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

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
MSE	$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

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

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

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

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

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

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

$$\alpha = \text{Learning rate}$$

- 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} (μ={mean:.3f}, σ={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} (μ={mean:.3f}, σ={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 layerdalam 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.

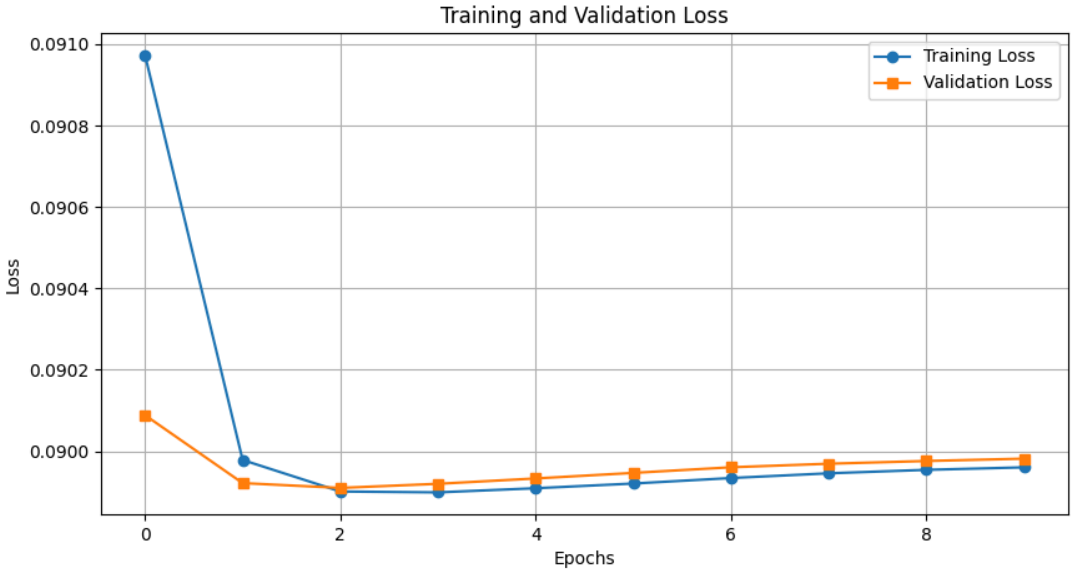
Langkah terakhir dari backward propagation adalah memanggil fungsi update_weights, yang bertanggung jawab untuk memperbarui bobot dan bias menggunakan learning rate yang telah ditentukan. Metode ini juga mengembalikan nilai loss, yang dihitung sebagai rata-rata kuadrat dari error, untuk memantau sejauh 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 2 hidden layer



Gambar 2.2.1.1.2 Hasil depth 3 hidden layer

Gambar 2.2.1.1.3 Hasil depth 4 hidden layer

Dari ketiga plot tersebut, kita dapat melakukan perbandingan:

- Semakin banyak hidden layer yang ada, loss semakin tinggi namun lebih cepat stabil.

2.2.1.2. Width

Gambar 2.2.1.2.1 Hasil plot 8 buah node

Gambar 2.2.1.2.2 Hasil plot 16 buah node

Gambar 2.2.1.2.3 Hasil plot 32 buah node

Dari ketiga plot tersebut, kita dapat melakukan perbandingan:

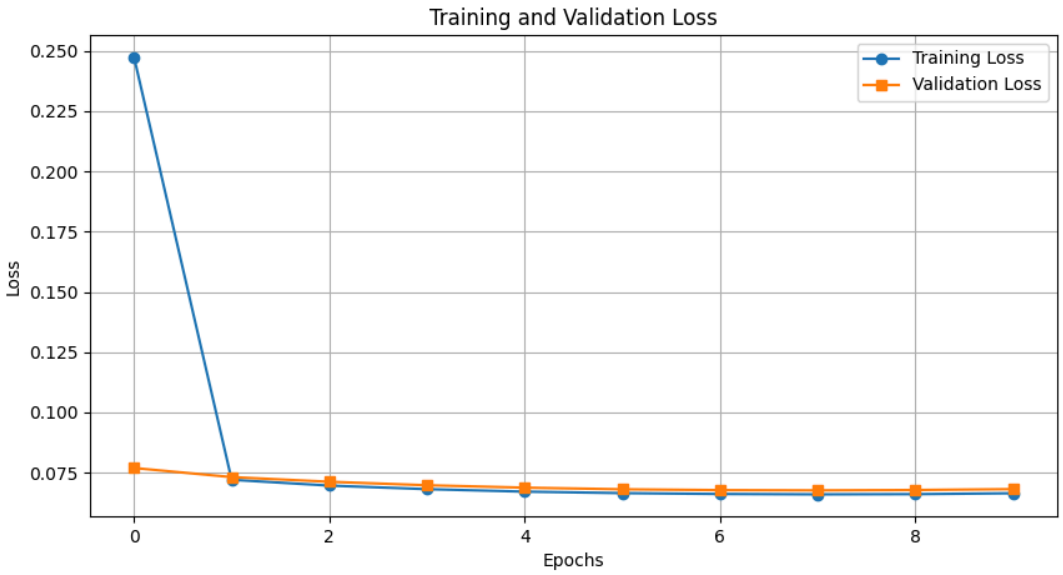
1. **Training History 1 (2 node, 1 hidden layer)**
   - Training Loss: Cepat turun hingga ~epoch 3, lalu datar dan sedikit naik di akhir.
   - Validation Loss: Mengikuti training loss dengan ketat, sedikit lebih tinggi di akhir.
   - Hal ini menunjukan bahwa model hampir tidak overfitting dan sangat stabil
2. **Training History 2 (3 node, 1 hidden layer)**
   - Training Loss: Lebih rendah dari grafik 1, tapi mulai naik setelah epoch 4.
   - Validation Loss: Jelas naik setelah titik minimum di epoch 3–4.
   - Hal ini menunjukan bahwa model mulai overfitting setelah titik minimum loss

3. **Training History 3 (4 node, 1 hidden layer)**
    - Training Loss: Terendah dari ketiganya — belajar paling dalam.
    - Validation Loss: Awalnya mengikuti dengan baik, lalu sedikit overfit di akhir (tapi lebih halus dibanding grafik 2).
    - Bisa ditarik fakta bahwa performanya baik di awal dan pertengahan epoch.

Dari fakta tersebut, bisa ditarik kesimpulan:

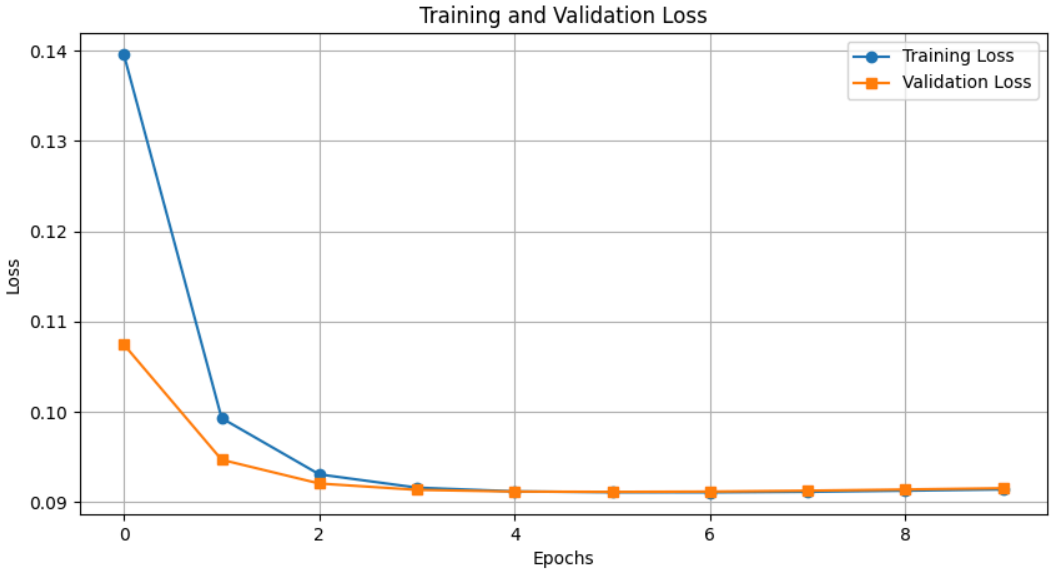
- Model 1 layer menunjukkan generalization yang sangat baik, tapi tampak kurang kuat dalam menangkap pola kompleks (sedikit underfit).
- Model 2 layer cepat belajar tapi overfit paling parah, menunjukkan bahwa jumlah dan/atau lebar neuron terlalu besar tanpa pengamanan.
- Model 3 layer mencapai training loss terendah dan validation loss yang relatif stabil, menjadi kandidat terbaik, asalkan overfitting dikendalikan.

2.2.2. Pengaruh fungsi aktivasi

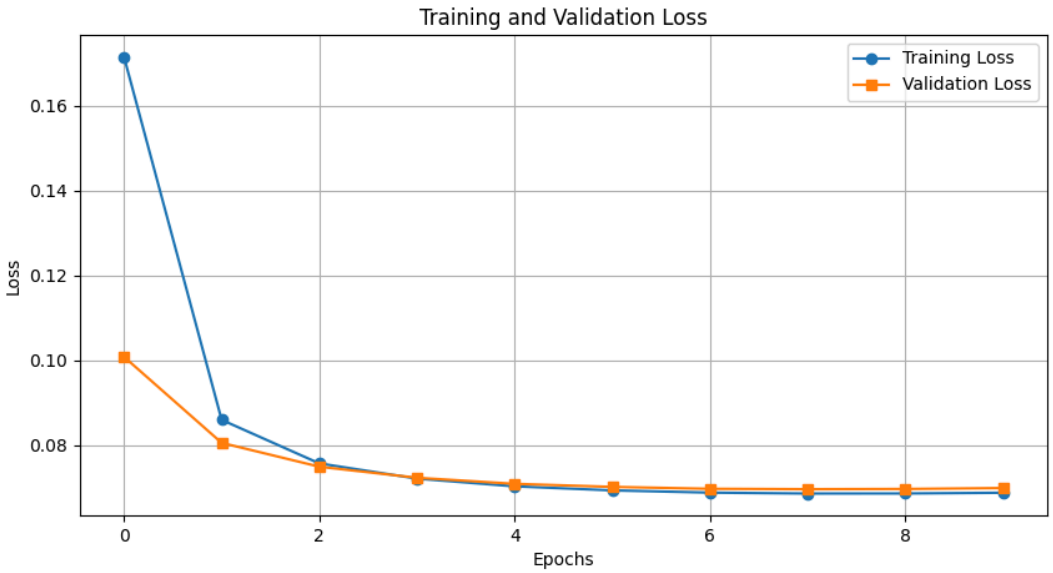


Gambar 2.2.2.1 Hasil aktivasi linear

Gambar 2.2.2.2 Hasil aktivasi ReLU



Gambar 2.2.2.3 Hasil aktivasi Sigmoid



Gambar 2.2.2.4 Hasil aktivasi Tanh

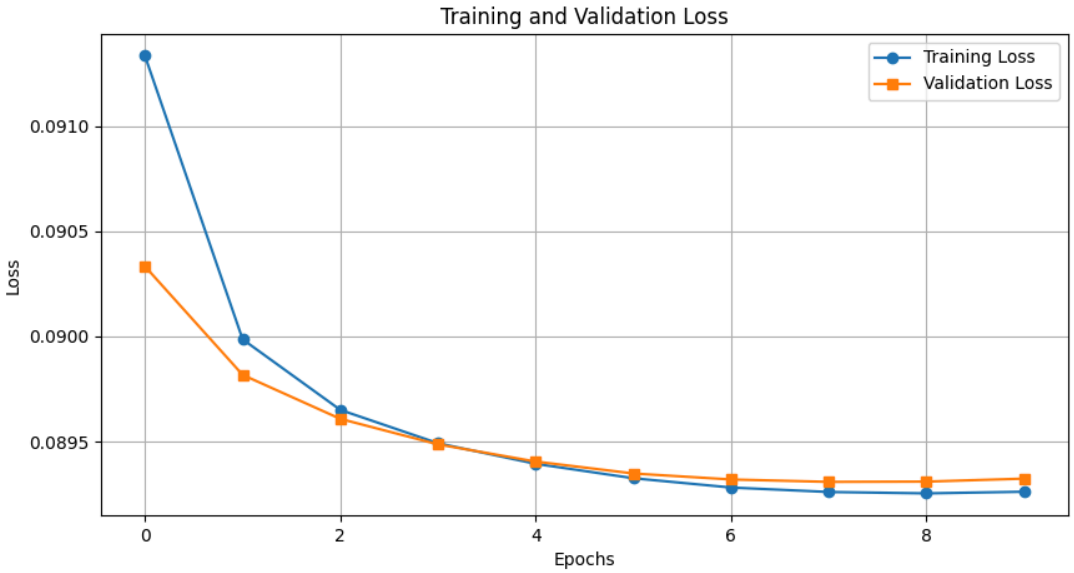
Dari keempat plot tersebut, kita dapat melakukan perbandingan:

1. **Linear**
   - Training Loss: Sangat tinggi di awal (~0.25), lalu turun drastis dan stagnan di sekitar 0.068.
   - Validation Loss: Relatif stabil sejak awal (~0.075) dan turun sedikit, lalu stagnan
2. **ReLU**
   - Training & Validation Loss: Sama-sama turun konsisten dan hampir sejajar, berhenti di sekitar 0.097–0.098.
3. **Sigmoid**
   - Training Loss: Turun drastis, lalu stagnan di sekitar 0.091.
   - Validation Loss: Hampir identik dengan training loss.
4. **Tanh**
   - Training & Validation Loss: Menurun dengan sangat cepat lalu stagnan di sekitar 0.07.

Dari hasil perbandingan tersebut, kita bisa mengambil kesimpulan:

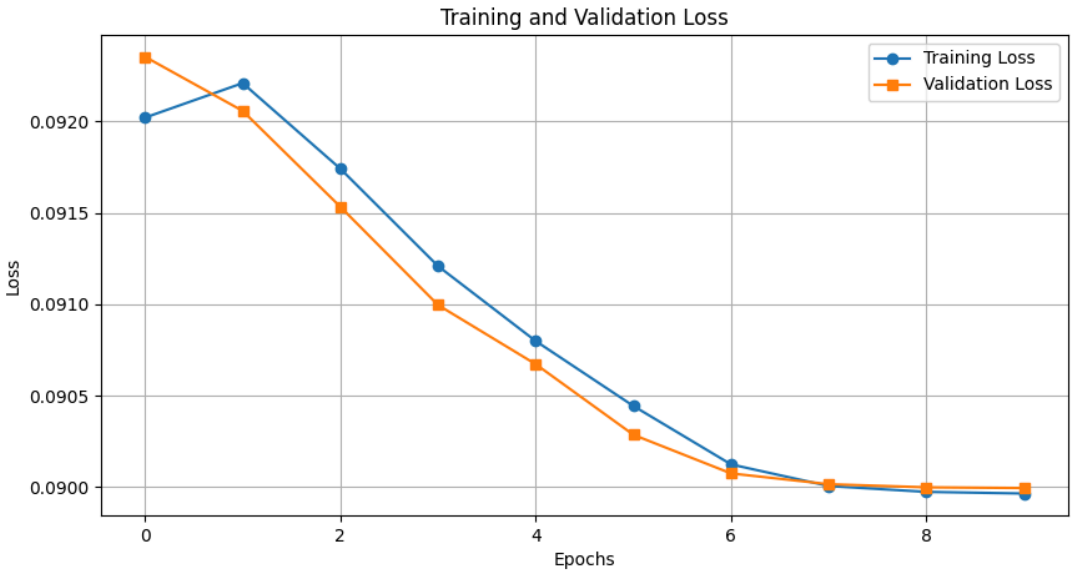
- ReLU secara konsisten memberikan hasil terbaik dalam hal:
  - Konvergensi cepat
  - Loss rendah
  - Tidak overfitting
- Linear activation sangat terbatas cocoknya untuk model sangat sederhana atau output layer dalam regresi.
- Sigmoid dan tanh stabil tapi bisa membatasi pembelajaran model dalam jaringan yang dalam, terutama tanpa teknik tambahan seperti batch normalization atau weight initialization yang tepat.

2.2.3. Pengaruh learning rate



Gambar 2.2.3.1 Hasil Learning Rate 0.005

Gambar 2.2.3.2 Hasil Learning Rate 0.01

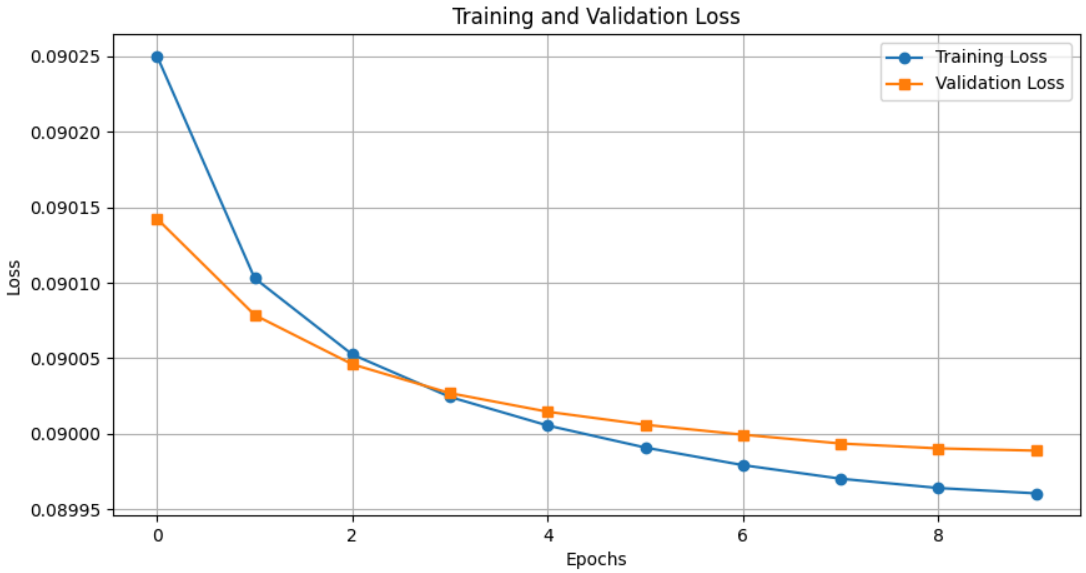


Gambar 2.2.3.3 Hasil Learning Rate 0.1

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 0.005
   - Training & Validation Loss: Konsisten menurun dan stabil mulai dari epoch ke-5, mendekati konvergen.
2. Learning Rate 0.01
   - Training Loss: Menurun hingga titik minimum (~epoch 3), lalu justru naik.
   - Validation Loss: Turun sebentar lalu naik terus bisa jadi dikarenakan overfitting dan kemungkinan bouncing.
3. Learning Rate 0.1
   - Training & Validation Loss: Menurun dengan konsisten, bahkan lebih halus dari LR 0.01.

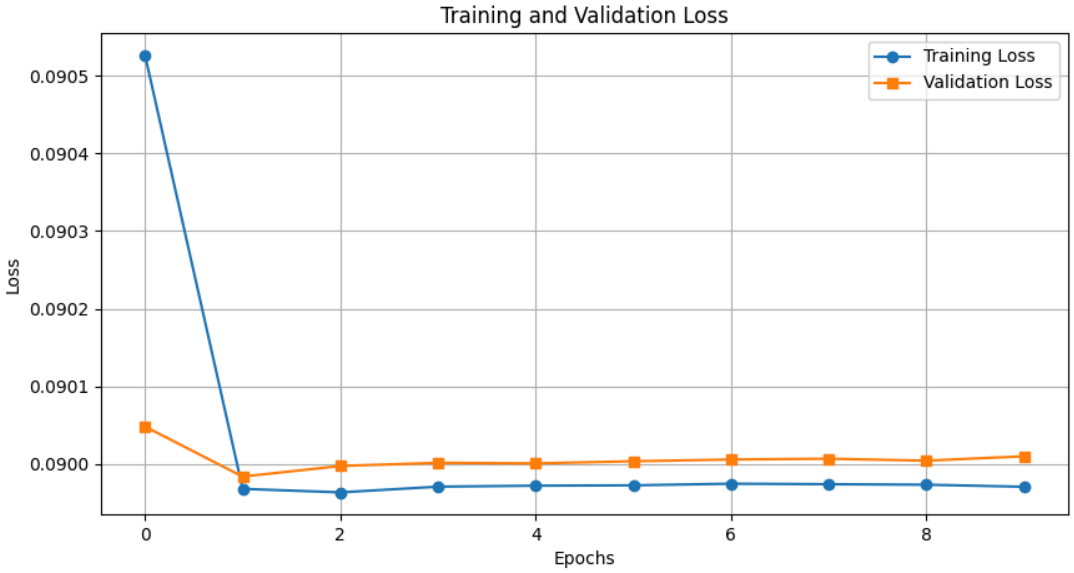
2.2.4. Pengaruh inisialisasi bobot



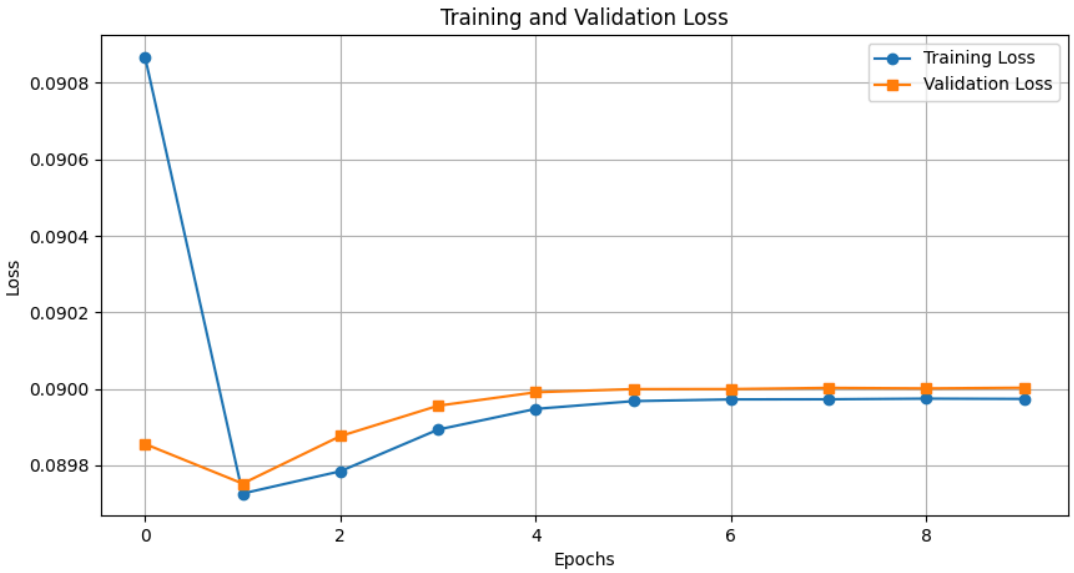
Gambar 2.2.4.1 Hasil inisialisasi bobot Zero

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

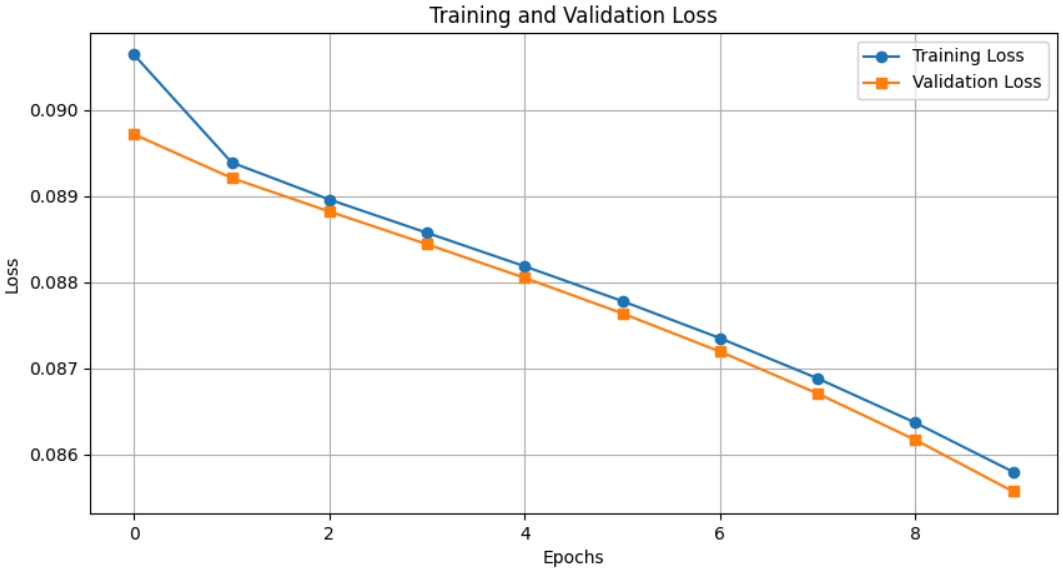
Dari kelima plot tersebut, kita dapat melakukan perbandingan:

1. **Bobot Zero**:
   - Training Loss: Menurun lambat dan stabil.
   - Validation Loss: Turun seiring training, sangat dekat dengan training loss.
2. **Bobot Uniform**:
   - Training Loss: Turun awal, lalu naik bisa jadi divergence.
   - Validation Loss: Ikut naik drastis setelah awal.
3. **Bobot Normal**:
   - Training & Validation Loss: Turun konsisten, lalu naik bisa jadi dikarenakan overfitting ringan.
4. **Bobot Xavier**:
   - Training & Validation Loss: Turun konsisten, lalu naik (indikasi overfitting ringan).
5. **Bobot He**:
   - Training & Validation Loss: Fluktuasi awal, tapi konvergen mendekati akhir.

2.2.5. Perbandingan dengan Regularisasi

Gambar 2.2.5.1 Hasil tanpa regularisasi

Gambar 2.2.5.2 Hasil L1

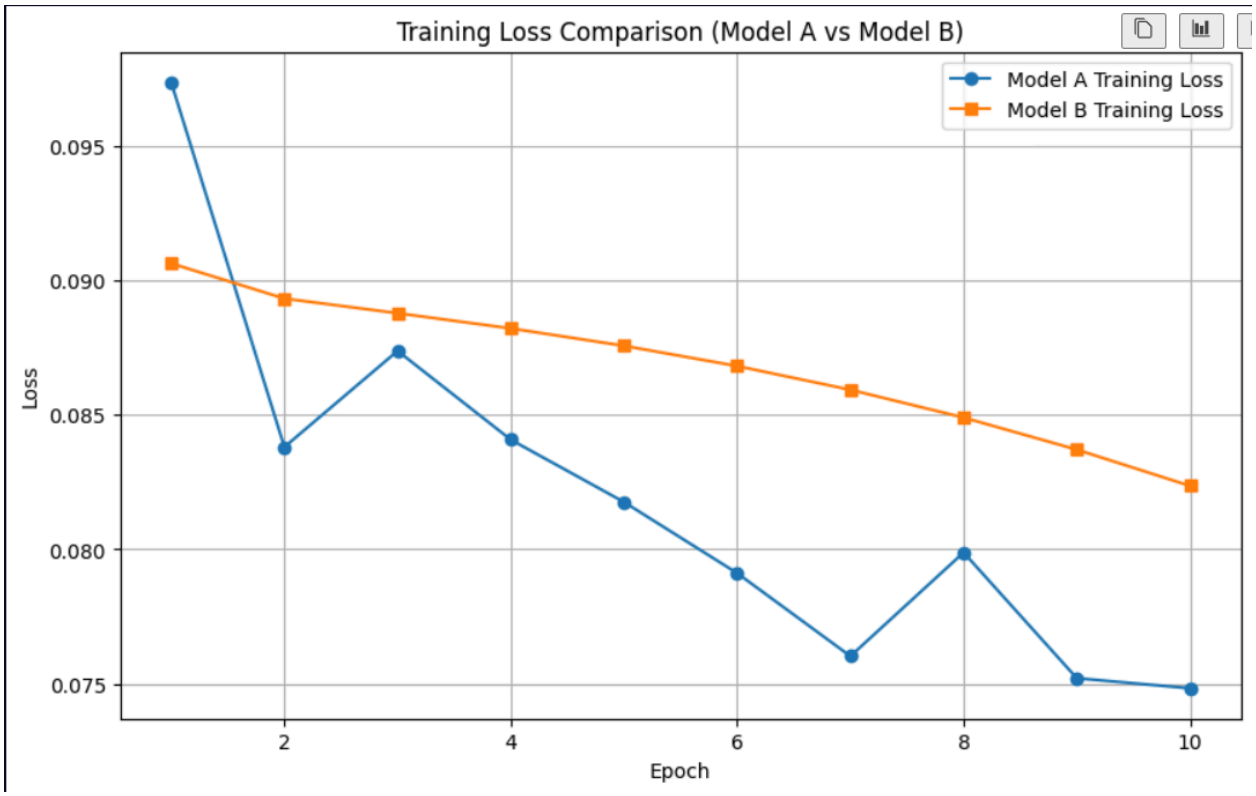


Gambar 2.2.5.3 Hasil L2

Dari ketiga plot tersebut, kita bisa membandingkan bagaimana pengaruh regularisasi terhadap training dan validation loss.

1. Non Regularisasi
   - Training Loss: Turun secara konsisten.
   - Validation Loss: Juga turun secara konsisten dan sedikit lebih rendah dari training loss di akhir.
2. L1
   - Training Loss: Turun lalu naik kembali setelah titik minimum.
   - Validation Loss: Ikut naik seiring training loss bisa jadi indikasi underfitting.
3. L2
   - Training & Validation Loss: Turun konsisten, sangat mirip dan stabil.

2.2.6. 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):
   - Loss menurun cepat di awal, bahkan lebih tajam dari model kita.
   - Namun, terjadi fluktuasi besar (naik-turun) antar epoch.
   - Pada epoch 10, loss berada di titik terendah (~0.075), berarti SKLearn mampu menemukan solusi yang lebih optimal secara angka.
2. Train Loss (Hijau - Sklearn):
   - Loss menurun konsisten dan stabil.
   - Tidak terjadi lonjakan atau penurunan tajam.
   - Pada epoch 10, loss sekitar ~0.082, sedikit lebih tinggi dari SKLearn tapi lebih halus.

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 respone; 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.	- Data structure - Fungsi yang ada di spek untuk kelas FFNN - GUI (Pensiun ToT) - Minor fixing - SKLearn check
13522155	Axel Santadi Warih	- Help Frontend - Help backend - Learning - Initiate weight
13522163	Atqiya Haydar Luqman	- 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