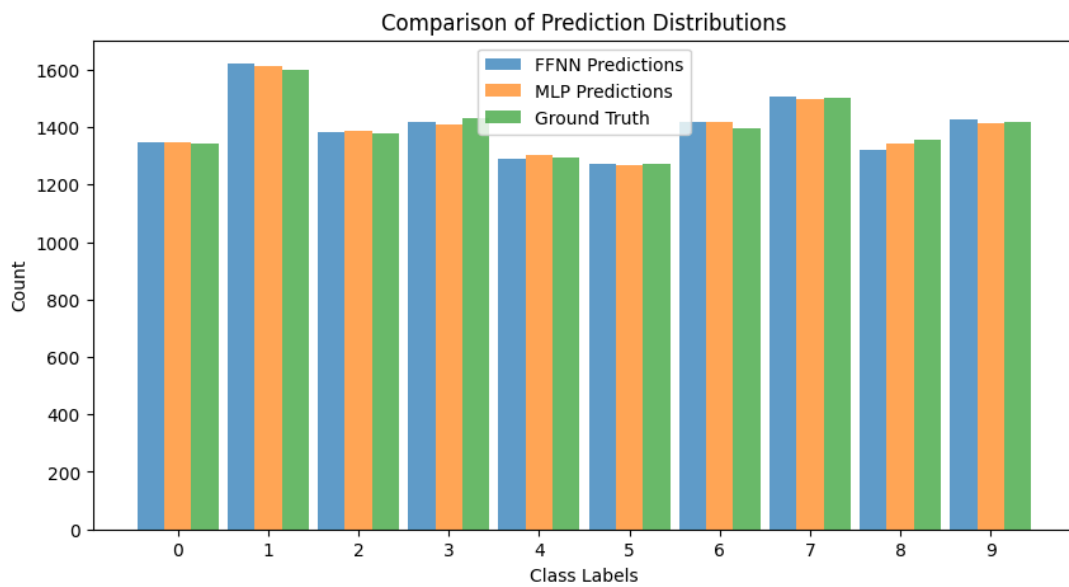
Meskipun tidak 100% identik *hyperparameter* dari kedua model, namun konfigurasi tersebut sudah mendekati. Salah satunya adalah inisialisasi bobot di mana MLP Classifier menggunakan Glorot/Xavier uniform, sedangkan implementasi FFNN harus dicek apakah benar-benar Xavier uniform atau tidak. Selain itu mekanisme update bobot juga bisa berbeda karena implementasi Scikit-Learn lebih kompleks. MLP Classifier menggunakan *learning rate schedule*, bahkan dalam "constant" mode sehingga bisa ada sedikit perbedaan efek dalam *training*.

Setelah dilakukan pengujian, berikut hasil akurasi dari kedua model.

Tabel 2.7.2. Perbandingan Accuracy Model

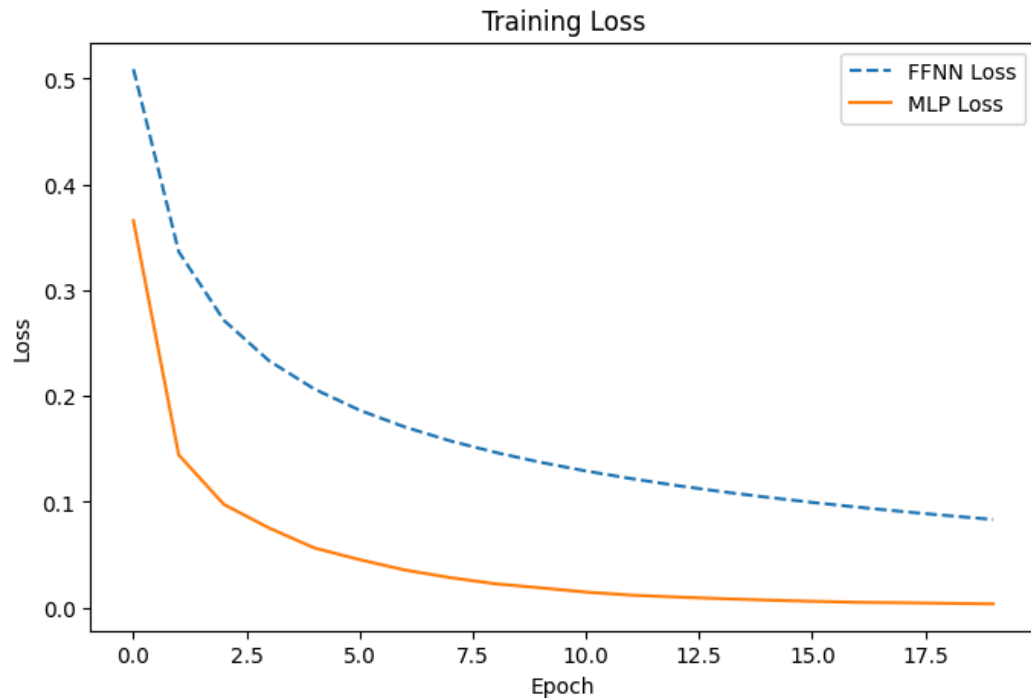
<i>Model</i>	<i>Accuracy</i>
MLP Classifier	0.9709285714285715
FFNN From Scratch	0.9589285714285715

Perbandingan distribusi prediksi yang diberikan oleh kedua model adalah sebagai berikut.



Gambar 2.7.1. Perbandingan Distribusi Kelas Prediksi dari Kedua Model

Sementara perbandingan *training loss* yang diberikan oleh kedua model adalah sebagai berikut.



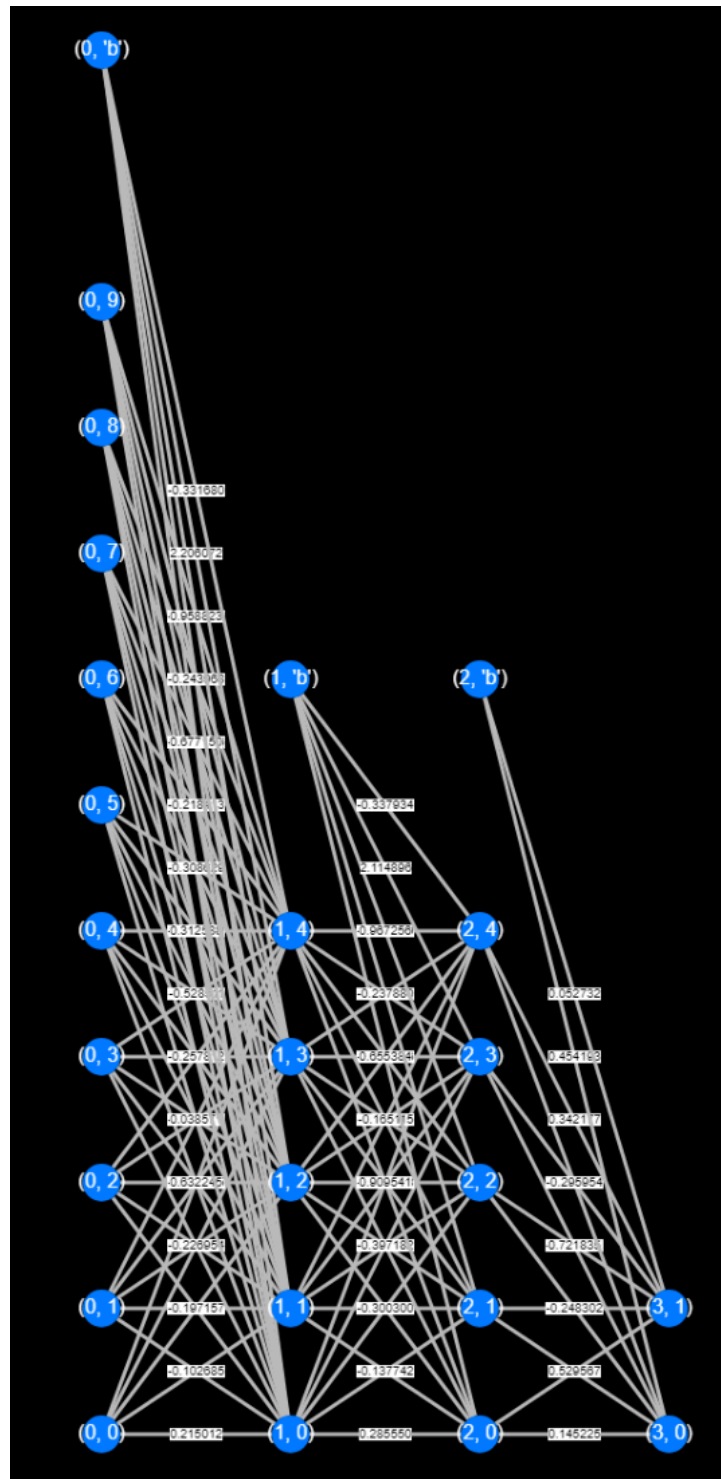
Gambar 2.7.2. Learning Curve Berupa Training Loss dari Kedua Model

Berdasarkan Gambar 2.7.1, distribusi prediksi kelas antara model FFNN *from scratch* dan MLP Classifier menunjukkan pola yang mirip dengan *ground truth*, menandakan bahwa kedua model mampu menangkap pola dalam data dengan baik. Namun, terdapat sedikit perbedaan dalam jumlah prediksi untuk setiap kelas, di mana MLP Classifier tampak lebih mendekati distribusi *ground truth* dibandingkan FFNN *from scratch*. Hal ini mengindikasikan bahwa MLP Classifier memiliki generalisasi yang lebih baik dibandingkan model yang diimplementasikan secara manual.

Pada Gambar 2.7.2, *learning curve* menunjukkan bahwa MLP Classifier memiliki *training loss* yang turun lebih cepat dan mencapai nilai *loss* yang lebih rendah dibandingkan FFNN *from scratch*. Ini menunjukkan bahwa model dari Scikit-Learn lebih efisien dalam proses optimisasi, kemungkinan disebabkan oleh mekanisme update bobot yang lebih kompleks dan penggunaan strategi learning rate schedule yang lebih optimal. Sementara itu, FFNN *from scratch* mengalami penurunan *loss* yang lebih lambat, yang dapat disebabkan oleh implementasi optimisasi yang lebih sederhana.

Dari hasil akurasi, MLP Classifier mencapai 97.09%, lebih tinggi dibandingkan FFNN *from scratch* yang hanya mencapai 95.89%. Meskipun perbedaannya tidak terlalu besar, hal ini menunjukkan bahwa penggunaan library yang sudah dioptimalkan seperti Scikit-Learn dapat meningkatkan performa model dibandingkan implementasi manual. Dengan demikian, meskipun FFNN from scratch dapat memberikan hasil yang kompetitif, model dari Scikit-Learn lebih efisien dalam hal konvergensi dan akurasi.

H. Visualisasi Arsitektur



Gambar 2.8.1. Visualisasi Arsitektur FFNN

III. KESIMPULAN DAN SARAN

Kesimpulan

Berdasarkan berbagai pengujian yang telah dilakukan terhadap model FFNN, beberapa kesimpulan utama dapat diperoleh sebagai berikut:

1. Pengaruh *Depth* dan *Width*

Struktur jaringan FFNN, terutama jumlah *hidden layer* dan jumlah neuron per layer (*width*), memiliki dampak signifikan terhadap kemampuan model dalam mempelajari pola data. Model dengan terlalu banyak lapisan dan neuron cenderung mengalami *overfitting*, sementara model yang terlalu sederhana tidak dapat menangkap kompleksitas data dengan baik (*underfitting*). Oleh karena itu, keseimbangan antara kompleksitas model dan performa generalisasi perlu diperhatikan dalam implementasi FFNN.

2. Pengaruh Fungsi Aktivasi

Fungsi aktivasi berperan penting dalam menentukan bagaimana sinyal diteruskan dalam jaringan. Hasil pengujian menunjukkan bahwa ReLU merupakan pilihan terbaik untuk *hidden layer* karena kemampuannya mengatasi *vanishing gradient problem*. Sementara itu, sigmoid dan tanh masih relevan untuk kasus tertentu, terutama pada lapisan output untuk tugas klasifikasi biner atau *multiclass*.

3. Pengaruh Learning Rate

Learning rate yang terlalu besar menyebabkan ketidakstabilan dalam pelatihan, sedangkan learning rate yang terlalu kecil memperlambat proses konvergensi. Hasil eksperimen menunjukkan bahwa pemilihan learning rate yang tepat sangat krusial untuk mencapai keseimbangan antara kecepatan konvergensi dan stabilitas pelatihan.

4. Pengaruh Inisialisasi Bobot

Inisialisasi bobot awal sangat mempengaruhi laju konvergensi dan stabilitas jaringan. Dari pengujian yang dilakukan, metode seperti Xavier Initialization dan He Initialization terbukti lebih efektif dibandingkan inisialisasi acak biasa, terutama dalam mempercepat proses pelatihan dan mencegah jaringan mengalami stuck pada kondisi sub-optimal.

5. Pengaruh Regularisasi

Teknik regularisasi seperti L1, L2 (*Ridge*), dan *Dropout* membantu meningkatkan kemampuan generalisasi model dengan mengurangi risiko *overfitting*. Dari hasil pengujian, penggunaan *dropout* dengan proporsi yang tepat mampu meningkatkan

akurasi model pada data uji tanpa mengorbankan kinerja pada data latih secara signifikan.

6. Pengaruh Normalisasi RMSNorm

Normalisasi menggunakan RMSNorm membantu menjaga kestabilan distribusi nilai dalam jaringan, yang pada akhirnya mempercepat konvergensi dan menghasilkan pelatihan yang lebih stabil. Namun, efektivitas metode ini tetap bergantung pada dataset dan arsitektur jaringan yang digunakan.

7. Perbandingan dengan Library Sklearn

Dibandingkan dengan model yang menggunakan library Sklearn, implementasi FFNN from scratch memberikan pemahaman lebih mendalam tentang bagaimana proses *feedforward*, *backpropagation*, dan optimasi bekerja. Namun, model dari Sklearn lebih praktis dan dapat memberikan hasil yang cukup baik dengan lebih sedikit usaha dalam tuning parameter.

Secara keseluruhan, implementasi FFNN from scratch dalam tugas ini memberikan wawasan penting mengenai pengaruh berbagai hyperparameter dan teknik optimasi terhadap kinerja model, sekaligus memperkuat pemahaman mengenai teori di balik *neural network* tiruan.

Saran

Berdasarkan hasil pengujian dan pembelajaran dari tugas ini, beberapa saran untuk pengembangan lebih lanjut dalam implementasi FFNN adalah sebagai berikut:

1. Optimasi Arsitektur Jaringan

Menggunakan metode *Grid Search* atau *Bayesian Optimization* dapat membantu dalam mencari kombinasi depth dan width yang optimal untuk berbagai jenis dataset. Eksperimen lebih lanjut diperlukan untuk memahami dampak variasi arsitektur pada performa model.

2. Pemilihan Fungsi Aktivasi yang Tepat

Pemilihan fungsi aktivasi harus disesuaikan dengan karakteristik dataset dan masalah yang ingin diselesaikan. ReLU tetap menjadi pilihan utama untuk *hidden layer*, sementara softmax atau sigmoid lebih cocok untuk tugas klasifikasi pada lapisan output.

3. Penyesuaian *Learning Rate* Secara Dinamis

Penggunaan *adaptive learning rate* seperti *Adam optimizer*, *learning rate decay*, atau

scheduler dapat meningkatkan efisiensi proses pelatihan dibandingkan menggunakan *learning rate* tetap.

4. Eksplorasi Teknik Inisialisasi Bobot yang Lebih Baik

Meskipun Xavier dan He Initialization sudah terbukti efektif, metode inisialisasi lain seperti LSUV (*Layer-Sequential Unit-Variance*) dapat diuji untuk meningkatkan performa *neural network* dalam kasus tertentu.

5. Penerapan Regularisasi yang Lebih Optimal

Regularisasi harus disesuaikan dengan ukuran dataset. Jika dataset kecil, penggunaan *dropout* yang lebih agresif dapat membantu mengurangi *overfitting*. Sementara itu, untuk dataset yang besar, L2 regularization dapat digunakan sebagai alternatif yang lebih stabil.

6. Eksplorasi Teknik Normalisasi yang Berbeda

Selain RMSNorm, teknik seperti *Batch Normalization*, *Layer Normalization*, atau *Group Normalization* dapat diuji untuk melihat perbedaannya dalam meningkatkan stabilitas dan efisiensi pelatihan model.

7. Perbandingan dengan Model yang Lebih Kompleks

Untuk mengukur seberapa baik performa FFNN yang telah diimplementasikan, disarankan untuk membandingkannya dengan model yang lebih kompleks seperti Convolutional Neural Network (CNN) atau Recurrent Neural Network (RNN) dalam skenario yang lebih kompleks.

Sebagai langkah lanjutan, implementasi FFNN dapat diuji dalam berbagai *problem domain* seperti *fraud detection*, analisis sentimen, atau prediksi harga saham untuk mengevaluasi efektivitasnya dalam skenario dunia nyata. Dengan menerapkan berbagai saran di atas, implementasi FFNN dapat terus ditingkatkan untuk menghasilkan model yang lebih akurat, efisien, dan mampu menangani berbagai jenis permasalahan secara optimal.

IV. PEMBAGIAN TUGAS


Berikut adalah tabel rincian mengenai pendistribusian kerja oleh anggota kelompok:

No	NIM Anggota	Nama Anggota	Deskripsi
1	13522070	Marzuli Suhada M	<p>Implementasi:</p> <ul style="list-style-type: none">- Implementasi fungsi aktivasi dan turunan Softmax- Implementasi inisialisasi bobot untuk Zero Initialization, Random Uniform, Random Normal- Implementasi method untuk save dan load- Instance model yang diinisialisasikan harus bisa menyimpan bobot & harus bisa menyimpan gradien bobot- Implementasi backward propagation dengan chain rule untuk menghitung gradien- Implementasi proses pelatihan dengan batch size, learning rate, jumlah epoch, dan verbose- Mengembalikan <i>history</i> pelatihan (training loss & validation loss)- Implementasi minimal 2 fungsi aktivasi lain yang sering digunakan <p>Pengujian:</p> <ul style="list-style-type: none">- Pengaruh regularisasi (jika mengerjakan)- Pengaruh normalisasi RMSNorm (jika mengerjakan)- Perbandingan dengan library Sklearn <p>Laporan</p>

2	13522072	Ahmad Mudabbir Arif	<p>Implementasi:</p> <ul style="list-style-type: none"> - Implementasi fungsi aktivasi dan turunan Sigmoid & Hyperbolic - Implementasi Loss Function dan turunan MSE - Implementasi method untuk menampilkan distribusi bobot dari tiap layer - Implementasi method untuk menampilkan distribusi gradien bobot dari tiap layer - Instance model yang diinisialisasikan harus bisa menyimpan bobot & harus bisa menyimpan gradien bobot - Implementasi weight update dengan gradient descent - Implementasi proses pelatihan dengan batch size, learning rate, jumlah epoch, dan verbose - Mengembalikan <i>history</i> pelatihan (training loss & validation loss) - Implementasi 2 metode inisialisasi bobot (<i>Bonus</i>) <p>Pengujian:</p> <ul style="list-style-type: none"> - Pengaruh learning rate - Pengaruh inisialisasi bobot <p>Laporan</p>
3	13522116	Naufal Adnan	<p>Implementasi:</p> <ul style="list-style-type: none"> - Setup inputan jumlah neuron - Implementasi fungsi aktivasi dan turunan Linear, ReLU - Implementasi Loss Function dan

			<p>turunan Binary Cross Entropy & Categorical Cross Entropy</p> <ul style="list-style-type: none"> - Implementasi method untuk menampilkan model berupa struktur jaringan beserta bobot dan gradien bobot tiap neuron dalam bentuk graf - Instance model yang diinisialisasikan harus bisa menyimpan bobot & harus bisa menyimpan gradien bobot - Implementasi forward propagation dengan batch input - Implementasi proses pelatihan dengan batch size, learning rate, jumlah epoch, dan verbose - Mengembalikan <i>history</i> pelatihan (training loss & validation loss) - Implementasi metode regularisasi L1 dan L2 - Implementasi metode normalisasi RMSNorm <p>Pengujian:</p> <ul style="list-style-type: none"> - Pengaruh depth dan width - Pengaruh fungsi aktivasi <p>Laporan</p>
--	--	--	--

V. REFERENSI

-  The spelled-out intro to neural networks and backpropagation: building micrograd
- <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