Laporan Tugas Besar 1

IF3270 Pembelajaran Mesin

Kelompok 39:

Dzaky Satrio Nugroho	13522059
Rafiki Prawhira Harianto	13522065
Julian Caleb Simandjuntak	13522099

Sekolah Teknik Elektro dan Informatika - Komputasi

Institut Teknologi Bandung

Deskripsi Persoalan

Tugas Besar I pada kuliah IF3270 Pembelajaran Mesin agar peserta kuliah mendapatkan wawasan tentang bagaimana cara mengimplementasikan Feedforward Neural Network (FFNN). Pada tugas ini, peserta kuliah akan ditugaskan untuk mengimplementasikan FFNN from *scratch*.

Pembahasan

Penjelasan implementasi

Deskripsi kelas beserta deskripsi atribut dan methodnya

1. Kelas ActivationFunction

```
class ActivationFunction:
    def __init__(self, activation_type):
        self.activation_type = activation_type

    def forward(self, x: np.ndarray) -> np.ndarray:
        if self.activation_type == 'linear':
            return ActivationFunction.__linear(x)
        elif self.activation_type == 'relu':
            return ActivationFunction.__relu(x)
        elif self.activation_type == 'sigmoid':
            return ActivationFunction.__sigmoid(x)
        elif self.activation_type == 'tanh':
            return ActivationFunction.__tanh(x)
        elif self.activation_type == 'softmax':
            return ActivationFunction.__softmax(x)
        elif self.activation_type == 'leaky_relu':
            return ActivationFunction.__leaky_relu(x)
        elif self.activation_type == 'swish':
            return ActivationFunction.__swish(x)
        else:
            raise ValueError(f"Activation function '{self.activation_type}' not
supported")

    def backward(self, x: np.ndarray) -> np.ndarray:
        if self.activation_type == 'linear':
            return ActivationFunction.__linear_derivative(x)
        elif self.activation_type == 'relu':
            return ActivationFunction.__relu_derivative(x)
        elif self.activation_type == 'sigmoid':
            return ActivationFunction.__sigmoid_derivative(x)
        elif self.activation_type == 'tanh':
            return ActivationFunction.__tanh_derivative(x)
```

```
        elif self.activation_type == 'softmax':
            return ActivationFunction.__softmax_derivative(x)
        elif self.activation_type == 'leaky_relu':
            return ActivationFunction.__leaky_relu_derivative(x)
        elif self.activation_type == 'swish':
            return ActivationFunction.__swish_derivative(x)
        else:
            raise ValueError(f"Activation function '{self.activation_type}' not
supported")

    def __linear(x: np.ndarray) -> np.ndarray:
        return x

    def __relu(x: np.ndarray) -> np.ndarray:
        return np.maximum(0, x)

    def __sigmoid(x: np.ndarray) -> np.ndarray:
        return 1 / (1 + np.exp(-x))

    def __tanh(x: np.ndarray) -> np.ndarray:
        return np.tanh(x)

    def __softmax(x: np.ndarray) -> np.ndarray:
        e_x = np.exp(x - np.max(x))
        return e_x / e_x.sum(axis=0)

    def __leaky_relu(x: np.ndarray, alpha=0.1) -> np.ndarray:
        return np.maximum(alpha*x, x)

    def __swish(x: np.ndarray) -> np.ndarray:
        return x * ActivationFunction.__sigmoid(x)

    def __linear_derivative(x: np.ndarray) -> np.ndarray:
        return np.ones_like(x)

    def __relu_derivative(x: np.ndarray) -> np.ndarray:
        return np.where(x > 0, 1, 0)
```

```python
    def __sigmoid_derivative(x: np.ndarray) -> np.ndarray:
        return x * (1 - x)

    def __tanh_derivative(x: np.ndarray) -> np.ndarray:
        return 1 - ActivationFunction.__tanh(x) ** 2

    def __softmax_derivative(x: np.ndarray) -> np.ndarray:
        softmax_x = ActivationFunction.__softmax(x)
        return softmax_x * (1 - softmax_x)

    def __leaky_relu_derivative(x: np.ndarray, alpha=0.1) -> np.ndarray:
        return np.where(x > 0, 1, alpha)

    def __swish_derivative(x: np.ndarray) -> np.ndarray:
        sigmoidx = ActivationFunction.__sigmoid(x)
        return sigmoidx * (1 + x * (1 - sigmoidx))
```

Kelas ini berisikan fungsi aktivasi yang dapat digunakan oleh model FFNN. Fungsi-fungsi yang ada diantaranya : Linear, ReLU, Sigmoid, Hyperbolic Tangent (tanh), softmax, Leaky ReLU, dan Swish. Selain fungsi aktivasi, di kelas ini juga terdapat turunan dari fungsi aktivasi yang ada.

2. Kelas LossFunction

```python
class LossFunction:
    def __init__(self, loss_type: str):
        self.loss_type = loss_type

    def loss(self, y_pred: np.ndarray, y_true: np.ndarray) -> float:
        if self.loss_type == 'mse':
            return LossFunction.__mse(y_pred, y_true)
        elif self.loss_type == 'bce':
            return LossFunction.__bce(y_pred, y_true)
        elif self.loss_type == 'cce':
            return LossFunction.__cce(y_pred, y_true)
        else:
            raise ValueError(f"Unknown loss function {self.loss_type}")
```

```python
    def loss_derivative(self, y_pred: np.ndarray, y_true: np.ndarray) ->
np.ndarray:
        if self.loss_type == 'mse':
            return LossFunction.__mse_derivative(y_pred, y_true)
        elif self.loss_type == 'bce':
            return LossFunction.__bce_derivative(y_pred, y_true)
        elif self.loss_type == 'cce':
            return LossFunction.__cce_derivative(y_pred, y_true)
        else:
            raise ValueError(f"Unknown loss function {self.loss_type}")

    # Mean Squared Error
    def __mse(y_pred: np.ndarray, y_true: np.ndarray) -> float:
        mse = np.sum((y_true - y_pred) ** 2) / len(y_true)
        return mse

    # Binary Cross-Entropy
    def __bce(y_pred: np.ndarray, y_true: np.ndarray) -> float:
        bce = -(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 -
y_pred)).mean()
        return bce

    # Categorical Cross-Entropy
    def __cce(y_pred: np.ndarray, y_true: np.ndarray) -> float:
        cce = -1 / len(y_true) * np.sum(np.sum(y_true * np.log(y_pred)))
        return cce

    def __mse_derivative(y_pred: np.ndarray, y_true: np.ndarray) -> np.ndarray:
        return -2 * (y_true - y_pred) / len(y_true) # times dy_pred/dw

    def __bce_derivative(y_pred: np.ndarray, y_true: np.ndarray) -> np.ndarray:
        return -1 * (y_pred - y_true) / (y_pred * (1 - y_pred) * len(y_true)) #
times dy_pred/dw

    def __cce_derivative(y_pred: np.ndarray, y_true: np.ndarray) -> np.ndarray:
        return -1 * (y_true / (y_pred * len(y_true))) # times dy_pred/dw
```

Kelas ini berisikan fungsi loss yang dapat digunakan oleh model FFNN. Fungsi-fungsi yang ada diantaranya : Mean Squared Error (MSE), Binary Cross-Entropy (BCE), dan Categorical Cross-Entropy (CCE). Selain fungsi aktivasi, di kelas ini juga terdapat turunan dari fungsi aktivasi yang ada.

3. Kelas WeightInitializer

```
class WeightInitializer:
    @staticmethod
    def zeros(shape):
        w = np.zeros(shape)
        b = np.zeros((1, shape[1]))
        return np.vstack((b, w))

    @staticmethod
    def uniform(shape, lower_bound=-0.1, upper_bound=0.1, seed=None):
        if seed is not None:
            np.random.seed(seed)
        w = np.random.uniform(lower_bound, upper_bound, shape)
        b = np.random.uniform(lower_bound, upper_bound, (1, shape[1]))
        return np.vstack((b, w))

    @staticmethod
    def normal(shape, mean=0.0, variance=1.0, seed=None):
        if seed is not None:
            np.random.seed(seed)
        std_dev = np.sqrt(variance)
        w = np.random.normal(mean, std_dev, shape)
        b = np.random.normal(mean, std_dev, (1, shape[1]))
        return np.vstack((b, w))

    @staticmethod
    def xavier_uniform(shape, seed=None):
        fan_in, fan_out = shape[0], shape[1]
        variance = 2.0 / (fan_in + fan_out)
        limit = np.sqrt(3.0 * variance)
        return WeightInitializer.uniform(shape, lower_bound=-limit,
```

```
upper_bound=limit, seed=seed)

    @staticmethod
    def xavier_normal(shape, seed=None):
        fan_in, fan_out = shape[0], shape[1]
        variance = 2.0 / (fan_in + fan_out)
        return WeightInitializer.normal(shape, mean=0.0, variance=variance,
seed=seed)

    @staticmethod
    def he_uniform(shape, seed=None):
        fan_in = shape[0]
        variance = 2.0 / fan_in
        limit = np.sqrt(3.0 * variance)
        return WeightInitializer.uniform(shape, lower_bound=-limit,
upper_bound=limit, seed=seed)

    @staticmethod
    def he_normal(shape, seed=None):
        fan_in = shape[0]
        variance = 2.0 / fan_in
        return WeightInitializer.normal(shape, mean=0.0, variance=variance,
seed=seed)
```

Kelas ini berisikan method untuk melakukan inisiasi bobot awal. Terdapat 7 method : Zeros, Uniform (Distribusi Uniform), dan Normal (Distribusi Normal), Xavier uniform, Xavier normal, He uniform, dan He normal.

4. Kelas FFNN

```
class FFNN:
    def __init__(self, layers, activations=None, loss="mse", initialization="uniform",
seed=39, batch_size=1, learning_rate=0.01, epochs=10, verbose=1, weights=None,
w_lower_bound=-0.1, w_upper_bound=0.1, w_mean=1.0, w_variance=0.0):
        self.layers = layers
        if activations:
            self.activations = [ActivationFunction(activation) for activation in
activations]
```

```
        else:
            self.activations = [ActivationFunction("sigmoid") for _ in
range(len(layers) - 1)]
        self.loss = LossFunction(loss_type=loss)
        self.initialization = initialization
        self.seed = seed
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.verbose = verbose
        self.value_matrix = []
        self.train_losses = []
        self.val_losses = []

        if self.initialization == 'custom':
            self.weights = weights
        else:
            self.weights = []
        self.gradients_w = []

        for i in range(1, len(self.layers)):
            in_size, out_size = self.layers[i - 1], self.layers[i]
            if self.initialization == 'zeros':
                w = WeightInitializer.zeros((in_size, out_size))
            elif self.initialization == 'uniform':
                w = WeightInitializer.uniform((in_size, out_size), seed=self.seed,
lower_bound=w_lower_bound, upper_bound=w_upper_bound)
            elif self.initialization == 'normal':
                w = WeightInitializer.normal((in_size, out_size), seed=self.seed,
mean=w_mean, variance=w_variance)
            elif self.initialization == 'custom':
                continue
            else:
                raise ValueError("Metode inisialisasi tidak valid.")

            self.weights.append(w)

    def forward_propagation(self, input_data)

    def backward_propagation(self, target_output)

    def update_weights(self)
```

```
    def train(self, X, y, val_split=0.2)

    def plot_loss(self)

    def predict(self, X)

    def visualize_weights(self, start=1, end=None, display_size=5,
gradient_graph=False)

    def visualize_gradient_weights(self, start=1, end=None, display_size=5)

    def save(self, file_path)

    @staticmethod
    def load(file_path)
```

Kelas FFNN memiliki beberapa atribut yaitu :

1. layers : berisi array of integer yang menyimpan banyak neuron untuk tiap layer
2. activations : berisi array of string yang menyimpan nama fungsi aktivasi untuk tiap layer
3. loss : berisi fungsi loss yang digunakan
4. initialization : berisi metode inisialisasi bobot awal
5. seed : berisi seed bobot jika initialization bukan zeros
6. batch_size : banyak data yang diproses dalam 1 iterasi
7. learning_rate : berisi learning rate
8. epochs : berisi jumlah epoch/iterasi
9. verbose : jika bernilai 1 maka menampilkan progress bar beserta kondisi training loss dan validation loss saat itu, jika 0 maka tidak menampilkan apa apa
10. value_matrix : array of array of int yang berisikan nilai dari tiap neuron
11. train_losses : array of array of float berisi nilai loss training set setiap epoch

12. val_losses : array of array of float berisi nilai loss validation set setiap epoch
13. weights : array of array of array of float yang menyimpan nilai semua bobot
14. gradients_w : array of array of array of float yang menyimpan nilai semua gradient weight

Kelas FFNN memiliki beberapa method, yaitu :

1. init : untuk menginisiasi semua atribut FFNN berdasarkan parameter yang diterima
2. forward_propagation : dijelaskan pada bagian Penjelasan forward propagation
3. backward_propagation : dijelaskan pada bagian Penjelasan backward propagation dan weight update
4. update_weights : dijelaskan pada bagian Penjelasan backward propagation dan weight update
5. train : fungsi utama untuk melatih model. Dipanggil forwardpropagataion dan backwardpropagation untuk 1 batch, lalu update_weight setiap epoch. Terdapat pula mekanisme untuk menampilkan progress bar jika verbose=1
6. plot_loss : Menampilkan grafik training loss dan validation loss terhadap epoch
7. predict : Menampilkan prediksi y untuk satu set X
8. visualize_weights : method untuk menghasilkan graf dan memvisualisasikan bobot dari keseluruhan model FFNN atau layer tertentu saja
9. visualize_gradient_weights : method untuk menghasilkan graf dan memvisualisasikan gradien bobot dari keseluruhan model FFNN atau layer tertentu saja

10. save : Menyimpan hasil training model ke suatu file menggunakan pickle
11. load : Load hasil save model dari suatu file menggunakan pickle

5. Kelas RNSNorm

```
class RMSNorm:
    def __init__(self, epsilon=1e-8):
        self.epsilon = epsilon

    def normalize(self, x):
        rms = np.sqrt(np.mean(x**2) + self.epsilon)
        return x / rms
```

Kelas RNSNorm melakukan normalisasi pada data input dengan membagi setiap elemen dengan akar dari nilai rata-rata kuadratnya (Root Mean Square) ditambah epsilon untuk stabilitas numerik.

Penjelasan forward propagation

```
    def forward_propagation(self, input_data):
        values = np.array(input_data)
        self.value_matrix = [values]

        for i in range(len(self.weights)):
            values = np.insert(values, 0, 1)
            z = np.dot(self.weights[i].T , values)
            new_values = self.activations[i].forward(z)
            values = new_values
            self.value_matrix.append(values)

        return self.value_matrix[-1]
```

Forward propagation berupa proses FFNN menghitung output berdasarkan input yang diberikan. Proses ini dilakukan dengan cara meneruskan “value” melalui setiap lapisan dalam jaringan, mulai dari input hingga output, dengan menggunakan bobot (weights) dan fungsi aktivasi. Net atau “z” dihitung menggunakan rumus w.T * x, yang diteruskan menggunakan activation function yang dipilih di fungsi self.activations[i].forward(z). Hasilnya disimpan dalam self.value_matrix, yang akan digunakan dalam backpropagation dan visualisasi. Return value disini hanya untuk menampilkan pada predict()

Penjelasan backward propagation dan weight update

```
    def backward_propagation(self, target_output):
        errors = [self.loss.loss_derivative(self.value_matrix[-1], target_output)]
        self.gradients_w = []

        for i in reversed(range(len(self.weights))):
            delta = errors[-1] * self.activations[i].backward(self.value_matrix[i + 1])
            prev_values = np.insert(self.value_matrix[i], 0, 1)  # Tambahkan bias
            grad = np.outer(prev_values, delta)
            self.gradients_w.insert(0, grad)
            errors.append(np.dot(self.weights[i], delta)[1:])
```

```
    def update_weights(self):
        for i in range(len(self.weights)):
            self.weights[i] -= self.learning_rate * self.gradients_w[i] /
self.batch_size
```

Backward propagation atau backpropagation adalah proses FFNN untuk menghitung gradien error dan memperbarui bobot berdasarkan error tersebut. Ini adalah tahap utama dalam pelatihan FFNN dengan gradient descent, dengan tujuan mengurangi loss setiap iterasinya. Errors dihitung berupa array loss_derivative dari setiap layer, dimulai dari layer terakhir. Selalu

dipastikan pada setiap iterasi backpropagation, gradien weight direset dengan array kosong.

Loop utama memanggil error layer terakhir dikali dengan derivative pada layer berikutnya (dari layer output ke input). Dipanggil self.activations[i].backward, yaitu activation derivative dari setiap layer yang dipilih. Np.outer dipakai untuk mengkalikan delta dengan dnet/dW atau Xij, menghasilkan dE/dw. self.gradients_w menyetor semua gradien dalam shape yang sama dengan self.weights, dan dipakai dalam visualisasi dan update_weights. update_weights akan mengurangi weights dengan gradien weight yang sesuai yang dikalikan dengan learning rate.

Hasil pengujian

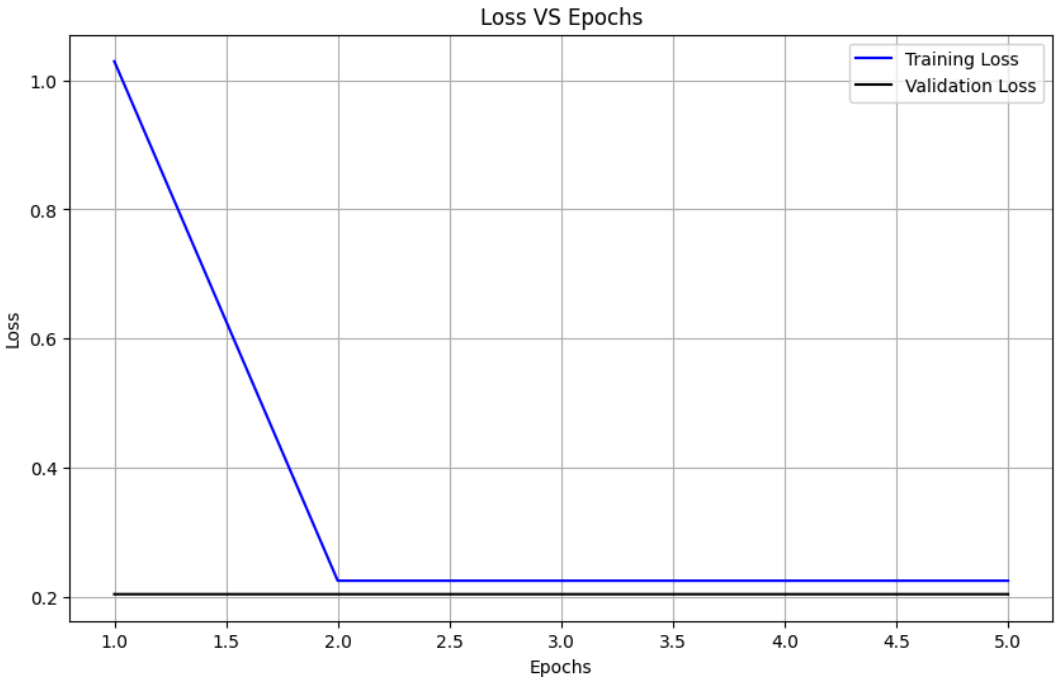
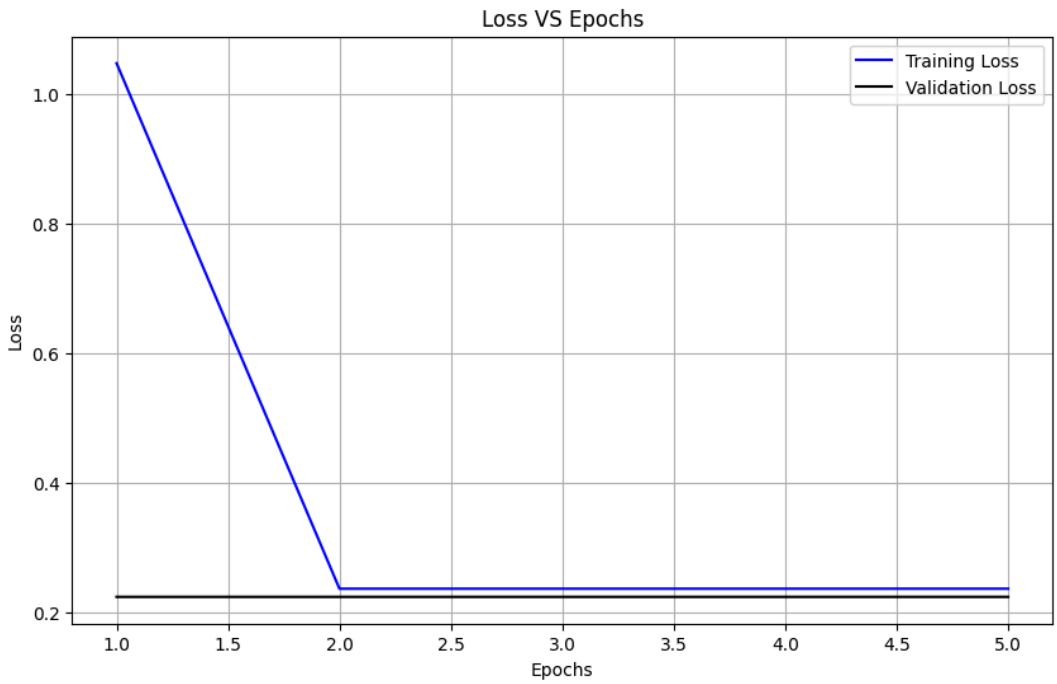
Pengaruh depth dan width

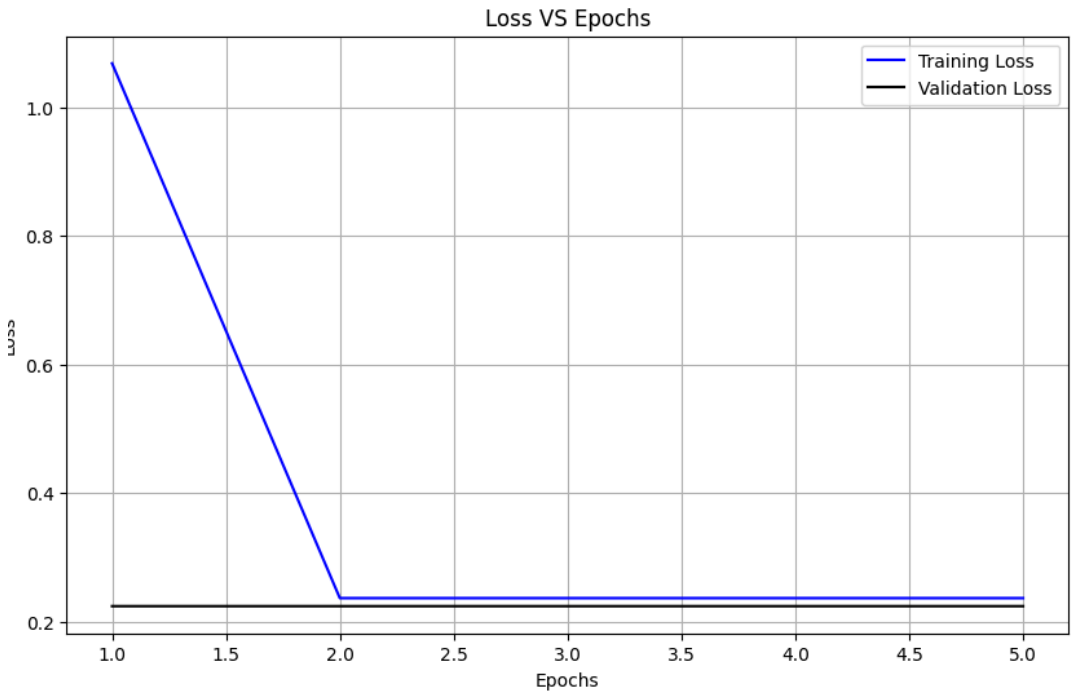
Dilakukan pengujian pengaruh depth dan width yang bervariasi terhadap hasil training dan prediksi model. Digunakan fungsi aktivasi sigmoid untuk semua proses aktivasi, fungsi inisialisasi bobot normal, dan fungsi loss MSE.

1. Pengaruh Depth

Tiap layer memiliki 2 Neuron

1 Layer	Akurasi: 0.12
2 Layer	Akurasi: 0.12
5 Layer	Akurasi: 0.12

1 Layer	
2 Layer	

5 Layer	

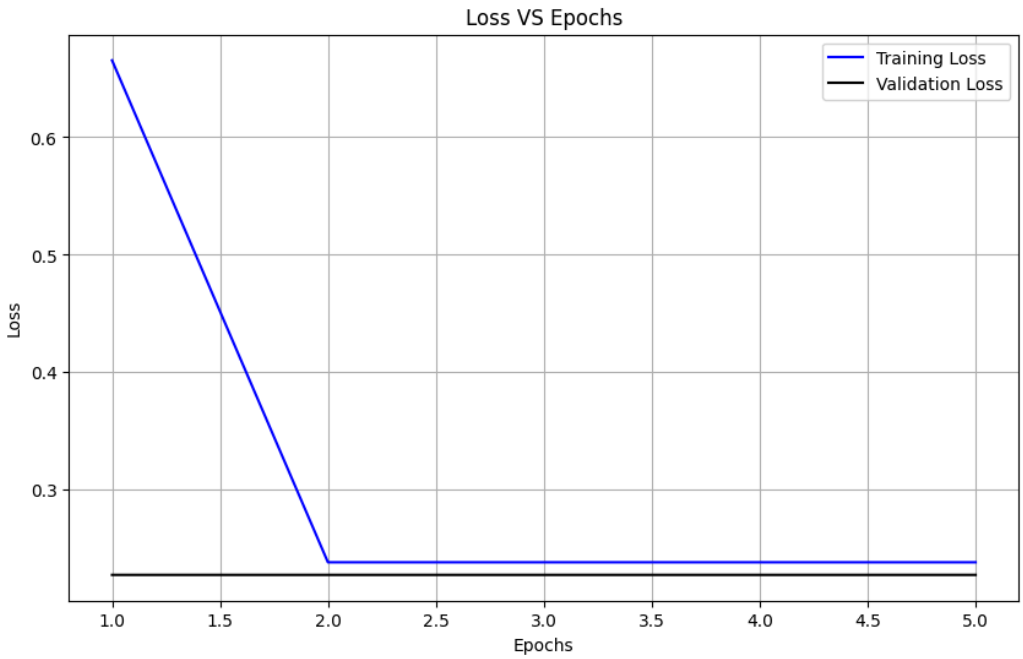
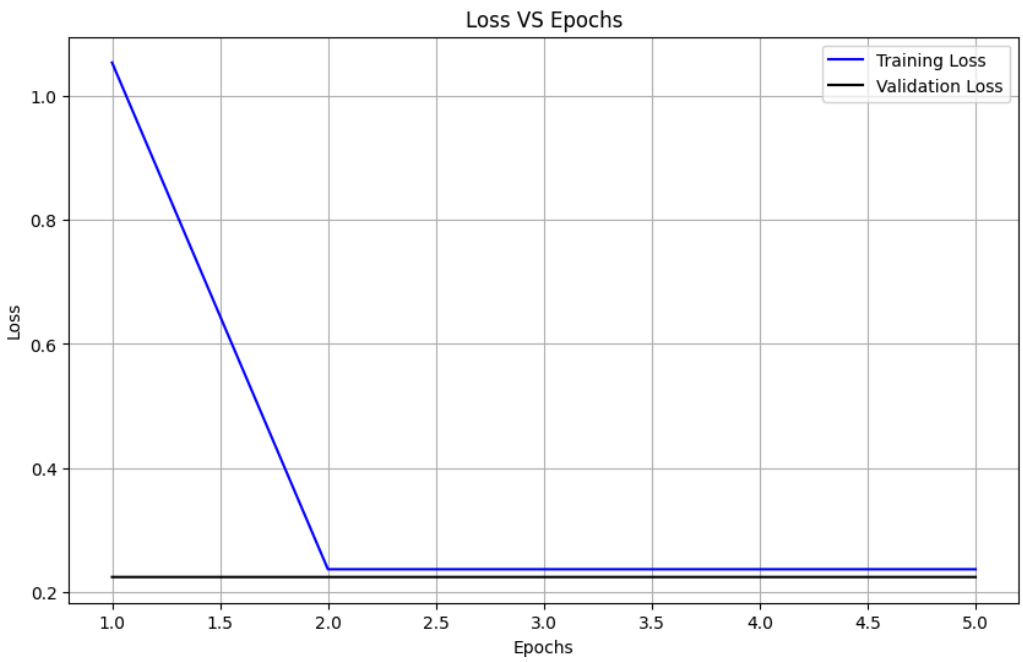
2. Pengaruh Width

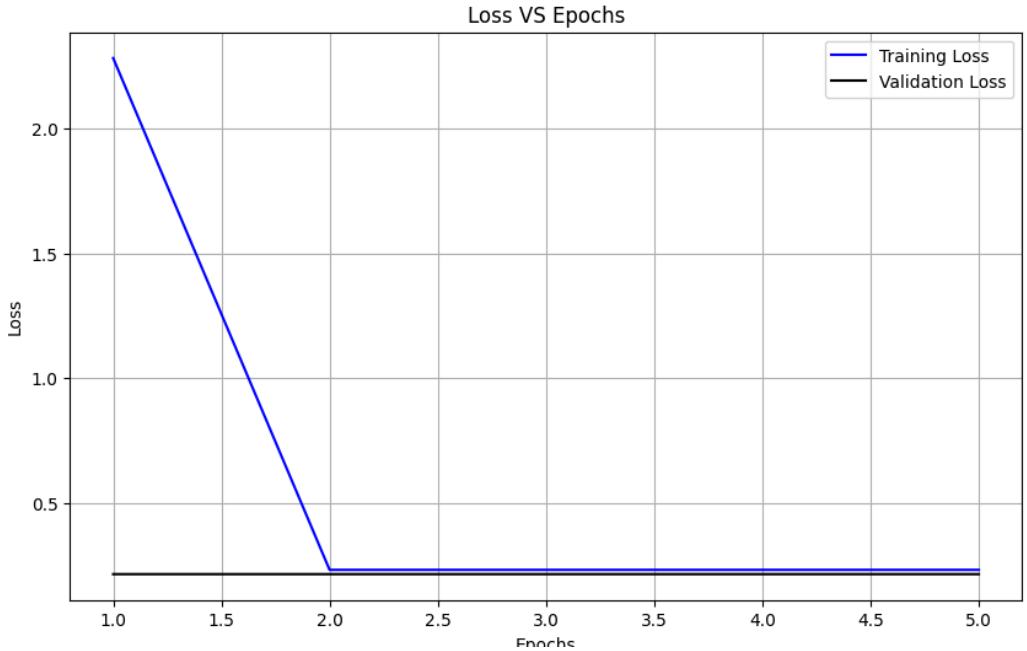
Dilakukan dengan banyak layer sebanyak 3

Hasil akhir prediksi

1 Layer	Akurasi: 0.12
2 Layer	Akurasi: 0.12
5 Layer	Akurasi: 0.12

Grafik training loss dan validation loss tiap epoch

1 Neuron	
2 Neuron	

5 Neuron	

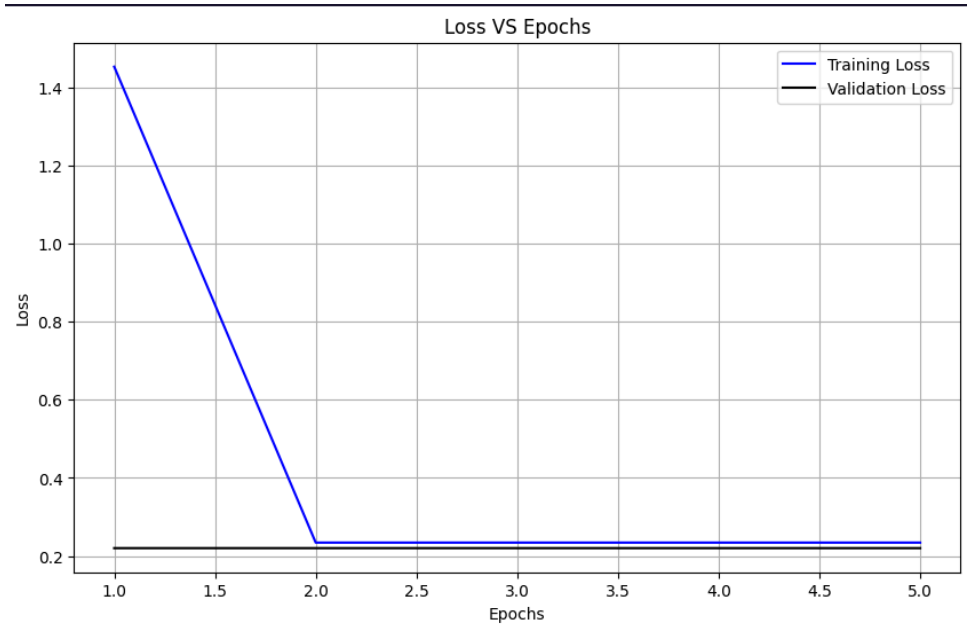
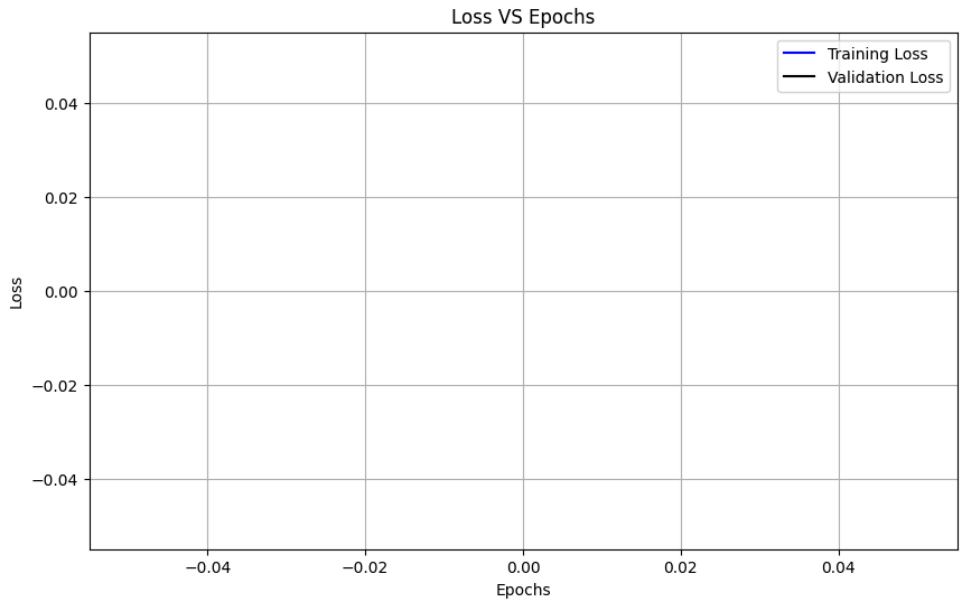
Pengaruh fungsi aktivasi

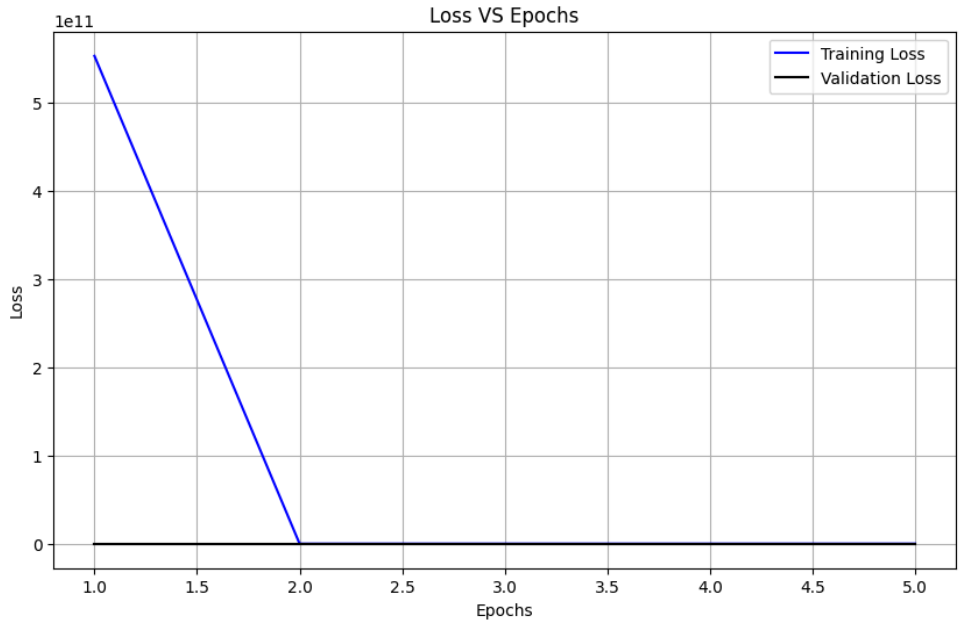
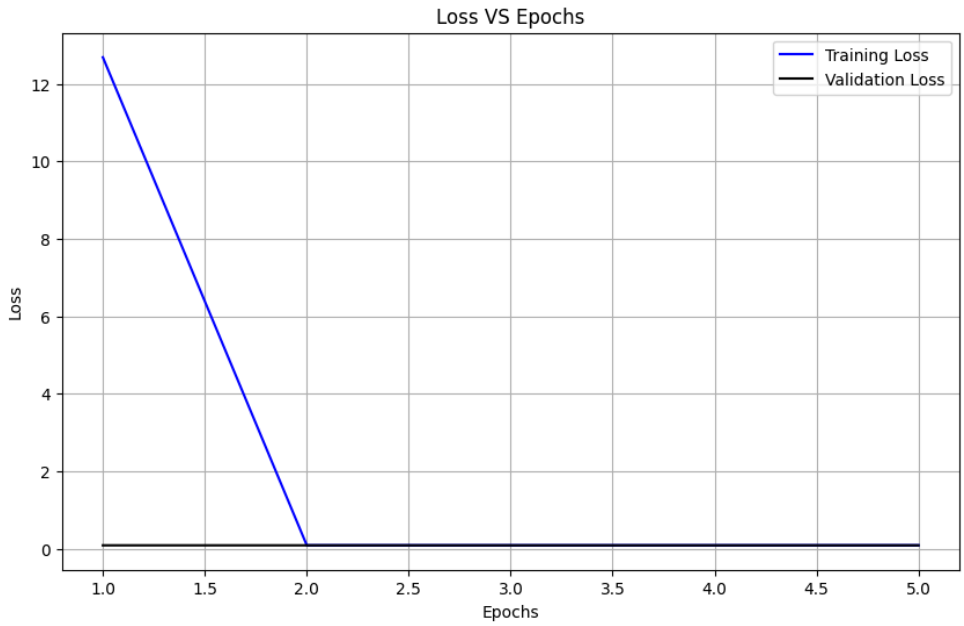
Dilakukan pengujian pengaruh fungsi aktivasi yang bervariasi terhadap hasil training dan prediksi model. Digunakan 3 layer dengan masing masing layer ada 3 neuron, fungsi inisialisasi bobot normal, dan fungsi loss MSE.

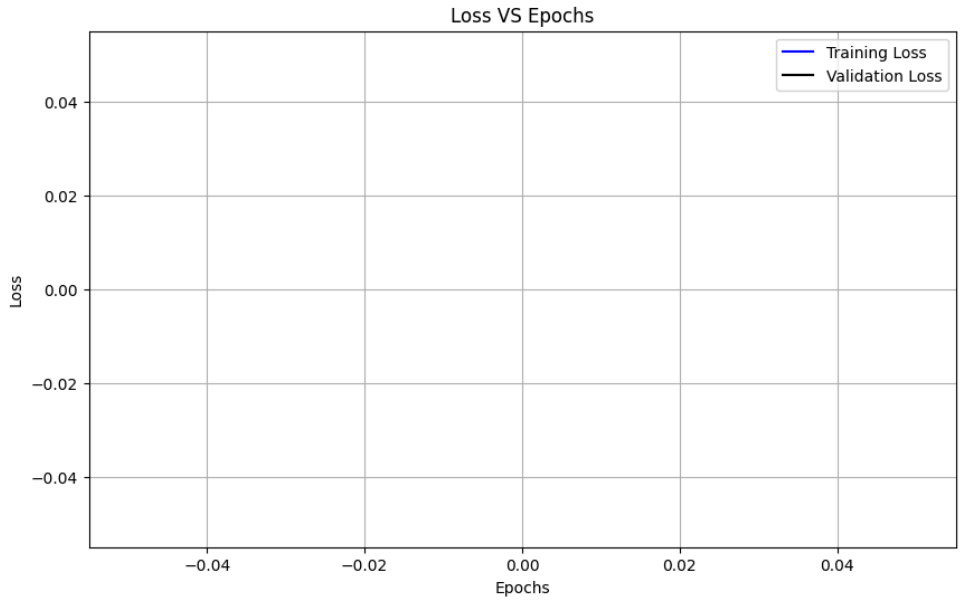
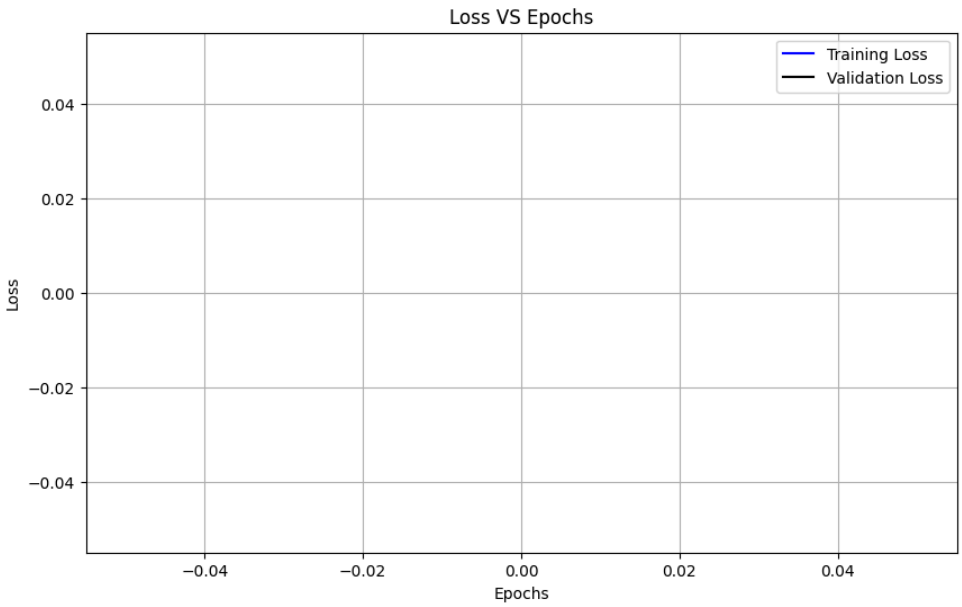
Hasil akhir prediksi

Sigmoid	Akurasi: 0.12
Linear	Akurasi: 0.10
RelU	Akurasi: 0.10
TanH	Akurasi: 0.10
Leaky RelU	Akurasi: 0.10
Swish	Akurasi: 0.10

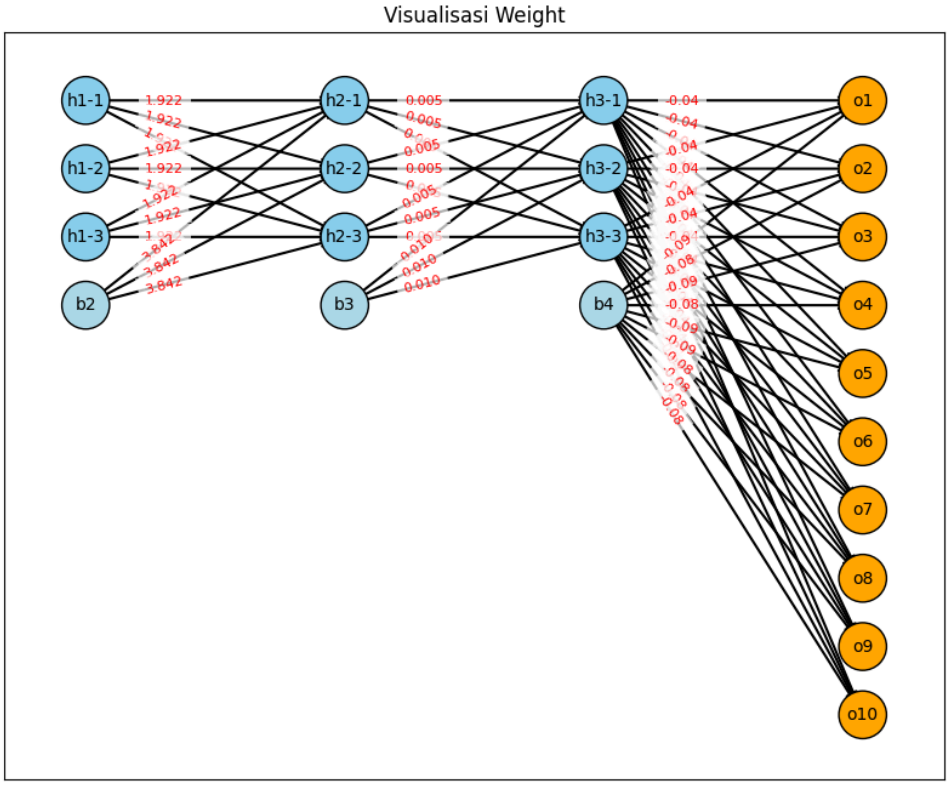
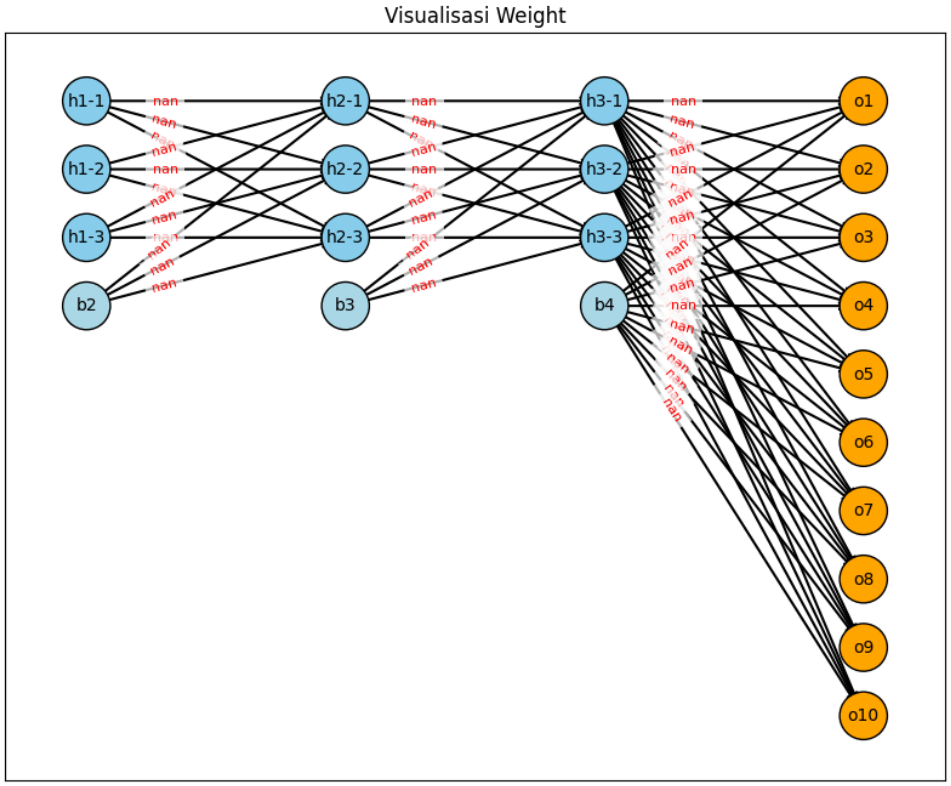
Grafik training loss dan validation loss tiap epoch

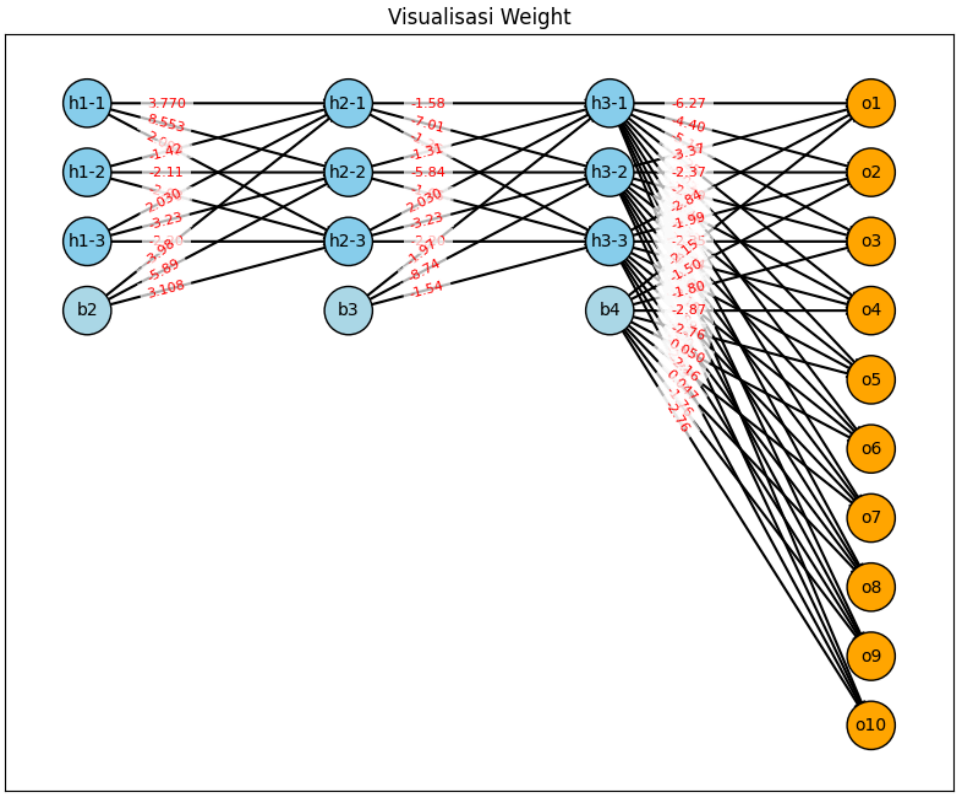
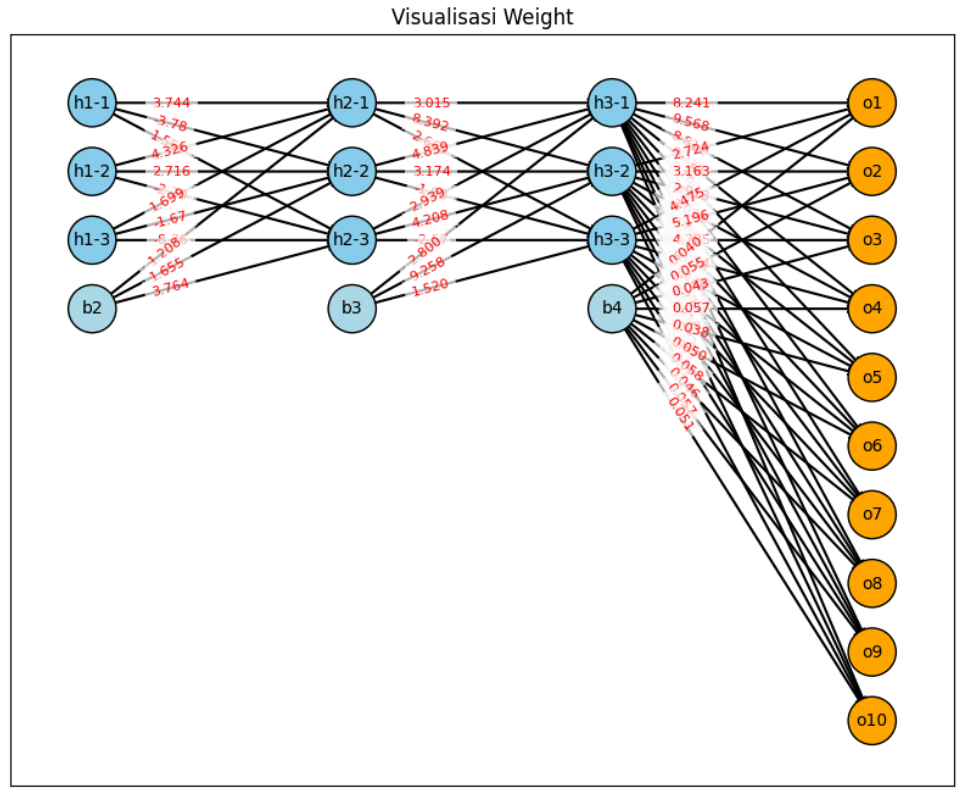
Sigmoid	
Linear	

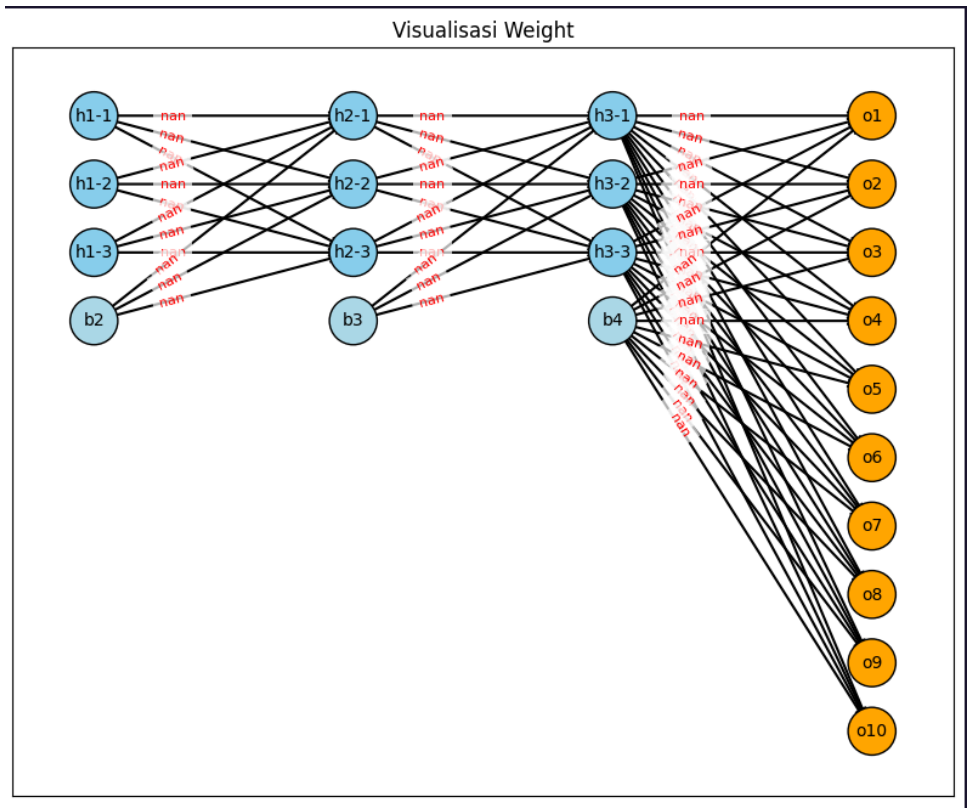
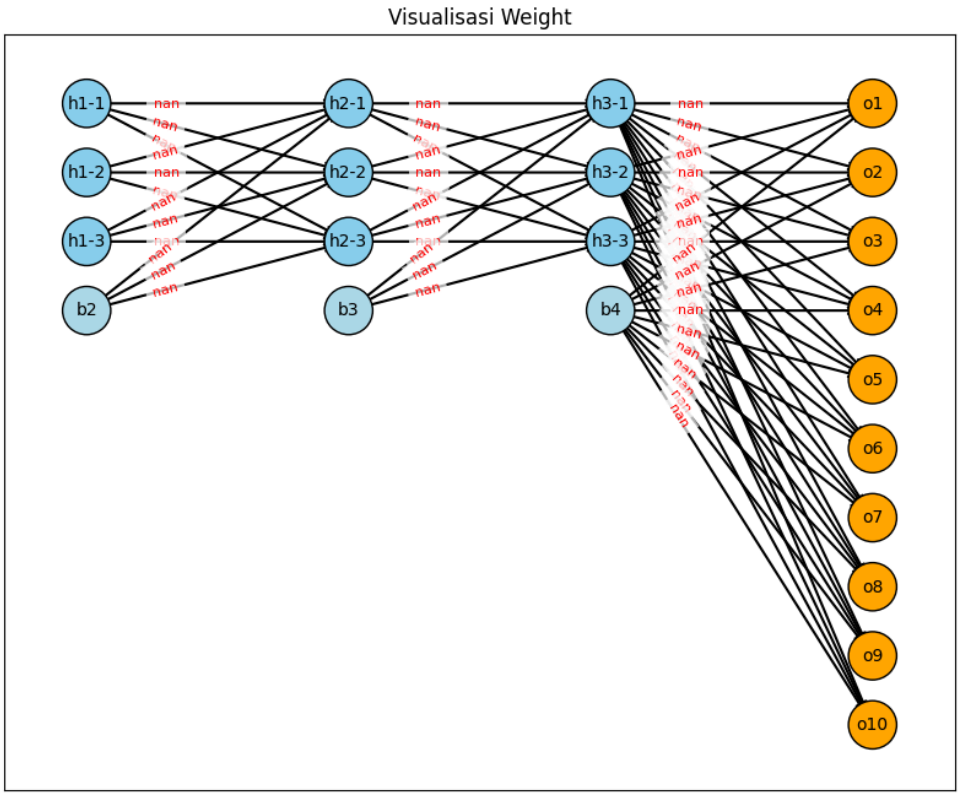
RelU	 Loss VS Epochs 1e11 Training Loss Validation Loss Loss Epochs 0 1 2 3 4 5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
TanH	 Loss VS Epochs Training Loss Validation Loss Loss Epochs 0 2 4 6 8 10 12 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

Leaky RelU	
Swish	

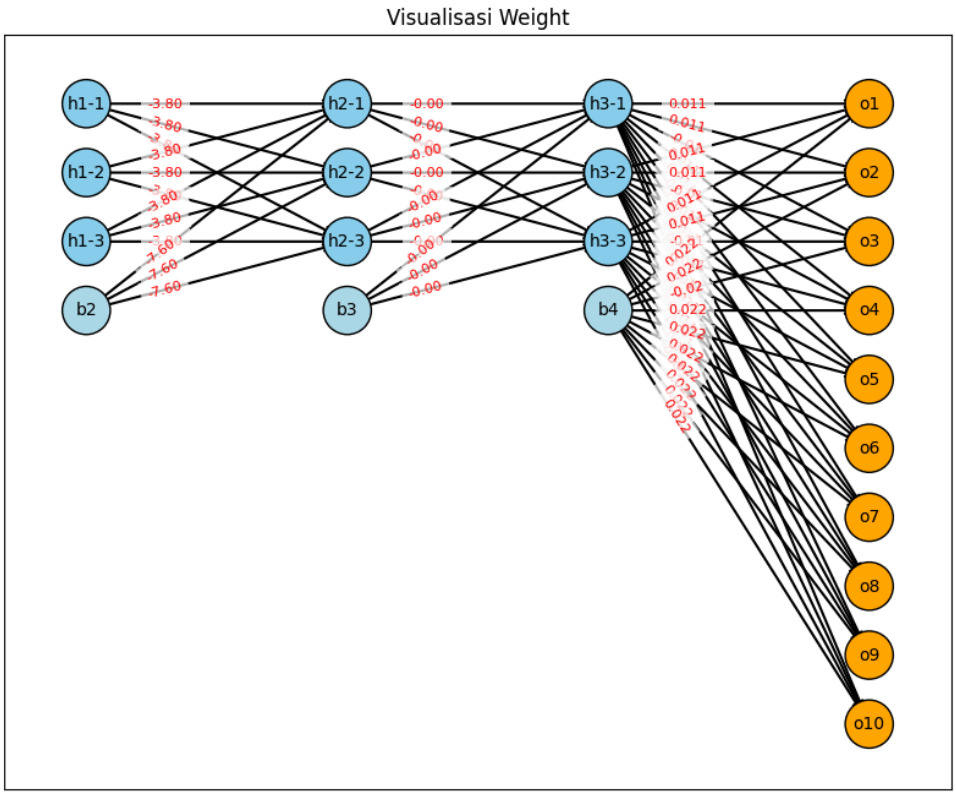
Distribusi bobot

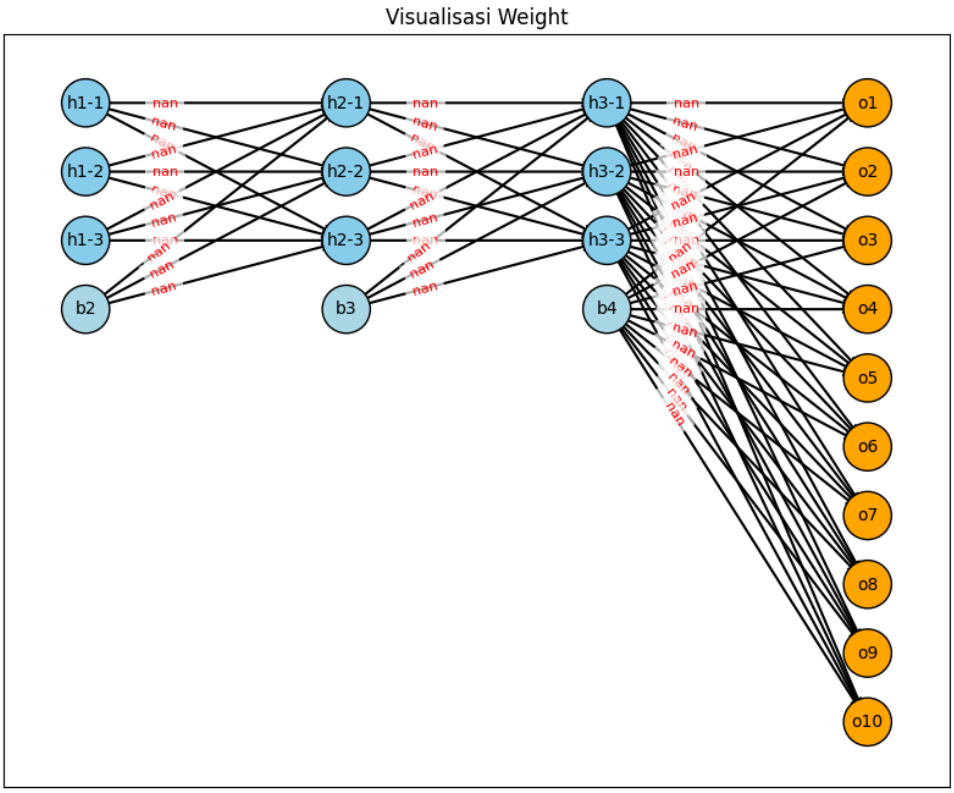
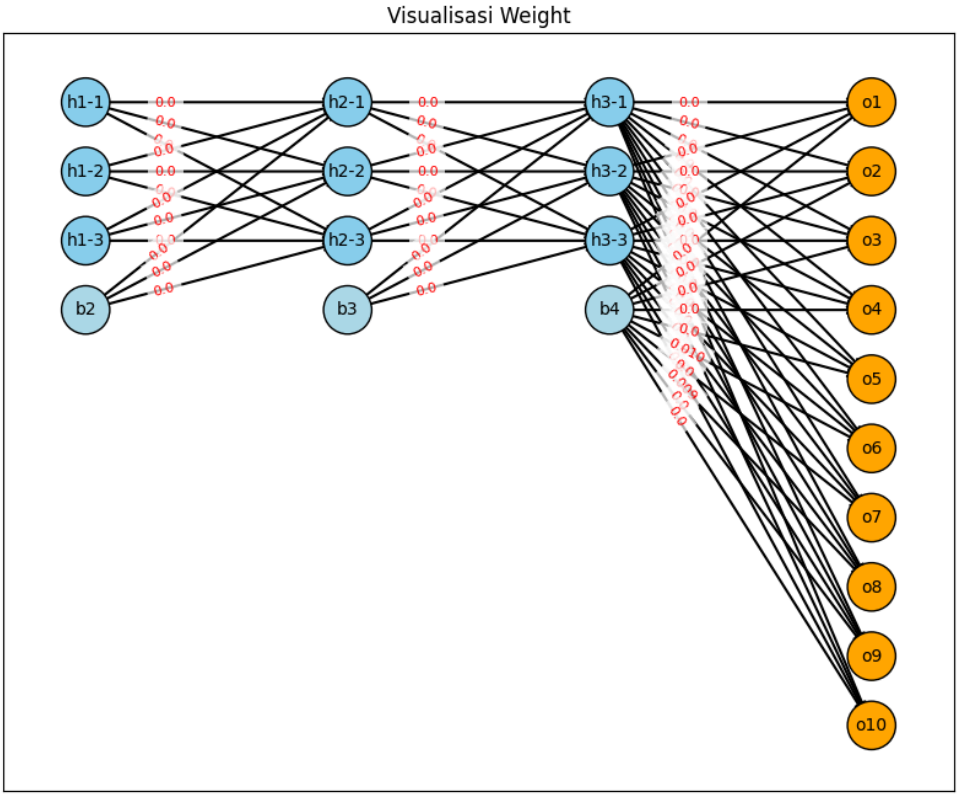
Sigmoid	
Linear	

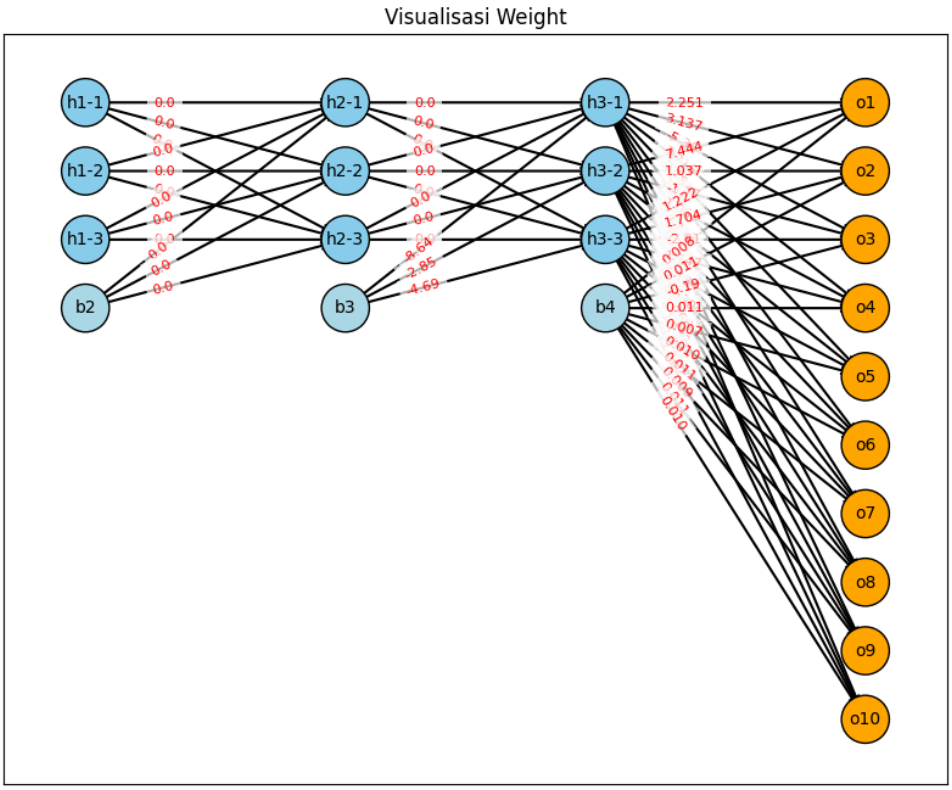
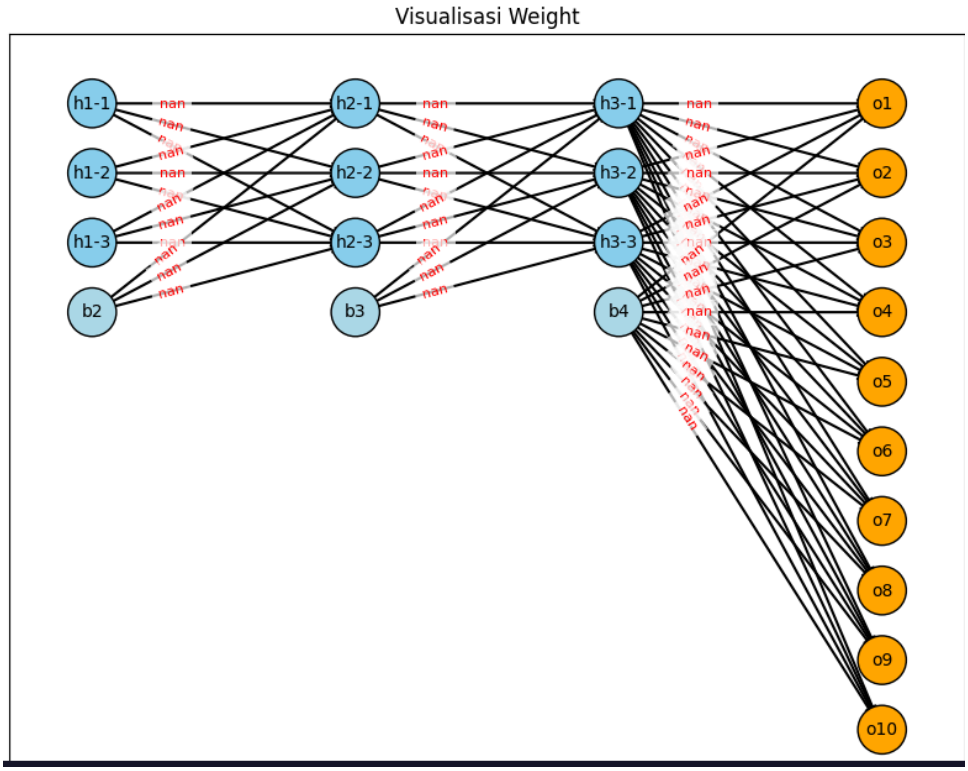
RelU	 Visualisasi Weight h1-1 h1-2 h1-3 b2 h2-1 h2-2 h2-3 b3 h3-1 h3-3 b4 o1 o2 o3 o4 o5 o6 o7 o8 o9 o10
TanH	 Visualisasi Weight h1-1 h1-2 h1-3 b2 h2-1 h2-2 h2-3 b3 h3-1 h3-2 h3-3 b4 o1 o2 o3 o4 o5 o6 o7 o8 o9 o10

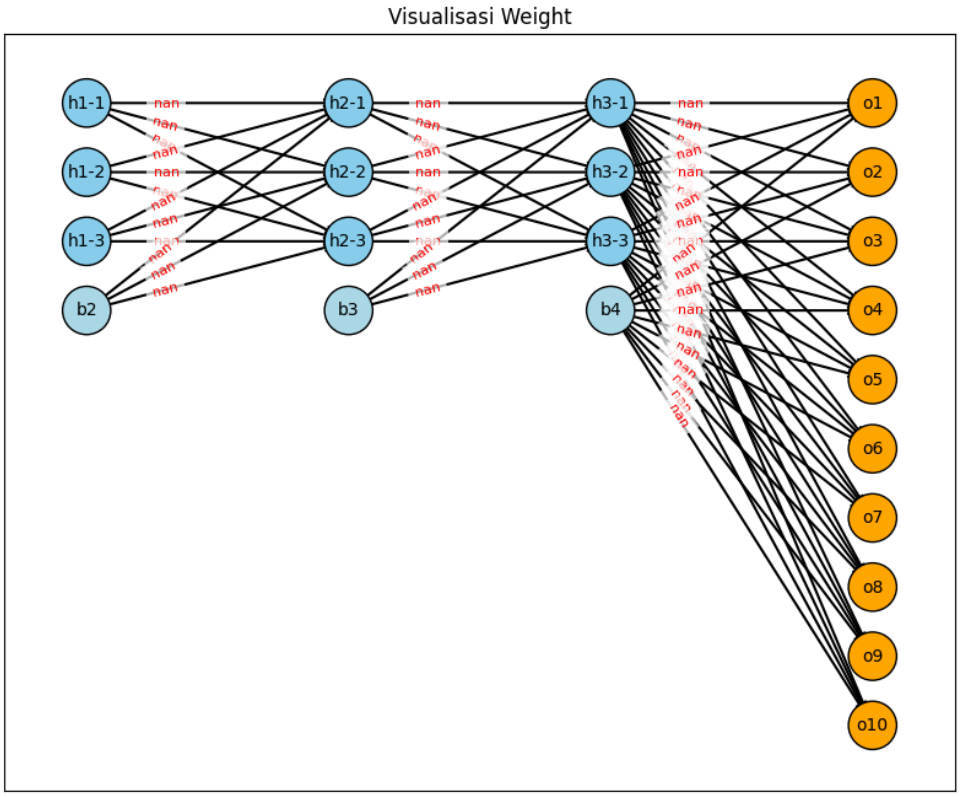
Leaky RelU	
Swish	

Distribusi gradien bobot

Sigmoid	

Linear	 Visualisasi Weight h1-1 h1-2 h1-3 b2 h2-1 h2-2 h2-3 b3 h3-1 h3-3 b4 o1 o2 o3 o4 o5 o6 o7 o8 o9 o10
RelU	 Visualisasi Weight h1-1 h1-2 h1-3 b2 h2-1 h2-2 h2-3 b3 h3-1 h3-3 b4 o1 o2 o3 o4 o5 o6 o7 o8 o9 o10

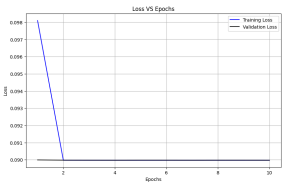
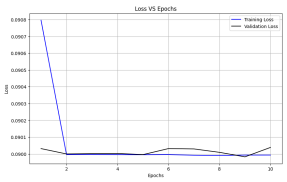
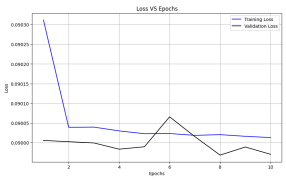
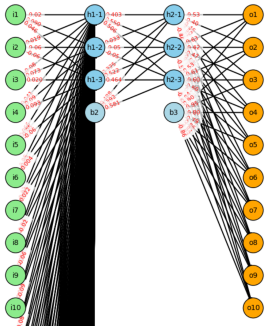
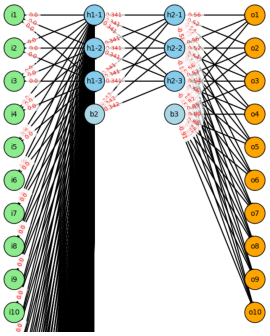
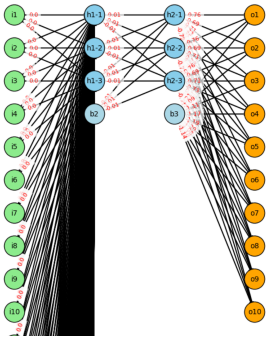
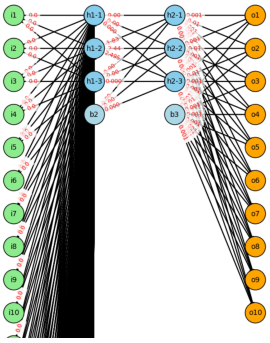
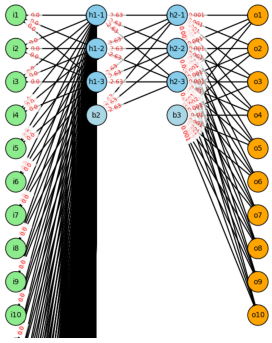
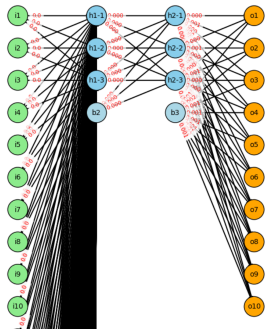
TanH	
Leaky RelU	

Swish	

Pengaruh learning rate

Dilakukan pengujian pengaruh learning rate yang bervariasi terhadap hasil training dan prediksi model, dengan mengambil 3 learning rate, yaitu 0.01, 0.1, 0.3. Digunakan 2 hidden layer dengan masing-masing layer mempunyai 3 neuron, fungsi aktivasi sigmoid untuk semua proses aktivasi, fungsi inisialisasi bobot normal, dan fungsi loss MSE.

	0.01	0.1	0.3
Hasil Akhir Prediksi	`Akurasi: 0.12`	`Akurasi: 0.11`	`Akurasi: 0.10`

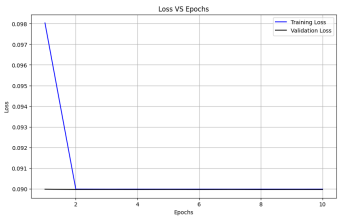
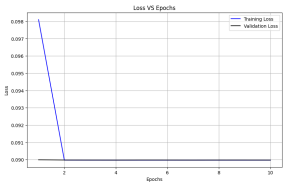
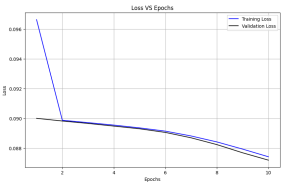
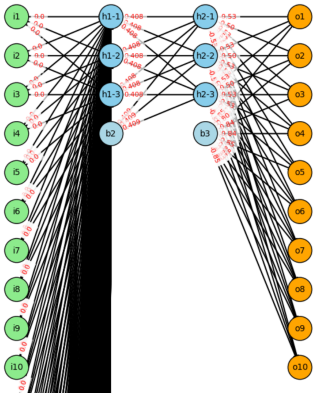
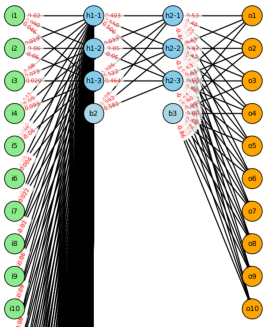
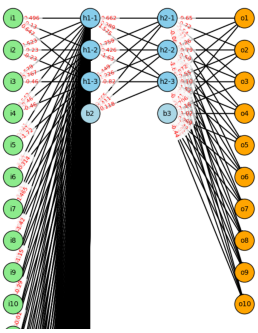
Grafik training loss dan validation loss			
Distribusi bobot			
Distribusi gradien bobot			

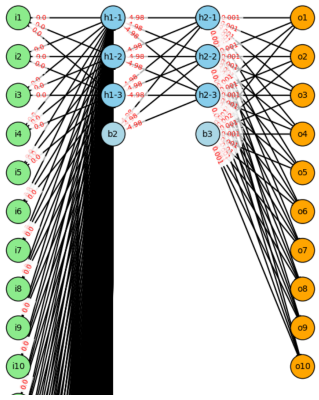
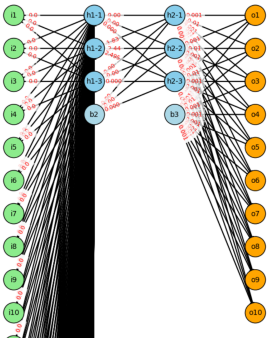
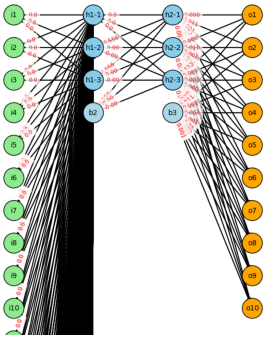
Learning rate mengontrol seberapa besar langkah yang diambil oleh model. Pada umumnya, learning rate yang lebih kecil memiliki konvergensi lebih stabil, namun training lambat dan mungkin terjebak di minimum lokal. Learning rate yang lebih tinggi mempercepat training, namun melewati minimum dan terus melompat-lompat.

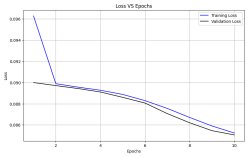
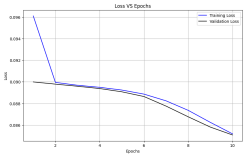
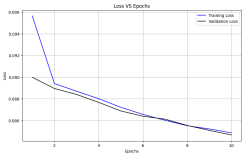
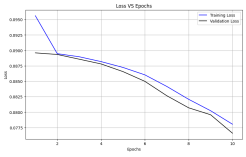
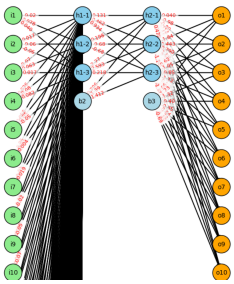
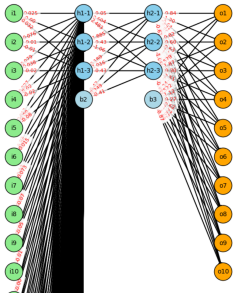
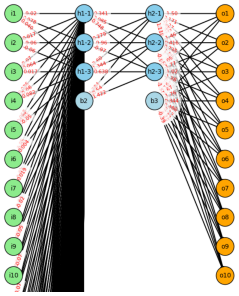
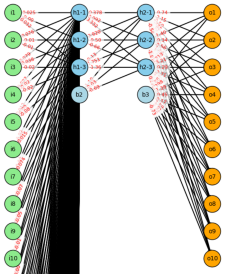
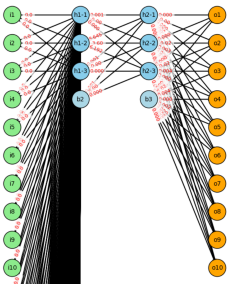
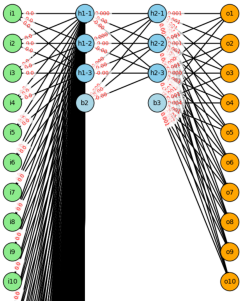
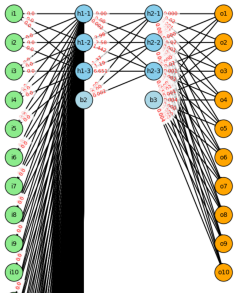
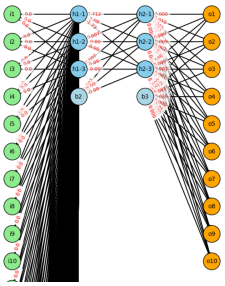
Dalam perbandingan pada model, terlihat perbedaannya paling jelas pada grafik training loss vs validation loss. Learning rate yang tinggi memiliki validation loss yang sering lompat-lompat antar epoch, sedangkan learning rate yang rendah memiliki validation loss yang lebih stabil.

Pengaruh inisialisasi bobot

Dilakukan pengujian pengaruh metode inisialisasi bobot yang bervariasi terhadap hasil training dan prediksi model, yaitu distribusi bobot zeros (semua bobot awal 0), uniform, Xavier, He, dan normal. Digunakan 2 hidden layer dengan masing-masing layer mempunyai 3 neuron, fungsi aktivasi sigmoid untuk semua proses aktivasi, learning rate 0.01, dan fungsi loss MSE.

	zeros	uniform	normal
Hasil Akhir Prediksi	Akurasi: 0.12	Akurasi: 0.12	Akurasi: 0.20
Grafik training loss dan validation loss			
Distribusi bobot			

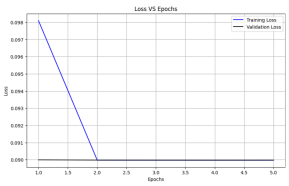
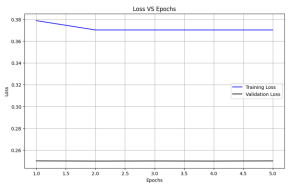
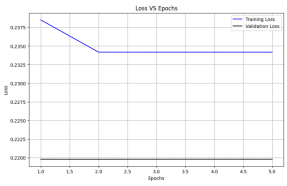
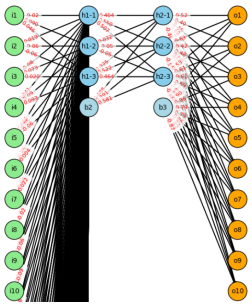
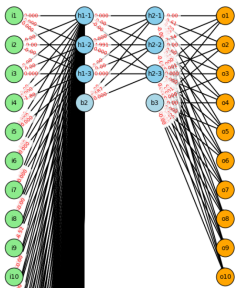
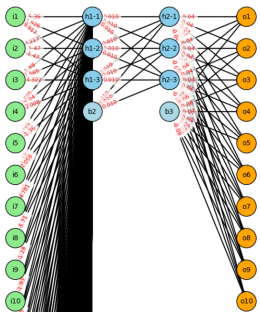
Distribusi gradien bobot			

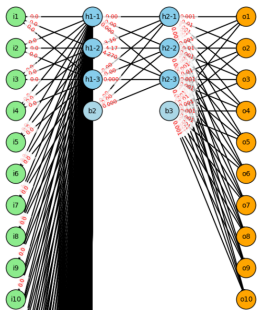
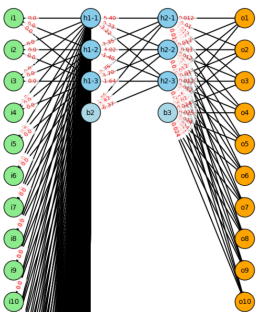
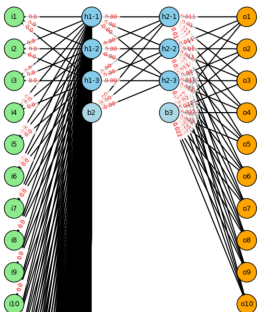
	Xavier uniform	Xavier normal	He uniform	He normal
Hasil Akhir Prediksi	Akurasi: 0.21	Akurasi: 0.26	Akurasi: 0.21	Akurasi: 0.43
Grafik training loss dan validation loss				
Distribusi bobot				
Distribusi gradien bobot				

Berdasarkan pengujian, pemilihan metode inisialisasi bobot sangat krusial dalam akurasi model dan grafik loss yang ideal. He Normal memiliki akurasi yang paling baik yaitu 0.43, dengan grafik menunjukkan test dan validation loss yang konstan turun. Xavier uniform, Xavier normal, He uniform, dan normal memiliki akurasi sekitar 0.2-0.26. Zeros dan uniform memiliki performa buruk yang serupa, akurasi 0.12 dengan grafik loss yang flatline di 0.09.

Pengaruh regularisasi

Dilakukan pengujian pengaruh regularisasi terhadap hasil training dan prediksi model, dengan menguji model tanpa regularisasi, dengan regularisasi L1, dan regularisasi L2, dengan masing-masing sebanyak 1 kali. Digunakan 2 hidden layer dengan masing-masing layer mempunyai 3 neuron, seed 42, weight inizialitation uniform, fungsi aktivasi sigmoid untuk semua proses aktivasi, 5 epoch, learning rate 0.01, dan fungsi loss MSE.

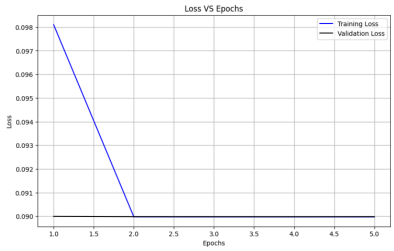
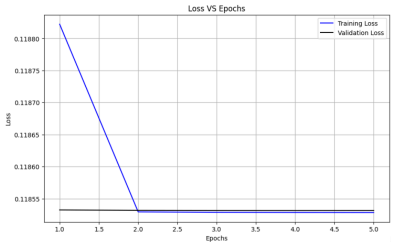
Pembanding	Tanpa regularisasi	L1	L2
Hasil akhir prediksi	`Akurasi: 0.12`	`Akurasi: 0.12`	`Akurasi: 0.10`
Grafik training loss dan validation loss			
Distribusi bobot			

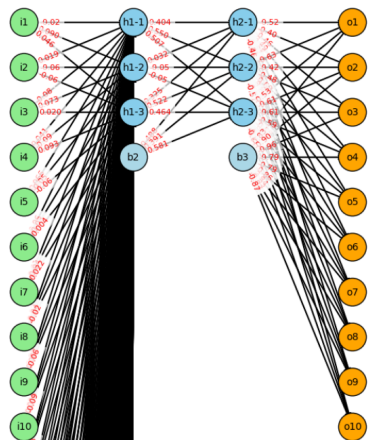
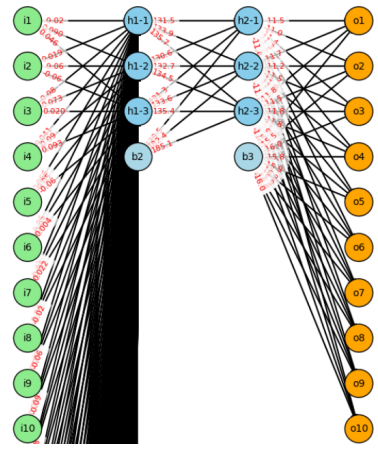
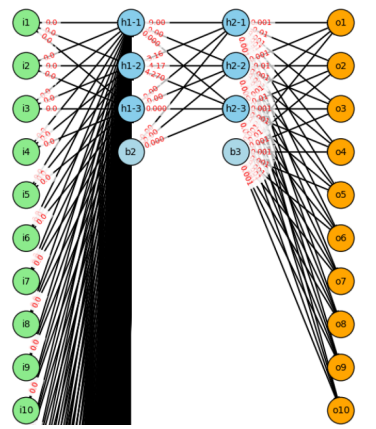
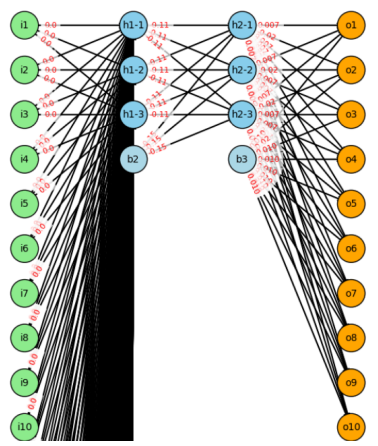
Distribusi gradien bobot			

Berdasarkan perbandingan di atas, tidak begitu besar perbedaan dari hasil prediksi antara model tanpa regularisasi, dengan regularisasi L1 maupun dengan regularisasi L2.

Pengaruh normalisasi RMSNorm

Dilakukan pengujian pengaruh normalisasi RMSNorm terhadap hasil training dan prediksi model dengan membandingkan model tanpa normalisasi dan dengan normalisasi. Digunakan 2 hidden layer dengan masing-masing layer mempunyai 3 neuron, seed 42, weight inizialitation uniform, fungsi aktivasi sigmoid untuk semua proses aktivasi, 5 epoch, learning rate 0.01, dan fungsi loss MSE.

Pembanding	Tanpa normalisasi	Normalisasi RNSNorm
Hasil akhir prediksi	`Akurasi: 0.12`	`Akurasi: 0.10`
Grafik training loss dan validation loss		

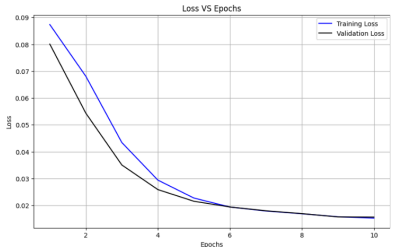
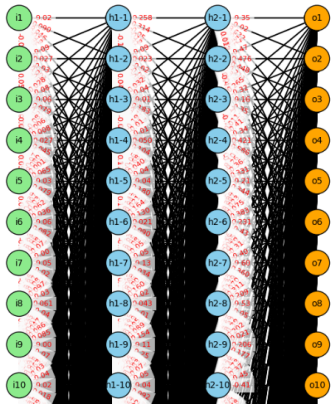
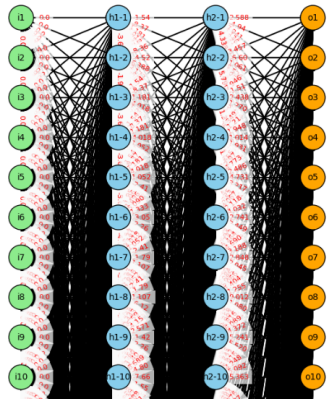
Distribusi bobot		
Distribusi gradien bobot		

Berdasarkan perbandingan di atas, tidak begitu besar perbedaan dari hasil prediksi antara model tanpa normalisasi dengan model menggunakan normalisasi RNSNorm.

Perbandingan dengan library sklearn

Dilakukan pengujian terhadap training dan prediksi model FFNN yang dibuat dengan dari library Scikit-Learn, dengan ketentuan 2 hidden layer, masing-masing 100 neuron dan fungsi aktivasi sigmoid, inisialisasi beban uniform, loss mse, dan learning rate 0.01. Hanya hasil akhir prediksi yang dibandingkan.

Pembanding	Tubes	Scikit-Learn

Hasil akhir prediksi	Akurasi: 0.91	Accuracy: 0.8940
Grafik training loss dan validation loss		-
Distribusi bobot		-
Distribusi gradien bobot		-

Berdasarkan perbandingan di atas, model yang kami buat sedikit lebih baik dibandingkan dengan pada scikit learn.

Kesimpulan dan Saran

Dalam Tugas Besar 1 ini, kelompok telah mengimplementasikan Feedforward Neural Network (FFNN) dari nol (from scratch), dengan fitur-fitur utama seperti berbagai fungsi aktivasi, fungsi loss, metode inisialisasi bobot, dan proses pelatihan menggunakan forward propagation dan backward

propagation. Pengujian dilakukan secara menyeluruh untuk menganalisis pengaruh **jumlah layer (depth), jumlah neuron per layer (width), fungsi aktivasi, learning rate, metode inisialisasi bobot,** dan **regularisasi** terhadap performa model.

Saran dari kami adalah sebaiknya pengujian ini dilakukan dengan model yang mempunyai layer dan neuron cukup besar, sehingga lebih terlihat hasil.

Pembagian tugas tiap anggota kelompok

Berikut adalah pembagian tugas tiap anggota kelompok:

Anggota	Tugas
Dzaky Satrio Nugroho 13522059	Turunan fungsi aktivasi Turunan fungsi inisialisasi bobot Visualisasi graf bobot dan gradiennya Pengujian depth, width, dan fungsi aktivasi
Rafiki Prawhira Harianto 13522065	Fungsi inisialisasi bobot Backpropagation Save and Load Pengujian learning rate dan inisialisasi bobot
Julian Caleb Simandjuntak 13522099	Fungsi aktivasi Forward propagation Visualisasi grafik loss Pengujian regularisasi, normalisasi, dan perbandingan dengan library sklearn Fungsi inisialisasi Xavier dan He Normalisasi RMSNorm Regularisasi L1 dan L2

Referensi

- https://joshnguyen.net/posts/backprop