

IF3270 Pembelajaran Mesin
Feedforward Neural Network

Tugas Besar 1

Disusun untuk memenuhi tugas mata kuliah Pembelajaran Mesin
pada Semester 2 (dua) Tahun Akademik 2024/2025

Tugas Besar IF3270 Pembelajaran Mesin



Oleh

Auralea Alvinia Syaikha 13522148

Muhammad Fauzan Azhim 13522153

Pradipta Rafa Mahesa 13522162

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2025

DAFTAR ISI

DAFTAR ISI	2
BAB 1 DESKRIPSI PERSOALAN	3
BAB 2 PEMBAHASAN	4
2.1 Penjelasan Implementasi	4
2.1.1 Deskripsi Kelas, Atribut, dan Method	4
2.1.2 Forward Propagation	19
2.1.3 Backward Propagation dan Weight Update	19
2.2 Hasil Pengujian	23
2.2.1 Pengaruh depth dan width	23
2.2.2 Pengaruh fungsi aktivasi	26
2.2.2.1 F1 Score	26
2.2.2.2 Train and Validation Loss	27
2.2.2.3 Distribusi Bobot	27
2.2.2.4 Distribusi Gradien	28
2.2.2.5 Analisis	29
2.2.3 Pengaruh learning rate	29
2.2.4 Pengaruh inisialisasi bobot	33
2.2.4.1 Hasil F1 Score	33
2.2.4.2 Train Loss dan Validation Loss	33
2.2.4.3 Distribusi Bobot	33
2.2.4.4 Distribusi Gradien Bobot	34
2.2.5.1 Hasil F1 Score	35
2.2.5.2 Train Loss dan Validation Loss	36
2.2.5.3 Distribusi Bobot	36
2.2.5.4 Distribusi Gradien Bobot	37
2.2.5.5 Analisis	37
2.2.6 Pengaruh RMS Prop	38
2.2.6.1 Hasil F1 Score	38
2.2.6.2 Train Loss dan Validation Loss	38
2.2.6.3 Distribusi Bobot	38
2.2.6.4 Distribusi Gradien Bobot	39
2.2.6.5 Analisis	39
2.2.7 Perbandingan dengan library sklearn	40
BAB 3 KESIMPULAN DAN SARAN	42
3.1 Kesimpulan	42
3.1 Saran	43
Pembagian Tugas	44
Referensi	45

BAB 1 DESKRIPSI PERSOALAN

Feedforward Neural Network (FFNN) adalah salah satu arsitektur dasar dalam jaringan saraf tiruan yang terdiri dari beberapa lapisan neuron yang saling terhubung secara penuh (fully connected) dari input ke output tanpa adanya umpan balik (loop). FFNN banyak digunakan dalam berbagai aplikasi machine learning, termasuk klasifikasi, regresi, dan pemrosesan sinyal. Dalam tugas besar ini, penulis diminta untuk mengimplementasikan modul FFNN dari nol tanpa menggunakan library deep learning seperti TensorFlow atau PyTorch. Implementasi ini harus mencakup berbagai fitur yang memungkinkan pengguna menyesuaikan struktur jaringan, memilih fungsi aktivasi, serta melatih model dengan metode optimasi berbasis gradient descent.

Modul FFNN yang dikembangkan harus mampu menerima konfigurasi jumlah neuron di setiap layer, termasuk lapisan input dan output. Fungsi aktivasi yang didukung mencakup Linear, ReLU, Sigmoid, Hyperbolic Tangent (tanh), dan Softmax. Selain itu, model harus mendukung beberapa fungsi loss seperti Mean Squared Error (MSE), Binary Cross-Entropy, dan Categorical Cross-Entropy dengan basis logaritma natural. Untuk inisialisasi bobot, metode yang dapat digunakan meliputi inisialisasi nol, distribusi uniform dengan batas tertentu, serta distribusi normal dengan parameter mean dan variance. Model ini juga harus dapat menampilkan struktur jaringan beserta bobot dan gradien dalam bentuk graf serta memiliki fitur penyimpanan dan pemuatan (save & load) model untuk digunakan kembali.

Implementasi harus mencakup forward propagation yang mendukung input dalam bentuk batch serta backward propagation yang menerapkan konsep chain rule untuk menghitung gradien bobot terhadap loss function. Model juga harus dapat memperbarui bobot menggunakan gradient descent dan mendukung berbagai parameter pelatihan seperti batch size, learning rate, jumlah epoch, serta opsi verbose untuk menampilkan progres selama training. Selama proses pelatihan, model juga harus mencatat training loss dan validation loss di setiap epoch.

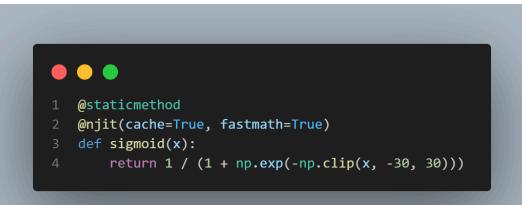
Untuk evaluasi, model akan diuji menggunakan dataset MNIST 784 yang dimuat melalui `fetch_openml` dari `sklearn`. Pengujian mencakup analisis pengaruh jumlah layer dan jumlah neuron per layer, perbandingan hasil dengan berbagai fungsi aktivasi, variasi learning rate, serta metode inisialisasi bobot. Selain itu, kinerja FFNN yang diimplementasikan akan dibandingkan dengan model `MLPClassifier` dari `sklearn` guna menilai efektivitas model yang dikembangkan dibandingkan dengan solusi yang telah tersedia dalam library.

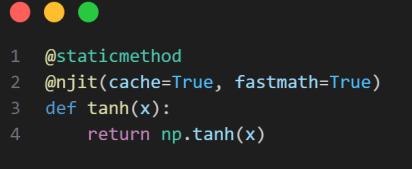
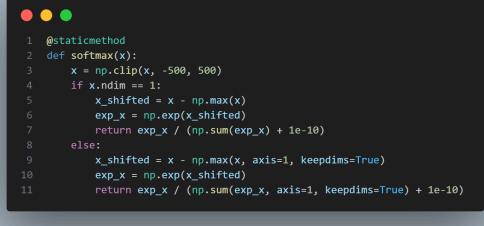
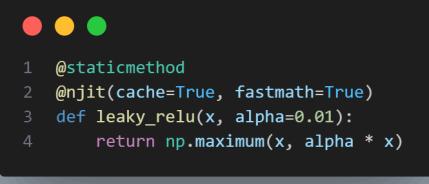
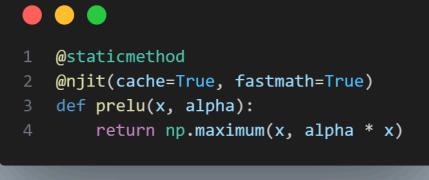
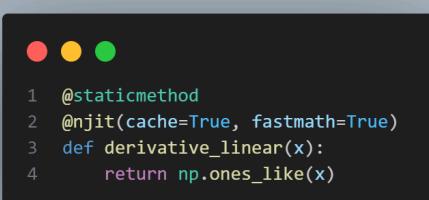
BAB 2 PEMBAHASAN

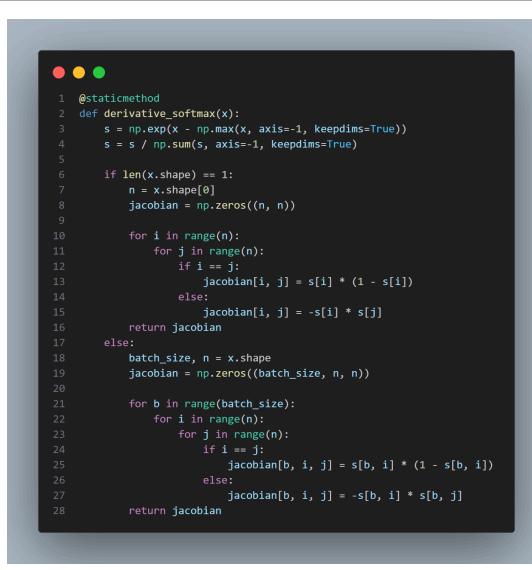
2.1 Penjelasan Implementasi

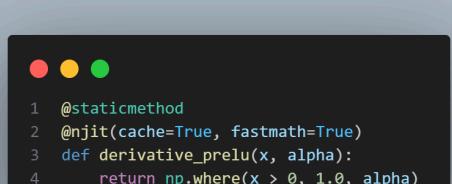
2.1.1 Deskripsi Kelas, Atribut, dan Method

Berikut adalah penjelasan dari kelas, atribut dari masing-masing kelas, dan method yang ada dalam masing-masing kelas.

Class ActivationFunction (ActivationFunction.py)	
Kelas ActivationFunction adalah kumpulan metode statis yang mengimplementasikan berbagai fungsi aktivasi dan turunannya untuk digunakan dalam model neural network.	
Fungsi	Deskripsi
	Fungsi yang mengembalikan nilai linear dari suatu input (x) yaitu mengembalikan input (x) tanpa perubahan, merepresentasikan fungsi aktivasi linear yang tidak melakukan transformasi pada nilai input.
	Fungsi aktivasi ReLU (Rectified Linear Unit) yang mengembalikan x jika $x > 0$, dan 0 jika $x \leq 0$...
	Fungsi aktivasi sigmoid yang mengubah input menjadi nilai dalam rentang (0,1), merepresentasikan probabilitas.

 <pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def tanh(x): 4 return np.tanh(x) </pre>	<p>Fungsi aktivasi tangens hiperbolik yang memetakan input ke dalam rentang (-1,1).</p>
 <pre> 1 @staticmethod 2 def softmax(x): 3 x = np.clip(x, -500, 500) 4 if x.ndim == 1: 5 x_shifted = x - np.max(x) 6 exp_x = np.exp(x_shifted) 7 return exp_x / (np.sum(exp_x) + 1e-10) 8 else: 9 x_shifted = x - np.max(x, axis=1, keepdims=True) 10 exp_x = np.exp(x_shifted) 11 return exp_x / (np.sum(exp_x, axis=1, keepdims=True) + 1e-10) </pre>	<p>Fungsi aktivasi softmax yang mengkonversi input menjadi distribusi probabilitas.</p>
 <pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def leaky_relu(x, alpha=0.01): 4 return np.maximum(x, alpha * x) </pre>	<p>Fungsi aktivasi Leaky ReLU yang mirip dengan ReLU tetapi mengizinkan nilai negatif dengan gradien kecil (alpha).</p>
 <pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def prelu(x, alpha): 4 return np.maximum(x, alpha * x) </pre>	<p>Fungsi aktivasi PReLU (Parametric ReLU) yang mirip dengan Leaky ReLU tetapi dengan alpha sebagai parameter yang dapat dilatih.</p>
 <pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def derivative_linear(x): 4 return np.ones_like(x) </pre>	<p>Fungsi turunan dari aktivasi linear yang selalu mengembalikan nilai 1.</p>

 <pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def derivative_relu(x): 4 return np.where(x > 0, 1, 0) </pre>	<p>Fungsi turunan dari ReLU yang mengembalikan 1 jika $x > 0$ dan 0 jika $x \leq 0$.</p>
 <pre> 1 @staticmethod 2 def derivative_sigmoid(x): 3 sig = ActivationFunction.sigmoid(x) 4 return sig * (1 - sig) </pre>	<p>Fungsi turunan dari sigmoid yang dihitung sebagai $\text{sigmoid}(x) * (1 - \text{sigmoid}(x))$.</p>
 <pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def derivative_tanh(x): 4 return 1 - np.tanh(x) ** 2 </pre>	<p>Fungsi turunan dari tanh yang dihitung sebagai $1 - \tanh^2(x)$.</p>
 <pre> 1 @staticmethod 2 def derivative_softmax(x): 3 s = np.exp(x - np.max(x, axis=-1, keepdims=True)) 4 s = s / np.sum(s, axis=-1, keepdims=True) 5 6 if len(x.shape) == 1: 7 n = x.shape[0] 8 jacobian = np.zeros((n, n)) 9 10 for i in range(n): 11 for j in range(n): 12 if i == j: 13 jacobian[i, j] = s[i] * (1 - s[i]) 14 else: 15 jacobian[i, j] = -s[i] * s[j] 16 return jacobian 17 else: 18 batch_size, n = x.shape 19 jacobian = np.zeros((batch_size, n, n)) 20 21 for b in range(batch_size): 22 for i in range(n): 23 for j in range(n): 24 if i == j: 25 jacobian[b, i, j] = s[b, i] * (1 - s[b, i]) 26 else: 27 jacobian[b, i, j] = -s[b, i] * s[b, j] 28 29 return jacobian </pre>	<p>Fungsi turunan dari softmax yang mengembalikan matriks Jacobian</p>

 <pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def derivative_prelu(x, alpha): 4 return np.where(x > 0, 1.0, alpha) </pre>	<p>Fungsi turunan dari PReLU yang mirip dengan turunan Leaky ReLU, tetapi dengan nilai α sebagai parameter yang dapat diubah.</p>

<h2>Class Initializer (Initializer.py)</h2>	
Fungsi	Deskripsi
 <pre> 1 @staticmethod 2 def init_weights(weight_init, param_1, param_2, input_size, num_neurons): 3 if weight_init == InitializerType.ZERO: 4 return np.zeros((num_neurons, input_size)) 5 elif weight_init == InitializerType.UNIFORM: 6 return np.random.uniform(param_1, param_2, (num_neurons, input_size)) * np.sqrt(2.0 / input_size) 7 elif weight_init == InitializerType.RANDOM_DIST_NORMAL: 8 return (np.random.normal(param_1, param_2, (num_neurons, input_size))) * np.sqrt(2.0 / input_size) 9 elif weight_init == InitializerType.XAVIER: 10 limit = np.sqrt(6 / (input_size + num_neurons)) 11 return np.random.uniform(-limit, limit, (num_neurons, input_size)) 12 elif weight_init == InitializerType.HE: 13 std_dev = np.sqrt(1.0 / input_size) 14 return np.random.normal(0, std_dev, (num_neurons, input_size)) </pre>	<p>Fungsi yang berguna untuk menginisialisasi bobot dari neuron-neuron pada suatu layer, mengembalikan array of float yang mana setiap elemen dalam float tersebut adalah weight suatu neuron. Fungsi dapat melakukan inisialisasi bobot dengan 5 metode yaitu zero(0), distribusi normal, distribusi uniform, xavier(glorot), dan he yang dapat dipilih berdasarkan input yang diterima.</p>

<pre> 1 @staticmethod 2 def init_bias(bias_init: InitializerType, param_1, param_2, num_neurons): 3 if bias_init == InitializerType.ZERO: 4 return np.zeros(num_neurons) 5 elif bias_init == InitializerType.RANDOM_DIST_UNIFORM: 6 return np.ones(num_neurons) * np.random.uniform(param_1, param_2) 7 elif bias_init == InitializerType.RANDOM_DIST_NORMAL: 8 return np.ones(num_neurons) * np.random.normal(param_1, param_2) 9 elif bias_init == InitializerType.XAVIER: 10 return np.zeros(num_neurons) 11 elif bias_init == InitializerType.HE: 12 return np.zeros(num_neurons) </pre>	<p>Fungsi yang berguna untuk menginisialisasi bobot bias dari suatu layer, mengembalikan array of float yang mana setiap elemen dalam float tersebut adalah bobot suatu bias. Fungsi dapat melakukan inisialisasi bobot dengan 5 metode yaitu zero(0), distribusi normal, dan distribusi uniform dapat dipilih berdasarkan input yang diterima. Xavier dan He menggunakan distribusi bobot bias zero.</p>
---	---

<h2>Class LossFunction (LossFunction.py)</h2>	
<p>Kelas LossFunction digunakan untuk mendefinisikan berbagai fungsi loss (kerugian) yang digunakan dalam pelatihan jaringan neural.</p>	
Fungsi	Deskripsi
<pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def mean_squared_error(y_pred, y_true, epsilon=1e-7): 4 return np.mean((y_pred - y_true) ** 2) </pre>	<p>Fungsi loss Mean Squared Error (MSE) yang menghitung rata-rata dari kuadrat selisih antara prediksi (y_{pred}) dan nilai sebenarnya (y_{true}). Digunakan untuk regresi dan mengukur seberapa jauh prediksi dari target.</p>
<pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def mean_squared_error_derivative(y_pred, y_true, epsilon=1e-7): 4 return 2 * (y_pred - y_true) </pre>	<p>Fungsi turunan dari Mean Squared Error (MSE) yang menghitung gradien loss terhadap prediksi, yaitu 2 kali selisih antara prediksi dan target. Digunakan saat pembaruan bobot selama backpropagation.</p>
<pre> 1 @staticmethod 2 @njit(cache=True, fastmath=True) 3 def binary_cross_entropy(y_pred, y_true, epsilon=1e-7): 4 y_pred = np.clip(y_pred, epsilon, 1.0 - epsilon) 5 return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred)) </pre>	<p>Fungsi loss Binary Cross-Entropy (BCE) yang menghitung kesalahan untuk tugas klasifikasi biner. Menggunakan logaritma dari prediksi yang dikalibrasi agar tetap dalam batas numerik yang stabil.</p>

```
1 @staticmethod
2     @njit(cache=True, fastmath=True)
3     def binary_cross_entropy_derivative(y_pred, y_true, epsilon=1e-7):
4         y_pred = np.clip(y_pred, epsilon, 1.0 - epsilon)
5         return -(y_true / y_pred) - ((1 - y_true) / (1 - y_pred))
```

Fungsi turunan dari Binary Cross-Entropy (BCE) yang menghitung gradien loss terhadap prediksi, membantu dalam pembaruan parameter model selama training.

```
1 @staticmethod
2     @njit(cache=True, fastmath=True)
3     def categorical_cross_entropy(y_pred, y_true, epsilon=1e-7):
4         y_pred = np.clip(y_pred, epsilon, 1.0 - epsilon)
5
6         if y_pred.ndim == 1:
7             # For 1D arrays (binary case)
8             return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
9         else:
10            # For 2D arrays (multi-class case)
11            return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
```

Fungsi loss Categorical Cross-Entropy (CCE) yang mengukur perbedaan antara distribusi probabilitas output model (y_{pred}) dan label sebenarnya (y_{true}). Digunakan untuk klasifikasi multi-kelas.

```
1 @staticmethod
2     @njit(cache=True, fastmath=True)
3     def categorical_cross_entropy_derivative(y_pred, y_true, epsilon=1e-7):
4         y_pred = np.clip(y_pred, epsilon, 1.0 - epsilon)
```

Fungsi turunan dari Categorical Cross-Entropy (CCE) yang menghitung selisih antara prediksi dan label sebenarnya, digunakan dalam backpropagation untuk memperbarui bobot model.

Class Layer (Layer.py)

Class Layer adalah kelas yang merepresentasikan suatu layer pada neural network

Atribut	Deskripsi
self.layer_name	Variabel yang menyimpan nama dari layer
self.activation_func	Variabel yang menyimpan fungsi aktivasi yang akan digunakan oleh layer tersebut (linear,relu,sigmoid,tanh)
self.derivative_activation	Variabel yang menyimpan turunan dari fungsi aktivasi yang terdapat pada atribut activation_func.
self.sum	Variabel yang digunakan untuk menyimpan hasil dari operasi linear pada input yaitu hasil dari perkalian matriks antara input x dengan bobot, ditambah bias,
self.output	Variabel yang digunakan untuk menyimpan hasil akhir dari lapisan setelah menerapkan fungsi aktivasi pada self.sum
self.alpha	Variabel yang menyimpan learning rate dari suatu layer
self.num_neurons	Variabel yang menyimpan informasi banyaknya jumlah neuron pada layer tersebut
self.weights	Variabel yang menyimpan bobot-bobot dari semua neuron pada layer tersebut
self.biases	Variabel yang menyimpan bobot-bobot dari semua bias pada layer tersebut
self.grad_weights	Variabel yang menyimpan gradien dari bobot-bobot dari semua neuron pada layer tersebut
self.last_input	Variabel yang digunakan untuk menyimpan input yang diterima oleh lapisan selama forward propagation. Variabel ini penting untuk proses backward propagation saat menghitung gradien untuk memperbarui bobot dan bias.
Fungsi	Deskripsi

<pre> 1 def forward(self, x): 2 self.last_input = x 3 self.sum = np.dot(x, self.weights.T) + self.biases 4 if self.alpha is not None: 5 self.output = ActivationFunction.prelu(self.sum, self.alpha) 6 else: 7 self.output = self.activation_func(self.sum) 8 return self.output </pre>	<p>Fungsi ini melakukan forward propagation yaitu dengan menghitung output lapisan dengan operasi linear dan fungsi aktivasi, serta menyimpan input untuk digunakan dalam pelatihan.</p>
<pre> 1 def backward(self, lr, delta_next): 2 if self.alpha is not None: 3 local_grad = self.derivative_activation(self.sum, self.alpha) 4 else: 5 local_grad = self.derivative_activation(self.sum) 6 7 local_grad = np.nan_to_num(local_grad, nan=0.0, posinf=1.0, neginf=-1.0) 8 delta_next = np.nan_to_num(delta_next, nan=0.0, posinf=1.0, neginf=-1.0) 9 10 delta = local_grad * delta_next 11 12 self.grad_weights = np.dot(delta.T, self.last_input) / self.last_input.shape[0] 13 grad_b = np.mean(delta, axis=0) 14 15 self.weights -= lr * self.grad_weights 16 self.biases -= lr * grad_b 17 18 delta_prev = np.dot(delta, self.weights) 19 return delta_prev </pre>	<p>Metode ini melakukan backward propagation yaitu dengan menghitung error lokal, gradien bobot/bias, memperbarui parameter, dan mengembalikan error untuk lapisan sebelumnya.</p>

<h2>Class InputLayer (InputLayer.py)</h2>	
<p>Class Output Layer adalah kelas yang merepresentasikan layer masukkan (input layer) pada neural network. Kelas ini digunakan untuk mendefinisikan atribut-atribut lapisan input. Kelas ini juga tidak memiliki bobot atau bias, tetapi dirancang agar kompatibel dengan lapisan lain dalam jaringan saraf.</p>	
Atribut	Deskripsi
input_size	Jumlah fitur dalam data input.
num_neurons	Sama dengan input_size, menunjukkan jumlah neuron di lapisan input (satu neuron untuk setiap fitur).
layer_name	Nama lapisan, default-nya adalah "Input Layer".
nodes	Placeholder untuk menyimpan node (jika diperlukan di masa depan).
alpha	Tidak digunakan di sini, tetapi relevan untuk kompatibilitas dengan lapisan lain.
Fungsi	Deskripsi

```

1  class InputLayer:
2      def __init__(self, input_size, layer_name="Input Layer"):
3          self.input_size = input_size
4          self.num_neurons = input_size
5          self.layer_name = layer_name
6          self.nodes = []
7          self.alpha = None
8
9      def forward(self, x):
10         return x
11
12     def backward(self, lr, delta):
13         return delta

```

Fungsi forward() menerima input x (data fitur) dan mengembalikannya langsung tanpa perubahan karena lapisan ini hanya bertugas meneruskan data ke lapisan berikutnya.

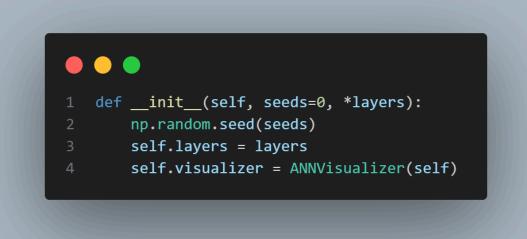
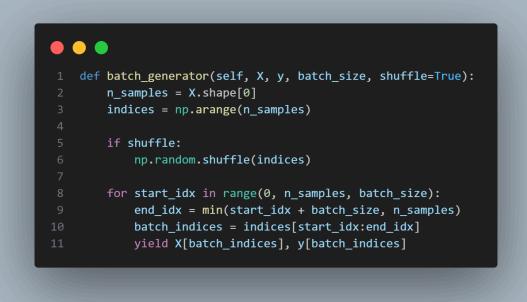
Fungsi backward() menerima gradien error (delta) dari lapisan berikutnya dan mengembalikannya tanpa perubahan karena lapisan input tidak memiliki parameter (seperti bobot atau bias) yang perlu diperbarui.

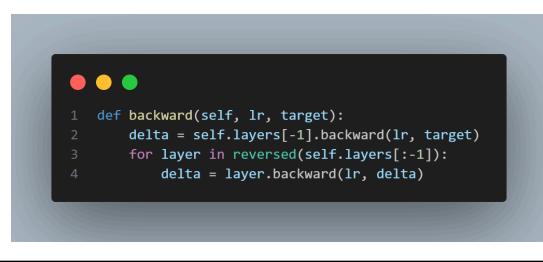
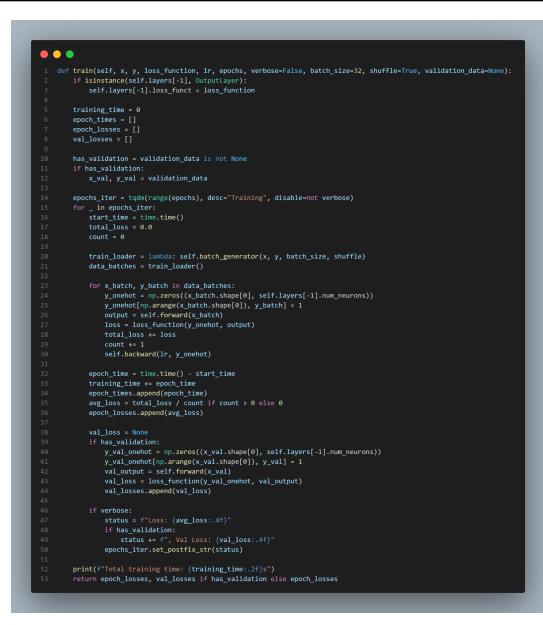
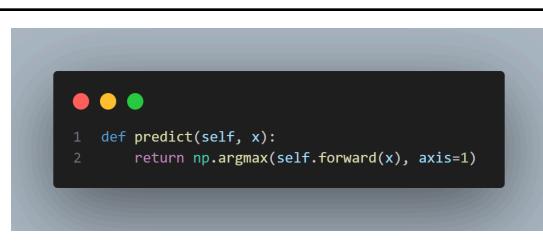
Class OutputLayer (OutputLayer.py)

Class Output Layer adalah kelas yang merepresentasikan layer terakhir (output layer) pada neural networknya. Kelas ini merupakan anak dari kelas Layer dan hanya berbeda pada fungsi backward, dan tidak memiliki fungsi forward.

Atribut	Deskripsi
Sama Seperti kelas Layer	Sama seperti kelas Layer
Fungsi	Deskripsi
<pre> 1 def backward(self, lr, target): 2 delta = self.derivative_loss(self.output, target) 3 4 self.grad_weights = np.dot(delta.T, self.last_input) / self.last_input.shape[0] 5 grad_b = np.mean(delta, axis=0) 6 7 self.weights -= lr * self.grad_weights 8 self.biases -= lr * grad_b 9 10 delta_prev = np.dot(delta, self.weights) </pre>	Fungsi ini melakukan backward propagation di output layer dengan menghitung error lokal (delta) menggunakan turunan fungsi loss. Lalu, menghitung gradien bobot dan bias. Kemudian, menambahkan regularisasi untuk mencegah overfitting, serta memperbarui bobot dan bias menggunakan optimizer. Fungsi ini akan mengembalikan error (delta_prev) untuk diteruskan ke lapisan sebelumnya.

Class ArtificialNeuralNetwork (ArtificialNeuralNetwork.py)

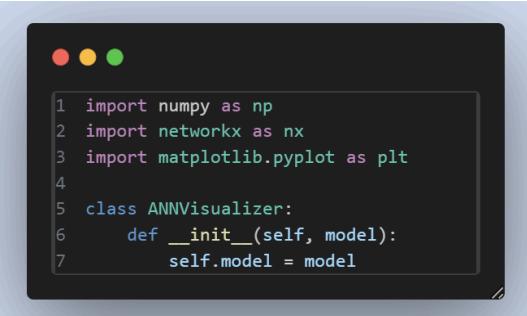
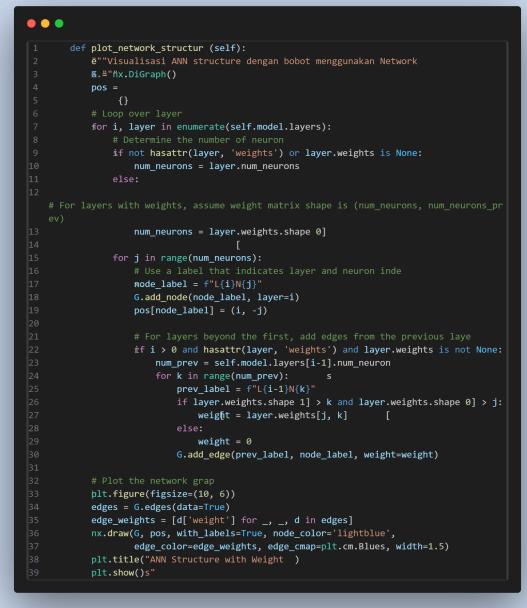
General Class Description here	
Atribut	Deskripsi
seeds	Variabel yang menyimpan seed(random state) yang akan digunakan.
self.layers	Variabel yang menyimpan layer-layer dari model tersebut
self.visualizer	Variabel yang menyimpan suatu instance dari kelas ANNVisualizer.
Fungsi	Deskripsi
 <pre> 1 def __init__(self, seeds=0, *layers): 2 np.random.seed(seeds) 3 self.layers = layers 4 self.visualizer = ANNVisualizer(self) </pre>	Fungsi konstruktor dari kelas ArtificialNeuralNetwork
 <pre> 1 def batch_generator(self, X, y, batch_size, shuffle=True): 2 n_samples = X.shape[0] 3 indices = np.arange(n_samples) 4 5 if shuffle: 6 np.random.shuffle(indices) 7 8 for start_idx in range(0, n_samples, batch_size): 9 end_idx = min(start_idx + batch_size, n_samples) 10 batch_indices = indices[start_idx:end_idx] 11 yield X[batch_indices], y[batch_indices] </pre>	Fungsi untuk menghasilkan mini-batches dari data (X, y) selama proses training untuk digunakan dalam proses mini-batch gradient descent agar training lebih efisien dan cepat dibandingkan full-batch.
 <pre> 1 def forward(self, x): 2 for layer in self.layers: 3 x = layer.forward(x) 4 5 return x </pre>	Fungsi yang memulai forward propagation pada setiap layer yang terdapat pada Atribut self.layers.

 <pre> 1 def train(self, x, y, loss_function, lr, epochs, verbose=False, batch_size=32, shuffle=True, validation_data=None): 2 if isinstance(self.layers[-1], OutputLayer): 3 self.layers[-1].loss_func = loss_function 4 5 training_time = 0 6 epoch_times = [] 7 epoch_losses = [] 8 val_losses = [] 9 10 has_validation = validation_data is not None 11 if has_validation: 12 x_val, y_val = validation_data 13 14 epochs_iter = tqdm(range(epochs), desc="Training", disable=not verbose) 15 for _ in epochs_iter: 16 start_time = time.time() 17 total_loss = 0.0 18 count = 0 19 20 train_loader = lambda: self.batch_generator(x, y, batch_size, shuffle) 21 data_batches = train_loader() 22 23 for x_batch, y_batch in data_batches: 24 y_onehot = np.zeros(x_batch.shape[0], self.layers[-1].num_neurons) 25 y_onehot[np.arange(x_batch.shape[0]), y_batch] = 1 26 neurons = self.feedforward(x_batch) 27 loss = loss_function(y_onehot, neurons) 28 total_loss += loss 29 count += 1 30 self.backward(lr, y_onehot) 31 32 epoch_time = time.time() - start_time 33 training_time += epoch_time 34 epoch_times.append(epoch_time) 35 avg_loss = total_loss / count if count > 0 else 0 36 epoch_losses.append(avg_loss) 37 38 val_loss = None 39 if has_validation: 40 y_val_onehot = np.zeros(x_val.shape[0], self.layers[-1].num_neurons) 41 y_val_onehot[np.arange(x_val.shape[0]), y_val] = 1 42 val_neurons = self.feedforward(x_val) 43 val_loss = loss_function(y_val_onehot, val_neurons) 44 val_losses.append(val_loss) 45 46 if verbose: 47 status = f"loss: {avg_loss:.4f}" 48 if has_validation: 49 status += f", val loss: {val_loss:.4f}" 50 epochs_iter.set_postfix_str(status) 51 52 print(f"Total training time: {training_time:.2f}") 53 return epoch_losses, val_losses if has_validation else epoch_losses </pre>	<p>Fungsi train melatih jaringan dengan optimasi berbasis gradient descent, menggunakan mini-batch dan opsi validasi jika data validasi tersedia. Ini juga mencatat waktu pelatihan dan loss setiap epoch untuk pemantauan performa model.</p>
 <pre> 1 def predict(self, x): 2 return np.argmax(self.forward(x), axis=1) </pre>	<p>Fungsi predict mengambil input dan mengembalikan label kelas yang diprediksi berdasarkan nilai tertinggi di output jaringan.</p>
 <pre> 1 def evaluate(self, x, y): 2 y_pred = self.predict(x) 3 accuracy = np.mean(y_pred == y) 4 return accuracy </pre>	<p>Fungsi evaluate menghitung akurasi model dengan membandingkan prediksi model dengan label sebenarnya dalam data uji.</p>

<pre> 1 def save(self, filename): 2 if not filename.endswith(".pkl"): 3 filename += ".pkl" 4 model_dir = "models" 5 if not os.path.exists(model_dir): 6 os.makedirs(model_dir) 7 filepath = os.path.join(model_dir, filename) 8 with open(filepath, 'wb') as f: 9 pickle.dump(self.layers, f) 10 print(f"Model saved to {filepath}") </pre>	<p>Fungsi untuk menyimpan sebuah model dalam bentuk file .pkl menggunakan library pickle</p>
<pre> 1 def load(self,filename): 2 filename = "models/" + filename 3 with open(filename, 'rb') as f: 4 self.layers = pickle.load(f) 5 for layer in self.layers: 6 print(layer) 7 print(layer.alpha) 8 print(f"Model loaded from {filename}") </pre>	<p>Fungsi untuk memuat sebuah model yang sudah disimpan ke dalam file .pkl, menggunakan pickle</p>
<pre> 1 def visualize_structure(self): 2 return self.visualizer.plot_network_structure() </pre>	<p>Fungsi untuk menampilkan model berupa struktur jaringan beserta bobot dan gradien bobot tiap neuron dalam bentuk graf.</p>
<pre> 1 def visualize_weight_table(self): 2 self.visualizer.plot_layer_tables() </pre>	<p>Fungsi yang menampilkan rata-rata bobot dan gradien untuk setiap neuron di setiap lapisan menggunakan fungsi dari kelas ANNVisualizer</p>
<pre> 1 def visualize_weight_distribution(self, layer_indices): 2 self.visualizer.plot_weight_distribution(layer_indices) </pre>	<p>Fungsi untuk menampilkan histogram distribusi bobot untuk lapisan tertentu. Menggunakan fungsi dari kelas ANNVisualizer</p>
<pre> 1 def visualize_gradient_distribution(self, layer_indices): 2 self.visualizer.plot_gradient_distribution(layer_indices) </pre>	<p>Fungsi untuk menampilkan histogram distribusi gradien bobot untuk lapisan tertentu. Menggunakan fungsi dari kelas ANNVisualizer</p>

Class ANNVisualizer (ANNVisualizer.py)

General Class Description here

Atribut	Deskripsi
self.model	Digunakan untuk menyimpan model jaringan saraf tiruan (ANN) yang akan divisualisasikan.
Fungsi	Deskripsi
	Fungsi konstruktor untuk menginisialisasi objek ANNVisualizer dengan menyimpan model ke dalam atribut self.model.
	Fungsi plot_network_structure() digunakan untuk membuat visualisasi struktur jaringan saraf menggunakan NetworkX. Setiap neuron direpresentasikan sebagai node, dan bobot antar neuron direpresentasikan sebagai edge. Pertama-tama fungsi ini akan membuat graf directed (DiGraph) menggunakan NetworkX. Lalu, node untuk setiap neuron di setiap lapisan ditambahkan. Kemudian, edge antar neuron berdasarkan bobot antar lapisan juga ditambahkan. Terakhir, Matplotlib digunakan untuk menggambar graf.

```

1  def plot_layer_table (self):
2      """Plot tabel average bobot dan gradient untuk setiap lapisan
3      #for i, layer in enumerate(self.model.layers):
4      #    if i == 0:
5      #        continue # Skip input lapisan
6      #    r
7
8      # Determine how many neurons this layer has
9      if not hasattr(layer, 'weights') or layer.weights is None:
10         num_neurons = layer.num_neurons
11     else:
12         num_neurons = layer.weights.shape[0]
13
14     # Build table data
15     table_data = []
16     for j in range(num_neurons):
17         neuron_label = f"Neuron {j}"
18
19         # Average weight
20         if hasattr(layer, 'weights') and layer.weights is not None:
21             avg_weight = np.mean(layer.weights[j, :])
22         else:
23             avg_weight = "N/A"
24
25         # Average grad weight
26         if hasattr(layer, 'grad_weight') and layer.grad_weight is not None:
27             avg_grad = np.mean(layer.grad_weight[j, :])
28         else:
29             avg_grad = "N/A"
30
31         avg_weight_st = f"(avg_weight:{.4f})" if isinstance(avg_weight, float)
32         avg_grad_st = f"(avg_grad:{.4f})" if isinstance(avg_grad, float) else
33         avg_grad
34
35         table_data.append([neuron_label, avg_weight_st, avg_grad_st])
36
37     base_width = 6
38     base_height = 2
39     row_height_facto = 0.25 # additional height per neuron
40     r = 0
41     fig_height = base_height + (num_neurons * row_height_facto )
42     fig = plt.subplots(figsize=(base_width, fig_height))
43
44     ax.axis('off')
45     ax.set_title(f"Layer {i} Weights Summary", fontsize=10, pad=10, y=1.02)
46
47     col_labels = ["Neuron", "Avg Weight", "Avg Grad Weight"]
48     col1_labels = ["", "t", "t"]
49     col2_labels = ["", "t", "t"]
50
51     # Styling
52     tbl = ax.table(
53         cellText=table_data,
54         colLabels=col_labels,
55         loc="center",
56         cellLoc="center",
57         edges="closed"
58     )
59
60     # Styling
61     #tbl.auto_set_font_size(False)
62     #tbl.set_fontsize(9)
63     for (row, col), cell in tbl.get_celld().items():
64         cell.set_edgecolor('black')
65         cell.set_linewidth(1)
66
67     for col_idx in range(len(col_labels)):
68         h = col1_labels if col_idx == 0 else col2_labels
69         tbl.auto_set_column_width (col_idx)
70
71     plt.tight_layout()
72     plt.show()

```

Fungsi `plot_layer_table()` digunakan untuk membuat tabel yang menampilkan rata-rata bobot dan gradien untuk setiap neuron di setiap lapisan.

Pertama-tama fungsi ini akan me-skip input layer karena input layer tidak memiliki bobot. Setelah itu, rata-rata bobot dan gradien untuk setiap neuron dihitung dan dilakukan plotting dalam bentuk tabel menggunakan Matplotlib untuk menampilkan datanya.

```

1  def plot_weight_distribution(self, layer_indices):
2      """Menampilkan distribusi bobot dari lapisan tertentu."""
3      for idx in layer_indices:
4          if idx >= len(self.model.layers):
5              print(f"Layer {idx} tidak ditemukan.")
6              continue
7
8          weights = self.model.layers[idx].weights.flatten()
9          plt.figure(figsize=(6, 4))
10         plt.hist(weights, bins=30, alpha=0.7, color='blue', edgecolor='black')
11         plt.title(f"Distribusi Bobot - Layer {idx}")
12         plt.xlabel('Weight Value')
13         plt.ylabel('Frequency')
14         plt.show()

```

Fungsi `plot_weight_distribution(layer_indices)` digunakan untuk menampilkan histogram distribusi bobot untuk lapisan tertentu. Fungsi ini menerima satu parameter yaitu 'layer_indices' yang berupa daftar indeks lapisan yang ingin divisualisasikan.

Pertama-tama fungsi akan mengecek apakah indeks lapisan valid. Kemudian, bobot dari lapisan yang dipilih diambil dan dilakukan plotting histogram distribusi bobot dengan menggunakan Matplotlib.

```

1  def plot_gradient_distribution(self, layer_indices):
2      """Menampilkan distribusi gradien bobot dari layer tertentu."""
3      for idx in layer_indices:
4          if idx >= len(self.model.layers):
5              print(f"Layer {idx} tidak ditemukan.")
6              continue
7
8          gradients = self.model.layers[idx].grad_weights.flatten()
9          plt.figure(figsize=(6, 4))
10         plt.hist(gradients, bins=30, alpha=0.7, color='red', edgecolor='black')
11         plt.title(f'Distribusi Gradien Bobot - Layer {idx}')
12         plt.xlabel('Gradient Value')
13         plt.ylabel('Frequency')
14         plt.show()

```

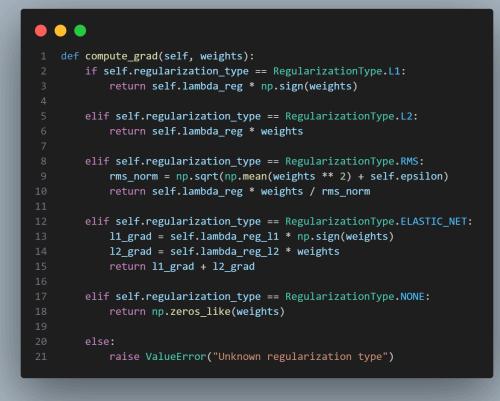
Fungsi
plot_gradient_distribution(layer_indices) digunakan untuk menampilkan histogram distribusi gradien bobot untuk lapisan tertentu. Fungsi ini menerima satu parameter yaitu 'layer_indices' yang berupa daftar indeks lapisan yang ingin divisualisasikan.

Pertama-tama fungsi akan mengecek apakah indeks lapisan valid. Kemudian, gradien dari lapisan yang dipilih diambil dan dilakukan plotting histogram distribusi gradien dengan menggunakan Matplotlib.

Class Regularizer (Regularizer.py)

General Class Description here

Atribut	Deskripsi
self.regularization_type	Menyimpan tipe regularisasi yang digunakan (dari enum RegularizationType) untuk menentukan metode penalti yang diterapkan pada bobot untuk mencegah overfitting
self.lambda_reg	Parameter pengali yang mengontrol kekuatan regularisasi (nilai default: 0.01). Digunakan untuk regularisasi L1 dan L2
self.lambda_reg_l1	Parameter pengali khusus untuk komponen L1 dalam Elastic Net. Mengontrol seberapa kuat penalti L1 (absolute value) diterapkan
self.lambda_reg_l2	Parameter pengali khusus untuk komponen L2 dalam Elastic Net. Mengontrol seberapa kuat penalti L2 (squared) diterapkan
self.epsilon	Nilai kecil untuk mencegah pembagian dengan nol
Fungsi	Deskripsi

 <pre> 1 def compute_grad(self, weights): 2 if self.regularization_type == RegularizationType.L1: 3 return self.lambda_reg * np.sign(weights) 4 5 elif self.regularization_type == RegularizationType.L2: 6 return self.lambda_reg * weights 7 8 elif self.regularization_type == RegularizationType.RMS: 9 rms_norm = np.sqrt(np.mean(weights ** 2) + self.epsilon) 10 return self.lambda_reg * weights / rms_norm 11 12 elif self.regularization_type == RegularizationType.ELASTIC_NET: 13 l1_grad = self.lambda_reg_l1 * np.sign(weights) 14 l2_grad = self.lambda_reg_l2 * weights 15 return l1_grad + l2_grad 16 17 elif self.regularization_type == RegularizationType.NONE: 18 return np.zeros_like(weights) 19 20 else: 21 raise ValueError("Unknown regularization type") </pre>	Fungsi compute_grad menghitung gradien regularisasi berdasarkan jenis regularisasi yang dipilih. Untuk L1, ia mengembalikan lambda dikalikan tanda bobot untuk mendorong sparsity; untuk L2, ia mengembalikan lambda dikalikan bobot untuk mengecilkan nilai bobot; sedangkan Elastic Net menggabungkan keduanya dengan masing-masing koefisien. Jika tidak ada regularisasi, fungsi mengembalikan array nol, dan akan menampilkan error jika tipe regularisasi tidak dikenali. Fungsi ini digunakan untuk menambahkan kontribusi regularisasi ke dalam proses pembaruan bobot model saat pelatihan.
--	--

2.1.2 Forward Propagation

Forward propagation adalah tahap pertama dalam proses perhitungan *feed forward neural network*, di mana data *input* diproses melalui setiap *layer* sampai menghasilkan output akhir berupa *output layer*. Proses ini sangat penting karena menentukan bagaimana informasi berupa bobot dan bias mengalir melalui *network* dan berpengaruh terhadap performa model dalam melakukan prediksi. Dalam *forward propagation*, *input* yang diberikan akan dikalikan dengan bobot masing-masing neuron, ditambahkan bias, dan kemudian dilanjutkan ke *layer* berikutnya melalui fungsi aktivasi untuk mendapatkan nilai output dari neuron tersebut. Hasil perhitungan ini kemudian diteruskan ke *layer* berikutnya hingga mencapai *layer output*. Berikut adalah hasil implementasi dari proses *forward propagation*.

2.1.3 Backward Propagation dan Weight Update

a. Backward Propagation

Backward propagation adalah mekanisme utama dalam pelatihan jaringan saraf untuk memperbaiki bobot dan bias berdasarkan kesalahan prediksi. Proses ini dimulai dari output layer dan bergerak mundur ke setiap hidden layer, menghitung gradien dan memperbarui parameter menggunakan algoritma gradient descent.

1. Output Layer: Menghitung Delta dari Loss Function

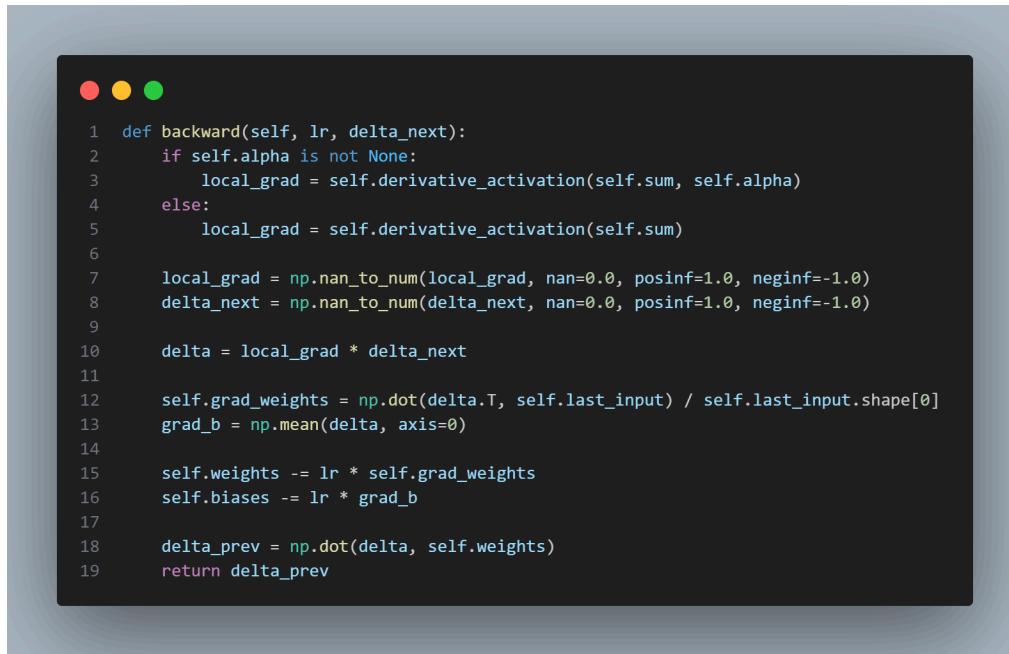
```
1  def backward(self, lr, target):
2      delta = self.derivative_loss(self.output, target)
3
4      self.grad_weights = np.dot(delta.T, self.last_input) / self.last_input.shape[0]
5      grad_b = np.mean(delta, axis=0)
6
7      self.weights -= lr * self.grad_weights
8      self.biases -= lr * grad_b
9
10     delta_prev = np.dot(delta, self.weights)
11     return delta_prev
```

Pada OutputLayer, backward propagation dimulai dengan menghitung turunan fungsi loss terhadap output prediksi. Ini dilakukan menggunakan fungsi turunan loss yang disesuaikan secara otomatis berdasarkan jenis loss function yang dipilih, seperti mean squared error atau categorical cross-entropy.

Nilai delta ini merepresentasikan sensitivitas error terhadap output layer, dan akan digunakan untuk menghitung gradien parameter

Bobot dan bias diperbarui dengan mengurangkan gradien yang telah dikalikan learning rate (lr). Hasil delta dari output layer ini akan dikirim ke layer sebelumnya untuk langkah backward berikutnya.

2. Hidden Layer: Menggabungkan Turunan Aktivasi dan Delta



```
1 def backward(self, lr, delta_next):
2     if self.alpha is not None:
3         local_grad = self.derivative_activation(self.sum, self.alpha)
4     else:
5         local_grad = self.derivative_activation(self.sum)
6
7     local_grad = np.nan_to_num(local_grad, nan=0.0, posinf=1.0, neginf=-1.0)
8     delta_next = np.nan_to_num(delta_next, nan=0.0, posinf=1.0, neginf=-1.0)
9
10    delta = local_grad * delta_next
11
12    self.grad_weights = np.dot(delta.T, self.last_input) / self.last_input.shape[0]
13    grad_b = np.mean(delta, axis=0)
14
15    self.weights -= lr * self.grad_weights
16    self.biases -= lr * grad_b
17
18    delta_prev = np.dot(delta, self.weights)
19    return delta_prev
```

Setiap Layer melakukan propagasi mundur dengan cara:

1. Menghitung turunan fungsi aktivasi terhadap input sum.
2. Mengalikan hasilnya dengan delta dari layer berikutnya.
3. Menghitung gradien terhadap bobot dan bias.
4. Menambahkan penalti regularisasi jika digunakan.
5. Memperbarui parameter.

Proses ini juga menangani kestabilan numerik dengan `np.nan_to_num` dan kliping nilai ekstrem, agar jaringan tetap stabil selama pelatihan.

3. Regularisasi dan RMSProp

Jika regularisasi diterapkan, maka gradien bobot akan ditambahkan penalti terhadap nilai bobot yang terlalu besar. Regularizer ini modular dan mendukung tipe-tipe seperti L1 atau L2.

Untuk stabilitas aktivasi saat forward pass, backward propagation bisa dikombinasikan dengan RMSProp-style normalization di Layer, yang secara tidak langsung mempengaruhi skala gradien karena normalisasi dilakukan sebelum fungsi aktivasi. Namun, aktualisasi RMSProp dalam backward belum dilakukan penuh (misal: update berdasarkan cache gradien), karena implementasinya lebih berfokus pada preprocessing input aktivasi.

4. Alur Lengkap dalam Jaringan

Dalam `ArtificialNeuralNetwork`, backward propagation dilakukan dari layer output ke input:

Proses ini akan dilakukan berulang kali di setiap epoch, hingga model mampu menghasilkan prediksi yang akurat dengan error seminimal mungkin.



```
1  def backward(self, lr, target):
2      delta = self.layers[-1].backward(lr, target)
3      for layer in reversed(self.layers[-1]):
4          delta = layer.backward(lr, delta)
5
6  def train(self, x, y, loss_function, lr, epochs, verbose=False, batch_size=32, shuffle=True, validation_data=None):
7      if isinstance(self.layers[-1], OutputLayer):
8          self.layers[-1].loss_funct = loss_function
9
10     training_time = 0
11     epoch_times = []
12     epoch_losses = []
13     val_losses = []
14
15     has_validation = validation_data is not None
16     if has_validation:
17         x_val, y_val = validation_data
18
19     epochs_iter = tqdm(range(epochs), desc="Training", disable=not verbose)
20     for _ in epochs_iter:
21         start_time = time.time()
22         total_loss = 0.0
23         count = 0
24
25         train_loader = lambda: self.batch_generator(x, y, batch_size, shuffle)
26         data_batches = train_loader()
27
28         for x_batch, y_batch in data_batches:
29             y_onehot = np.zeros((x_batch.shape[0], self.layers[-1].num_neurons))
30             y_onehot[np.arange(x_batch.shape[0]), y_batch] = 1
31             output = self.forward(x_batch)
32             loss = loss_function(y_onehot, output)
33             total_loss += loss
34             count += 1
35             self.backward(lr, y_onehot)
36
37             epoch_time = time.time() - start_time
38             training_time += epoch_time
39             epoch_times.append(epoch_time)
40             avg_loss = total_loss / count if count > 0 else 0
41             epoch_losses.append(avg_loss)
42
43             val_loss = None
44             if has_validation:
45                 y_val_onehot = np.zeros((x_val.shape[0], self.layers[-1].num_neurons))
46                 y_val_onehot[np.arange(x_val.shape[0]), y_val] = 1
47                 val_output = self.forward(x_val)
48                 val_loss = loss_function(y_val_onehot, val_output)
49                 val_losses.append(val_loss)
50
51             if verbose:
52                 status = f"Loss: {avg_loss:.4f}"
53                 if has_validation:
54                     status += f", Val Loss: {val_loss:.4f}"
55                 epochs_iter.set_postfix_str(status)
56
57             print(f"Total training time: {training_time:.2f}s")
58     return epoch_losses, val_losses if has_validation else epoch_losses
```

b. Weight Update

Setelah *backward propagation* menghitung gradien bobot, langkah berikutnya adalah memperbarui bobot menggunakan metode optimasi, seperti *Gradient Descent*. Tujuan dari tahap ini adalah menyesuaikan bobot agar output prediksi semakin mendekati nilai target dengan meminimalkan error. Berikut adalah hasil implementasi untuk *update* bobot.

2.2 Hasil Pengujian

2.2.1 Pengaruh depth dan width

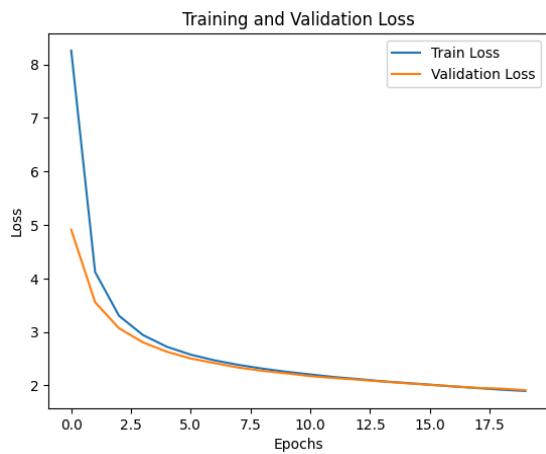
2.2.1.1 Depth

a. Depth = 1 hidden layer dengan width = 20

i. Hasil akhir prediksi

F1 Score: 0.9314411872953958

ii. Grafik loss pelatihannya

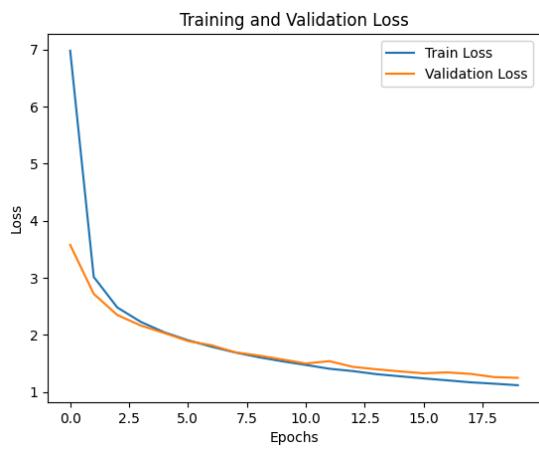


b. Depth = 3 layer dengan width = 20

i. Hasil akhir prediksi

F1 Score: 0.9496421025330465

ii. Grafik loss pelatihannya

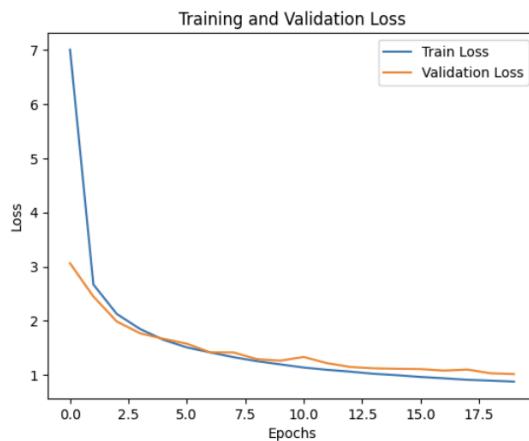


c. Depth = 5 layer dengan width = 20

i. Hasil akhir prediksi

F1 Score: 0.9537950284967625

ii. Grafik loss pelatihannya



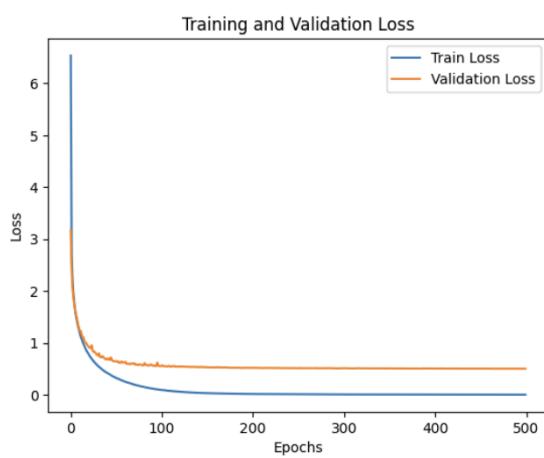
2.2.1.2 Width

a. Width = 64 neuron dengan depth = 3

i. Hasil akhir prediksi

F1 Score: 0.9698560420453983

ii. Grafik loss pelatihan

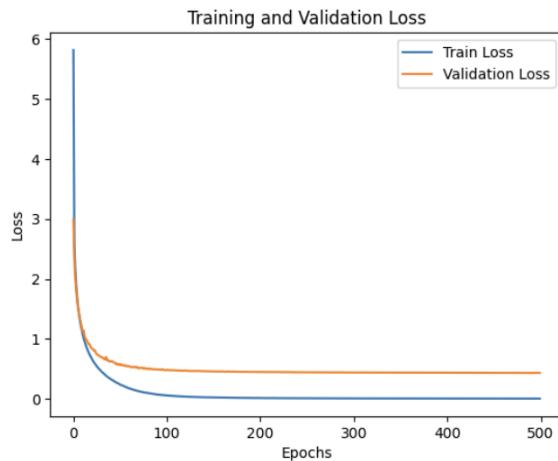


b. Width = 128 neuron dengan depth = 3

i. Hasil akhir prediksi

F1 Score: 0.9743308237271447

ii. Grafik loss pelatihan

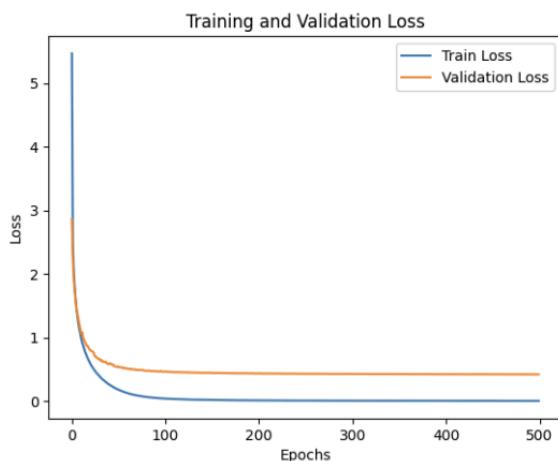


c. Width = 256 neuron dengan depth = 3

i. Hasil akhir prediksi

F1 Score: 0.9754137606912996

ii. Grafik loss pelatihan



2.2.1.3 Hasil Analisis

a. Pengaruh Jumlah Neuron (depth)

Penambahan jumlah hidden layer secara umum meningkatkan performa model hingga titik tertentu. Hal ini terlihat dari peningkatan F1 Score saat depth bertambah dari 1 (F1: 0.9314) menjadi 3 (F1: 0.9496), dan kembali meningkat saat depth menjadi 5 (F1: 0.9538). Hal ini menunjukkan bahwa model dengan kedalaman lebih mampu menangkap kompleksitas data dengan lebih baik.

Selain itu, dari grafik loss, semua model menunjukkan tren konvergensi yang baik, namun model dengan depth lebih tinggi memiliki loss training dan validasi yang lebih rendah secara konsisten.

b. Pengaruh Jumlah Neuron (width)

Berdasarkan hasil observasi diatas, dapat disimpulkan bahwa seiring bertambahnya jumlah neuron dalam setiap hidden layer, F1 Score juga mengalami peningkatan. Hal ini menunjukkan bahwa model dengan lebih banyak neuron memiliki kapasitas lebih tinggi untuk menangkap pola dalam data.

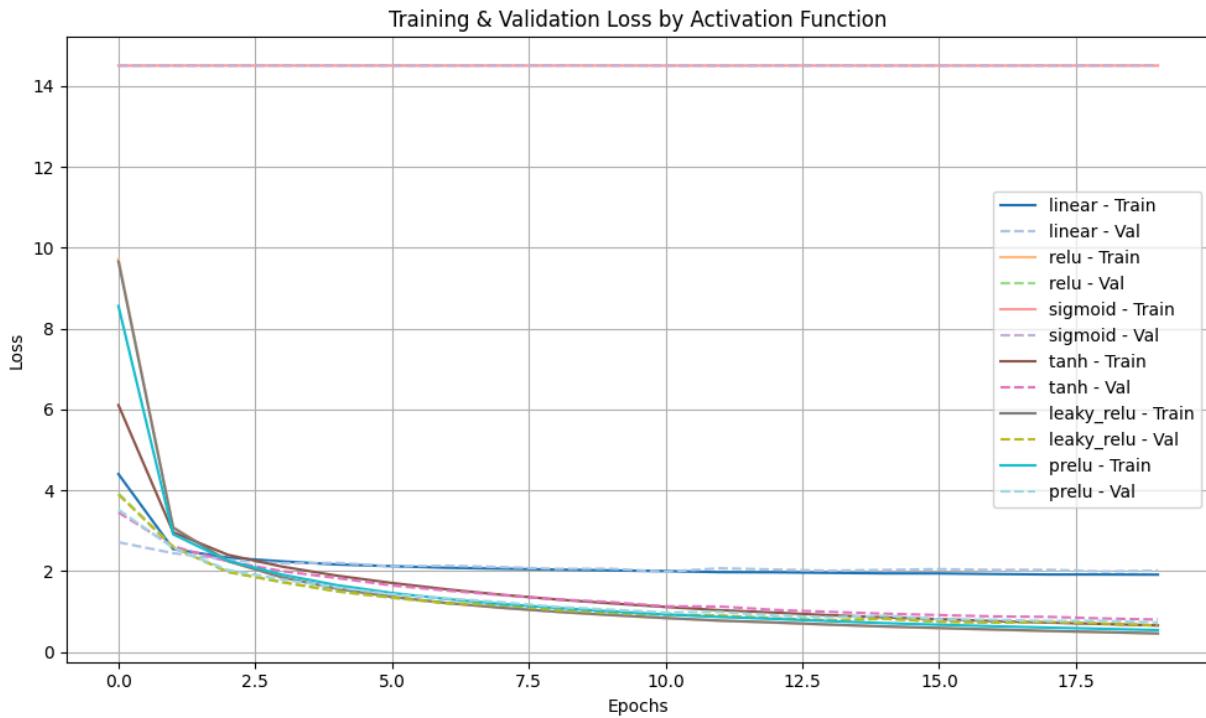
Selain itu, dari grafik loss juga dapat dilihat bahwa semakin banyak neuron, maka train loss cenderung lebih rendah di setiap epochnya, yang menandakan bahwa model lebih mampu menyesuaikan diri dengan data training.

2.2.2 Pengaruh fungsi aktivasi

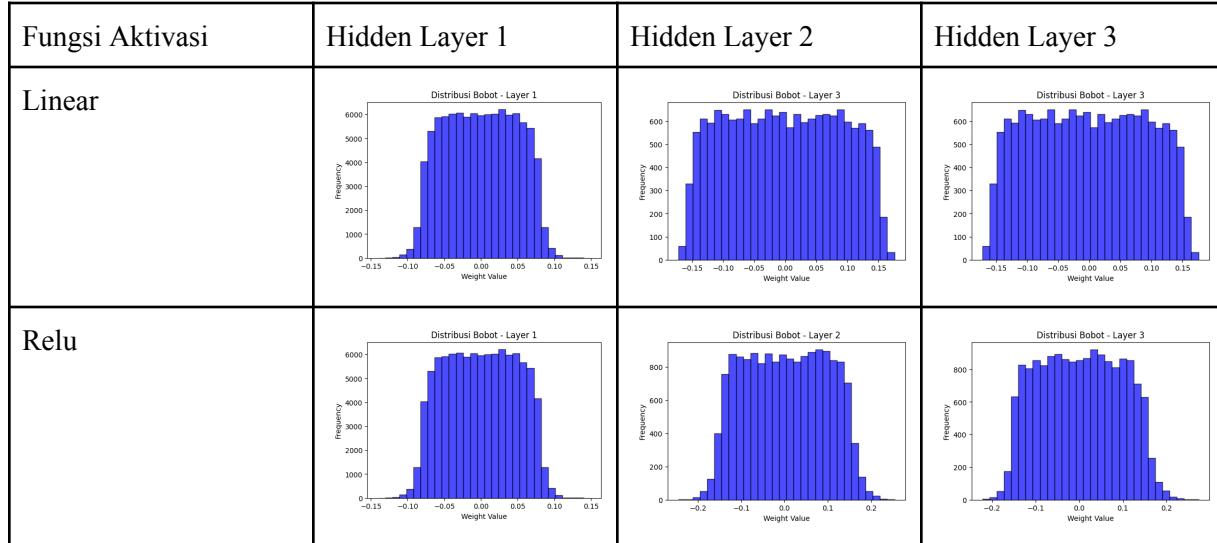
2.2.2.1 F1 Score

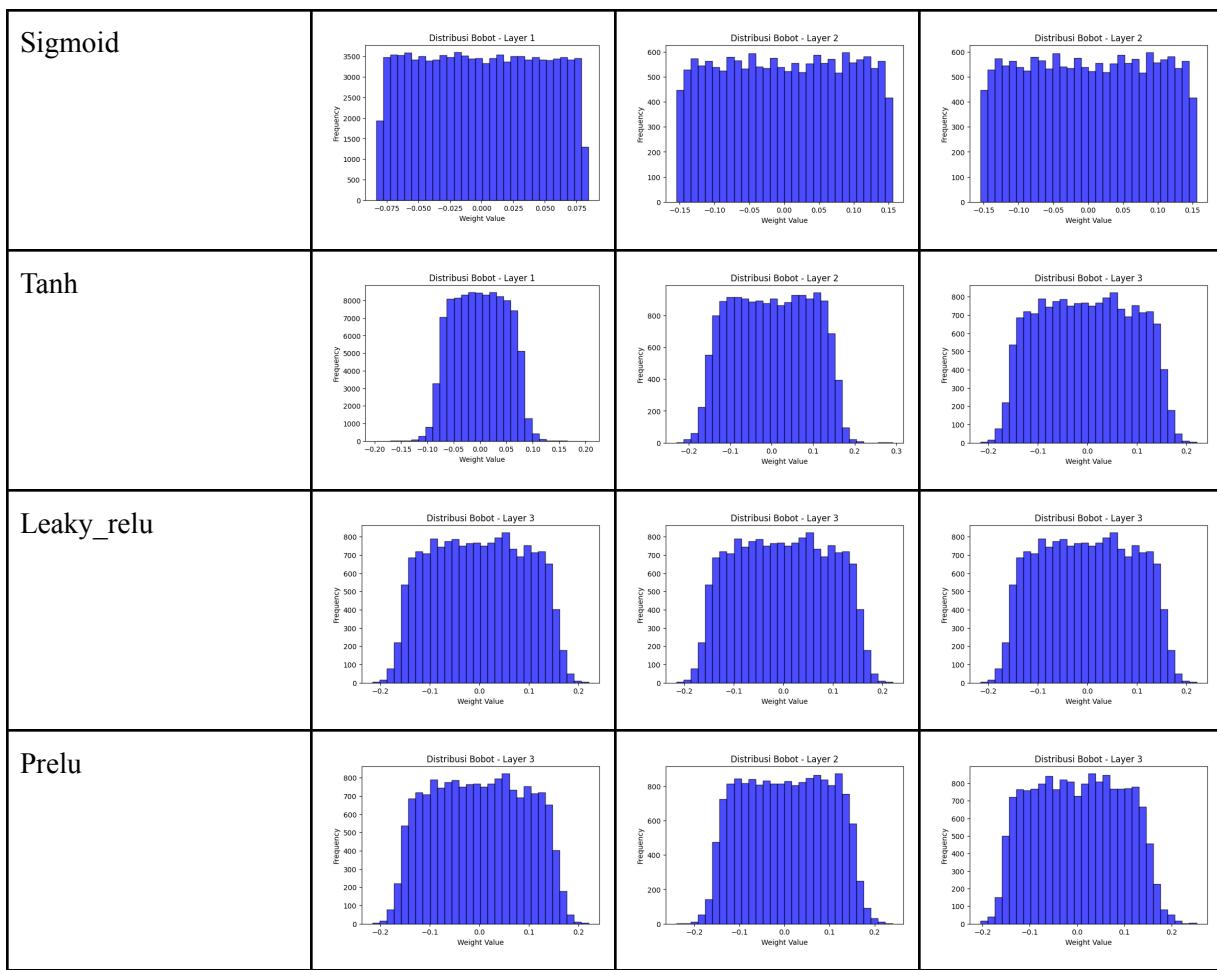
1. linear: 0.9175
2. relu: 0.9708
3. sigmoid: 0.0169
4. tanh: 0.9671
5. leaky_relu: 0.9708
6. prelu: 0.9675

2.2.2.2 Train and Validation Loss

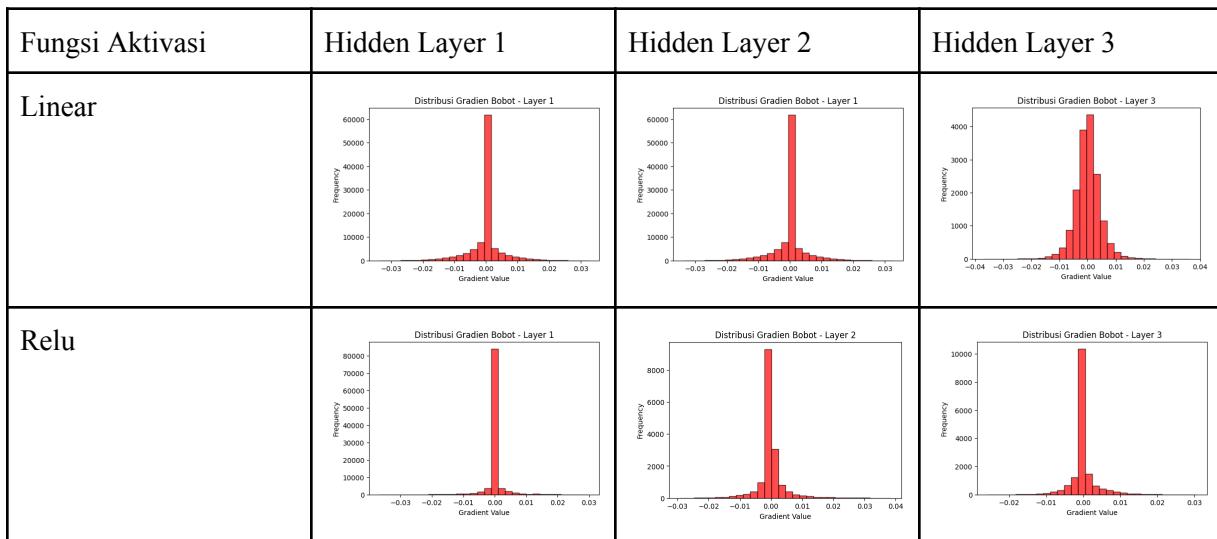


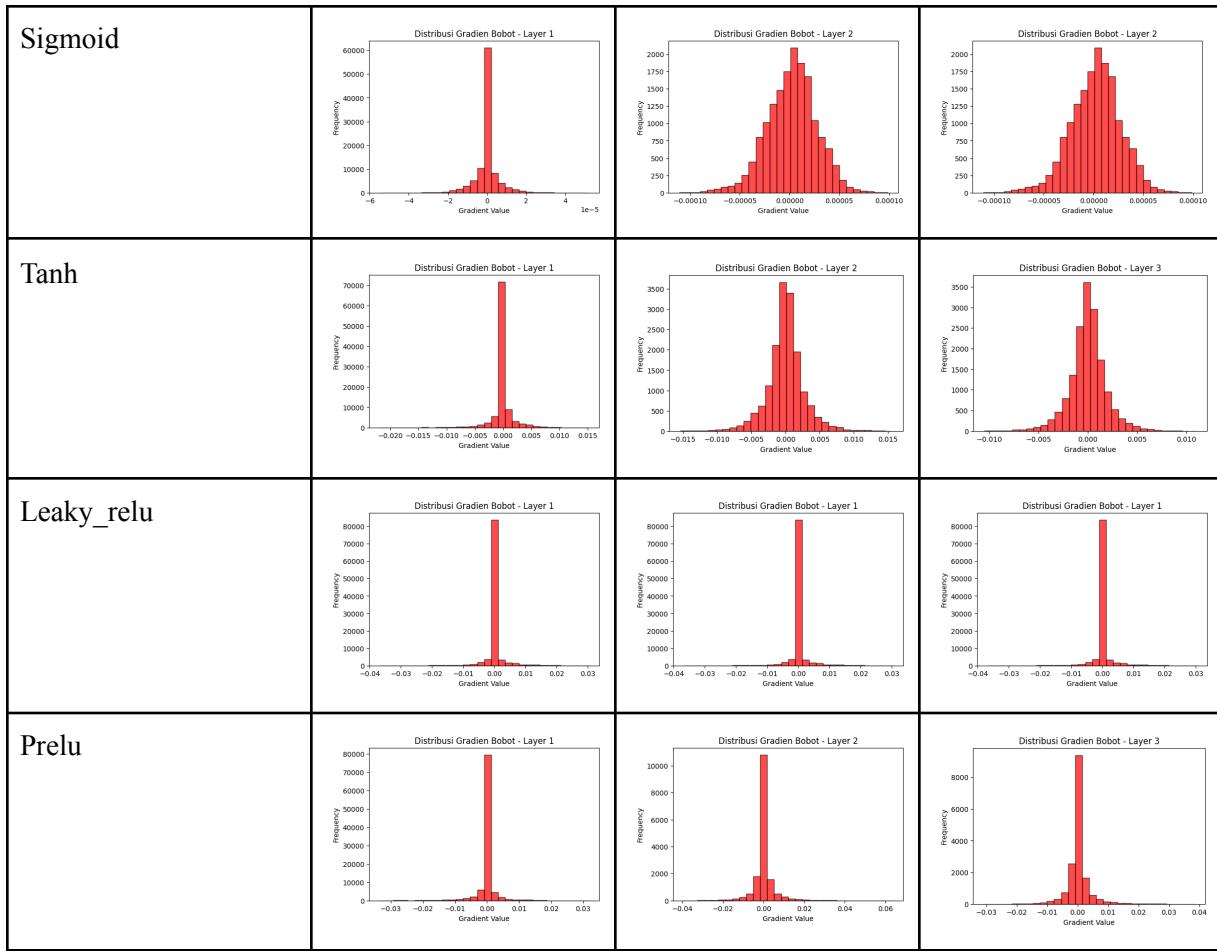
2.2.2.3 Distribusi Bobot





2.2.2.4 Distribusi Gradien





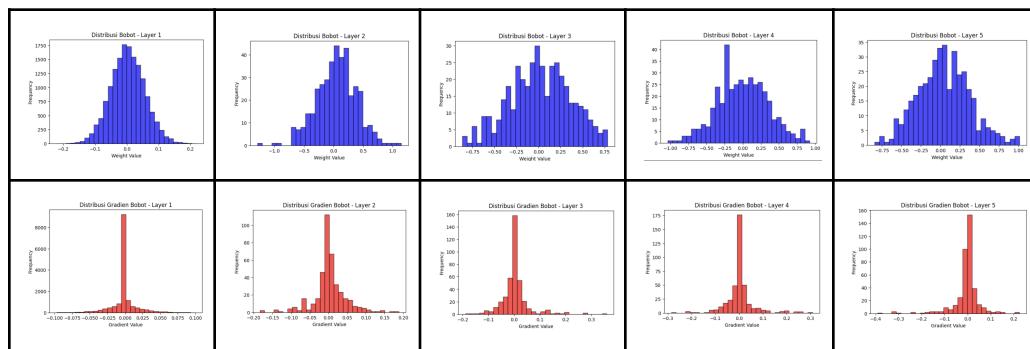
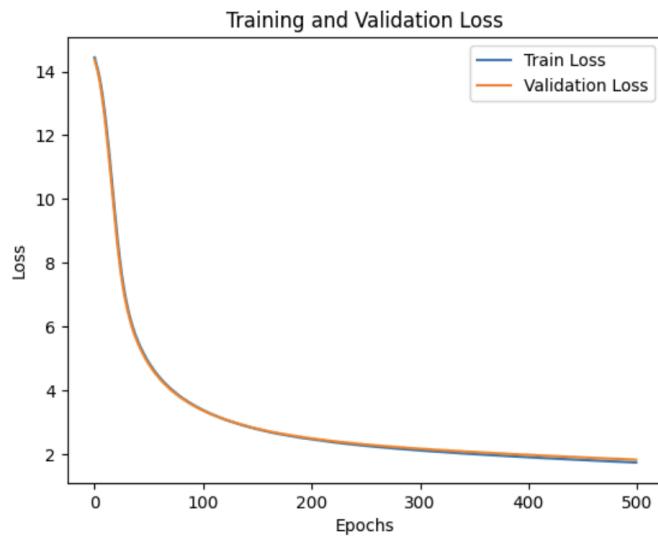
2.2.2.5 Analisis

Aktivasi ReLU dan Leaky ReLU menghasilkan performa tertinggi dengan nilai F1 sebesar 0.9708, menunjukkan kemampuannya dalam menangani vanishing gradient dan mempercepat konvergensi. Tanh dan PReLU juga menunjukkan performa yang sangat baik dengan F1 Score masing-masing 0.9671 dan 0.9675, meskipun sedikit lebih rendah dibanding ReLU. Sementara itu, Linear sebagai fungsi aktivasi menghasilkan F1 Score yang cukup baik (0.9175), namun tidak seoptimal fungsi non-linear. Sebaliknya, Sigmoid memiliki performa yang sangat rendah dengan F1 Score hanya 0.0169, yang kemungkinan besar disebabkan oleh masalah vanishing gradient dan output yang cenderung saturasi, terutama pada jaringan yang lebih dalam.

2.2.3 Pengaruh learning rate

a. Learning rate = 0.0001

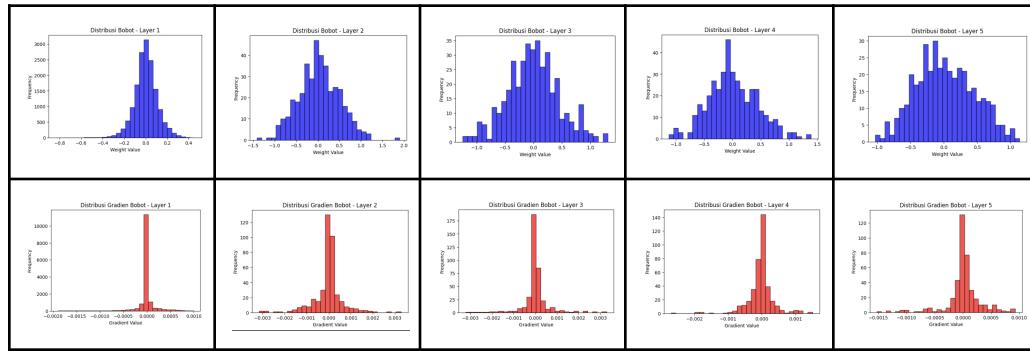
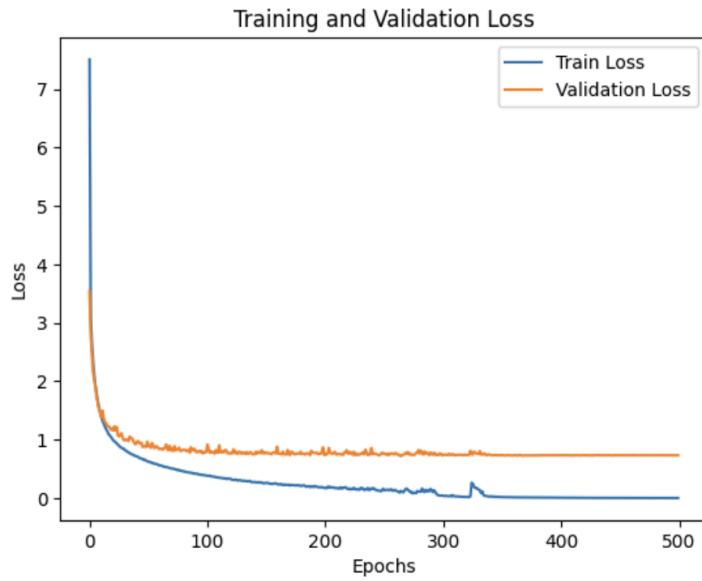
F1 Score: 0.9302532990200199



Dengan learning rate sebesar 0.0001, akurasi yang dihasilkan adalah 93.02%

b. Learning rate = 0.01

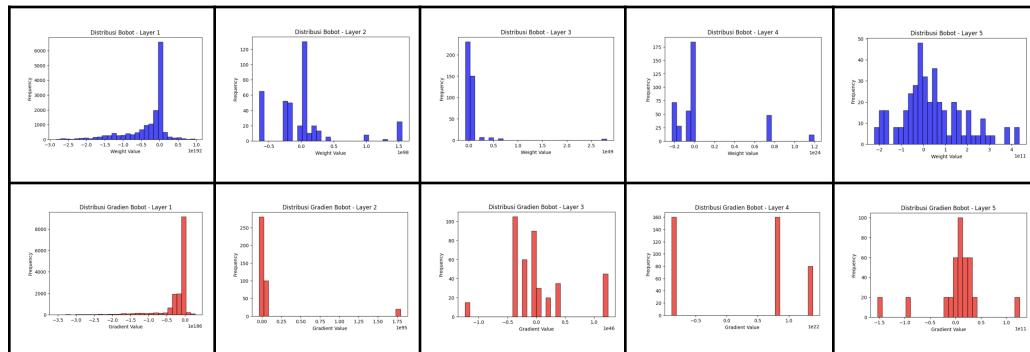
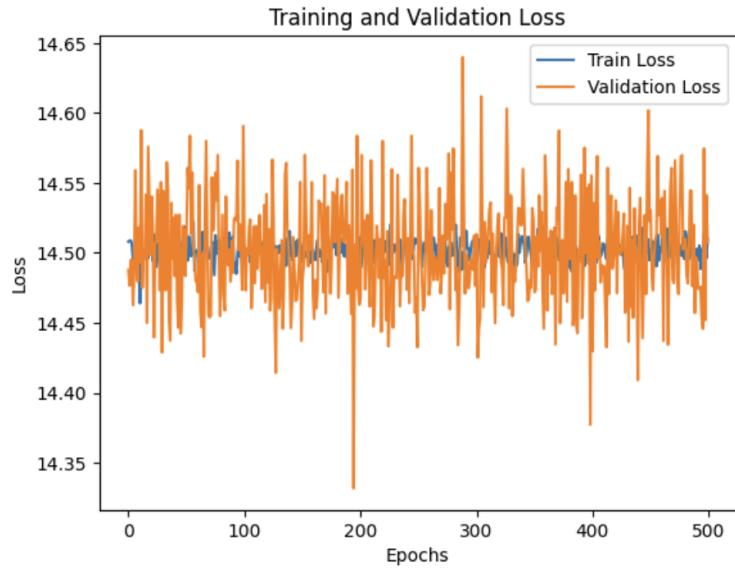
F1 Score: 0.9550249525342206



Dengan learning rate sebesar 0.01, akurasi yang dihasilkan adalah 95.50%

c. Learning rate = 1

F1 Score: 0.017945383615084527



Dengan learning rate sebesar 1, akurasi yang dihasilkan adalah 1.79%

d. Hasil Analisis

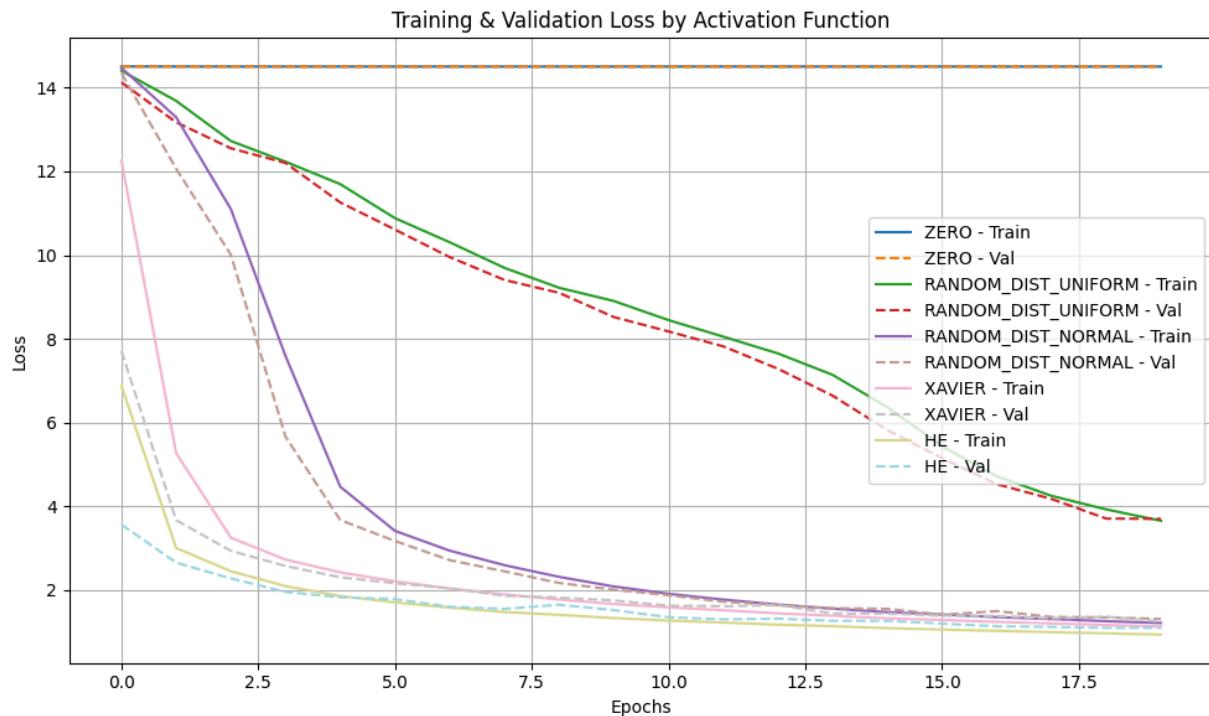
Learning rate 0.01 menghasilkan performa terbaik (F1 Score: 0.955) karena memberikan update bobot yang cukup besar namun stabil, memungkinkan model belajar secara efektif. Learning rate 0.0001 (F1 Score: 0.930) terlalu kecil sehingga proses belajar lambat dan mungkin belum mencapai konvergensi optimal dalam 500 epoch (model belum cukup untuk mengeksplorasi dan menemukan pola terbaik di data). Sementara itu, learning rate 1.0 (F1 Score: 0.017) terlalu besar, menyebabkan model tidak stabil dan gagal belajar.

2.2.4 Pengaruh inisialisasi bobot

2.2.4.1 Hasil F1 Score

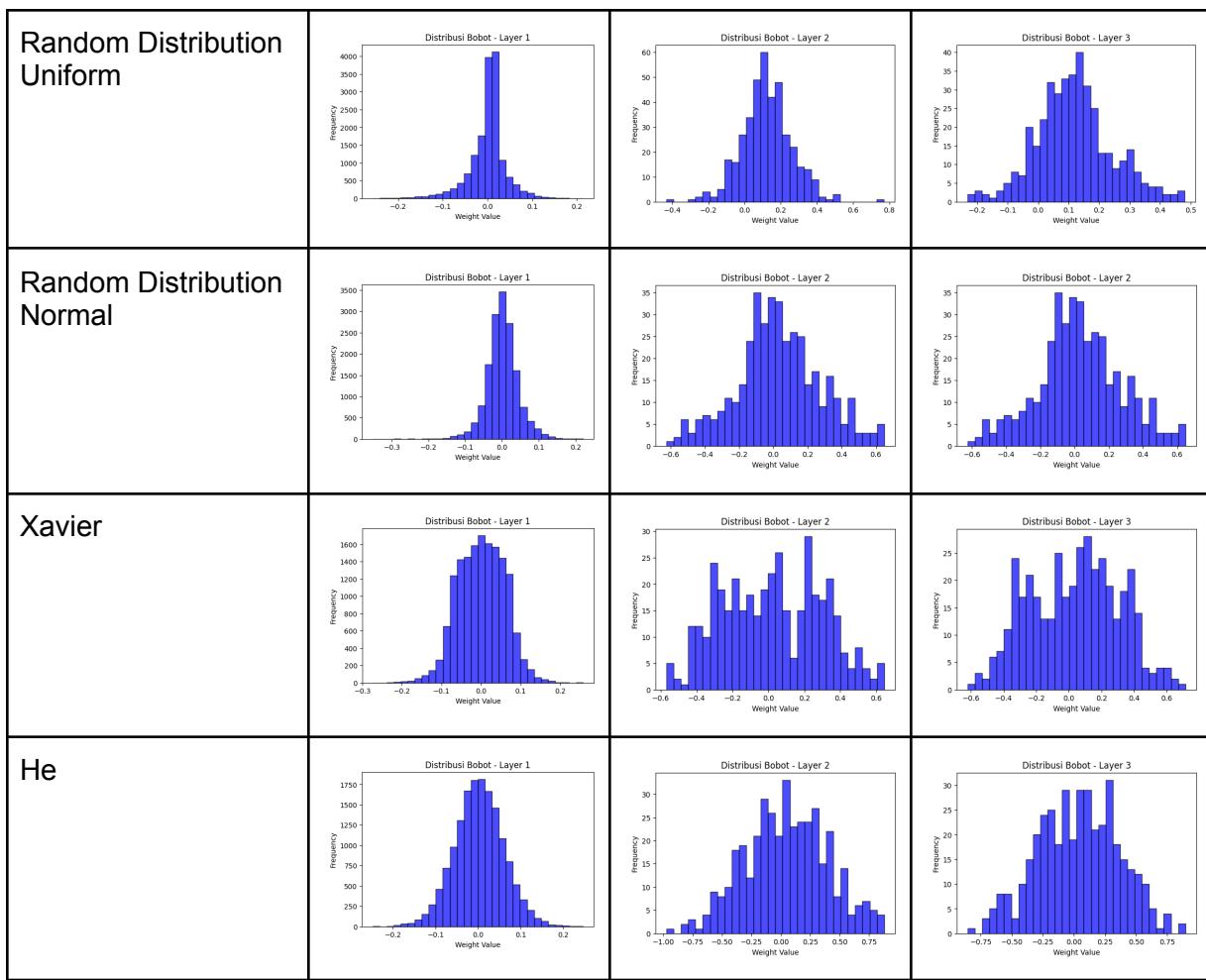
- ZERO : 0.0205
- RANDOM_DIST_UNIFORM : 0.8644
- RANDOM_DIST_NORMAL : 0.9452
- XAVIER : 0.9462
- HE : 0.9505

2.2.4.2 Train Loss dan Validation Loss

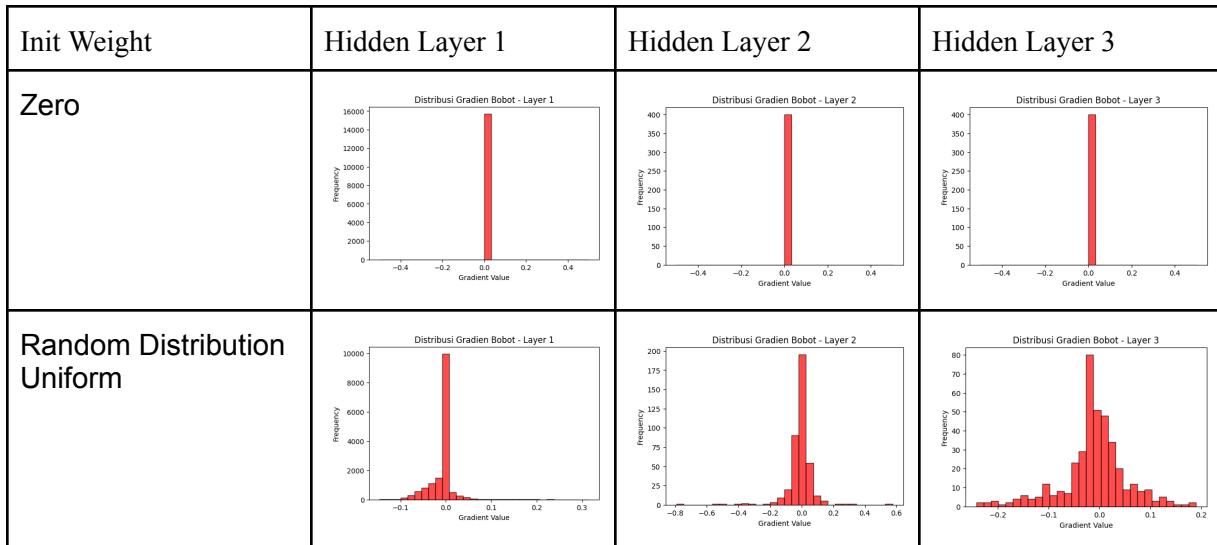


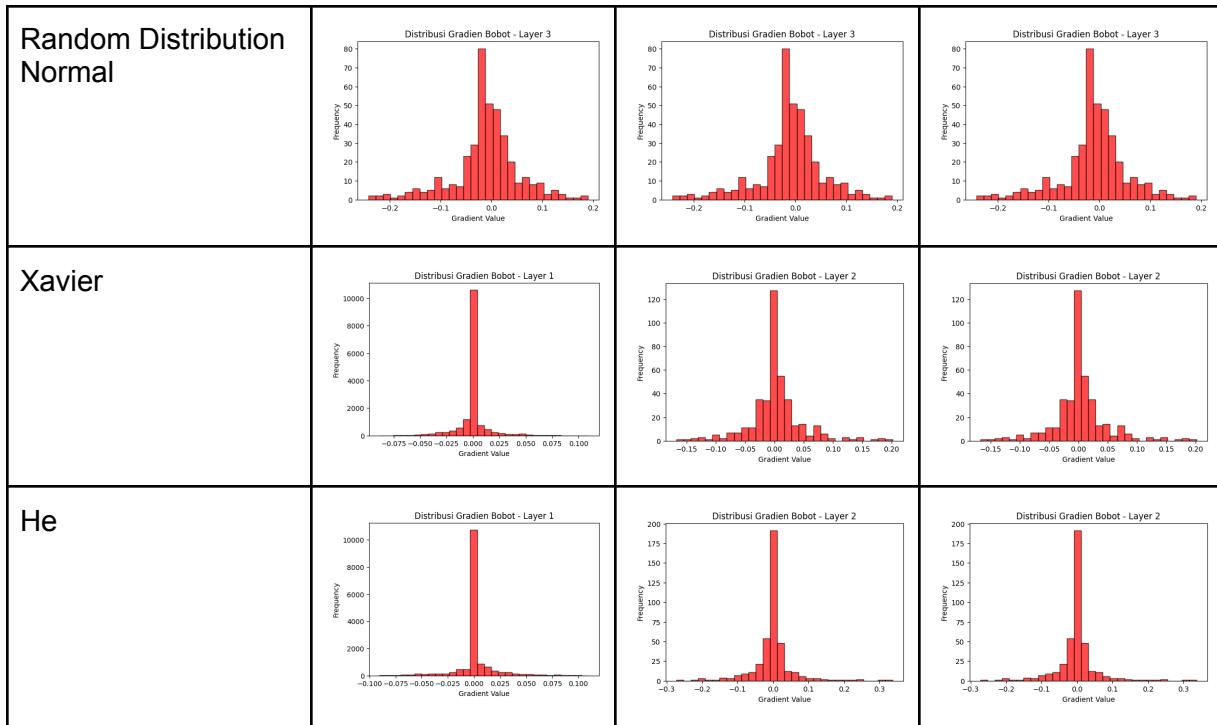
2.2.4.1 Distribusi Bobot

Init Weight	Hidden Layer 1	Hidden Layer 2	Hidden Layer 3
Zero			



2.2.4.1 Distribusi Gradien Bobot





a. Hasil Analisis

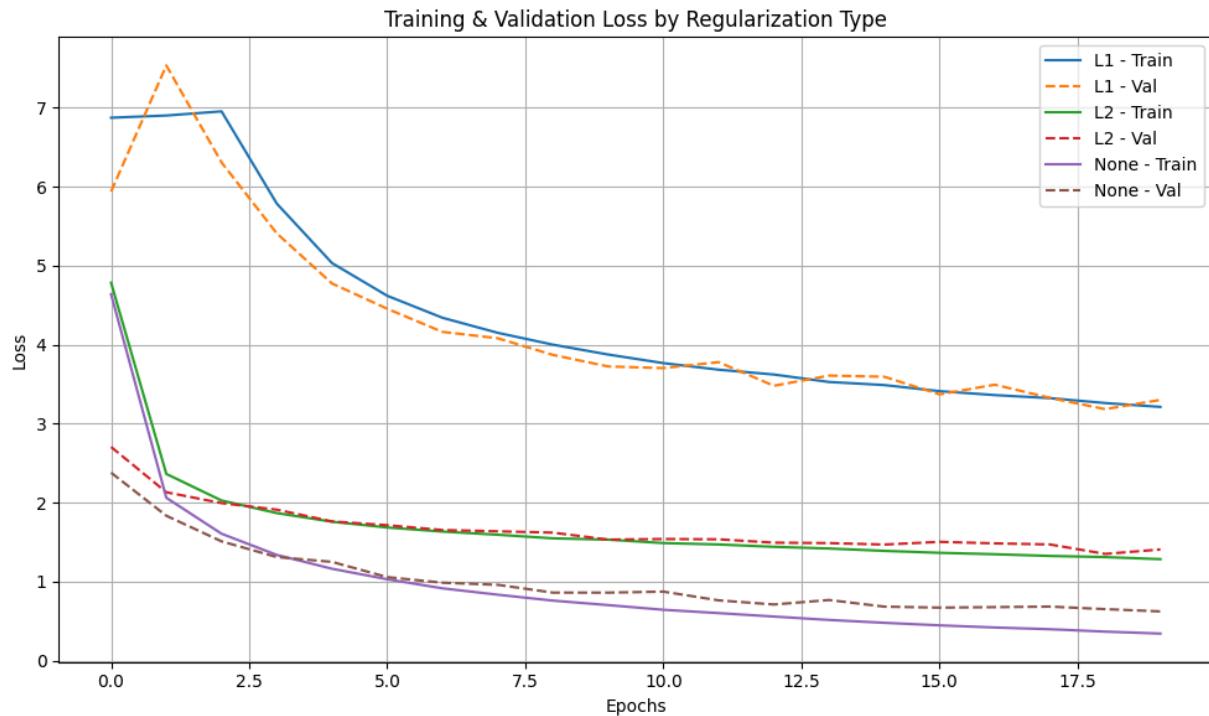
Inisialisasi dengan nilai nol (ZERO) menghasilkan performa yang sangat buruk (F1 Score = 0.0205), yang sesuai dengan teori bahwa bobot nol menyebabkan semua neuron belajar dengan cara yang sama, sehingga tidak terjadi pembelajaran yang efektif. Sementara itu, inisialisasi acak menggunakan distribusi uniform dan normal memperbaiki performa secara signifikan, dengan F1 Score masing-masing sebesar 0.8644 dan 0.9452. Metode Xavier dan He initialization, yang dirancang khusus untuk mengatasi permasalahan vanishing dan exploding gradient pada jaringan dalam, memberikan hasil terbaik dengan F1 Score sebesar 0.9462 dan 0.9505.

2.2.5 Pengaruh Regularisasi

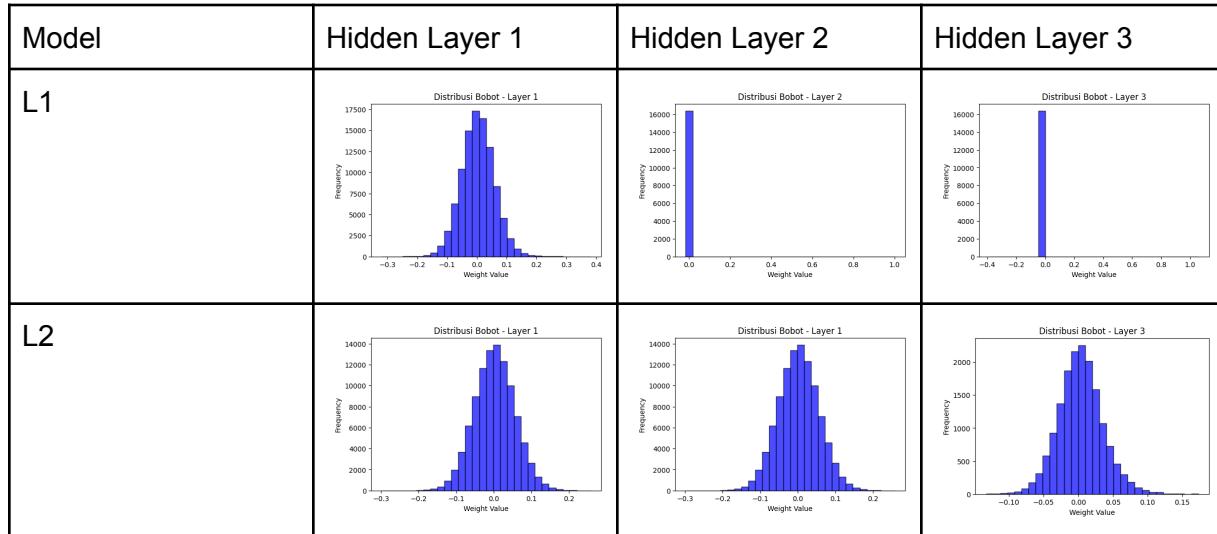
2.2.5.1 Hasil F1 Score

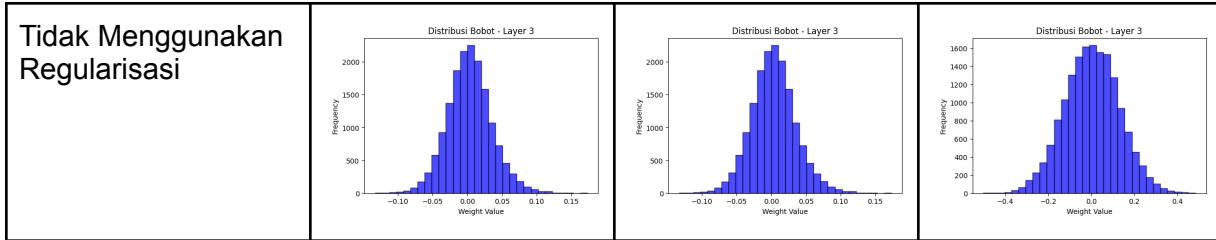
- L1 Regularization - F1 Score: 0.8798
- L2 Regularization - F1 Score: 0.9541
- None Regularization - F1 Score: 0.9693

2.2.5.2 Train Loss dan Validation Loss

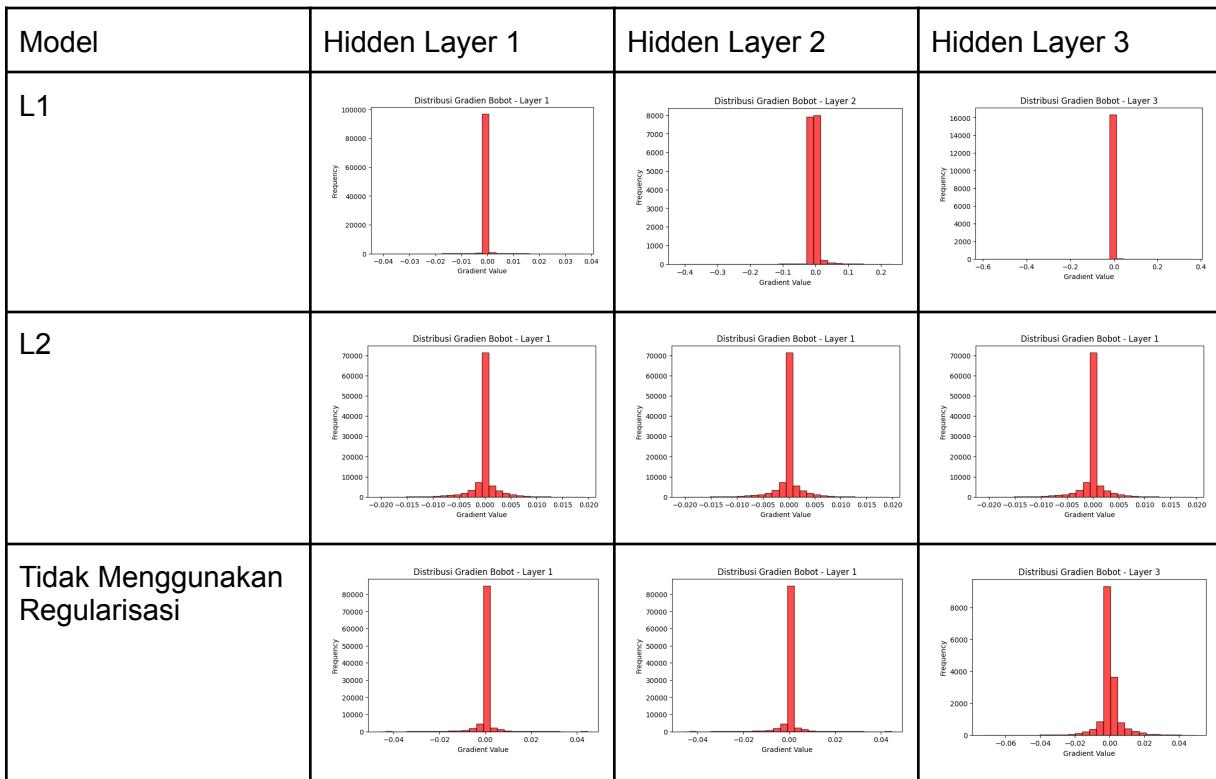


2.2.5.3 Distribusi Bobot





2.2.5.4 Distribusi Gradien Bobot



2.2.5.5 Analisis

Hasil F1 Score pada pengujian pengaruh regularisasi menunjukkan bahwa penggunaan regularisasi dapat menurunkan performa model jika tidak disesuaikan dengan kompleksitas data. Model tanpa regularisasi menghasilkan performa tertinggi dengan F1 Score sebesar 0.9693, diikuti oleh L2 regularization (0.9541), dan terakhir L1 regularization (0.8798). L1 cenderung memberikan penalti yang lebih besar terhadap bobot kecil sehingga banyak bobot menjadi nol, yang dapat menyebabkan hilangnya informasi penting, terutama jika fitur relevan tersebar. L2 lebih stabil karena menghukum bobot besar tanpa memaksanya menjadi nol, sehingga cenderung mempertahankan generalisasi yang baik. Namun, jika data tidak terlalu kompleks atau

tidak rentan terhadap overfitting, seperti pada kasus ini, maka penggunaan regularisasi justru dapat menghambat kinerja model.

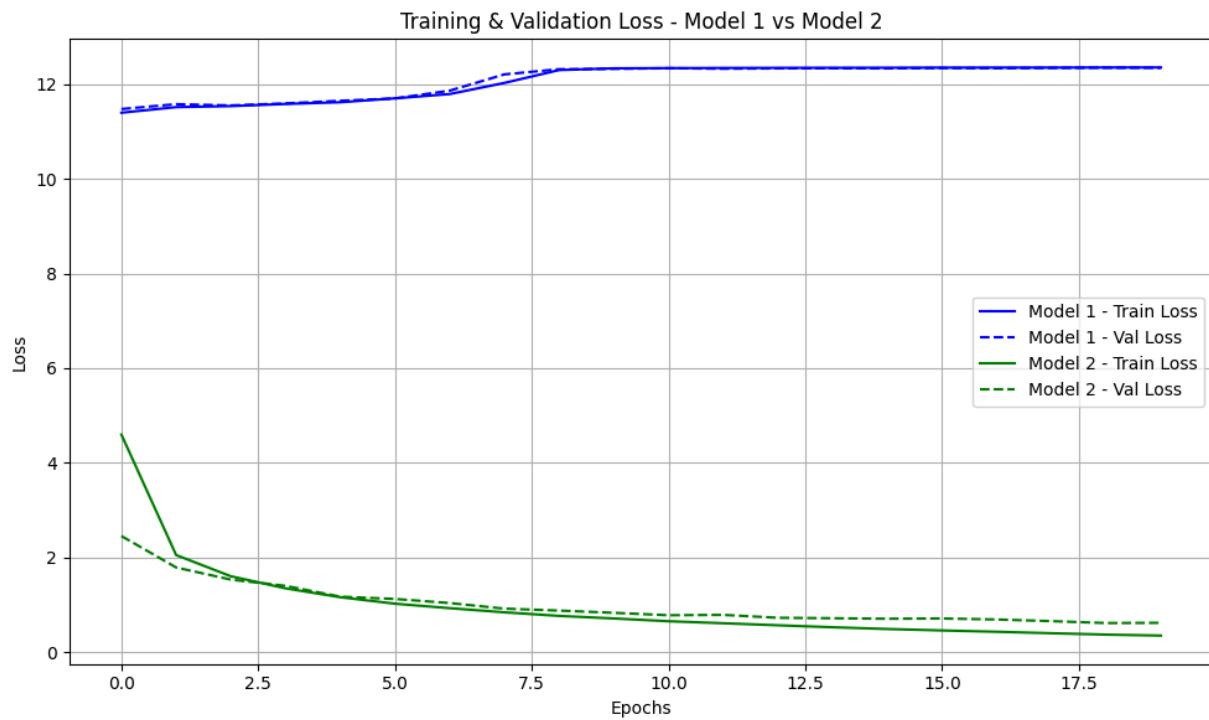
2.2.6 Pengaruh RMS Prop

2.2.6.1 Hasil F1 Score

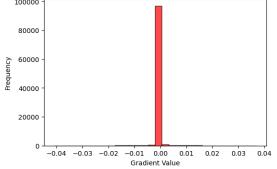
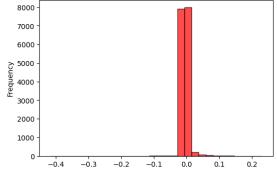
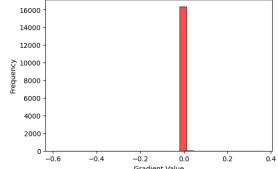
F1 Score dengan RMS Prop: 0.0726

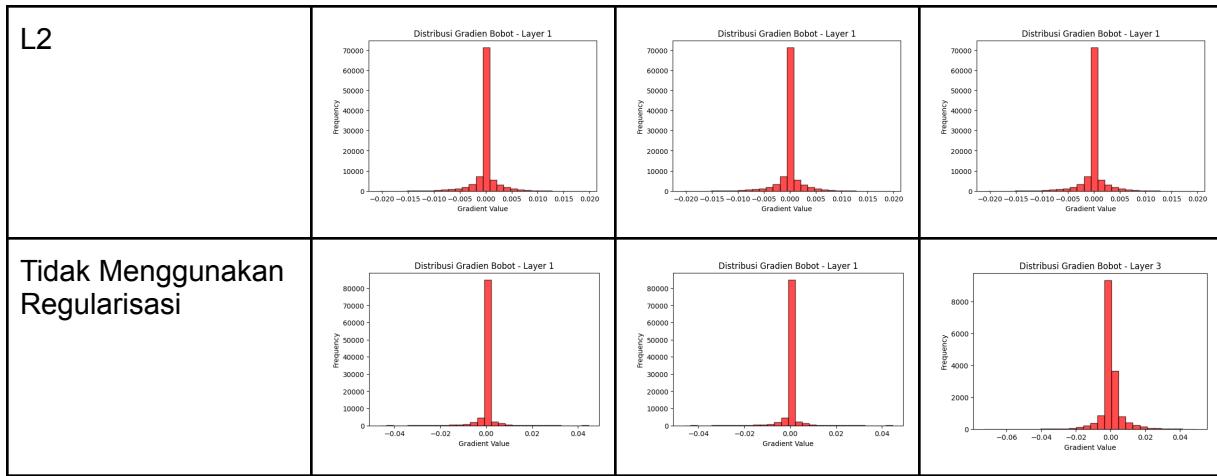
F1 Score tanpa RMS Prop: 0.9700

2.2.6.2 Train Loss dan Validation Loss

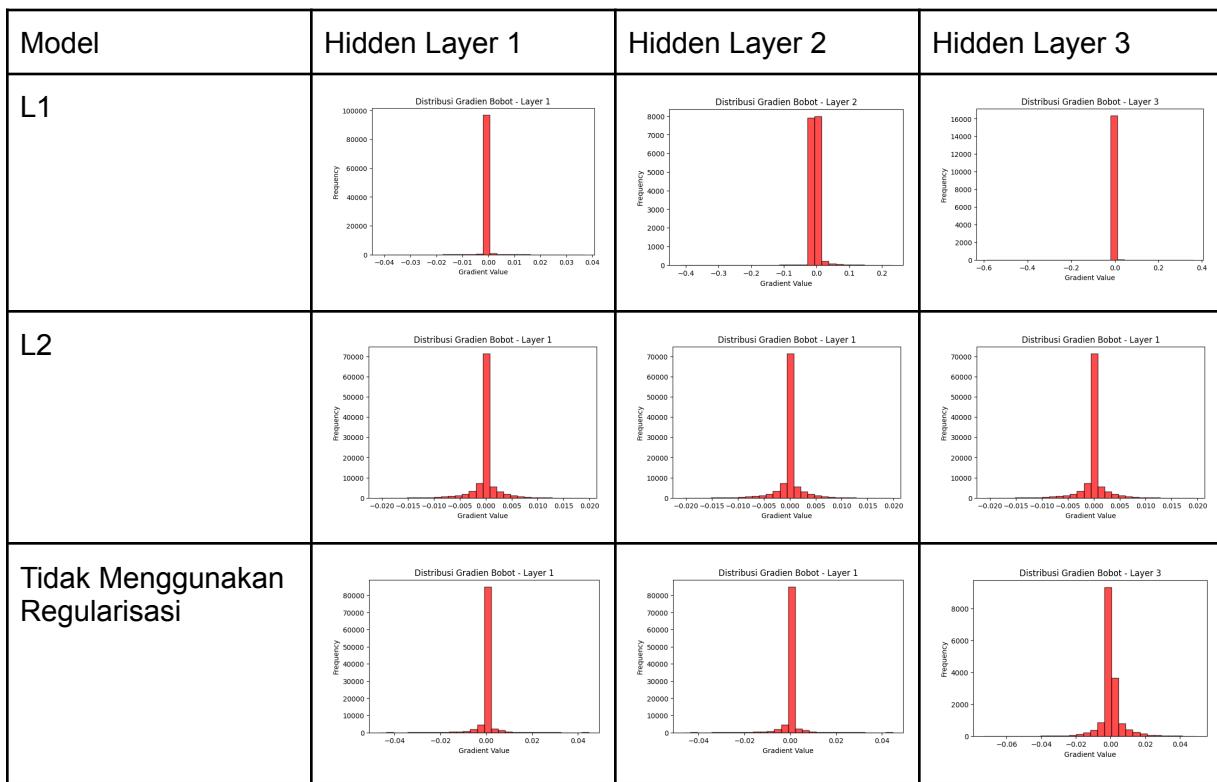


2.2.6.3 Distribusi Bobot

Model	Hidden Layer 1	Hidden Layer 2	Hidden Layer 3
L1			



2.2.6.4 Distribusi Gradien Bobot



2.2.6.5 Analisis

Hasil F1 Score menunjukkan perbedaan yang sangat signifikan antara model yang menggunakan RMSProp dan yang tidak. Tanpa RMSProp, model mampu mencapai F1 Score sebesar 0.9700, yang menandakan performa klasifikasi yang sangat baik dan akurat. Sebaliknya, saat RMSProp diaktifkan dalam proses forward pass, F1 Score justru merosot tajam menjadi 0.0726, yang hampir setara dengan prediksi acak dan mengindikasikan kegagalan model dalam belajar. Hal ini kemungkinan besar disebabkan oleh kesalahan dalam implementasi RMSProp. Pada dasarnya, RMSProp adalah algoritma optimisasi yang digunakan untuk mengatur pembaruan bobot berdasarkan rata-rata kuadrat dari gradien.

2.2.7 Perbandingan dengan library sklearn

a. Parameter

```
input_size = 784
hidden_layers = 5
hidden_size = 128 (banyak neuron pada suatu hidden layer)
output_size = 10 (banyak neuron pada output layer)
learning_rate = 0.01
param_1 = 0
param_2 = 0.5
batch_size = 64
Epoch = 20
Activation function = tanh
```

b. Scratch

F1 Score: 0.9670747601934998

c. Sklearn

```
Iteration 1, loss = 0.31902645
Iteration 2, loss = 0.14478043
Iteration 3, loss = 0.10597684
Iteration 4, loss = 0.08302081
Iteration 5, loss = 0.06492869
Iteration 6, loss = 0.05416916
Iteration 7, loss = 0.04421797
Iteration 8, loss = 0.03745102
Iteration 9, loss = 0.03054358
Iteration 10, loss = 0.02470356
Iteration 11, loss = 0.02102427
Iteration 12, loss = 0.01715281
Iteration 13, loss = 0.01315105
Iteration 14, loss = 0.01105560
Iteration 15, loss = 0.00852507
Iteration 16, loss = 0.00687729
Iteration 17, loss = 0.00540345
Iteration 18, loss = 0.00402926
Iteration 19, loss = 0.00269943
Iteration 20, loss = 0.00219982
c:\Users\DELL\AppData\Local\Programs\Python\Python37\lib\site-packages\sklearn\linear_model\logistic.py:415: UserWarning: Optimized solver reached a limit of 20 iterations without converging to a better local minimum. This may indicate a local minimum or that the algorithm has not converged.
  warnings.warn("Optimized solver reached a limit of %d iterations without
F1 Score: 0.9792651443089493
```

d. Hasil Analisis

Berdasarkan hasil perbandingan antara model yang dibangun secara scratch dan model dari library sklearn, dapat disimpulkan bahwa keduanya mampu memberikan performa klasifikasi yang sangat baik, ditunjukkan dengan nilai F1 Score yang tinggi. Model scratch menghasilkan F1 Score sebesar 0.9671, sementara model sklearn menghasilkan nilai yang sedikit lebih tinggi, yaitu 0.9793. Selisih ini kemungkinan disebabkan oleh optimasi internal dan efisiensi implementasi dari sklearn yang sudah sangat matang dan teruji secara luas. Selain itu, proses pelatihan pada model sklearn tampak menunjukkan penurunan loss yang konsisten selama 20 iterasi, mengindikasikan proses konvergensi yang stabil. Secara keseluruhan, hasil ini menunjukkan bahwa implementasi *scratch* yang dilakukan sudah cukup baik dan mendekati performa dari library populer, serta mencerminkan pemahaman mendalam terhadap mekanisme jaringan saraf buatan.

BAB 3 KESIMPULAN DAN SARAN

3.1 Kesimpulan

Berdasarkan hasil implementasi dan pengujian Feedforward Neural Network (FFNN) dari awal tanpa bantuan library deep learning seperti TensorFlow atau PyTorch, dapat disimpulkan bahwa:

- a. Model FFNN scratch berhasil dibangun secara modular dan fleksibel, mendukung berbagai konfigurasi jaringan seperti jumlah layer, jumlah neuron per layer, fungsi aktivasi, fungsi loss, metode inisialisasi bobot, hingga teknik regularisasi dan optimisasi.
- b. Forward propagation dan backward propagation telah diimplementasikan dengan sukses, mendukung input dalam bentuk batch dan mampu menghitung gradien bobot terhadap fungsi loss dengan baik menggunakan chain rule.
- c. Hasil pengujian menunjukkan bahwa struktur jaringan berpengaruh signifikan terhadap performa model. Penambahan depth dan width mampu meningkatkan F1 Score, menunjukkan bahwa jaringan yang lebih kompleks lebih mampu menangkap representasi data yang lebih kaya.
- d. Fungsi aktivasi non-linear seperti ReLU, Leaky ReLU, Tanh, dan PReLU memberikan hasil yang jauh lebih baik dibandingkan Linear atau Sigmoid, dengan ReLU dan Leaky ReLU mencatat performa terbaik (F1 Score: 0.9708).
- e. Learning rate memiliki pengaruh besar terhadap kestabilan dan kecepatan konvergensi model. Learning rate optimal berada di angka 0.01; nilai yang terlalu kecil menghambat pembelajaran, sedangkan yang terlalu besar membuat model tidak stabil.
- f. Inisialisasi bobot yang baik sangat penting. Metode seperti He dan Xavier initialization terbukti memberikan performa optimal, sedangkan inisialisasi nol menyebabkan kegagalan total dalam pembelajaran.
- g. Penggunaan regularisasi harus disesuaikan dengan kebutuhan. Tanpa regularisasi justru memberikan hasil terbaik dalam kasus ini, karena data MNIST relatif bersih dan tidak terlalu kompleks. Penggunaan L1 atau L2 perlu dikaji ulang sesuai kompleksitas dan potensi overfitting dataset.
- h. Implementasi RMSProp belum optimal, bahkan menurunkan performa secara drastis, menandakan adanya potensi kesalahan dalam cara integrasinya terhadap alur training.
- i. Perbandingan dengan model MLPClassifier dari sklearn menunjukkan performa yang kompetitif. Model dari awal (scratch) berhasil mendekati performa library populer, yang mengindikasikan keberhasilan dalam memahami dan membangun arsitektur jaringan saraf buatan secara mendalam.

3.1 Saran

Untuk meningkatkan performa pada model Feed Forward Neural Network, berikut beberapa saran agar performa model dapat menjadi lebih baik.

1. Perbaiki dan lengkapi implementasi RMSProp dan teknik optimasi lainnya, seperti Adam atau AdaGrad, untuk meningkatkan fleksibilitas dan efektivitas pelatihan model.
2. Tambahkan fitur early stopping dan learning rate scheduler agar pelatihan lebih efisien dan terhindar dari overfitting atau underfitting.
3. Uji model pada dataset lain di luar MNIST agar dapat menilai generalisasi model terhadap data yang lebih kompleks, noisy, atau memiliki distribusi yang berbeda.

Dengan berbagai eksperimen dan analisis yang telah dilakukan, tugas besar ini tidak hanya memberikan pemahaman mendalam tentang cara kerja jaringan saraf tiruan, namun juga pengalaman langsung dalam membangun sistem machine learning dari awal secara menyeluruh.

Pembagian Tugas

Nama	NIM	Tugas
Auralea Alvinia S	13522148	<ul style="list-style-type: none">- Memvisualisasikan struktur graf beserta bobot tiap neuron pada layer, distribusi bobot, distribusi gradien- Membuat input layer
M. Fauzan Azhim	13522153	<ul style="list-style-type: none">- Membuat regularization, normalization, activation function, loss function- Membuat forward propagation dan back propagation
Pradipta rafa Mahesa	13522162	<ul style="list-style-type: none">- Membuat fitur safe dan load- Melakukan inisialisasi bobot

Referensi

- <https://www.jasonosajima.com/forwardprop>
- <https://www.jasonosajima.com/backprop>
- <https://numpy.org/doc/2.2/>
- https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
- <https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>
- <https://douglasorr.github.io/2021-11-autodiff/article.html>
- https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2022/tutorials/tut01.pdf
- <https://www.geeksforgeeks.org/training-and-validation-loss-in-deep-learning/>
- <https://builtin.com/data-science/l2-regularization>