

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

Dosen:

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

Kelompok :

13522130 Justin Aditya Putra P.

13522155 Axel Santadi Warih

13522163 Atqiya Haydar Luqman

PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
SEMESTER II TAHUN 2023/2024

Daftar Isi

| | |
|--|-----------|
| Daftar Isi | 2 |
| 1. Deskripsi Persoalan | 3 |
| 2. Pembahasan | 8 |
| 2.1. Penjelasan Implementasi | 8 |
| 2.1.1. Deskripsi kelas beserta deskripsi atribut dan methodnya | 8 |
| 2.1.2. Penjelasan forward propagation | 16 |
| 2.1.3. Penjelasan backward propagation dan weight update | 17 |
| 2.2. Hasil pengujian | 19 |
| 2.2.1. Pengaruh depth dan width | 19 |
| 2.2.2. Pengaruh fungsi aktivasi | 24 |
| 2.2.3. Pengaruh learning rate | 26 |
| 2.2.4. Pengaruh inisialisasi bobot | 28 |
| 2.2.5. Perbandingan dengan library sklearn | 31 |
| 3. Kesimpulan dan Saran | 33 |
| 4. Pembagian Tugas | 34 |
| 5. Referensi | 35 |

1. Deskripsi Persoalan

Implementasikan suatu modul FFNN yang memenuhi ketentuan-ketentuan berikut:

- FFNN yang diimplementasikan dapat **menerima jumlah neuron dari tiap layer** (termasuk input layer dan output layer)
- FFNN yang diimplementasikan dapat **menerima fungsi aktivasi dari tiap layer**. Pilihan fungsi aktivasi yang harus diimplementasikan adalah sebagai berikut:

| Nama Fungsi Aktivasi | Definisi Fungsi |
|---------------------------|---|
| Linear | $Linear(x) = x$ |
| ReLU | $ReLU(x) = \max(0, x)$ |
| Sigmoid | $\sigma(x) = \frac{1}{1 + e^{-x}}$ |
| Hyperbolic Tangent (tanh) | $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ |
| Softmax | Untuk vector $\vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$, $softmax(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$ |

- FFNN yang diimplementasikan dapat **menerima fungsi loss** dari model tersebut. Pilihan loss function yang harus diimplementasikan adalah sebagai berikut:

| Nama Fungsi Loss | Definisi Fungsi |
|------------------|--|
| <u>MSE</u> | $MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$ |

| | |
|----------------------------------|--|
| <u>Binary Cross-Entropy</u> | $\mathcal{L}_{BCE} = -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$ <p> y_i = Actual binary label (0 or 1) \hat{y}_i = Predicted value of y_i n = Batch size </p> |
| <u>Categorical Cross-Entropy</u> | $\mathcal{L}_{CCE} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^C (y_{ij} \log \hat{y}_{ij})$ <p> y_{ij} = Actual value of instance i for class j \hat{y}_{ij} = Predicted value of y_{ij} C = Number of classes n = Batch size </p> |

- Catatan:
 - Binary cross-entropy merupakan kasus khusus categorical cross-entropy dengan kelas sebanyak 2
 - Log yang digunakan merupakan logaritma natural (logaritma dengan basis e)
- Terdapat mekanisme untuk **inisialisasi bobot** tiap neuron (termasuk bias). Pilihan metode inisialisasi bobot yang harus diimplementasikan adalah sebagai berikut:
 - **Zero initialization**
 - Random dengan distribusi **uniform**.
 - Menerima parameter lower bound (batas minimal) dan upper bound (batas maksimal)
 - Menerima parameter seed untuk reproducibility
 - Random dengan distribusi **normal**.
 - Menerima parameter mean dan variance
 - Menerima parameter seed untuk reproducibility
- Instance model yang diinisialisasikan harus bisa **menyimpan bobot** tiap neuron (termasuk bias)
- Instance model yang diinisialisasikan harus bisa **menyimpan gradien bobot** tiap neuron (termasuk bias)
- Instance model memiliki method untuk **menampilkan model** berupa **struktur jaringan** beserta **bobot** dan **gradien bobot** tiap neuron **dalam bentuk graf**. (Format graf dibebaskan)
- Instance model memiliki method untuk **menampilkan distribusi bobot** dari tiap layer.
 - Menerima masukan berupa list of integer (bisa disesuaikan ke struktur data lain sesuai kebutuhan) yang menyatakan layer mana saja yang distribusinya akan di-plot

- Instance model memiliki method untuk **menampilkan distribusi gradien bobot** dari tiap layer.
 - Menerima masukan berupa list of integer (bisa disesuaikan ke struktur data lain sesuai kebutuhan) yang menyatakan layer mana saja yang distribusinya akan di-plot
- Instance model memiliki method untuk **save** dan **load**
- Model memiliki implementasi **forward propagation** dengan ketentuan sebagai berikut:
 - Dapat menerima input berupa **batch**.
- Model memiliki implementasi **backward propagation** untuk menghitung perubahan gradien:
 - Dapat menangani perhitungan perubahan gradien untuk input data **batch**.
 - Gunakan konsep **chain rule** untuk menghitung gradien tiap bobot terhadap loss function.
 - Berikut merupakan **turunan pertama** untuk setiap fungsi aktivasi:

| Nama Fungsi Aktivasi | Turunan Pertama |
|---------------------------|---|
| Linear | $\frac{d(\text{Linear}(x))}{dx} = 1$ |
| ReLU | $\frac{d(\text{ReLU}(x))}{dx} = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$ |
| Sigmoid | $\frac{d(\sigma(x))}{dx} = \sigma(x)(1 - \sigma(x))$ |
| Hyperbolic Tangent (tanh) | $\frac{d(\tanh(x))}{dx} = \left(\frac{2}{e^x - e^{-x}} \right)^2$ |
| Softmax | <p>Untuk vector $\vec{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$,</p> $\frac{d(\text{softmax}(\vec{x}))}{d\vec{x}} = \begin{bmatrix} \frac{\partial(\text{softmax}(\vec{x})_1)}{\partial x_1} & \dots & \frac{\partial(\text{softmax}(\vec{x})_1)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial(\text{softmax}(\vec{x})_n)}{\partial x_1} & \dots & \frac{\partial(\text{softmax}(\vec{x})_n)}{\partial x_n} \end{bmatrix}$ <p>Dimana untuk $i, j \in \{1, \dots, n\}$,</p> $\frac{\partial(\text{softmax}(\vec{x})_i)}{\partial x_j} = \text{softmax}(\vec{x})_i (\delta_{i,j} - \text{softmax}(\vec{x})_j)$ $\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$ |

- Berikut merupakan **turunan pertama** untuk setiap fungsi loss terhadap bobot suatu FFNN (lanjutan sisanya menggunakan chain rule):

| Nama Fungsi Loss | Definisi Fungsi |
|----------------------------------|--|
| <u>MSE</u> | $\frac{\partial \mathcal{L}_{MSE}}{\partial W} = -\frac{2}{n} \sum_{i=1}^n \frac{\partial \mathcal{L}_{MSE}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial W} = -\frac{2}{n} \sum_{i=1}^n (y_i - \hat{y}_i) \frac{\partial \hat{y}_i}{\partial W}$ |
| <u>Binary Cross-Entropy</u> | $\frac{\partial \mathcal{L}_{BCE}}{\partial W} = -\frac{1}{n} \sum_{i=1}^n \frac{\partial \mathcal{L}_{BCE}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial W} = -\frac{1}{n} \sum_{i=1}^n \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} \frac{\partial \hat{y}_i}{\partial W}$ |
| <u>Categorical Cross-Entropy</u> | $\frac{\partial \mathcal{L}_{CCE}}{\partial W} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^c \frac{\partial \mathcal{L}_{CCE}}{\partial \hat{y}_{ij}} \frac{\partial \hat{y}_{ij}}{\partial W} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^c \frac{y_{ij}}{\hat{y}_{ij}} \frac{\partial \hat{y}_{ij}}{\partial W}$ |

- Model memiliki implementasi **weight update** dengan menggunakan **gradient descent** untuk memperbarui bobot berdasarkan gradien yang telah dihitung, berikut persamaannya:

$$W_{new} = W_{old} - \alpha \left(\frac{\partial \mathcal{L}}{\partial W_{old}} \right)$$

α = Learning rate

- Implementasi untuk pelatihan model harus memenuhi ketentuan berikut:
 - Dapat menerima parameter berikut:
 - Batch size
 - Learning rate
 - Jumlah epoch
 - Verbose
 - Verbose 0 berarti tidak menampilkan apa-apa selama pelatihan
 - Verbose 1 berarti hanya menampilkan progress bar beserta dengan kondisi training loss dan validation loss saat itu
 - Proses pelatihan mengembalikan **histori dari proses pelatihan** yang berisi **training loss** dan **validation loss tiap epoch**.
- Lakukan **pengujian** terhadap implementasi FFNN dengan ketentuan sebagai berikut:
 - Analisis pengaruh beberapa hyperparameter sebagai berikut:
 - Pengaruh **depth (banyak layer)** dan **width (banyak neuron per layer)**
 - Pilih 3 variasi kombinasi width (depth tetap) dan 3 variasi depth (width semua layer tetap)
 - Bandingkan hasil akhir prediksinya
 - Bandingkan grafik loss pelatihannya
 - Pengaruh **fungsi aktivasi** hidden layer

- Lakukan untuk setiap fungsi aktivasi yang diimplementasikan **kecuali softmax**.
 - Bandingkan hasil akhir prediksinya
 - Bandingkan grafik loss pelatihannya
 - Bandingkan distribusi bobot dan gradien bobot dari beberapa/semua layer pada model
- Pengaruh **learning rate**
 - Lakukan 3 variasi learning rate (nilainya dibebaskan)
 - Bandingkan hasil akhir prediksinya
 - Bandingkan grafik loss pelatihannya
 - Bandingkan distribusi bobot dan gradien bobot dari beberapa/semua layer pada model
- Pengaruh **inisialisasi bobot**
 - Lakukan untuk setiap metode inisialisasi bobot yang diimplementasikan
 - Bandingkan hasil akhir prediksinya
 - Bandingkan grafik loss pelatihannya
 - Bandingkan distribusi bobot dan gradien bobot dari beberapa/semua layer pada model
- Analisis perbandingan hasil prediksi dengan library sklearn MLP
 - Lakukan satu kali pelatihan dengan hyperparameter yang sama untuk kedua model
 - Hyperparameter yang digunakan dibebaskan
 - Bandingkan hasil akhir prediksinya saja
- Gunakan dataset berikut untuk menguji model: mnist_784
 - Gunakan method fetch_openml dari sklearn untuk memuat dataset
 - Berikut contoh untuk memuat dataset: Contoh

○ Akan ada beberapa **test case** yang akan diberikan oleh tim asisten (menyusul) Note: Tetap akan ada test case untuk penilaian, akan dilakukan saat asisten memeriksa tugas:

Pengujian dilakukan di file .ipynb terpisah

2. Pembahasan

2.1. Penjelasan Implementasi

2.1.1. Deskripsi kelas beserta deskripsi atribut dan methodnya

```
class NNetwork:
    def __init__(self, num_of_layers: int, layer_sizes: list[int], activation_functions: list[str] = None,
        verbose=False, weights: list[list[list[float]]] = None, biases: list[list[float]] = None):
        self.layer_sizes = layer_sizes
        self.verbose = verbose

        if activation_functions is None:
            activation_functions = ["sigmoid"] * (num_of_layers - 2) + ["softmax"] # Default: Sigmoid untuk hidden,
            Softmax untuk output
        elif len(activation_functions) != num_of_layers-1:
            raise ValueError(f"Jumlah fungsi aktivasi harus {num_of_layers-1}, bukan {len(activation_functions)}.")
        self.activation_functions = activation_functions

        self.layers: list[list[NNode]] = []
        for i in range(len(layer_sizes)):
            print("inisiasi layer", i)
            layer = [
                NNode(
                    weights=np.random.randn(layer_sizes[i-1]) if i > 0 else [],
                    bias=np.random.randn()
                )
                for _ in range(layer_sizes[i])
            ]
            print("inisiasi node selesai")
            self.layers.append(layer)
        print("inisiasi layer selesai")

        if self.verbose:
            print(f"Jaringan saraf dengan {num_of_layers} layer berhasil dibuat!")
            for i, layer in enumerate(self.layers):
                if i == 0:
                    print(f"Layer {i} (Input) - {len(layer)} neurons")
                else:
                    print(f"Layer {i} - {len(layer)} neurons, Aktivasi: {self.activation_functions[i-1]}")

    def initialize_weights(self, method: str = "zero", lower: float = -0.5, upper: float = 0.5, mean: float = 0.0,
        variance: float = 0.1, seed: int = None, verbose: bool = False):
        rng = np.random.default_rng(seed)

        for layer_idx in range(1, len(self.layers)):
            prev_layer_size = len(self.layers[layer_idx - 1])

            for node_idx, node in enumerate(self.layers[layer_idx]):
                num_weights = prev_layer_size

                weight_init_methods = {
                    "zero": lambda: np.zeros(num_weights),
                    "uniform": lambda: rng.uniform(lower, upper, num_weights),
                    "normal": lambda: rng.normal(mean, np.sqrt(variance), num_weights),
                }
```



```

        "xavier": lambda: rng.normal(0, np.sqrt(1 / prev_layer_size), num_weights),
        "he": lambda: rng.normal(0, np.sqrt(2 / prev_layer_size), num_weights)
    }

    if method not in weight_init_methods:
        raise ValueError(f'Metode inisialisasi '{method}' tidak dikenali. Gunakan 'zero', 'uniform', 'normal',
        'xavier', atau 'he'.')

    node.weights = weight_init_methods[method]()
    node.bias = rng.normal(0, np.sqrt(variance)) if method in ["normal", "xavier", "he"] else
rng.uniform(lower, upper)

    if verbose:
        print(f'Layer {layer_idx} - Node {node_idx}: weights={node.weights.tolist()},
bias={node.bias:.4f}')

def plot_network_graph(self):
    G = nx.DiGraph()
    pos = {}
    node_labels = {}

    layer_spacing = 2.0
    node_spacing = 1.5
    node_colors = []

    color_map = ["lightgreen", "lightblue", "salmon"]

    for layer_idx, layer in enumerate(self.layers):
        for node_idx, node in enumerate(layer):
            node_id = f"L{layer_idx}N{node_idx}"

            G.add_node(node_id, layer=layer_idx)
            pos[node_id] = (layer_idx * layer_spacing, -node_idx * node_spacing)
            node_labels[node_id] = f"N{node_idx}\nB: {node.bias:.2f}"

            if layer_idx == 0:
                node_colors.append(color_map[0])
            elif layer_idx == len(self.layers) - 1:
                node_colors.append(color_map[2])
            else:
                node_colors.append(color_map[1])

            if layer_idx > 0:
                prev_layer = self.layers[layer_idx - 1]
                for prev_idx, prev_node in enumerate(prev_layer):
                    prev_id = f"L{layer_idx-1}N{prev_idx}"
                    weight = node.weights[prev_idx]

                    G.add_edge(prev_id, node_id, weight=f"{weight:.2f}")

    plt.figure(figsize=(12, 6))
    nx.draw(
        G, pos, with_labels=True, labels=node_labels, node_color=node_colors,
        edge_color="gray", node_size=2000, font_size=10
    )

```

```

edge_labels = {(u, v): d["weight"] for u, v, d in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8)

plt.title("Struktur Jaringan Saraf Tiruan", fontsize=14)
plt.show()

def plot_weight_distribution(self, layers: list[int], show_grid: bool = True):
    plt.figure(figsize=(10, 5))

    found_data = False
    colors = plt.cm.viridis(np.linspace(0, 1, len(layers)))

    for idx, layer_idx in enumerate(layers):
        if layer_idx < 1 or layer_idx >= len(self.layers):
            print(f"Layer {layer_idx} tidak valid.")
            continue

        weights = np.array([weight for node in self.layers[layer_idx] for weight in node.weights])

        if weights.size == 0:
            print(f"Layer {layer_idx} tidak memiliki bobot.")
            continue

        found_data = True

        mean = np.mean(weights)
        std = np.std(weights)

        plt.hist(weights, bins=20, alpha=0.6, label=f"Layer {layer_idx} ( $\mu$ ={{mean:.3f}},  $\sigma$ ={{std:.3f}})",
            color=colors[idx])

    if not found_data:
        print("Tidak ada data bobot yang dapat ditampilkan.")
        return

    plt.xlabel("Nilai Bobot")
    plt.ylabel("Frekuensi")
    plt.legend()
    plt.title("Distribusi Bobot per Layer")
    if show_grid:
        plt.grid(True, linestyle="--", alpha=0.5)
    plt.show()

def plot_gradient_distribution(self, layers: list[int], show_grid: bool = True):
    plt.figure(figsize=(10, 5))

    found_data = False
    colors = plt.cm.plasma(np.linspace(0, 1, len(layers)))

    for idx, layer_idx in enumerate(layers):
        if layer_idx < 1 or layer_idx >= len(self.layers):
            print(f"Layer {layer_idx} tidak valid.")
            continue

```

```

gradients = np.array([grad for node in self.layers[layer_idx] for grad in node.gradients])

if gradients.size == 0:
    print(f"Layer {layer_idx} tidak memiliki gradien.")
    continue

found_data = True

mean = np.mean(gradients)
std = np.std(gradients)

plt.hist(gradients, bins=20, alpha=0.6, label=f"Layer {layer_idx} ( $\mu$ ={{mean:.3f}},  $\sigma$ ={{std:.3f}})",
color=colors[idx])

if not found_data:
    print("Tidak ada data gradien yang dapat ditampilkan.")
    return

plt.xlabel("Nilai Gradien")
plt.ylabel("Frekuensi")
plt.legend()
plt.title("Distribusi Gradien Bobot per Layer")
if show_grid:
    plt.grid(True, linestyle="--", alpha=0.5)
plt.show()

def save_model(self, filename: str, verbose: bool = True):
    if not filename.endswith(".pkl"):
        filename += ".pkl"

    try:
        with open(filename, 'wb') as f:
            pickle.dump(self, f, protocol=pickle.HIGHEST_PROTOCOL)
        if verbose:
            print(f"✅ Model berhasil disimpan ke '{filename}'")
    except Exception as e:
        if verbose:
            print(f"❌ Gagal menyimpan model: {e}")

    @staticmethod
    def load_model(filename: str, verbose: bool = True):
        if not filename.endswith(".pkl"):
            filename += ".pkl"

        if not os.path.exists(filename):
            if verbose:
                print(f"❌ File '{filename}' tidak ditemukan.")
            return None

        with suppress(pickle.UnpicklingError, EOFError, Exception):
            with open(filename, 'rb') as f:
                model = pickle.load(f)

        if isinstance(model, NNetwork):
            if verbose:

```

```

        print(f"✅ Model berhasil dimuat dari '{filename}')"
        return model
    else:
        if verbose:
            print(f"❌ File '{filename}' bukan model NNetwork yang valid.")

    if verbose:
        print(f"❌ Gagal memuat model: File '{filename}' mungkin korup atau tidak kompatibel.")
    return None

def update_weights(self, learning_rate: float = 0.01):
    for layer_idx in range(1, len(self.layers)):
        for node in self.layers[layer_idx]:
            if node.gradients is not None and node.bias_gradient is not None:
                node.weights -= learning_rate * node.gradients
                node.bias -= learning_rate * node.bias_gradient

        node.reset_gradients()

```

Kelas NNetwork adalah sebuah implementasi Neural Network yang memiliki beberapa layers, masing-masing layers memiliki neuron yang terhubung melalui bobot dan bias.

Atribut:

1. layer_sizes

Menyimpan ukuran setiap lapisan dalam jaringan saraf, yang menunjukkan jumlah neuron pada setiap lapisan.

2. verbose

Sebuah parameter boolean untuk menentukan apakah informasi lebih lanjut tentang proses pembuatan dan pelatihan jaringan akan ditampilkan.

3. activation_functions

Menyimpan daftar fungsi aktivasi untuk setiap lapisan, dengan fungsi aktivasi default adalah sigmoid untuk lapisan tersembunyi dan softmax untuk lapisan output.

4. layers

sebuah daftar dua dimensi yang menyimpan objek NNode untuk setiap lapisan. Setiap lapisan berisi neuron-neuron yang diwakili oleh objek NNode dengan bobot dan bias terkait.

Method:

1. __init__(self, num_of_layers: int, layer_sizes: list[int], activation_functions: list[str] = None, verbose=False, weights: list[list[list[float]]] = None, biases: list[list[float]] = None)

Konstruktor yang digunakan untuk menginisialisasi jaringan saraf. Menerima parameter untuk menentukan jumlah lapisan, ukuran lapisan, fungsi aktivasi, dan verbose untuk menampilkan status saat proses inisialisasi.

2. **initialize_weights(self, method: str = "zero", lower: float = -0.5, upper: float = 0.5, mean: float = 0.0, variance: float = 0.1, seed: int = None, verbose: bool = False)**
Digunakan untuk menginisialisasi bobot pada jaringan saraf menggunakan berbagai metode, seperti zero, uniform, normal, xavier, atau he. Metode ini mengubah bobot dan bias neuron berdasarkan parameter yang diberikan.
3. **plot_network_graph(self)**
Metode ini menghasilkan visualisasi grafik jaringan saraf, dengan node yang mewakili neuron dan edge yang mewakili bobot antar neuron. Visualisasi menggunakan jaringan berarah (directed graph) yang menggambarkan hubungan antar neuron di setiap lapisan.
4. **plot_weight_distribution(self, layers: list[int], show_grid: bool = True)**
Menampilkan distribusi bobot dalam bentuk histogram untuk lapisan yang ditentukan, memungkinkan visualisasi penyebaran bobot antar neuron pada lapisan-lapisan tertentu.
5. **plot_gradient_distribution(self, layers: list[int], show_grid: bool = True)**
Menampilkan distribusi gradien bobot dalam bentuk histogram untuk lapisan-lapisan yang ditentukan, memungkinkan visualisasi bagaimana gradien berubah seiring dengan pelatihan model.
6. **save_model(self, filename: str, verbose: bool = True)**
Menyimpan model jaringan saraf yang sudah dilatih ke file menggunakan format pickle. Nama file akan memiliki ekstensi .pkl jika belum ada.
7. **load_model(filename: str, verbose: bool = True)**
Memuat model yang sudah disimpan sebelumnya dari file pickle dan mengembalikan objek model yang dimuat.
8. **update_weights(self, learning_rate: float = 0.01)**
Mengupdate bobot dan bias setiap neuron dengan menggunakan gradien yang dihitung dan parameter learning_rate. Bias dan bobot diperbarui berdasarkan gradien dari lapisan-lapisan yang ada.

```
class NNode:
    _id_counter = 0

    def __init__(self, weights: list[float] = None, bias: float = 0.0):
        self.id = NNode._id_counter
        NNode._id_counter += 1
        self.weights = np.array(weights) if weights is not None else np.array([])
        self.bias = bias
        self.gradients = np.zeros_like(self.weights)
        self.bias_gradient = 0.0
```

```
def reset_gradients(self):
    """Reset gradien bobot dan bias setelah update parameter."""
    self.gradients = np.zeros_like(self.weights)
    self.bias_gradient = 0.0

def __repr__(self):
    return f"NNode(id={self.id}, weights={self.weights}, bias={self.bias})"
```

Kelas NNode mewakili sebuah neuron dalam Neural Network. Setiap neuron memiliki bobot, bias, serta gradien bobot dan bias yang dihitung selama proses pelatihan backpropagation.

Atribut:

1. `_id_counter`

Sebuah variabel kelas yang digunakan untuk menghasilkan ID unik untuk setiap objek NNode. Setiap kali neuron baru dibuat, nilai `_id_counter` akan bertambah satu untuk memastikan setiap neuron memiliki ID yang berbeda.

2. `id`

ID unik untuk neuron yang dihasilkan secara otomatis saat objek NNode diinisialisasi. ID ini digunakan untuk membedakan setiap neuron dalam jaringan.

3. `weights`

Daftar atau array NumPy yang menyimpan bobot-bobot yang menghubungkan neuron ini dengan neuron-neuron di lapisan sebelumnya. Jika bobot tidak diberikan, maka akan diinisialisasi sebagai array kosong.

4. `bias`

Nilai bias untuk neuron ini. Bias adalah nilai yang ditambahkan pada hasil output fungsi aktivasi, yang memungkinkan model untuk belajar lebih baik pada data yang tidak terpusat di sekitar nol. Bias ini diinisialisasi dengan nilai default 0.0 jika tidak diberikan.

5. `gradients`

Gradien untuk bobot neuron, yang diinisialisasi dengan array nol yang memiliki ukuran yang sama dengan `weights`. Gradien ini dihitung selama proses backpropagation dan digunakan untuk memperbarui bobot.

6. `bias_gradient`

Gradien untuk bias neuron yang juga diinisialisasi dengan nilai 0.0. Ini digunakan untuk memperbarui nilai bias selama proses pelatihan.

Method:

1. `__init__(self, weights: list[float] = None, bias: float = 0.0)`

Konstruktor untuk membuat neuron baru. Jika bobot tidak diberikan, maka bobot akan diinisialisasi sebagai array kosong. Bias diinisialisasi dengan nilai yang diberikan atau default 0.0. Gradien bobot dan bias juga diinisialisasi sebagai nol.

2. `reset_gradients(self)`

Metode ini digunakan untuk mereset gradien bobot dan bias setelah pembaruan dilakukan selama proses pelatihan. Gradien diatur kembali menjadi nol untuk mempersiapkan proses perhitungan gradien berikutnya.

3. `__repr__(self)`

Metode ini mendefinisikan bagaimana objek NNode akan direpresentasikan dalam bentuk string. Ini berguna untuk debugging atau saat mencetak objek neuron. Menampilkan ID neuron, bobot, dan bias neuron tersebut.

Implementasi Fungsi Aktivasi

```
import numpy as np

def linear(x):
    return x

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

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

def relu_derivative(x):
    return np.where(x > 0, 1, 0) # Turunan ReLU: 1 untuk x > 0, 0 untuk x <= 0

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

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x)) # Turunan Sigmoid

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

def tanh_derivative(x):
    return 1 - np.tanh(x)**2 # Turunan Tanh

def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

def softmax_derivative(x):
    return softmax(x) * (1 - softmax(x)) # Ini hanya berlaku untuk loss MSE, gunakan Jacobian untuk cross-entropy

activation_functions = {
    "linear": linear,
    "relu": relu,
```

```

"sigmoid": sigmoid,
"tanh": tanh,
"softmax": softmax
}

activation_derivatives = {
    "linear": linear_derivative,
    "relu": relu_derivative,
    "sigmoid": sigmoid_derivative,
    "tanh": tanh_derivative,
    "softmax": softmax_derivative
}

```

Implementasi Fungsi Loss

```

import numpy as np

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

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

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

```

2.1.2. Penjelasan forward propagation

```

def forward_propagation(self, inputs: np.ndarray):
    if len(inputs.shape) == 1:
        inputs = inputs.reshape(1, -1)

    if len(inputs.shape) != 2:
        raise ValueError("Input harus berbentuk (batch_size, input_size)")

    current_input = inputs

    for layer_idx in range(1, len(self.layers)):
        activation_function = self.activation_functions[layer_idx - 1]

        weights = np.array([node.weights for node in self.layers[layer_idx]])
        biases = np.array([node.bias for node in self.layers[layer_idx]])

        z = np.dot(current_input, weights.T) + biases

```



```

if activation_function == "sigmoid":
    current_input = FA.sigmoid(z)
elif activation_function == "relu":
    current_input = FA.relu(z)
elif activation_function == "tanh":
    current_input = FA.tanh(z)
elif activation_function == "linear":
    current_input = FA.linear(z)
elif activation_function == "softmax":
    current_input = FA.softmax(z)
else:
    raise ValueError(f'Fungsi aktivasi '{activation_function}' tidak dikenali.')

return current_input

```

Metode `forward_propagation` bertujuan untuk menghitung keluaran neural network berdasarkan input yang diberikan. Proses diawali dengan penyesuaian bentuk input agar sesuai dengan format (`batch_size`, `input_size`). Selanjutnya, input awal disimpan dalam variabel `current_input`, yang akan diproses melalui setiap layers, kecuali input layer.

Pada setiap layer, metode ini mengambil fungsi aktivasi yang telah ditentukan dalam atribut `self.activation_functions`, serta bobot dan bias dari setiap neuron yang disusun dalam bentuk array atau matriks. Selanjutnya, nilai aktivasi dihitung menggunakan operasi matriks $Z = WX + B$, di mana W adalah bobot, X adalah input, dan B adalah bias. Nilai hasil perhitungan tersebut kemudian diterapkan ke fungsi aktivasi yang sesuai, seperti sigmoid, ReLU, tanh, linear, atau softmax, untuk menentukan respons neuron terhadap input yang diberikan.

Output yang dihasilkan dari setiap layer digunakan sebagai input untuk layer berikutnya hingga mencapai layer terakhir. Akhirnya, metode ini mengembalikan hasil perhitungan sebagai output akhir, yang merepresentasikan prediksi model berdasarkan input yang diberikan.

2.1.3. Penjelasan backward propagation dan weight update

```

def backward_propagation(self, inputs: np.ndarray, targets: np.ndarray, learning_rate: float = 0.01):
    batch_size = inputs.shape[0]

    if batch_size == 0:
        raise ValueError("Batch size tidak boleh nol")

    if len(inputs.shape) == 1:
        inputs = inputs.reshape(1, -1)

    if len(inputs.shape) != 2:
        raise ValueError("Input harus berbentuk (batch_size, input_size)")

    activations = [inputs]
    current_input = inputs

```

```

for layer_idx in range(1, len(self.layers)):
    activation_function = self.activation_functions[layer_idx - 1]

    weights = np.array([node.weights for node in self.layers[layer_idx]])
    biases = np.array([node.bias for node in self.layers[layer_idx]])

    z = np.dot(current_input, weights.T) + biases

    if activation_function == "sigmoid":
        current_input = FA.sigmoid(z)
    elif activation_function == "relu":
        current_input = FA.relu(z)
    elif activation_function == "tanh":
        current_input = FA.tanh(z)
    elif activation_function == "linear":
        current_input = FA.linear(z)
    elif activation_function == "softmax":
        current_input = FA.softmax(z)
    else:
        raise ValueError(f'Fungsi aktivasi '{activation_function}' tidak dikenali.')

    activations.append(current_input)

errors = [None] * len(self.layers)

output_activations = activations[-1]
loss_derivative = output_activations - targets

if self.activation_functions[-1] == "softmax":
    errors[-1] = loss_derivative
else:
    errors[-1] = loss_derivative *
FA.activation_derivatives[self.activation_functions[-1]](output_activations)

for layer_idx in range(len(self.layers) - 2, 0, -1):
    error_signal = errors[layer_idx + 1]
    activation_derivative = FA.activation_derivatives[self.activation_functions[layer_idx - 1]](activations[layer_idx])

    weights_next_layer = np.array([node.weights for node in self.layers[layer_idx + 1]])
    errors[layer_idx] = np.dot(error_signal, weights_next_layer) * activation_derivative

for layer_idx in range(1, len(self.layers)):
    prev_activation = activations[layer_idx - 1]
    error_signal = errors[layer_idx]

    for node_idx, node in enumerate(self.layers[layer_idx]):
        node.gradients = np.dot(prev_activation.T, error_signal[:, node_idx]) / batch_size

        node.bias_gradient = np.mean(error_signal[:, node_idx], axis=0)

```

```
self.update_weights(learning_rate)

return np.mean(loss_derivative**2)
```

Metode `backward_propagation` bertujuan untuk menghitung dan memperbarui bobot serta bias jaringan saraf berdasarkan perbedaan antara output prediksi dan target yang diinginkan. Proses ini diawali dengan memastikan bahwa input memiliki bentuk yang sesuai, yaitu `(batch_size, input_size)`, agar dapat diproses dengan benar. Jika batch size bernilai nol, metode ini akan mengeluarkan error karena proses pembelajaran memerlukan sampel data yang valid.

Langkah pertama dalam proses ini adalah melakukan forward propagation untuk menyimpan aktivasi setiap layer dalam daftar `activations`. Aktivasi ini digunakan dalam perhitungan gradien selama backpropagation. Setelah seluruh layer diproses, metode menghitung error pada output layer berdasarkan selisih antara hasil prediksi (`output_activations`) dan target yang diberikan (`targets`). Jika output layer menggunakan fungsi aktivasi softmax, error dihitung langsung dari selisih tersebut. Jika menggunakan fungsi aktivasi lain, error dikalikan dengan turunan dari fungsi aktivasi yang sesuai untuk mendapatkan sinyal error.

Selanjutnya, metode ini melakukan backpropagation of errors, yaitu menghitung error pada setiap hidden layer dengan menggunakan rantai propagasi error. Error dari layer berikutnya dikalikan dengan bobot terkait, lalu dikalikan dengan turunan fungsi aktivasi dari layer saat ini. Proses ini berjalan mundur dari output layer hingga hidden layer pertama, sehingga error dapat menyebar ke seluruh jaringan secara terstruktur.

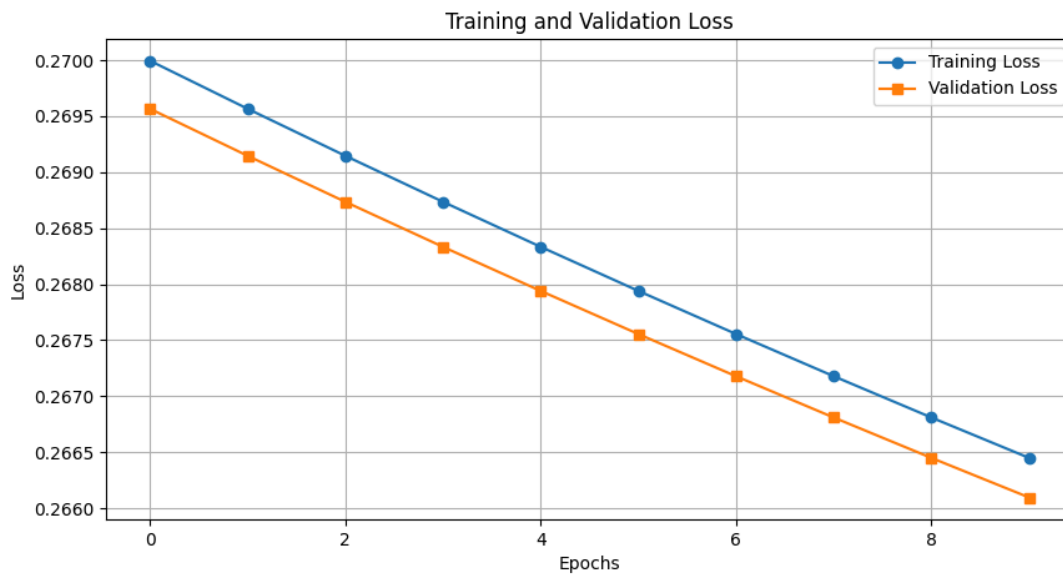
Setelah error dihitung untuk setiap layer, metode ini memperbarui gradien bobot dan bias dengan menggunakan rata-rata propagasi error yang dihitung sebelumnya. Bobot diperbarui berdasarkan hasil perkalian antara input sebelumnya dengan error sinyal, sedangkan bias diperbarui dengan rata-rata error sinyal. Semua perhitungan ini dibagi dengan `batch_size` agar pembaruan lebih stabil dalam kasus mini-batch gradient descent.

Langkah terakhir dari backward propagation adalah memanggil fungsi `update_weights`, yang bertanggung jawab untuk memperbarui bobot dan bias menggunakan learning rate yang telah ditentukan. Metode ini juga mengembalikan nilai loss, yang dihitung sebagai rata-rata kuadrat dari error, untuk memantau sejauh mana model telah belajar dari data yang diberikan.

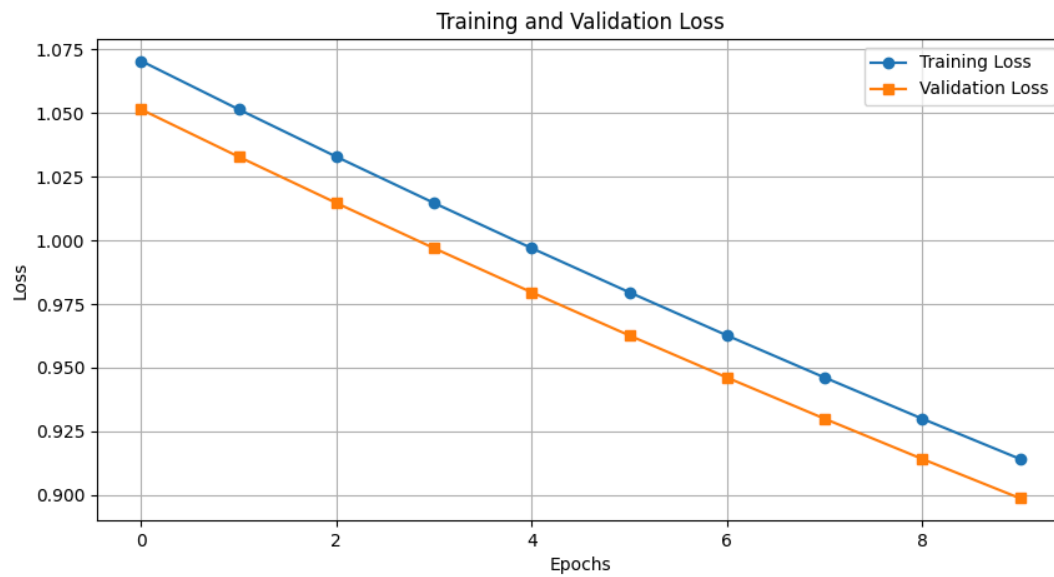
2.2. Hasil pengujian

2.2.1. Pengaruh depth dan width

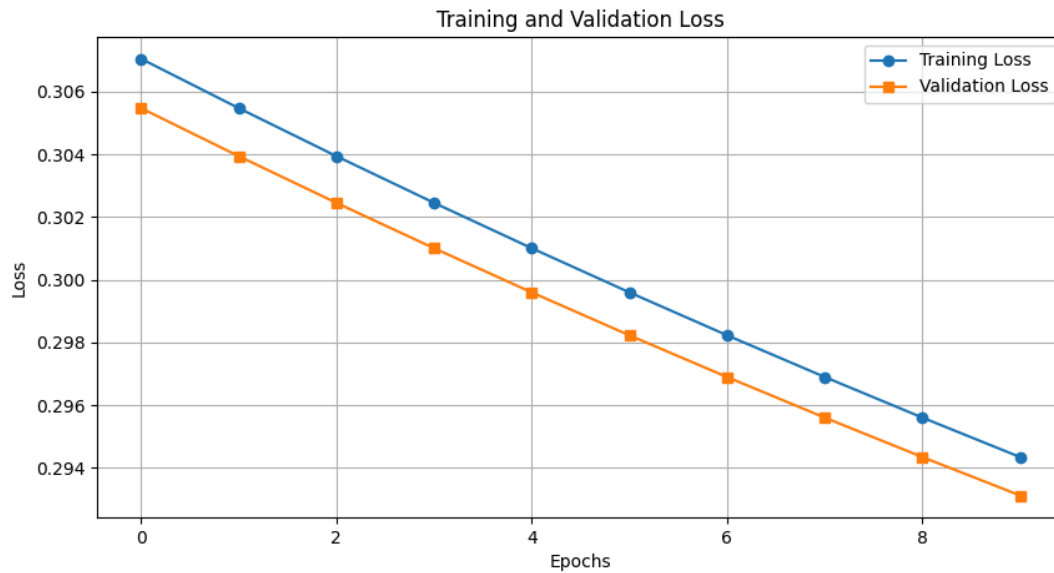
2.2.1.1. Depth



Gambar 2.2.1.1.1 Hasil depth 1 hidden layer



Gambar 2.2.1.1.2 Hasil depth 2 hidden layer

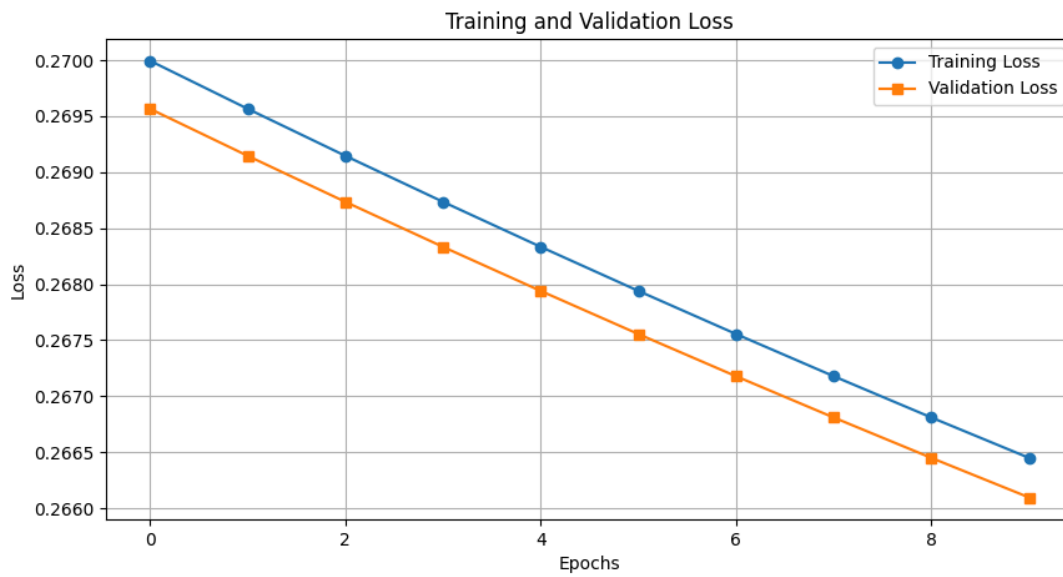


Gambar 2.2.1.1.3 Hasil depth 3 hidden layer

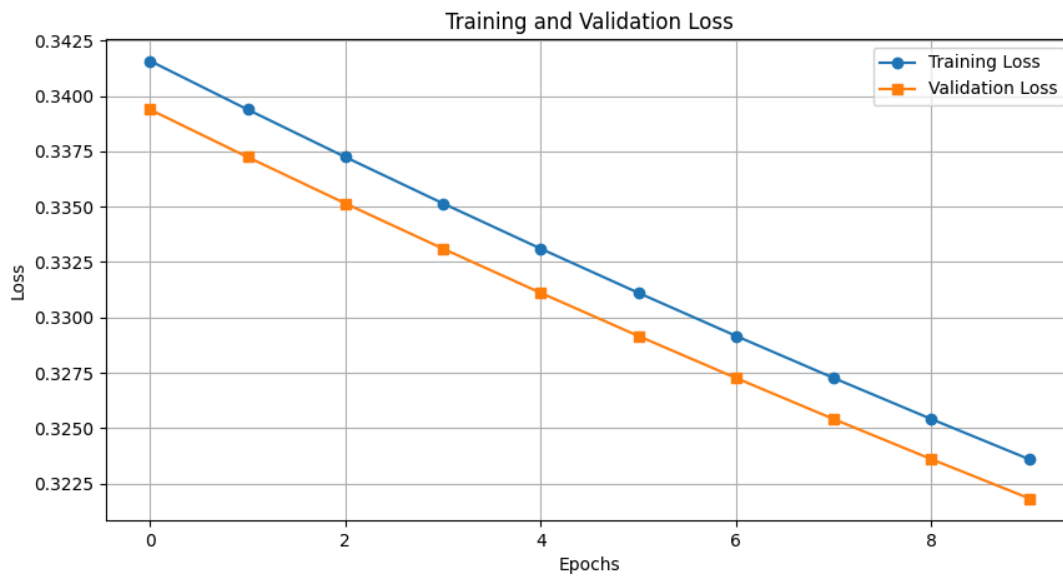
Dari ketiga plot tersebut, kita dapat melakukan perbandingan:

1. 1 hidden layer
 - Loss awal paling rendah dibanding model lainnya.
 - Kurva menunjukkan pola yang tetap stabil dengan penurunan loss yang cukup baik.
 - Validasi loss tetap dekat dengan training loss, menunjukkan generalisasi yang cukup baik.
2. 2 hidden layer
 - Loss awal paling tinggi dibandingkan model lainnya.
 - Penurunan loss lebih tajam pada epoch awal tetapi tetap menurun stabil di epoch selanjutnya.
 - Ada kemungkinan model lebih kompleks, sehingga bisa meningkatkan performa tetapi juga berisiko overfitting jika jumlah data tidak mencukupi.
3. 3 hidden layer
 - Loss awal cukup tinggi, tetapi tidak setinggi 2 hidden layer..
 - Kurva training loss dan validation loss tetap berdekatan, menunjukkan bahwa model tidak mengalami overfitting.
 - Konvergensi cukup baik, tetapi kemungkinan performa kurang optimal dibanding model dengan lebih banyak hidden layer.

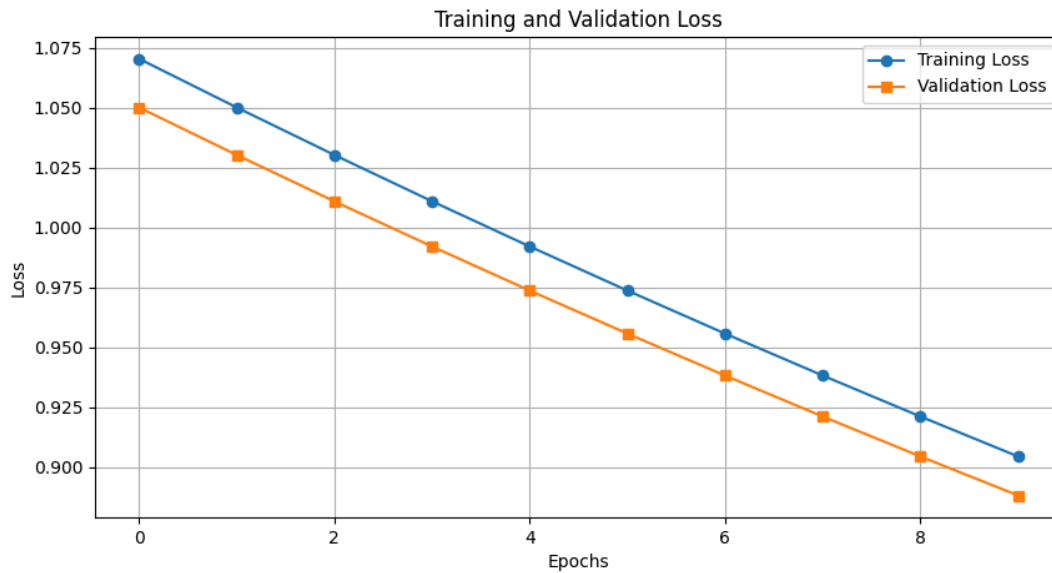
2.2.1.2. Width



Gambar 2.2.1.2.1 Hasil plot 2 buah node



Gambar 2.2.1.2.2 Hasil plot 3 buah node



Gambar 2.2.1.2.3 Hasil plot 4 buah node

Dari ketiga plot tersebut, kita dapat melakukan perbandingan:

1. Training History 1 (2 node, 1 hidden layer)

- Loss awal lebih rendah dibandingkan model dengan lebih banyak node.
- Penurunan loss cukup stabil.
- Tidak ada indikasi overfitting.

2. Training History 2 (3 node, 1 hidden layer)

- Loss awal lebih tinggi dibandingkan model dengan 2 node, tetapi turun dengan pola yang sama.
- Model lebih kompleks, namun tidak menunjukkan peningkatan drastis dalam konvergensi.

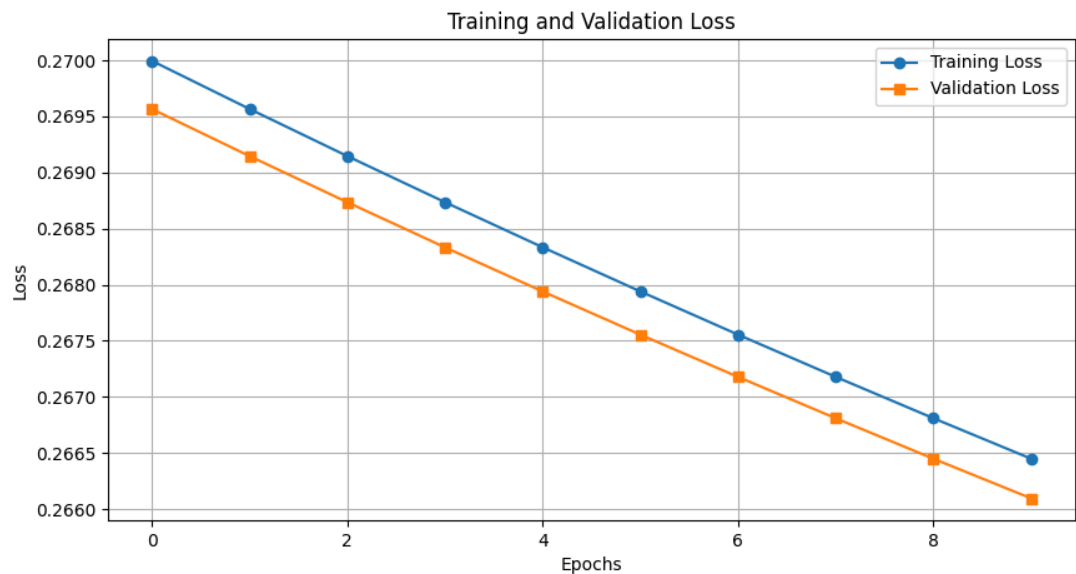
3. Training History 3 (4 node, 1 hidden layer)

- Loss awal jauh lebih tinggi dibandingkan kedua model sebelumnya.
- Meskipun loss terus berkurang, model tampaknya mengalami kesulitan dalam menyesuaikan diri secara optimal.

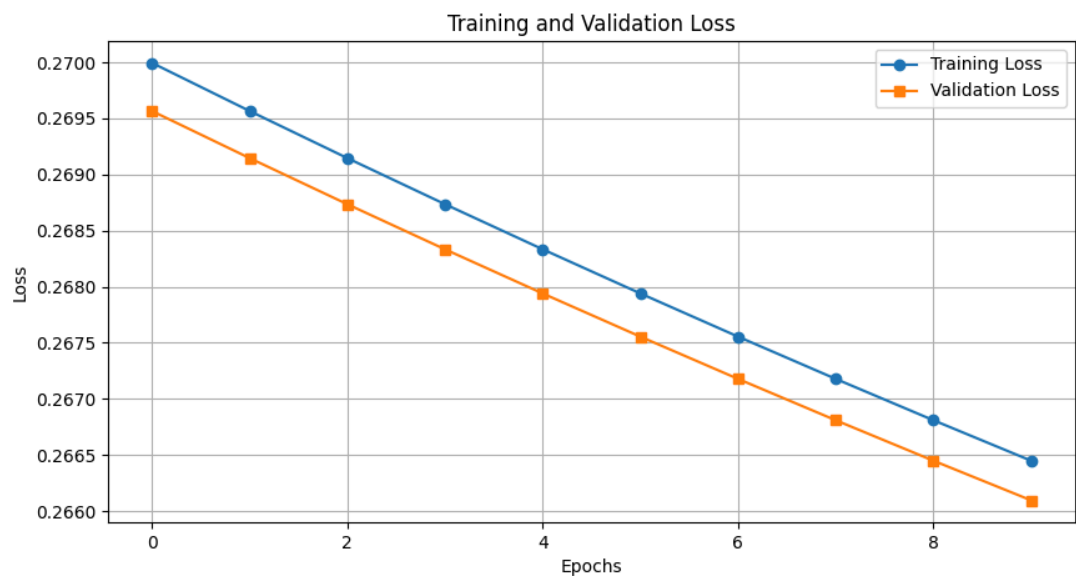
Dari fakta tersebut, bisa ditarik kesimpulan:

- Menambah jumlah node dalam satu hidden layer tidak selalu memberikan peningkatan performa yang signifikan.
- Model dengan 2 node tampaknya lebih efisien dalam menurunkan loss tanpa meningkatkan kompleksitas terlalu banyak.
- Model dengan 4 node memiliki loss awal yang tinggi, menunjukkan bahwa peningkatan jumlah node bisa menyebabkan model lebih sulit dikonfigurasi dengan baik.

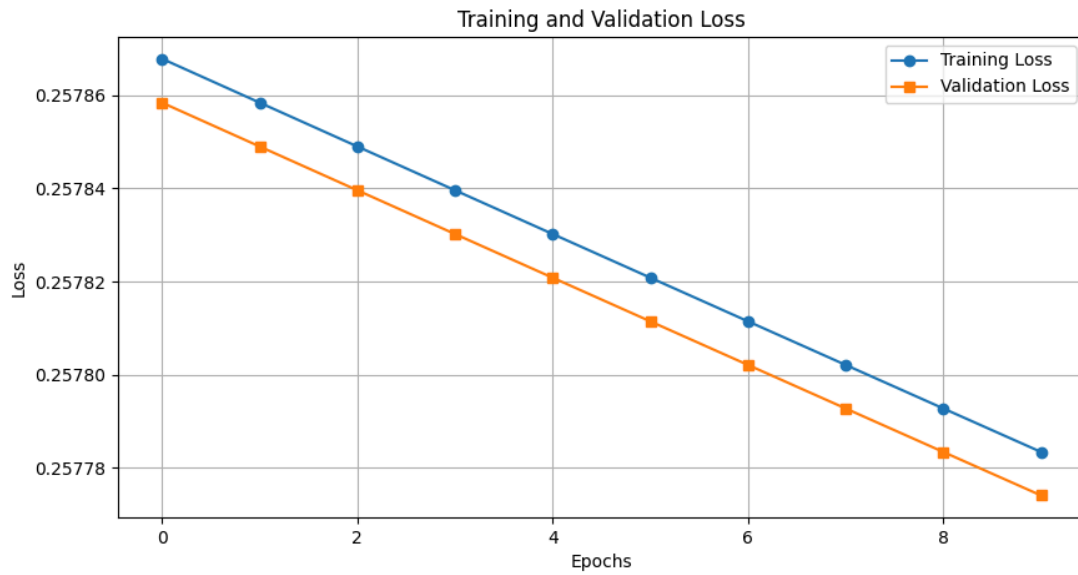
2.2.2. Pengaruh fungsi aktivasi



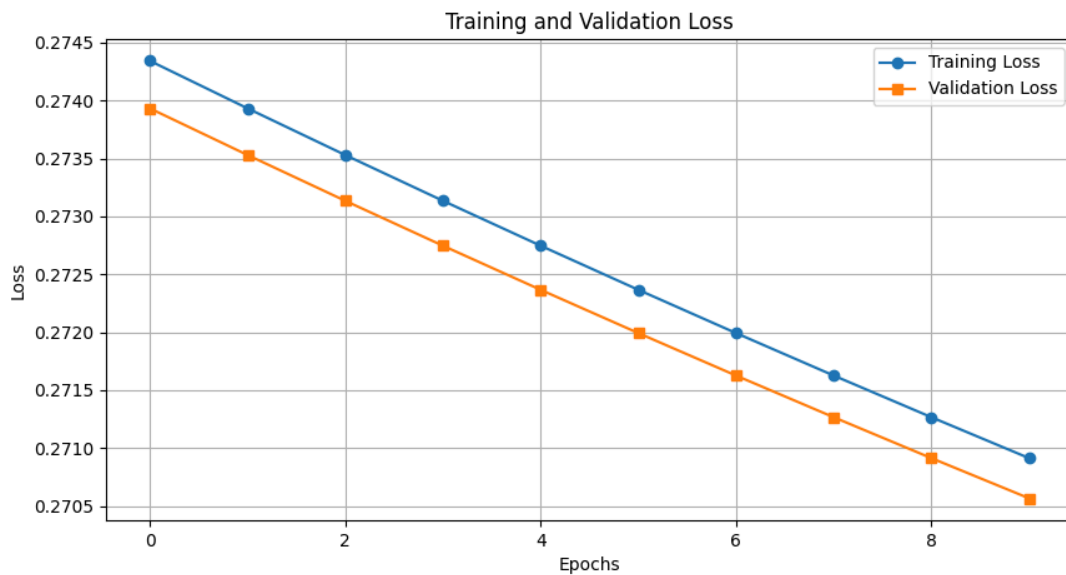
Gambar 2.2.2.1 Hasil aktivasi linear



Gambar 2.2.2.2 Hasil aktivasi ReLU



Gambar 2.2.2.3 Hasil aktivasi Sigmoid



Gambar 2.2.2.4 Hasil aktivasi Tanh

Dari keempat plot tersebut, kita dapat melakukan perbandingan:

1. Linear

- Loss awal: sekitar 0.2700
- Loss akhir: sekitar 0.2660
- Kurva penurunan loss berjalan stabil tanpa adanya plateau atau fluktuasi yang berarti.

2. ReLU

- Loss awal: sekitar 0.2700
- Loss akhir: sekitar 0.2660

- Kurva sangat mirip dengan Linear, menunjukkan performa yang hampir serupa dalam skenario ini.

3. Sigmoid

- Loss awal: sekitar 0.25786
- Loss akhir: sekitar 0.25778
- Penurunan loss lebih kecil dibandingkan dengan Linear dan ReLU, menunjukkan bahwa model dengan aktivasi Sigmoid mengalami saturasi lebih cepat.

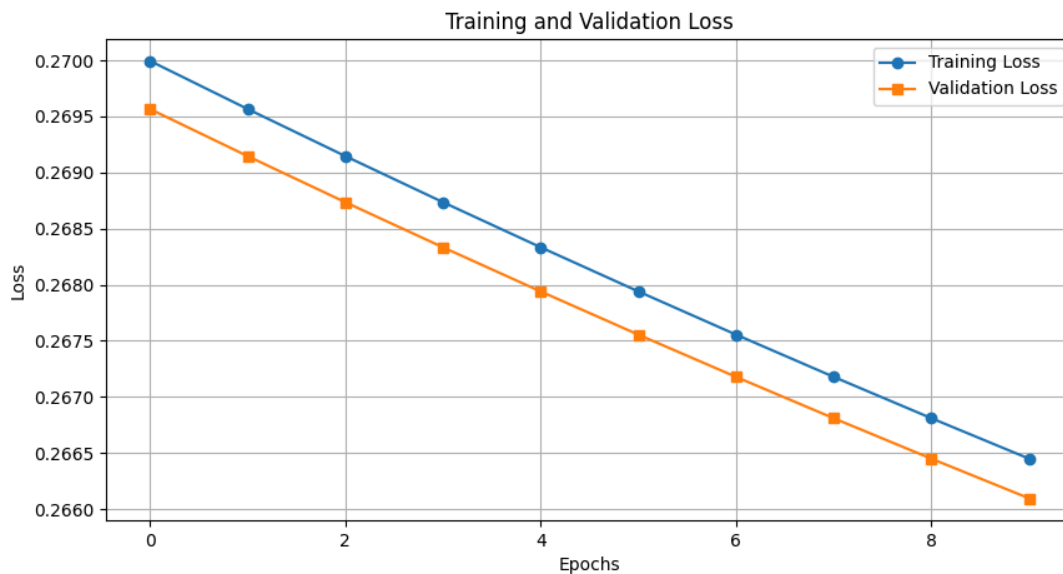
4. Tanh

- Loss awal: sekitar 0.2745
- Loss akhir: sekitar 0.2705
- Penurunan loss berjalan stabil, mirip dengan Linear dan ReLU, tetapi nilai loss awal dan akhirnya lebih tinggi dibandingkan keduanya.

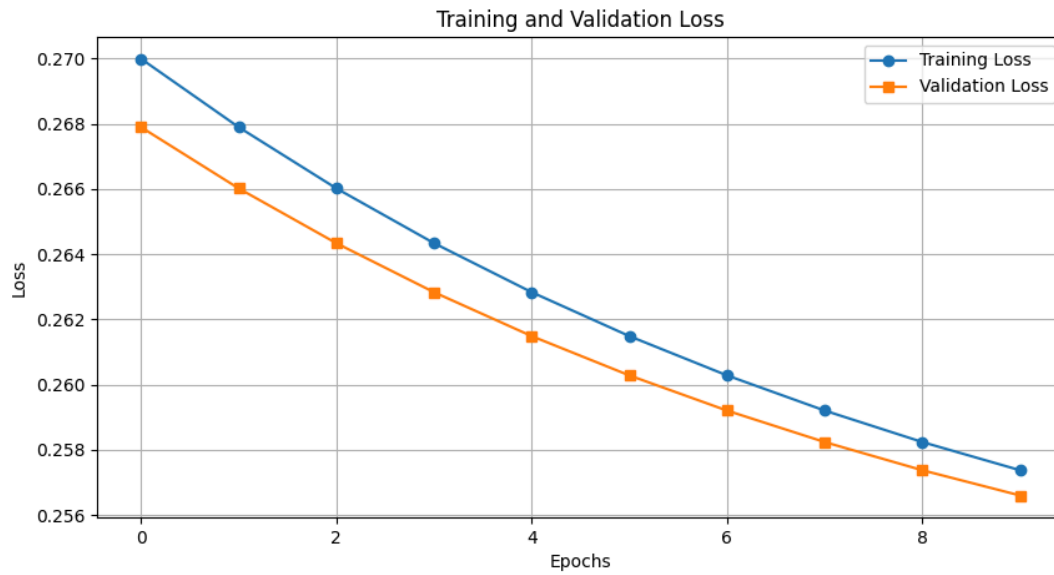
Dari hasil perbandingan tersebut, kita bisa mengambil kesimpulan:

- Linear dan ReLU menunjukkan pola penurunan loss yang mirip, menandakan bahwa ReLU tidak memberikan keunggulan yang signifikan dalam konteks model ini.
- Sigmoid memiliki loss awal dan akhir yang lebih kecil, tetapi rentang perbaikannya sangat kecil, mengindikasikan potensi saturasi gradien.
- Tanh menunjukkan performa lebih baik daripada Sigmoid, tetapi masih memiliki loss yang lebih tinggi dibandingkan Linear dan ReLU.

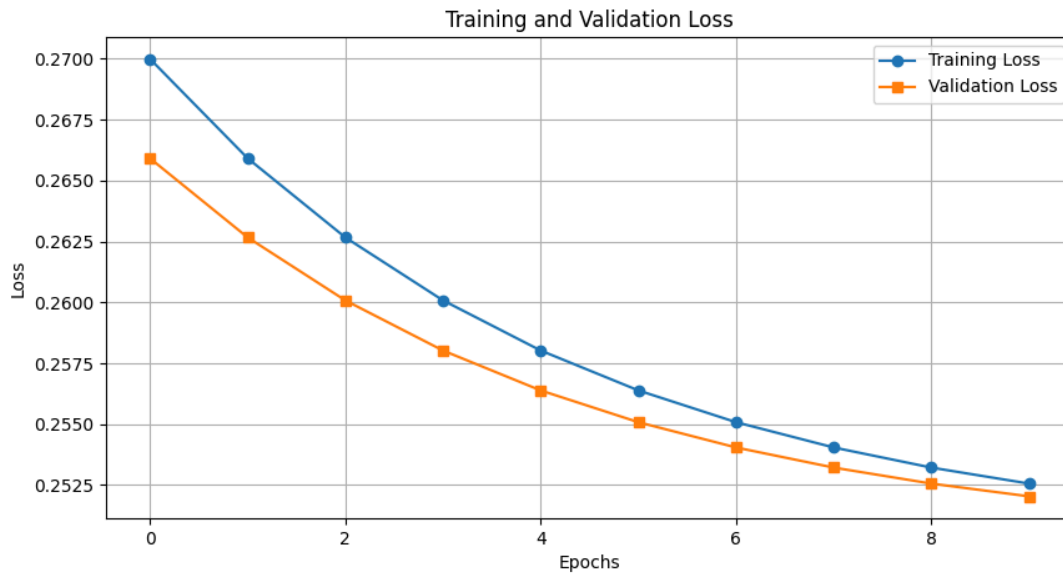
2.2.3. Pengaruh learning rate



Gambar 2.2.3.1 Hasil Learning Rate 0.01



Gambar 2.2.3.2 Hasil Learning Rate 0.05



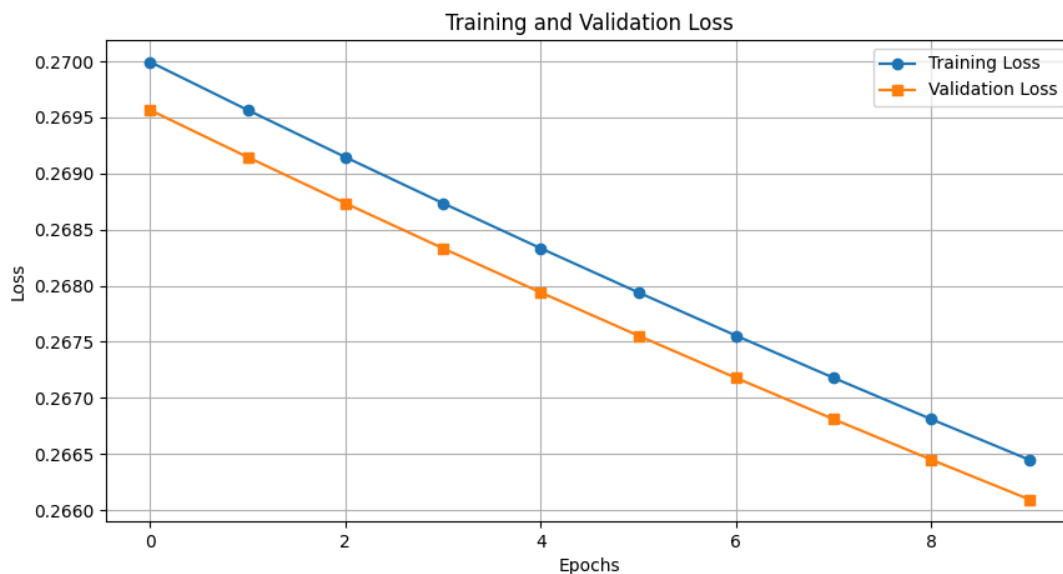
Gambar 2.2.3.3 Hasil Learning Rate 0.1

Dari ketiga plot tersebut, kita bisa membandingkan bagaimana pengaruh learning rate (0.01, 0.05, dan 0.1) terhadap training dan validation loss.

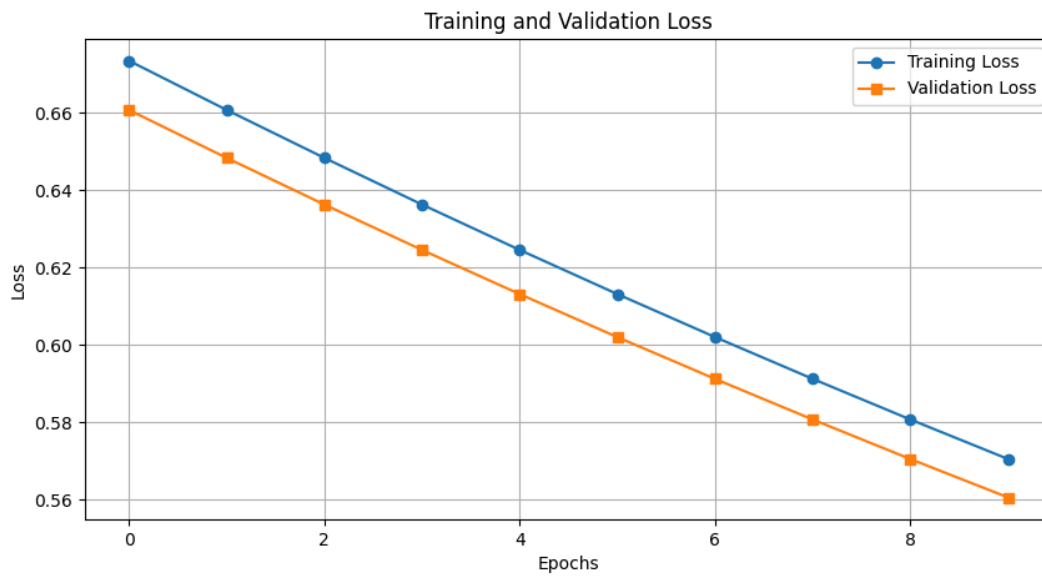
1. Learning Rate 0.01 (training_history_2.png)
 - Training loss dan validation loss turun secara stabil.
 - Kurva menunjukkan konvergensi yang konsisten dan tidak ada tanda-tanda fluktuasi besar.

- Learning rate ini tampaknya cukup aman untuk digunakan karena tidak menyebabkan osilasi.
2. Learning Rate 0.05 (training_history_3.png)
 - Loss berkurang lebih cepat dibandingkan learning rate 0.01.
 - Tidak ada osilasi signifikan, tetapi ada kemungkinan sedikit lebih cepat mencapai titik minimum.
 - Ini bisa menjadi pilihan yang lebih optimal jika ingin mempercepat training tanpa terlalu banyak risiko overshooting.
 3. Learning Rate 0.1 (training_history_4.png)
 - Training dan validation loss turun lebih cepat pada awal training.
 - Namun, grafik menunjukkan indikasi konvergensi yang lebih cepat tetapi mungkin belum optimal.
 - Ada risiko bahwa learning rate yang terlalu tinggi bisa melewati titik optimal sebelum stabilisasi.

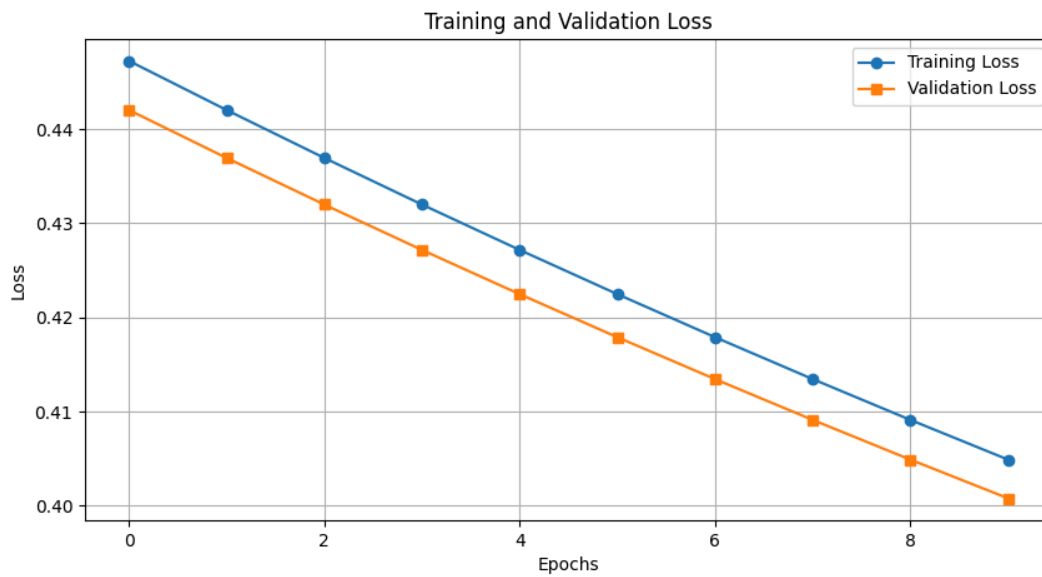
2.2.4. Pengaruh inisialisasi bobot



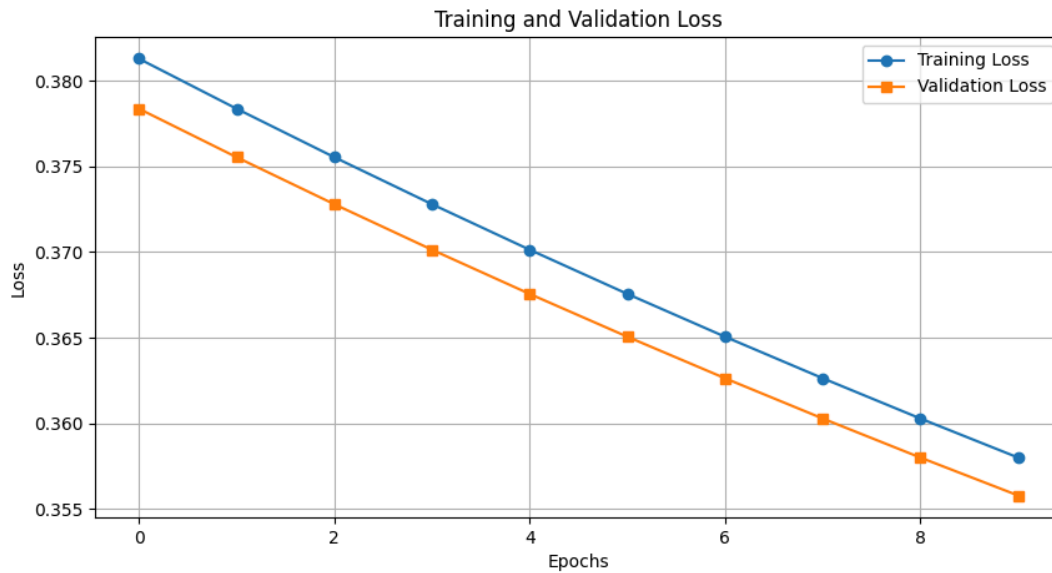
Gambar 2.2.4.1 Hasil inisialisasi bobot Zero



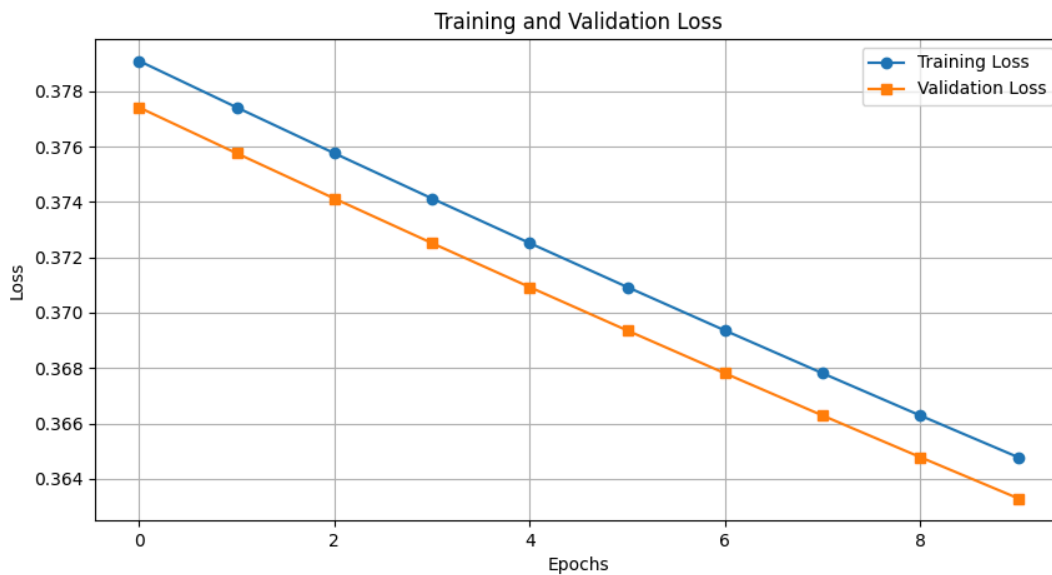
Gambar 2.2.4.2 Hasil inisialisasi bobot Uniform



Gambar 2.2.4.3 Hasil inisialisasi bobot Normal



Gambar 2.2.4.4 Hasil inisialisasi bobot xavier



Gambar 2.2.4.5 Hasil inisialisasi bobot He

Dari kelima plot tersebut, kita dapat melakukan perbandingan:

1. **Bobot Zero:**

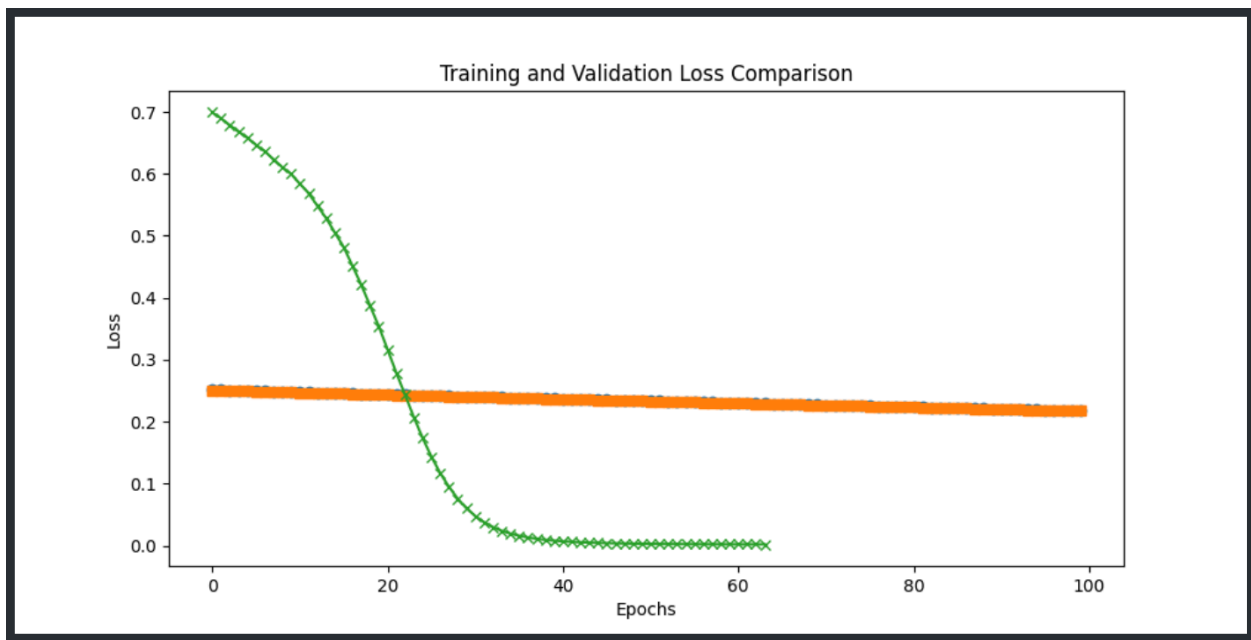
- Loss awal: Sekitar 0.2700
- Penurunan loss terlihat konsisten
- Loss akhir lebih tinggi dibanding metode lainnya
- Mungkin mengalami masalah dengan pembelajaran karena semua bobot sama pada awalnya

2. **Bobot Uniform:**

- Loss awal: Sekitar 0.66
- Penurunan loss lebih cepat dibanding bobot zero

- Hasil akhir lebih rendah dari bobot zero
3. **Bobot Normal:**
 - Loss awal: Sekitar 0.44
 - Penurunan loss tetap stabil
 - Hasil lebih baik dibandingkan uniform dan zero
 4. **Bobot Xavier:**
 - Loss awal: Sekitar 0.38
 - Loss lebih rendah dibanding normal
 - Lebih stabil dalam pelatihan
 5. **Bobot He:**
 - Loss awal: Sekitar 0.378
 - Hasil akhir mirip dengan Xavier, tetapi sedikit lebih rendah
 - Biasanya bagus untuk jaringan dengan aktivasi ReLU

2.2.5. Perbandingan dengan library sklearn



Gambar 2.2.5.1 Hasil perbandingan dengan library sklearn

Dapat dilihat dari gambar tersebut bahwa:

1. Train Loss (Kuning - Model Kami):
 - Hampir datar sepanjang training.
 - Tidak menunjukkan penurunan signifikan.
 - Ini bisa mengindikasikan bahwa model mengalami *underfitting*.

2. Train Loss (Hijau - Sklearn):

- Mulai tinggi (~ 0.7) dan menurun drastis seiring bertambahnya epoch.
- Konvergen mendekati nol setelah sekitar 40 epoch.
- Ini menunjukkan bahwa model dari sklearn belajar dengan baik dan berhasil menurunkan error dengan cepat.

Dari perbandingan tersebut, dapat diambil kesimpulan bahwa:

- Model dari sklearn jelas lebih efektif karena loss menurun drastis dan mencapai nilai yang sangat rendah.
- Model kamui(warna kuning) sepertinya tidak belajar dengan baik karena loss tetap konstan.

3. Kesimpulan dan Saran

Dari hasil yang didapatkan, dapat ditarik kesimpulan bahwa kompleksitas tidak selalu menghasilkan model yang baik. Dalam konteks permasalahan kecil atau khusus, hasil dari model yang lebih simpel dengan jumlah layer dan node yang lebih sedikit dapat menghasilkan performa yang lebih baik dibandingkan dengan model lain yang memiliki jumlah layer maupun node yang tinggi.

Layaknya seperti neuron manusia yang dapat membawa seorang manusia dari berbahasa bayi hingga mengerjakan tugas besar IF3270 machine learning. Model Yang sekhilas hanya stimulus dan response; input ke output yang pendek dapat menghasilkan prediksi yang bagus, selama mutu dari inputnya bisa dijaga.

Kedepannya, sebaiknya pelatihan model FFNN dilakukan dengan parameter kecil, dan di tingkatkan ukurannya berdasarkan keperluan, atau dengan penggabungan/metode seperti ensemble methods.

4. Pembagian Tugas

| NIM | Nama | Tugas |
|----------|------------------------|---|
| 13522130 | Justin Aditya Putra P. | <ul style="list-style-type: none">- Data structure- GUI |
| 13522155 | Axel Santadi Warih | <ul style="list-style-type: none">- Help Frontend- Help backend- Learning- Initiate weight |
| 13522163 | Atqiya Haydar Luqman | <ul style="list-style-type: none">- Mengerjakan semua kebutuhan fungsi yang ada di spesifikasi |

5. Referensi

- [Stack overflow](#)
- <https://www.youtube.com/watch?v=VMj-3S1tku0>
- [Vue Documentation](#)
- Slide Perkuliahan IF3270 ANN:FFNN