

**TUGAS BESAR 1**  
**IF3270 Pembelajaran Mesin**  
**Feedforward Neural Network**



**Disusun Oleh:**  
**13522061      Maximilian Sulistiyo**  
**13522075      Marvel Pangondian**  
**13522101      Abdullah Mubarak**

**Teknik Informatika**  
**Sekolah Teknik Elektro dan Informatika**  
**Institut Teknologi Bandung**

## DAFTAR ISI

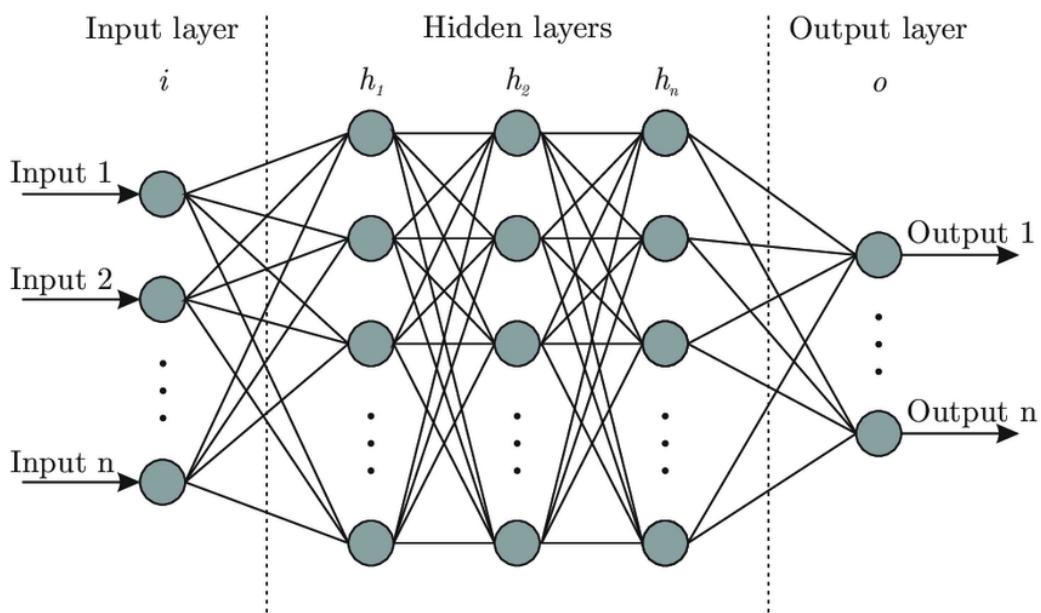
DAFTAR ISI.....	2
BAB 1.....	4
BAB 2.....	6
2.1. Penjelasan Implementasi.....	6
2.1.1. Kelas Value.....	6
2.1.2. Kelas Dense Layer.....	14
2.1.3. Kelas Neural Network.....	18
2.1.4. Kelas Loss Function.....	32
2.2. Hasil Pengujian.....	36
2.2.1. Pengaruh Depth & Width.....	36
2.2.1.1. Pengaruh Width.....	36
i. Tes 1 - 32 neuron setiap layer.....	36
ii. Tes 2 - 64 neuron setiap layer.....	37
iii. Tes 3 - 128 neuron setiap layer.....	38
iv. Kesimpulan - Pengaruh Width.....	39
2.2.1.2. Pengaruh Depth.....	40
i. Tes 1 - 1 hidden layer.....	40
ii. Tes 2 - 3 hidden layer.....	40
iii. Tes 3 - 5 hidden layer.....	41
iv. Kesimpulan - Pengaruh Depth.....	41
2.2.2. Pengaruh Fungsi Aktivasi.....	42
2.2.2.1. Tes 1 - Linear Activation Function.....	42
2.2.2.2. Tes 2 - ReLU Activation Function.....	43
2.2.2.3. Tes 3 - Sigmoid Activation Function.....	44
2.2.2.4. Tes 4 - Tanh Activation Function.....	45
2.2.2.5. Kesimpulan - Activation Function.....	46
2.2.3. Pengaruh Learning Rate.....	47
2.2.3.1. Tes 1 - Learning rate 0.01.....	47
2.2.3.2. Tes 2 - Learning rate 0.05.....	48
2.2.3.3. Tes 3 - Learning rate 0.25.....	49
2.2.3.4. Kesimpulan - Pengaruh Learning Rate.....	50
2.2.4. Pengaruh Inisialisasi Bobot.....	51
2.2.4.1. Tes 1 - Inisialisasi Zero.....	51
2.2.4.2. Tes 2 - Inisialisasi Uniform.....	52
2.2.4.3. Tes 3 - Inisialisasi Normal.....	53
2.2.4.4. Tes 4 - Inisialisasi Xavier.....	54
2.2.4.5. Tes 5 - Inisialisasi He.....	55
2.2.4.6. Kesimpulan - Pengaruh Inisialisasi Bobot.....	55
2.2.5. Pengaruh Regulation Parameter.....	57

2.2.5.1. Tes 1 - Tanpa Regulation.....	58
2.2.5.2. Tes 2 - Regulation L1.....	58
2.2.5.3. Tes 3 - Regulation L2.....	59
2.2.5.4. Kesimpulan - Pengaruh Inisialisasi Bobot.....	60
2.2.6. Perbandingan dengan Library sklearn.....	61
2.2.6.1. Hasil training model sklearn.....	62
2.2.6.2. Hasil training model ANN Kelompok.....	62
2.2.6.3. Kesimpulan - Perbandingan dengan Library sklearn.....	63
BAB 3.....	64
3.1. Kesimpulan.....	64
Berdasarkan eksperimen kami dapat menyimpulkan beberapa hal:.....	64
3.2. Saran.....	65
BAB 4.....	66
LAMPIRAN.....	67
DAFTAR PUSTAKA.....	68

# BAB 1

## Deskripsi Persoalan

ANN terbuat dari individual Perceptron yang merupakan single unit/neuron, estimasi function (hypothesis). Artificial Neural yang terinspirasi dari cara kerja otak dimana unit komputasi yang berbeda menjadi intelligent dengan cara berinteraksi satu sama lain. Network atau jaringan artinya terdiri dari beberapa fungsi yang berbeda



Ilustrasi Artificial Neural Network

*Feedforward Neural Network* (FFNN) merupakan salah satu arsitektur dari ANN. Pada arsitektur ini koneksi bersifat satu arah sehingga membentuk directed acyclic graph dengan simpul input dan output yang telah ditentukan. Salah satu contohnya adalah multilayer perceptron merupakan fully connected feedforward dengan non linear activation function. Informasi bergerak melalui network dari input ke output nodes dan tidak ada loop

Cara inferensi dari FFNN adalah menggunakan *forward propagation* dimana informasi mengalir melalui fungsi yang mengevaluasi vektor  $x$ , melalui *intermediate computation* untuk mendefinisikan  $f$  dan kemudian mendapatkan hasil  $y$

*Backpropagation* pada FFNN adalah algoritma yang digunakan untuk training network dengan mengoptimalkan bobot (weights) untuk meminimalkan loss function. Tahap pertama dari *backpropagation* adalah *forward propagation* yang telah dijelaskan sebelumnya.

Setelah mendapatkan output, error dihitung dengan membandingkan output yang diprediksi dengan output yang diharapkan (target). Error ini kemudian "dipropagasi kembali" melalui jaringan dari output layer ke input layer.

Disini kami diberi tugas untuk mengimplementasikan FFNN *from scratch* menggunakan bahasa *python* dengan mengimplementasikan beberapa fungsi aktivasi yang sering digunakan (*linear, relu, sigmoid, tanh, dan softmax*) dan fungsi loss yang sering digunakan (*mse, binary cross entropy, dan categorical cross entropy*)

## BAB 2

### Pembahasan

#### 2.1. Penjelasan Implementasi

##### 2.1.1. Kelas Value

Kelas Value merupakan kelas yang diimplementasikan untuk membangun sistem *automatic differentiation* dengan melacak semua operasi matematika dan turunannya dalam bentuk sebuah *computational graph*

```
import numpy as np

class Value:
    def __init__(self, data, _children=(), _op="", label ""):
        self.data = np.array(data)
        self.grad = np.zeros_like(self.data, dtype=float)
        self._backward = lambda: None
        self._prev = set(_children)
        self._op = _op
        self.label = label

    def __repr__(self):
        return f"Value(data={shape{self.data.shape}}, {self.data})"

    def __add__(self, other):
        other_data = other.data if isinstance(other, Value) else
np.array(other)
        out = Value(
            self.data + other_data,
            (self, other) if isinstance(other, Value) else
(self, ),
            "+",
        )
        def _backward():
            grad_shape = np.broadcast(
                self.data, other_data if isinstance(other,
```

```

Value) else other
        ).shape

        if self.grad.shape != grad_shape:
            axes_to_sum = tuple(
                i
                for i, (a, b) in
enumerate(zip(self.grad.shape, grad_shape))
                if a != b
            )
            sum_grad = np.sum(out.grad, axis=axes_to_sum,
keepdims=True)
            self.grad += sum_grad
        else:
            self.grad += out.grad

        if isinstance(other, Value):
            if other.grad.shape != grad_shape:
                axes_to_sum = tuple(
                    i
                    for i, (a, b) in
enumerate(zip(other.grad.shape, grad_shape))
                    if a != b
                )
                sum_grad = np.sum(out.grad,
axis=axes_to_sum, keepdims=True)
                other.grad += sum_grad
            else:
                other.grad += out.grad

        out._backward = _backward
        return out

    def mean(self, axis=None, keepdims=False):
        n = self.data.size if axis is None else
self.data.shape[axis]
        t = np.mean(self.data, axis=axis, keepdims=keepdims)
        out = Value(t, (self,), "mean")

    def _backward():

```

```
        grad_broadcasted = np.ones_like(self.data) *
out.grad / n
        self.grad += grad_broadcasted

        out._backward = _backward
        return out

    def __radd__(self, other):
        return self.__add__(other)

    def __sub__(self, other):
        other_data = other.data if isinstance(other, Value) else
np.array(other)
        out = Value(
            self.data - other_data,
            (self, other) if isinstance(other, Value) else
(self,),
            "-",
        )

        def _backward():
            self.grad += out.grad
            if isinstance(other, Value):
                other.grad -= out.grad

            out._backward = _backward
            return out

    def __rsub__(self, other):
        other_data = other if not isinstance(other, Value) else
other.data
        out = Value(
            other_data - self.data,
            (self,) if not isinstance(other, Value) else (other,
self),
            "-",
        )

        def _backward():
            self.grad -= out.grad
```

```
        if isinstance(other, Value):
            other.grad += out.grad

        out._backward = _backward
        return out

    def __mul__(self, other):
        other_data = other.data if isinstance(other, Value) else
np.array(other)
        out = Value(
            self.data * other_data,
            (self, other) if isinstance(other, Value) else
(self,),
            "*",
        )

        def _backward():
            self.grad += other_data * out.grad
            if isinstance(other, Value):
                other.grad += self.data * out.grad

            out._backward = _backward
            return out

    def __neg__(self):
        out = Value(-self.data, (self,), "-")

        def _backward():
            self.grad -= out.grad

        out._backward = _backward
        return out

    def __rmul__(self, other):
        return self.__mul__(other)

    def matmul(self, other):
        out = Value(np.matmul(self.data, other.data), (self,
other), "@")
```

```
def __backward():
    self.grad += np.matmul(out.grad, other.data.T)
    other.grad += np.matmul(self.data.T, out.grad)

out.__backward = __backward
return out

def tanh(self):
    t = np.tanh(self.data)
    out = Value(t, (self,), "tanh")

    def __backward():
        self.grad += (1 - t**2) * out.grad

    out.__backward = __backward
    return out

def sigmoid(self):
    t = 1 / (1 + np.exp(-self.data))
    out = Value(t, (self,), "sigmoid")

    def __backward():
        self.grad += (t * (1 - t)) * out.grad

    out.__backward = __backward
    return out

def relu(self):
    t = np.maximum(0, self.data)
    out = Value(t, (self,), "relu")

    def __backward():
        self.grad += (self.data > 0) * out.grad

    out.__backward = __backward
    return out

def linear(self):
    out = Value(self.data, (self,), "linear")
```

```

def __backward():
    self.grad += out.grad

    out.__backward = __backward
    return out

def softmax(self):
    exp_x = np.exp(self.data - np.max(self.data, axis=1,
keepdims=True))
    t = exp_x / np.sum(exp_x, axis=1, keepdims=True)
    out = Value(t, (self,), "softmax")

    def __backward():
        n = self.data.shape[0]

        for i in range(n):
            softmax_i = t[i].reshape(-1, 1)
            grad_i = out.grad[i].reshape(-1, 1)

            jacobian = np.diagflat(softmax_i) -
np.dot(softmax_i, softmax_i.T)

            self.grad[i] += np.dot(jacobian,
grad_i).flatten()

        out.__backward = __backward
        return out

    def log(self):
        t = np.log(self.data)
        out = Value(t, (self,), "log")

        def __backward():
            self.grad += (1 / self.data) * out.grad

        out.__backward = __backward
        return out

    def sum(self, axis=None, keepdims=False):

```

```
t = np.sum(self.data, axis=axis, keepdims=keepdims)
out = Value(t, (self,), "sum")

def _backward():
    if axis is not None and not keepdims:
        expanded_grad = out.grad

        axes = [axis] if not isinstance(axis, tuple)
    else list(axis)
        axes.sort(reverse=True)

        for ax in axes:
            expanded_grad =
np.expand_dims(expanded_grad, axis=ax)

            self.grad += expanded_grad
    else:
        self.grad += out.grad

    out._backward = _backward
    return out

def clip(self, min_val, max_val):
    clipped_data = np.clip(self.data, min_val, max_val)
    out = Value(clipped_data, (self,), "clip")

    def _backward():
        grad_mask = (self.data >= min_val) & (self.data <=
max_val)
        self.grad += grad_mask * out.grad

    out._backward = _backward
    return out

def backward(self):
    topo = []
    visited = set()

    def build_topo(v):
        if v not in visited:
```

```

        visited.add(v)
        for child in v._prev:
            build_topo(child)
        topo.append(v)

    build_topo(self)

    self.grad = np.ones_like(self.data)
    for node in reversed(topo):
        node._backward()

```

Kelas ini menyimpan beberapa atribut:

1. **data**: isi dari nilai yang disimpan dalam bentuk numpy array
2. **grad**: gradient dari nilai tersebut terhadap output
3. **\_backward**: fungsi yang akan dijalankan saat backward propagation, diinisialisasi dengan fungsi kosong
4. **\_prev**: kumpulan node-node yang menjadi input untuk operasi yang menghasilkan nilai ini
5. **\_op**: string yang menyimpan operasi yang menghasilkan nilai ini
6. **label**: nama atau label untuk nilai ini

Kemudian terdapat method-method cara melakukan operasi matematika dasar seperti add (penjumlahan), sub (pengurangan), mul (perkalian), truediv (pembagian). Terdapat juga implementasi dari fungsi aktivasi seperti linear, tanh, sigmoid, dan softmax yang masing-masing dilengkapi dengan perhitungan gradiennya. Pada fungsi operasi-operasi tersebut terdapat dua hal yang terjadi:

1. **Operasi forward**: Perhitungan nilai output berdasarkan input dengan menerapkan operasi matematika yang sesuai dan menyimpan informasi yang diperlukan untuk komputasi gradien nanti. Setiap operasi juga membangun bagian dari grafik komputasi dengan menyimpan referensi ke nilai-nilai input.
2. **Operasi backward**: Operasi ini digunakan untuk menghitung gradien dari operasi matematika tersebut sesuai dengan aturan rantai (chain rule) dalam kalkulus. Fungsi backward mendefinisikan bagaimana gradien dari output akan diproses kembali dan didistribusikan ke operand-operand yang terlibat

dalam operasi. Misalnya pada operasi penjumlahan, gradien output akan didistribusikan sama rata ke kedua input, sedangkan pada perkalian, gradien didistribusikan sesuai dengan nilai operand lainnya.

Terakhir terdapat method `backward()` yang berfungsi untuk melakukan backpropagation pada seluruh *computational graph*. Pertama method ini melakukan topo sort untuk mendapatkan urutan node. Kemudian akan menginisialisasi gradien output dengan nilai 1 (karena  $\frac{\partial o}{\partial o} = 1$ ) dan setelah itu menelusuri urutan node tadi secara reverse dan memanggil method `_backward()` pada setiap node untuk menghitung dan mengakumulasi gradien pada setiap node. Hal ini akan mendistribusikan gradien ke seluruh parameter sesuai dengan *chain rule*.

### 2.1.2. Kelas Dense Layer

Kelas ini merupakan implementasi dari fully connected layer pada sebuah FFNN. Layer ini menghubungkan setiap neuron pada layer sebelumnya dengan setiap neuron yang terdapat dalam layer ini.

```
import numpy as np
from value import Value

class DenseLayer:
    def __init__(
        self,
        output_size,
        activation=None,
        init="random",
        bias_init="Zero",
        mean=0.0,
        var=1.0,
        lower_bound=-0.5,
        upper_bound=0.5,
        seed=None,
```

```
        reg_type=None,
        reg_param=0.0,
    ):

        self.output_size = output_size
        self.activation = activation

        self.init = init
        self.bias_init = bias_init
        self.mean = mean
        self.var = var
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound
        self.seed = seed

        self.reg_type = reg_type
        self.reg_param = reg_param

        self.weights = None
        self.biases = None

    def _initialize_weights(self, input_size):
        if self.seed is not None:
            np.random.seed(self.seed)

        if self.init == "random":
            weights_data = np.random.randn(input_size,
self.output_size) - 0.5
        elif self.init == "Xavier":
            weights_data = np.random.randn(input_size,
self.output_size) * np.sqrt(
                1.0 / input_size
            )
        elif self.init == "He":
            weights_data = np.random.randn(input_size,
self.output_size) * np.sqrt(
                2.0 / input_size
            )
        elif self.init == "Zero":
            weights_data = np.zeros((input_size,
self.output_size))
```

```

        elif self.init == "One":
            weights_data = np.ones((input_size,
self.output_size))
        elif self.init == "uniform":
            weights_data = np.random.uniform(
                low=self.lower_bound,
                high=self.upper_bound,
                size=(input_size, self.output_size),
            )
        elif self.init == "normal":
            weights_data = np.random.normal(
                loc=self.mean,
                scale=np.sqrt(self.var),
                size=(input_size, self.output_size),
            )
        else:
            raise ValueError(f"Invalid weight initialization
method: {self.init}")
        self.weights = Value(weights_data, label="weights")

    def _initialize_bias(self, input_size):
        if self.seed is not None:
            np.random.seed(self.seed + 1)
        if self.bias_init == "random":
            bias_data = np.random.randn(1, self.output_size) -
0.5
        elif self.bias_init == "Xavier":
            bias_data = np.random.randn(1, self.output_size) *
np.sqrt(1.0 / input_size)
        elif self.bias_init == "He":
            bias_data = np.random.randn(1, self.output_size) *
np.sqrt(2.0 / input_size)
        elif self.bias_init == "Zero":
            bias_data = np.zeros((1, self.output_size))
        elif self.bias_init == "One":
            bias_data = np.ones((1, self.output_size))
        elif self.bias_init == "uniform":
            bias_data = np.random.uniform(
                low=self.lower_bound, high=self.upper_bound,
                size=(1, self.output_size)

```

```

        )
    elif self.bias_init == "normal":
        bias_data = np.random.normal(
            loc=self.mean, scale=np.sqrt(self.var), size=(1,
self.output_size)
        )
    else:
        raise ValueError(f"Unknown bias initialization
method: {self.bias_init}")

    self.biases = Value(bias_data, label="biases")

def forward(self, input_value):
    if not isinstance(input_value, Value):
        input_value = Value(input_value, label="input")

    self.input = input_value

    if self.weights is None or self.biases is None:
        self.input_size = input_value.data.shape[1]
        self._initialize_weights(self.input_size)
        self._initialize_bias(self.input_size)

        linear_output = self.input.matmul(self.weights) +
self.biases

        if self.activation:
            return self.activation(linear_output)
    else:
        return linear_output

```

Kelas ini terdiri atas beberapa atribut :

1. output\_size : Jumlah neuron pada layer ini
2. activation : Fungsi aktivasi yang digunakan pada layer ini (Linear, ReLu, Sigmoid, Tanh, Softmax). Softmax hanya dapat digunakan sebagai fungsi aktivasi layer output.

3. init : Metode inisialisasi bobot ( random, Xavier, He, Zero, One, Uniform, Normal)
4. init\_bias : Metode inisialisasi bobot bias ( random, Xavier, He, Zero, One, Uniform, Normal)
5. mean, var, lower\_bound, upper\_bound : Parameter tambahan untuk metode inisialisasi bobot normal/uniform
6. seed : Mengatur seed untuk *reproducibility* agar eksperimen dapat direproduksi
7. weights, biases : Bobot dan bias dalam bentuk obyek *Value* untuk automatic differentiation.

Kelas ini memiliki beberapa method utama, antara lain :

1. \_\_init\_\_() : Menginisialisasi konfigurasi layer, tetapi belum menginisialisasi bobot dan bias, baru akan dibuat saat *forward*
2. \_initialize\_weights(input\_size) : Menginisialisasi bobot pada layer tersebut berdasarkan atribut init .
3. \_initialize\_bias(input\_size) : Menginisialisasi bias pada layer tersebut berdasarkan atribut init bias
4. forward(input\_value) : Melakukan operasi feedforward yang dimulai dengan mengecek apakah bobot dan bias sudah diinisialisasi sebelumnya. Jika bobot dan bias belum diinisialisasi, maka memanggil fungsi \_initialize\_weights dan \_initialize\_bias. Setelah itu, melakukan perhitungan linear\_output dengan melakukan perkalian matrix antara bobot (self.weights) dan input ditambah dengan bias (linear\_output = self.input.matmul(self.weights) + self.biases). Hasil dari linear\_output (merupakan net neuron) diproses lebih lanjut dengan fungsi aktivasi pada layer tersebut.

### **2.1.3. Kelas Neural Network**

Kelas ini merupakan implementasi dari Neural Network yang menggabungkan beberapa layer (input, hidden, output) untuk membentuk arsitektur yang lengkap. Kelas ini bertindak sebagai kontainer yang mengatur aliran data dari input hingga output dan mengelola proses pembelajaran.

```
import numpy as np
from dense_layer import DenseLayer
from loss_functions import LossFunction
from value import Value
import time
import matplotlib.pyplot as plt
from visualizer import visualize_ann
import pickle
from activations import *

# ANN Class
class NeuralNetwork:
    def __init__(self, loss_function_option):
        self.layers = []
        self.loss_functions = {
            "mse": LossFunction.mse,
            "binary_cross_entropy":
LossFunction.binary_cross_entropy,
            "categorical_cross_entropy":
LossFunction.categorical_cross_entropy,
        }
        self.loss_function =
self.loss_functions[loss_function_option]
        self.loss = loss_function_option
        self.last_gradients = []
        self.current_history = None
        self.n_features = None

    def visualize(self, output_dir=None, filename="ann"):
        if self.n_features is not None:
            visualize_ann(
                model=self,
                input_shape=self.n_features,
                filename=filename,
                output_dir=output_dir,
            )
```

```

def add_layer(self, layer):
    self.layers.append(layer)

def forward(self, input_data):
    if not isinstance(input_data, Value):
        x = Value(input_data)
    else:
        x = input_data

    for layer in self.layers:
        x = layer.forward(x)
    return x

def _progress_bar(
    self, current, total, bar_length=50,
    training_loss=None, val_loss=None
):
    progress = min(1.0, current / total)
    arrow = "=" * int(round(progress * bar_length)) - 1
    + ">"
    spaces = " " * (bar_length - len(arrow))

    if training_loss is not None and val_loss is not
None:
        print(
            f"\r[{arrow + spaces}] {int(progress * 100)}% - loss: {training_loss:.4f} - val_loss:
{val_loss:.4f}",
            end="",
        )
    else:
        print(f"\r[{arrow + spaces}] {int(progress * 100)}%", end="")

    if current == total:
        print()

def train(
    self,
    x,

```

```
Y,
epochs=1000,
learning_rate=0.01,
batch_size=32,
verbose=1,
validation_data=None,
isOneHot=False,
) :
    self.n_features = X.shape[1]
    n_samples = X.shape[0]
    n_batches = int(np.ceil(n_samples / batch_size))

    y_orig = Y.copy()

    if isOneHot:
        Y = one_hot(Y) if not hasattr(self, "_one_hot")
    else self._one_hot(Y)

    history = {"loss": [], "val_loss": [], "accuracy": [],
               "val_accuracy": []}

    has_validation = validation_data is not None

    if has_validation:
        X_val, Y_val = validation_data
        y_val_orig = Y_val.copy()
        if isOneHot:
            Y_val = (
                one_hot(Y_val)
                if not hasattr(self, "_one_hot")
                else self._one_hot(Y_val)
            )

    for epoch in range(epochs):
        epoch_start_time = time.time()
        total_loss = 0

        indices = np.random.permutation(n_samples)
        X_shuffled = X[indices]
        Y_shuffled = Y[indices]
```

```

        for batch_idx in range(n_batches):
            start = batch_idx * batch_size
            end = min(start + batch_size, n_samples)

            X_batch = X_shuffled[start:end]
            Y_batch = Y_shuffled[start:end]

            # Forward pass
            Y_pred = self.forward(X_batch)

            # Compute loss
            loss = self.loss_function(Y_batch, Y_pred)

            # Backward pass - compute gradients
            loss.backward()

            # Update weights using gradients
            self._update_parameters(learning_rate)

            total_loss += loss.data

            if verbose == 1:
                self._progress_bar(batch_idx + 1,
n_batches)

            avg_loss = total_loss / n_batches
            history["loss"].append(avg_loss)

            train_output = self.forward(X)
            train_pred = (
                np.argmax(train_output.data, axis=1)
                if train_output.data.shape[1] > 1
                else (train_output.data > 0.5).astype(int)
            )
            train_accuracy = np.sum(train_pred == y_orig) /
y_orig.size
            history["accuracy"].append(train_accuracy)

            val_loss = None

```

```

        val_accuracy = None
        if has_validation:
            val_output = self.forward(X_val)
            val_loss = self.loss_function(Y_val,
val_output).data
            history["val_loss"].append(val_loss)

            val_pred = (
                np.argmax(val_output.data, axis=1)
                if val_output.data.shape[1] > 1
                else (val_output.data > 0.5).astype(int)
            )
            val_accuracy = np.sum(val_pred ==
y_val_orig) / y_val_orig.size
            history["val_accuracy"].append(val_accuracy)

        epoch_time = time.time() - epoch_start_time

        if verbose == 1:
            if has_validation:
                print(
                    f"\rEpoch {epoch+1}/{epochs} - "
{epoch_time:.2f}s - loss: {avg_loss:.4f} - accuracy:
{train_accuracy:.4f} - val_loss: {val_loss:.4f} -
val_accuracy: {val_accuracy:.4f}"
                )
            else:
                print(
                    f"\rEpoch {epoch+1}/{epochs} - "
{epoch_time:.2f}s - loss: {avg_loss:.4f} - accuracy:
{train_accuracy:.4f}"
                )
        elif verbose > 1 and ((epoch + 1) % 10 == 0 or
epoch == 0):
            print(
                f"Epoch {epoch+1}/{epochs}, Loss:
{avg_loss:.4f}, Accuracy: {train_accuracy:.4f}"
            )

        self.current_history = history
    
```

```
        return history

    def _update_parameters(self, learning_rate):
        # Make sure last_gradient size is as loong as needed
        if len(self.last_gradients) < len(self.layers):
            for _ in range(len(self.layers) -
len(self.last_gradients)):
                self.last_gradients.append(None)

        for i, layer in enumerate(self.layers):
            if hasattr(layer, "weights") and hasattr(layer,
"biases"):
                self.last_gradients[i] =
layer.weights.grad.copy()

                layer.weights.data -= learning_rate *
layer.weights.grad
                layer.biases.data -= learning_rate *
layer.biases.grad

                layer.weights.grad =
np.zeros_like(layer.weights.data)
                layer.biases.grad =
np.zeros_like(layer.biases.data)

    def predict(self, X):
        return self.forward(X).data

    def plot_weight_distribution(self, layer_indices=None,
title=""):
        # Find layers with weights
        if layer_indices is None:
            layer_indices = [
                i for i, layer in enumerate(self.layers) if
hasattr(layer, "weights")
            ]
        num_layers = len(layer_indices)
        if num_layers == 0:
            print("No layers with weights found.")
```

```

        return

    fig, axes = plt.subplots(1, num_layers, figsize=(5 * num_layers, 5))
    fig.suptitle(title, fontsize=16)
    if num_layers == 1:
        axes = [axes]

    # Create subplot for each layer
    for i, layer_idx in enumerate(layer_indices):
        if layer_idx >= len(self.layers) or not hasattr(
            self.layers[layer_idx], "weights"
        ):
            print(f"Layer {layer_idx} does not exist or doesn't have weights.")
            continue

        weights =
self.layers[layer_idx].weights.data.flatten()

        axes[i].hist(weights, bins=50, alpha=0.7)
        axes[i].set_title(f"Layer {layer_idx} Weight Distribution")
        axes[i].set_xlabel("Weight Value")
        axes[i].set_ylabel("Frequency")
        axes[i].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    def plot_gradient_distribution(self, layer_indices=None,
log_scale=True, title=""):
        if not hasattr(self, "last_gradients"):
            print(
                "No gradients have been stored yet. Run at least one training step first."
            )
            return

        # Find all layers with weights

```

```

        if layer_indices is None:
            layer_indices = [
                i for i, layer in enumerate(self.layers) if
hasattr(layer, "weights")
            ]

        num_layers = len(layer_indices)
        if num_layers == 0:
            print("No layers with weights found.")
            return

        # Create subplot grid
        fig, axes = plt.subplots(1, num_layers, figsize=(5 *
num_layers, 5))
        fig.suptitle(title, fontsize=16)
        if num_layers == 1:
            axes = [axes]

        for i, layer_idx in enumerate(layer_indices):
            if layer_idx >= len(self.layers) or not hasattr(
                self.layers[layer_idx], "weights"
            ):
                print(f"Layer {layer_idx} does not exist or
doesn't have weights.")
                continue

            gradients =
self.last_gradients[layer_idx].flatten()

            n, bins, patches = axes[i].hist(gradients,
bins=50, alpha=0.7)

            # Set log if needed
            if log_scale:
                axes[i].set_yscale("log")

            axes[i].set_title(f"Layer {layer_idx} Gradient
Distribution")
            axes[i].set_xlabel("Gradient Value")
            axes[i].set_ylabel("Frequency (log scale)" if

```

```
log_scale else "Frequency")  
  
        axes[i].ticklabel_format(axis="x", style="sci",  
scilimits=(-4, 4))  
  
        axes[i].grid(True, alpha=0.3)  
  
    plt.tight_layout()  
    plt.show()  
  
def plot_training(self, title=""):  
    if self.current_history is not None:  
        fig, (ax1, ax2) = plt.subplots(1, 2,  
figsize=(15, 5))  
        fig.suptitle(title, fontsize=16)  
        ax1.plot(self.current_history["loss"],  
label="Train Loss")  
        if "val_loss" in self.current_history and  
self.current_history["val_loss"]:  
            ax1.plot(self.current_history["val_loss"],  
label="Validation Loss")  
            ax1.set_xlabel("Epoch")  
            ax1.set_ylabel("Loss")  
            ax1.set_title("Loss per Epoch")  
            ax1.legend()  
            ax1.grid(True)  
  
            # Accuracy plot  
            ax2.plot(self.current_history["accuracy"],  
label="Train Accuracy")  
            if (  
                "val_accuracy" in self.current_history  
                and self.current_history["val_accuracy"]  
) :  
                ax2.plot(  
                    self.current_history["val_accuracy"],  
label="Validation Accuracy"  
                )  
                ax2.set_xlabel("Epoch")  
                ax2.set_ylabel("Accuracy")
```

```

        ax2.set_title("Accuracy per Epoch")
        ax2.legend()
        ax2.grid(True)

        plt.tight_layout()
        plt.show()

    def save(self, filename):
        # Create a dictionary to store model state
        model_state = {
            "loss_function_option": (
                self.loss.__name__ if hasattr(self.loss,
"__name__") else str(self.loss)
            ),
            "layer_states": [],
            "last_gradients": [
                grad.tolist() if grad is not None else None
                for grad in self.last_gradients
            ],
            "current_history": self.current_history,
            "n_features": self.n_features,
        }

        # Save each layer state
        for layer in self.layers:
            if hasattr(layer, "weights") and hasattr(layer,
"biases"):
                layer_state = {
                    "type": "dense",
                    "weights": layer.weights.data,
                    "biases": layer.biases.data,
                    "output_size": layer.output_size,
                    "init": layer.init,
                    "bias_init": layer.bias_init,
                    "mean": layer.mean,
                    "var": layer.var,
                    "lower_bound": layer.lower_bound,
                    "upper_bound": layer.upper_bound,
                    "seed": layer.seed,
                    "reg_type": layer.reg_type,
                }

```

```

        "reg_param": layer.reg_param,
    }

        if hasattr(layer, "activation") and
layer.activation is not None:
            if layer.activation == relu:
                layer_state["activation"] = "relu"
            elif layer.activation == sigmoid:
                layer_state["activation"] =
"sigmoid"
            elif layer.activation == tanh:
                layer_state["activation"] = "tanh"
            elif layer.activation == softmax:
                layer_state["activation"] =
"softmax"
            elif layer.activation == linear:
                layer_state["activation"] = "linear"
            else:
                layer_state["activation"] =
"unknown"

model_state["layer_states"].append(layer_state)

with open(filename, "wb") as f:
    pickle.dump(model_state, f)

print(f"Model saved to {filename}")

@staticmethod
def load(filename):
    with open(filename, "rb") as f:
        model_state = pickle.load(f)

    loaded_model =
NeuralNetwork(model_state["loss_function_option"])
    if "last_gradients" in model_state:
        loaded_model.last_gradients = [
            np.array(grad) if grad is not None else None
            for grad in model_state["last_gradients"]
        ]

```

```
]

    if "current_history" in model_state:
        loaded_model.current_history =
model_state["current_history"]

    if "n_features" in model_state:
        loaded_model.n_features =
model_state["n_features"]

    activation_map = {
        "relu": relu,
        "sigmoid": sigmoid,
        "tanh": tanh,
        "softmax": softmax,
        "linear": linear,
    }

# Recreate the layers with the saved configurations
for layer_state in model_state["layer_states"]:
    if layer_state["type"] == "dense":
        activation_func = None
        if (
            "activation" in layer_state
            and layer_state["activation"] in
activation_map
        ):
            activation_func =
activation_map[layer_state["activation"]]

        layer = DenseLayer(
            output_size=layer_state["output_size"],
            activation=activation_func,
            init=layer_state["init"],
            bias_init=layer_state["bias_init"],
            mean=layer_state["mean"],
            var=layer_state["var"],
            lower_bound=layer_state["lower_bound"],
            upper_bound=layer_state["upper_bound"],
            seed=layer_state["seed"],
```

```

        reg_type=layer_state["reg_type"] ,
        reg_param=layer_state["reg_param"] ,
    )

    layer.weights =
Value(layer_state["weights"])
    layer.biases = Value(layer_state["biases"])

loaded_model.add_layer(layer)

print(f"Model loaded from {filename}")
return loaded_model


def print_parameters(layer, num_elements=5):
    print(f"First {num_elements} weights:
{layer.weights.flatten()[:num_elements]}")
    print(f"First {num_elements} biases:
{layer.biases.flatten()[:num_elements]}")



def get_predictions(A2):
    return np.argmax(A2, axis=1)


def get_accuracy(predictions, Y):
    return np.sum(predictions == Y) / Y.size


def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    return one_hot_Y

```

Kelas ini terdiri dari beberapa atribut:

1. layers: Menyimpan dense layer dari neural network

2. loss\_functions: Menyimpan loss function yang dapat digunakan neural network ini
3. loss\_function: Menyimpan fungsi loss function yang sedang digunakan neural network
4. loss: Menyimpan loss apa yang sedang digunakan neural network
5. last\_gradients: Menyimpan gradient terakhir hasil training
6. current\_history: Menyimpan loss hasil training data
7. n\_features: Menyimpan banyaknya fitur dari input data

Kelas ini memiliki beberapa method utama, antara lain :

1. \_\_init\_\_(): Menginisialisasi neural network dengan loss function spesifik. Membuat sebuah list kosong untuk layers dan juga menyiapkan tempat untuk history dan last gradient
2. add\_layer(layer): Menambahkan layer baru ke arsitektur neural network
3. forward(input\_data): Memproses input\_data melalui semua layer yang terdapat pada neural network.
4. train(): Melatih neural network menggunakan data yang diberikan
5. \_update\_parameters(learning\_rate): Mengubah bobot dan bias berdasarkan gradient yang dihitung
6. predict(x): Memasukkan data melalui neural network untuk mendapatkan hasil
7. save(filename): Menserialkan neural network menjadi sebuah file yang dapat di load kembali
8. load(filename): Fungsi statik yang akan membuat ulang neural network berdasarkan data serial yang terdapat pada save file
9. visualize(): Memvisualisasikan arsitektur dari neural network

#### 2.1.4. Kelas Loss Function

Kelas ini berisi implementasi dari loss function menggunakan Value class yang sudah dibuat

```
import numpy as np
from value import Value
```

```

class LossFunction:

    @staticmethod
    def mse(y_true, y_pred):
        if not isinstance(y_true, Value):
            y_true = Value(y_true)

        if len(y_true.data.shape) == 1 and
len(y_pred.data.shape) == 2:
            num_classes = y_pred.data.shape[1]
            batch_size = y_true.data.shape[0]

            one_hot = np.zeros((batch_size, num_classes))
            for i in range(batch_size):
                if 0 <= y_true.data[i] < num_classes:
                    one_hot[i, int(y_true.data[i])] = 1

            y_true = Value(one_hot)

        diff = y_pred - y_true
        squared = diff * diff
        return squared.mean()

    @staticmethod
    def binary_cross_entropy(y_true, y_pred):
        if not isinstance(y_true, Value):
            y_true = Value(y_true)

        # Add small epsilon to avoid log(0)
        epsilon = 1e-7
        clipped_pred = y_pred.clip(epsilon, 1 - epsilon)

        # Use multiplication by -1 instead of negation
        term1 = y_true * clipped_pred.log()
        term2 = (1 - y_true) * (1 - clipped_pred).log()
        loss = (term1 + term2) * (-1)  # Multiply by -1
        instead of using negation

        return loss.mean()

    @staticmethod

```

```

def categorical_crossentropy(y_true, y_pred):
    if not isinstance(y_true, Value):
        y_true = Value(y_true)

    # Add small epsilon to avoid log(0)
    epsilon = 1e-7
    clipped_pred = y_pred.clip(epsilon, 1 - epsilon)

    # Use multiplication by -1 instead of negation
    log_probs = clipped_pred.log()
    weighted_log_probs = y_true * log_probs
    summed = weighted_log_probs.sum(axis=1)
    loss = summed * (-1)

    return loss.mean()

# Wont be used with autodiff
@staticmethod
def mse_derivative(y_true, y_pred):
    return 2 * (y_pred - y_true) / y_true.size

@staticmethod
def binary_crossentropy_derivative(y_true, y_pred,
epsilon=1e-15):
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return ((1 - y_true) / (1 - y_pred) - y_true /
y_pred) / np.size(y_true)

@staticmethod
def categorical_crossentropy_derivative(y_true,
y_pred):
    return (y_pred - y_true) / y_true.shape[0]

```

### 2.1.5. Kelas Activation

Kelas ini merupakan implementasi dari fungsi aktivasi.

```
class Activation:
    def __init__(self, activation_function, name):
        self.activation_function = activation_function
        self.name = name

    def __call__(self, x):
        return self.activation_function(x)

    def forward(self, x):
        return self.activation_function(x)

    def get_name(self):
        return self.name

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

def sigmoid_activation(x):
    return x.sigmoid()

def relu_activation(x):
    return x.relu()

def linear_activation(x):
    return x.linear()

def softmax_activation(x):
    return x.softmax()

tanh = Activation(tanh_activation, 'tanh')
sigmoid = Activation(sigmoid_activation, 'sigmoid')
relu = Activation(relu_activation, 'relu')
linear = Activation(linear_activation, 'linear')
softmax = Activation(softmax_activation, 'softmax')
```



Kelas ini terdiri dari beberapa atribut:

1. activation\_function: Menyimpan fungsi aktivasi yang digunakan
2. name: Menyimpan nama dari fungsi aktivasi yang digunakan

Kelas ini memiliki beberapa method utama, antara lain :

1. \_\_init\_\_(activation\_function, name): Menginisialisasi atribut dari kelas ini
2. \_\_call\_\_(x): Membuat object ini dapat dipanggil bagaikan dia adalah sebuah fungsi dimana ia akan apply fungsi aktivasinya terhadap x
3. forward(x): Berfungsi sama dengan call hanya beda penamaan
4. get\_name(): Mengembalikan nama fungsi aktivasi yang digunakan

## 2.2. Hasil Pengujian

### 2.2.1. Pengaruh *Depth & Width*

Pengujian terhadap pengaruh *Depth & Width* dilakukan melalui dua tes utama, tes pertama dilakukan dengan menjaga *Depth* (jumlah layer yang ada) dan memvariasikan *width* (*jumlah neuron dalam setiap hidden layer*), test kedua dilakukan dengan menjaga *width* dan memvariasikan *depth*.

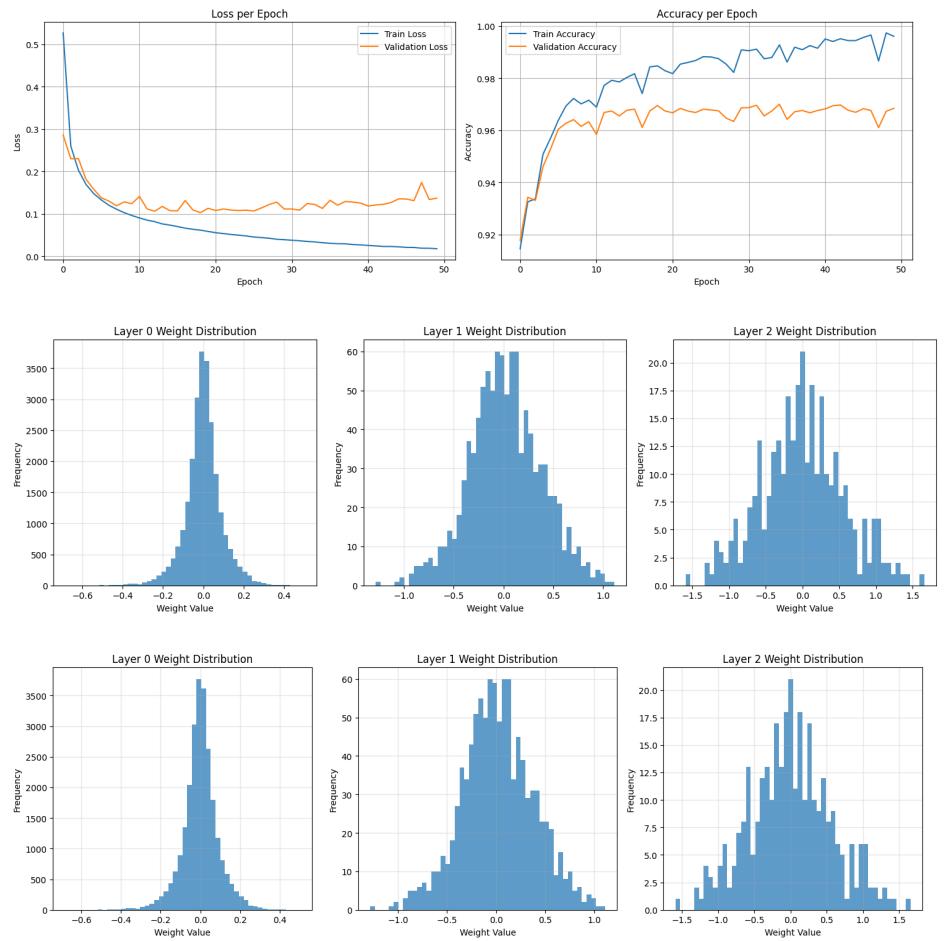
#### 2.2.1.1. Pengaruh Width

Tes pertama ini dilakukan melalui beberapa kriteria dan sub tes lainnya, yakni :

- Jumlah hidden layer : 2
- Tes 1 : 32 neuron setiap layer
- Tes 2 : 64 neurons setiap layer
- Tes 3 : 128 neuron setiap layer

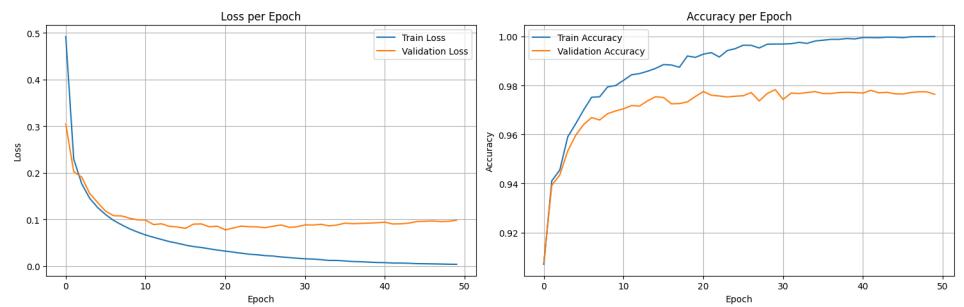
##### i. Tes 1 - 32 neuron setiap layer

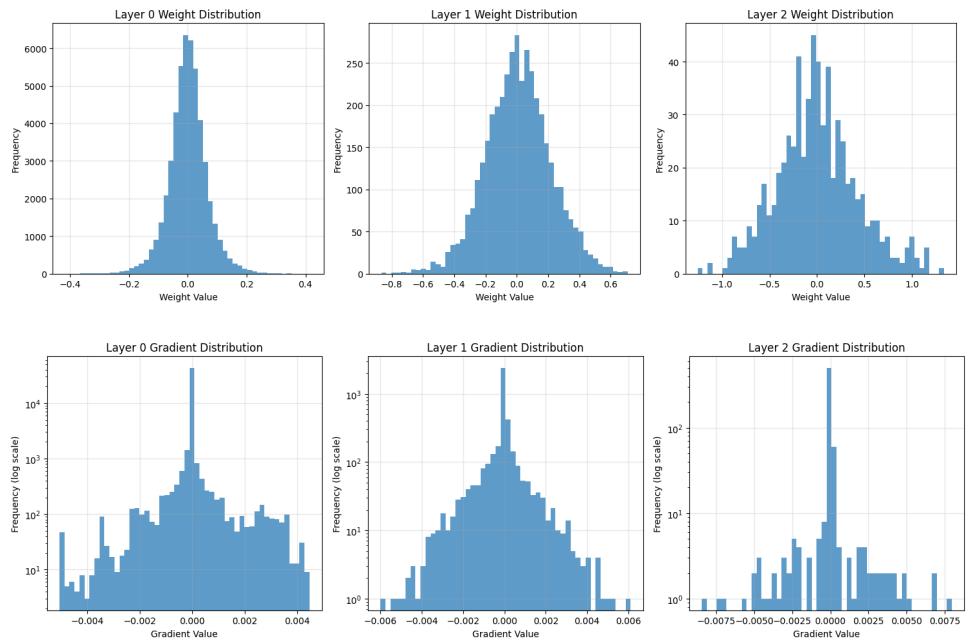
*Validation accuracy:* 0.9684



## ii. Tes 2 - 64 neuron setiap layer

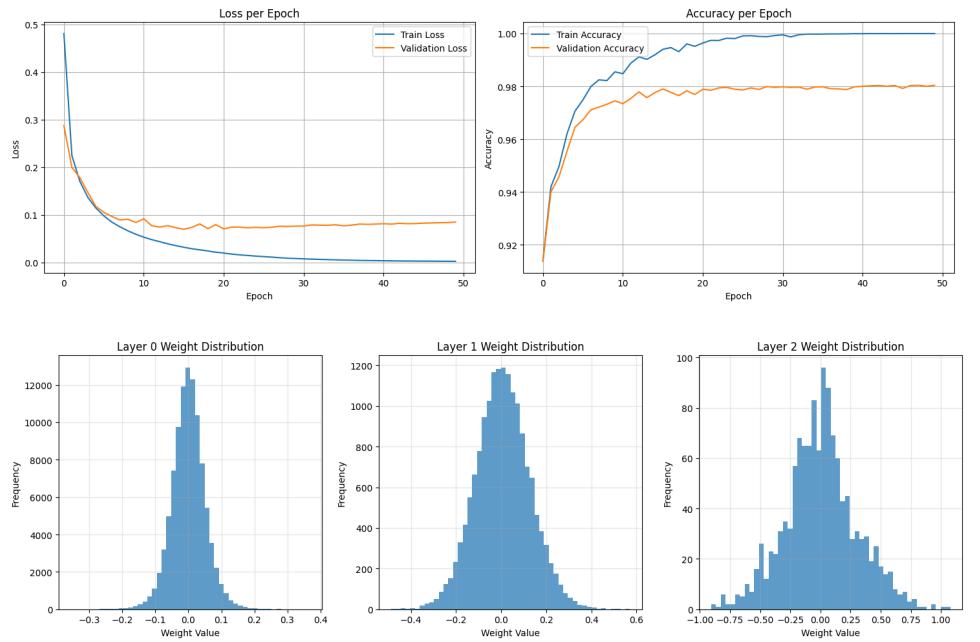
*Validation accuracy : 0.9764*

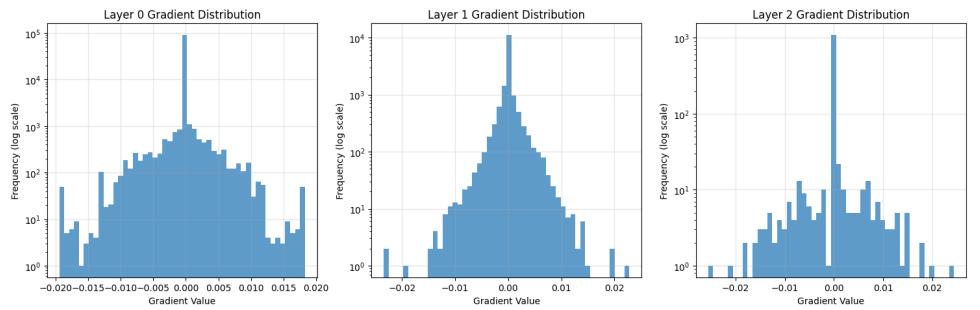




### iii. Tes 3 - 128 neuron setiap layer

*Validation accuracy : 0.9804*





#### iv. Kesimpulan - Pengaruh *Width*

Berdasarkan eksperimen kami menggunakan berbagai model dengan *depth* yang bervariasi (jumlah neuron per *layer*), jaringan yang lebih luas menunjukkan kinerja yang lebih baik, dengan akurasi validasi yang lebih tinggi dan kurva pembelajaran yang lebih stabil. Visualisasi masing - masing model tersebut mengungkapkan:

1. Jaringan yang lebih besar belajar lebih baik: Jaringan dengan lebih banyak neuron mencapai akurasi yang lebih tinggi. Jaringan tersebut juga belajar lebih cepat dan lebih lancar.
2. Prediksi yang lebih baik: Jaringan dengan lebih banyak neuron memberikan prediksi yang lebih baik, dengan model ketiga (jaringan terluas) mencapai akurasi tertinggi sebesar 98,04%.
3. *Smoothen loss curves*: Jaringan yang lebih luas menunjukkan *final loss values* yang lebih rendah dan kurva *loss* yang lebih mulus, sementara jaringan yang lebih sempit (jumlah neuron sedikit) menunjukkan lebih banyak fluktuasi.
4. Efek pada distribusi bobot: Jaringan yang lebih luas memiliki distribusi bobot yang lebih berpusat di sekitar nol

### 2.2.1.2.

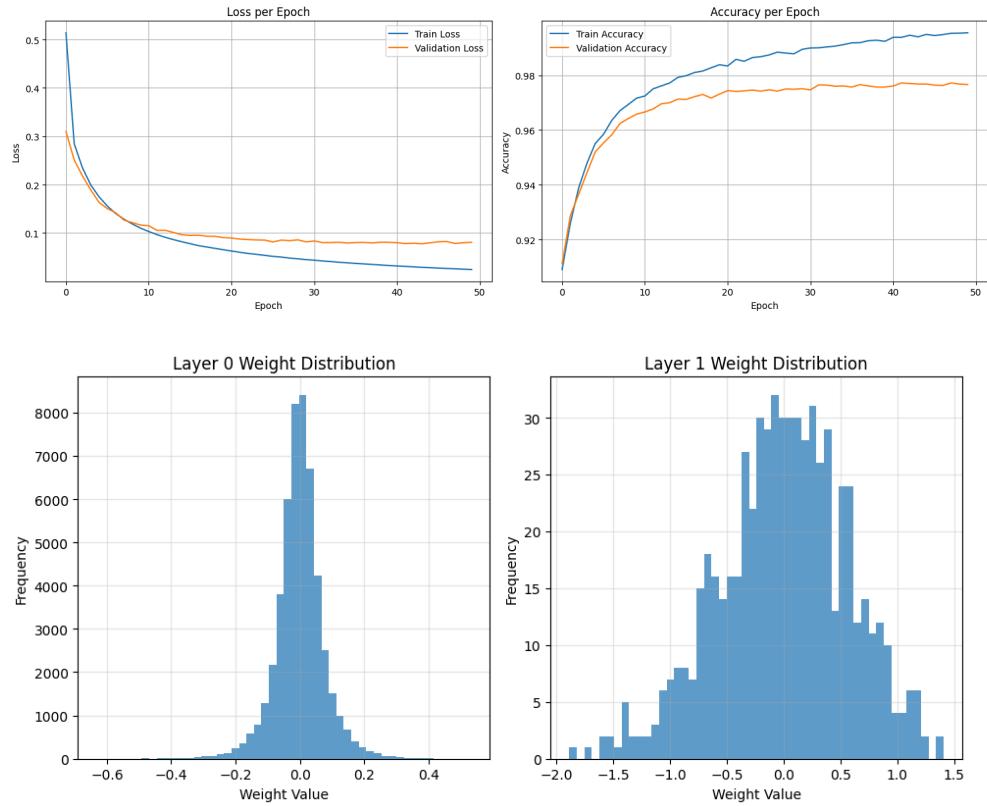
#### Pengaruh Depth

Tes kedua ini dilakukan melalui beberapa kriteria dan sub tes lainnya, yakni :

- Jumlah neuron setiap *layer*: 64
- Tes 1 : 1 *hidden layer*
- Tes 2 : 3 *hidden layers*
- Tes 3 : 5 *hidden layers*

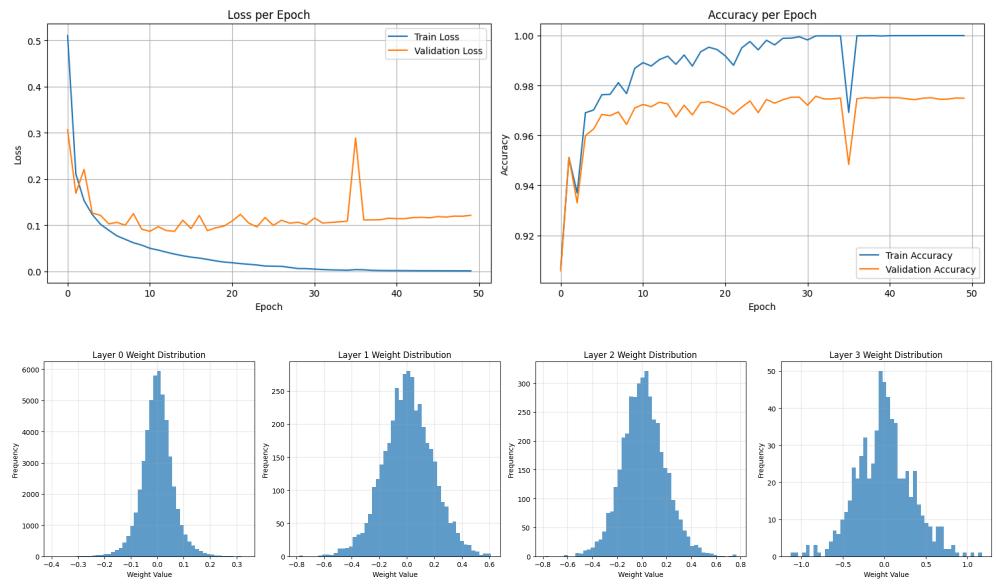
##### i. Tes 1 - 1 *hidden layer*

*Validation accuracy*: 0.9766



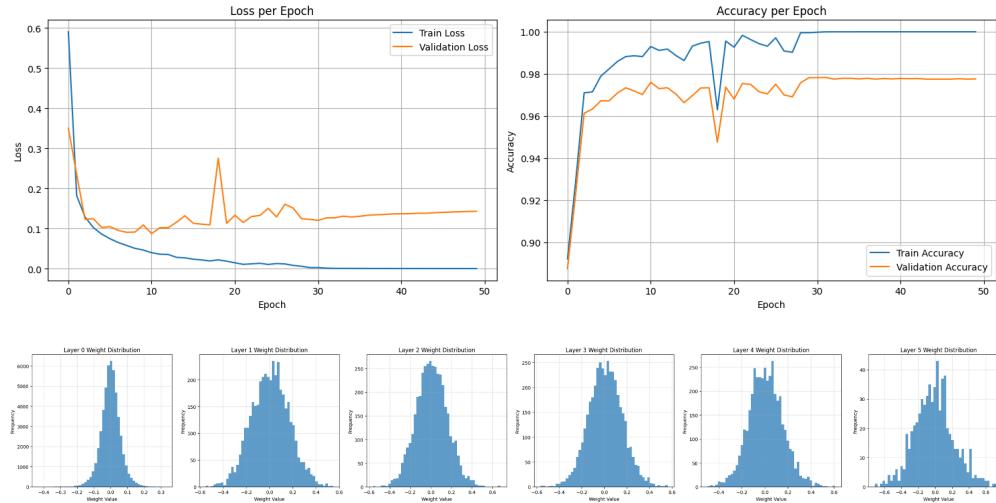
##### ii. Tes 2 - 3 *hidden layer*

*Validation accuracy*: 0.9749



### iii. Tes 3 - 5 hidden layer

*Validation accuracy: 0.9776*



### iv. Kesimpulan - Pengaruh *Depth*

Berdasarkan eksperimen yang dilakukan pada model dengan kedalaman (jumlah layer) yang bervariasi, jaringan yang lebih dalam tidak selalu menghasilkan model yang lebih baik. Bahkan, jaringan yang lebih dalam justru menunjukkan **ketidakstabilan** yang lebih besar pada grafik loss-nya, yang menandakan bahwa pola pelatihan yang kurang stabil. Visualisasi model menunjukkan :

1. Peningkatan performa tidak selalu *one to one* dengan penambahan *layer*: Menambahkan lebih banyak layer tidak selalu membuat model yang memiliki performa yang lebih baik. Model pertama, yang hanya memiliki satu *hidden layer*, justru mencapai akurasi lebih tinggi dibandingkan model kedua yang memiliki tiga *hidden layer*.
2. Pelatihan yang tidak stabil: Penambahan jumlah layer cenderung menyebabkan **fluktuasi** yang lebih drastis pada grafik loss. Hal ini menunjukkan bahwa jaringan yang lebih dalam mengalami pola pelatihan yang lebih tidak stabil.

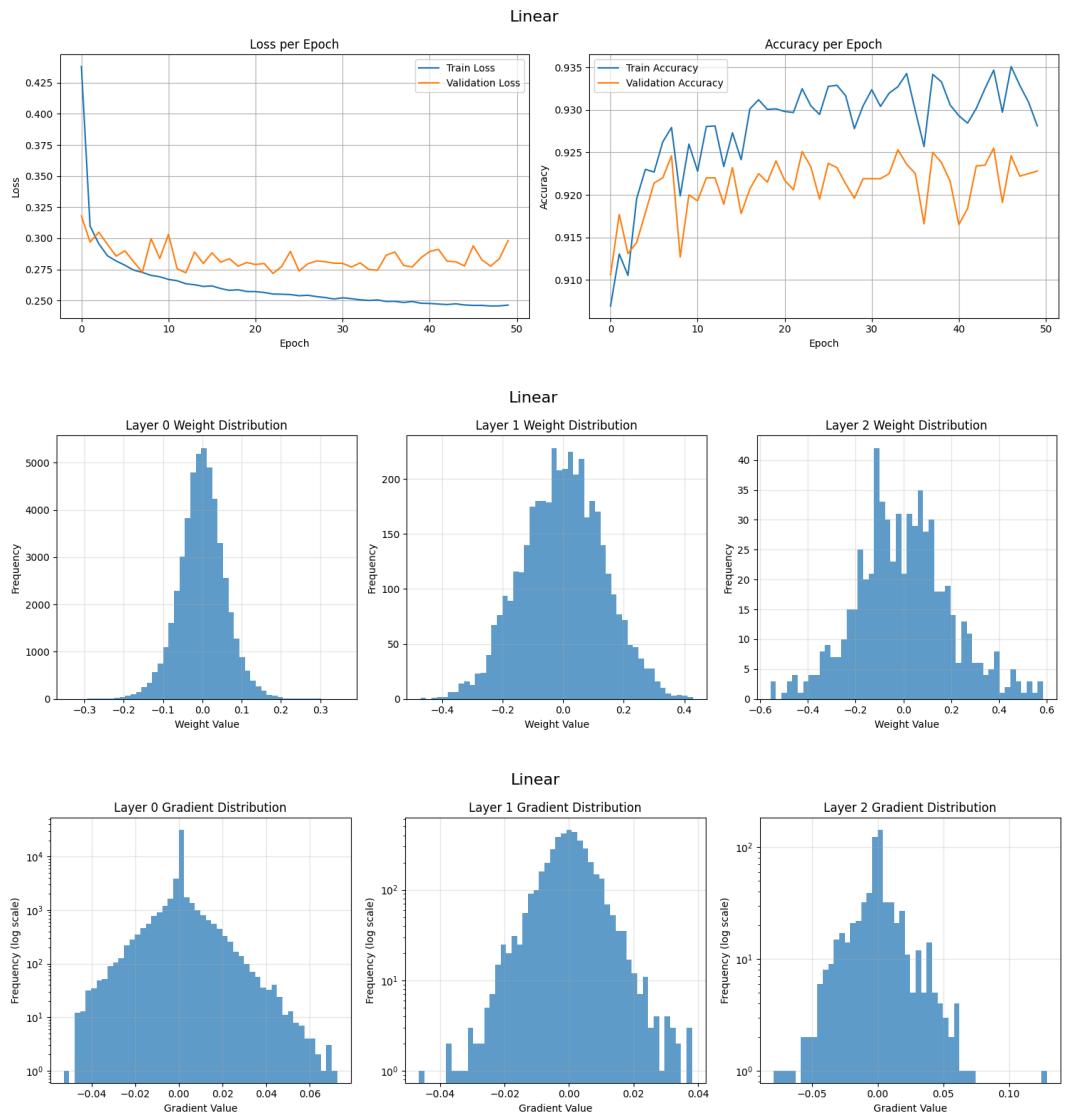
### 2.2.2. Pengaruh Fungsi Aktivasi

Pengujian terhadap pengaruh Fungsi Aktivasi dilakukan melalui empat tes dengan kriteria berikut :

- 2 hidden layers
- 64 neurons per layer
- Tes 1 : Linear
- Tes 2 : ReLu
- Tes 3 : Sigmoid
- Tes 4 : Tanh

#### 2.2.2.1. Tes 1 - Linear Activation Function

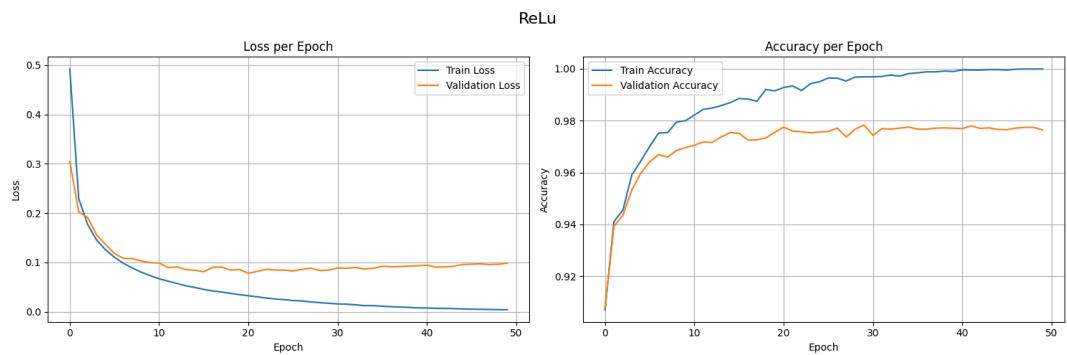
*Validation accuracy: 0.9228*

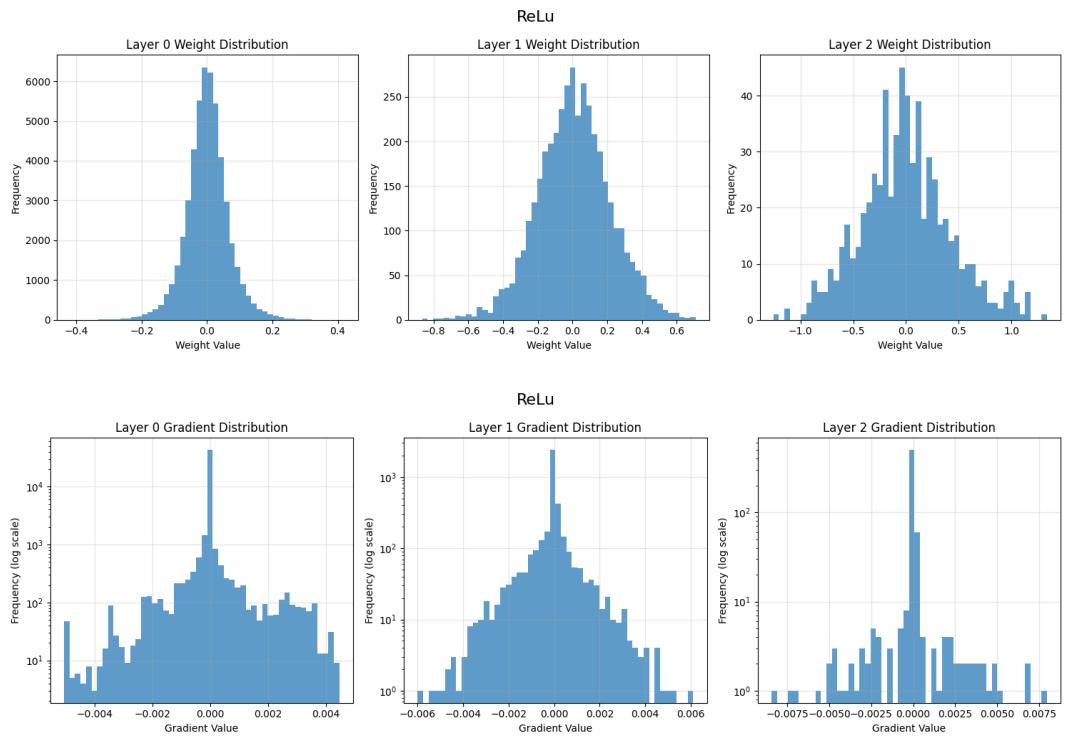


### 2.2.2.2.

### Tes 2 - ReLU Activation Function

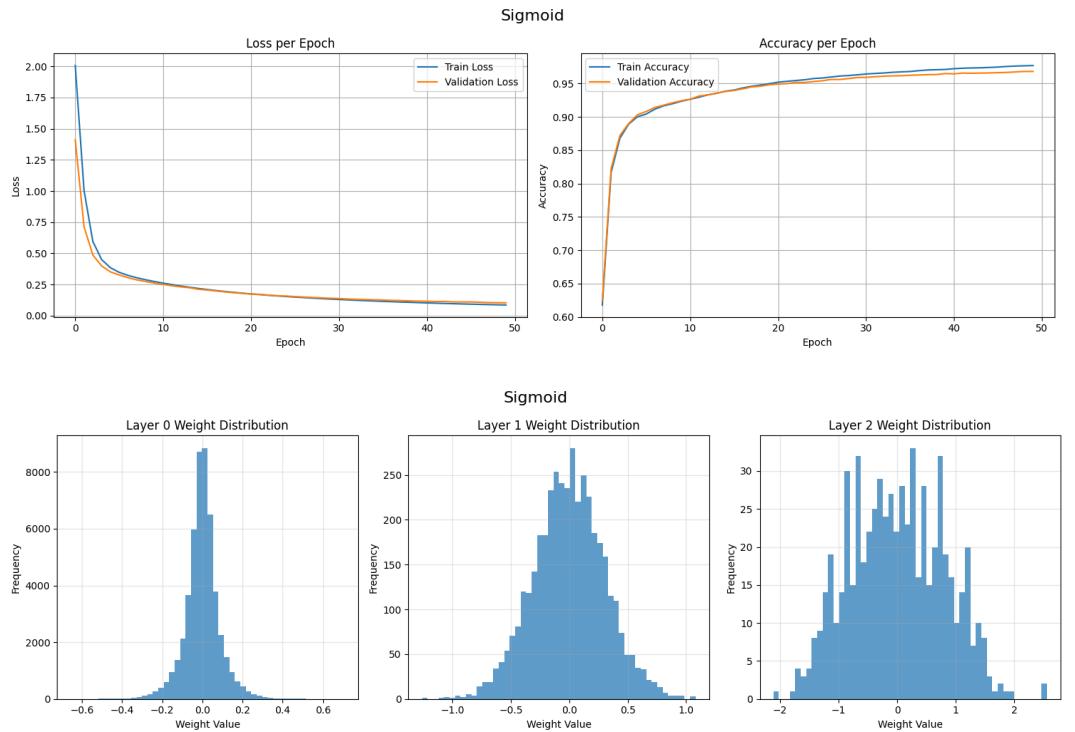
*Validation accuracy: 0.9764*

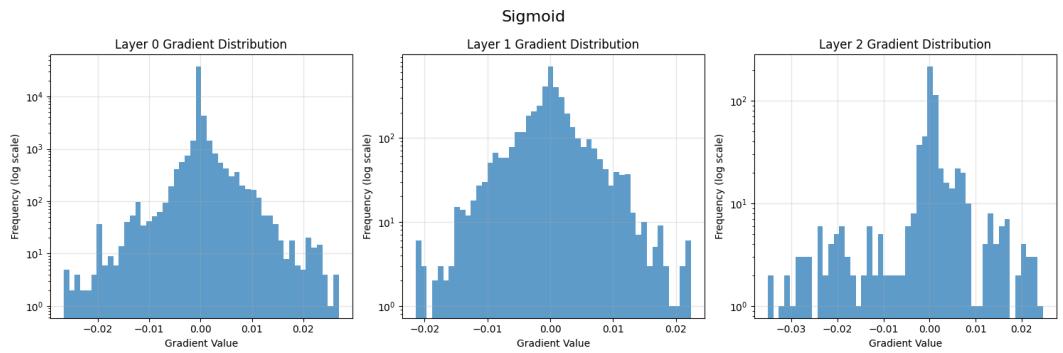




### 2.2.2.3. Tes 3 - Sigmoid Activation Function

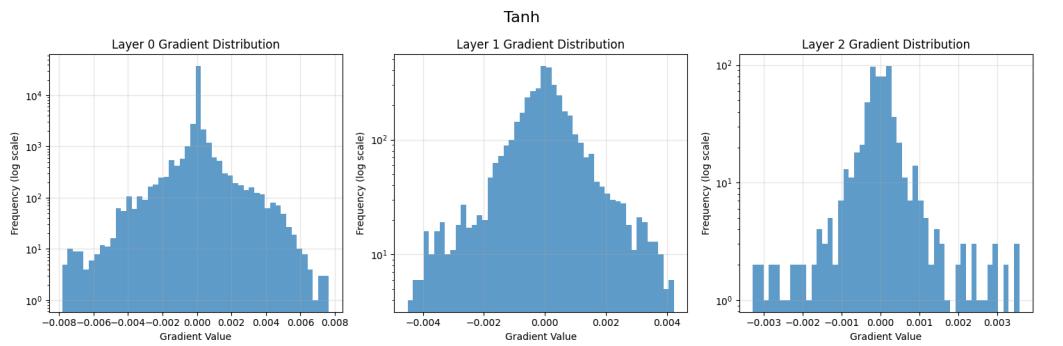
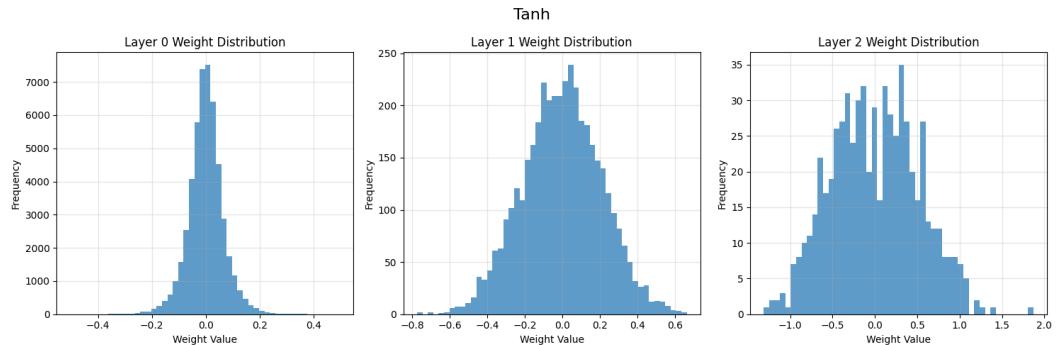
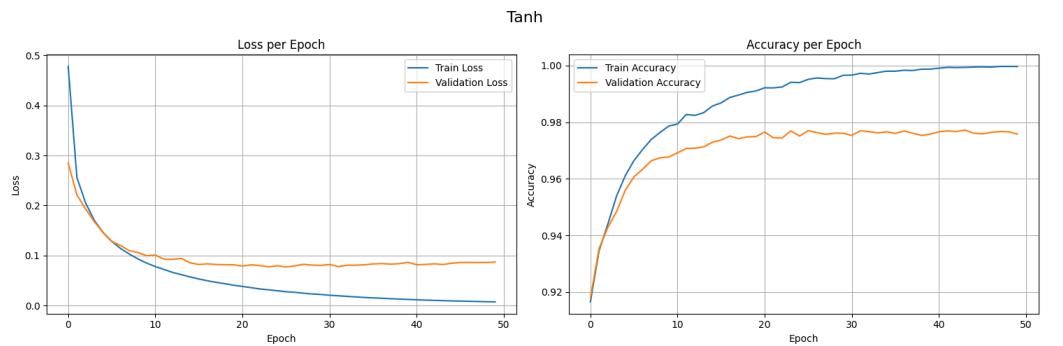
*Validation accuracy: 0.968*





#### 2.2.2.4. Tes 4 - Tanh Activation Function

*Validation accuracy: 0.9758*



### **2.2.2.5.**

#### **Kesimpulan - Activation Function**

Berdasarkan eksperimen yang kami lakukan pada model dengan berbagai fungsi aktivasi, kami menemukan bahwa fungsi aktivasi yang berbeda memberikan dampak yang signifikan terhadap performa model. Visualisasi kami mengungkapkan empat temuan utama berikut:

1. Prediksi Akhir (Akurasi Validasi):
  - a. ReLU: 97.64%
  - b. Tanh: 97.58%
  - c. Sigmoid: 96.8%
  - d. Linear: 92.28%
2. Pola *training loss* :
  - a. ReLU dan Tanh memiliki kurva loss pelatihan yang mirip, keduanya menunjukkan bahwa mereka proses belajar yang efisien (dimulai dari sekitar 0.4–0.5 dan menurun dengan cepat).
  - b. Sigmoid menunjukkan pembelajaran awal yang lebih lambat, namun pada akhirnya mencapai performa yang cukup baik.
  - c. Linear menunjukkan kurva pembelajaran yang tidak stabil dan berfluktuasi secara signifikan.
3. Distribusi Bobot (Weight):
  - a. Sigmoid menghasilkan distribusi bobot yang paling lebar, terutama pada layer *output*
  - b. ReLU dan Tanh menunjukkan distribusi bobot yang lebih *moderate*, dengan Tanh sedikit lebih menyebar pada layer *output*
  - c. Linear mempertahankan distribusi bobot yang paling sempit, terkonsentrasi di sekitar nol
4. Distribusi Gradien:
  - a. Linear memiliki gradien yang sangat besar, menyebabkan proses pelatihan menjadi tidak stabil

- b. Sigmoid menunjukkan terjadinya *vanishing gradient* (terlihat dari grafik, gradien semakin terkonsentrasi di sekitar nol pada layer-layer yang lebih dalam)
- c. ReLU dan Tanh mempertahankan magnitudo gradien yang seimbang di seluruh layer

Performa dari ReLU dan Tanh yang lebih bagus dibandingkan dengan model lainnya dapat dikaitkan dengan kemampuan mereka dalam menjaga gradien yang sehat sambil memberikan keragaman yang cukup. Distribusi gradien yang seimbang, distribusi bobot yang sesuai, serta kurva pembelajaran yang halus dan efisien berkontribusi terhadap performa yang lebih tinggi.

Performa Sigmoid yang sedikit lebih rendah disebabkan oleh masalah *vanishing gradient*, sementara performa Linear yang buruk membuktikan diperlukannya *non-linearity* dalam sebuah model ANN yang baik.

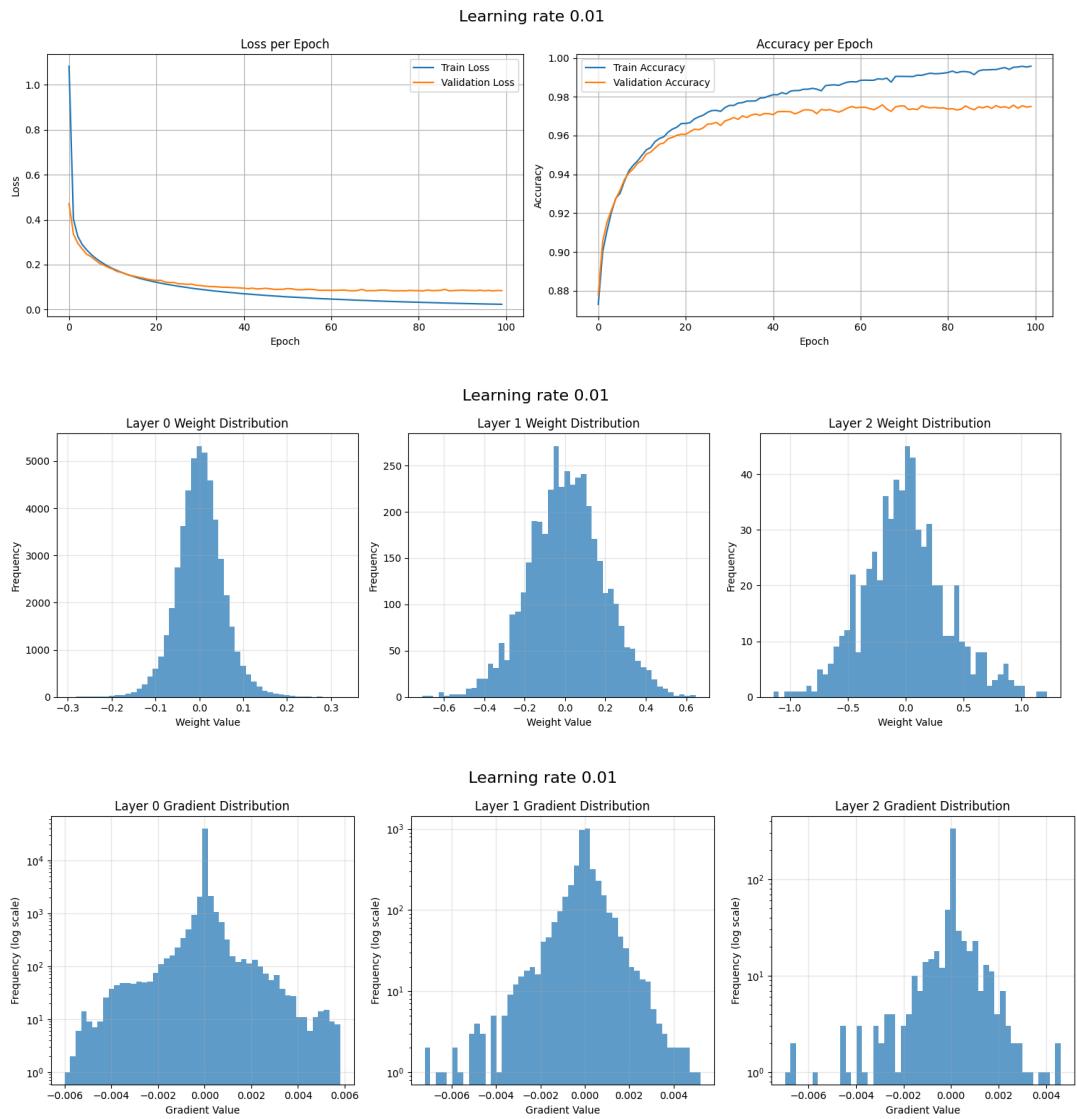
### **2.2.3. Pengaruh Learning Rate**

Pengujian terhadap pengaruh Learning rate dilakukan melalui tiga tes dengan kriteria berikut :

- 2 hidden layers
- 64 neurons per layer
- Tes 1 : Learning rate 0.01
- Tes 2 : Learning rate 0.05
- Tes 3 : Learning rate 0.25

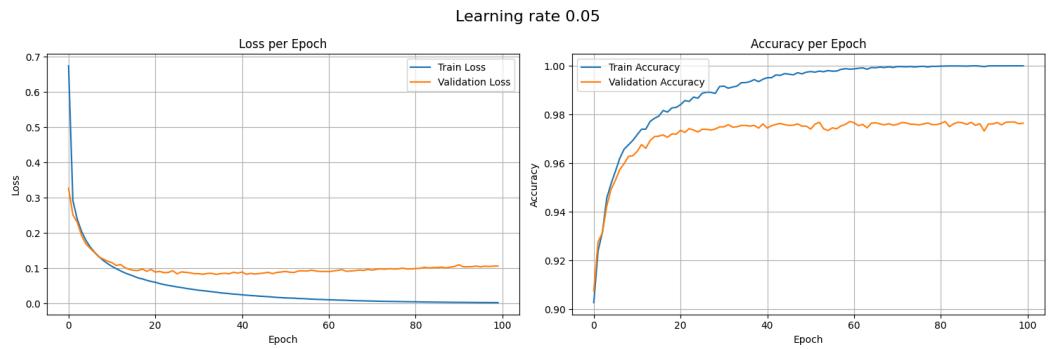
#### **2.2.3.1. Tes 1 - Learning rate 0.01**

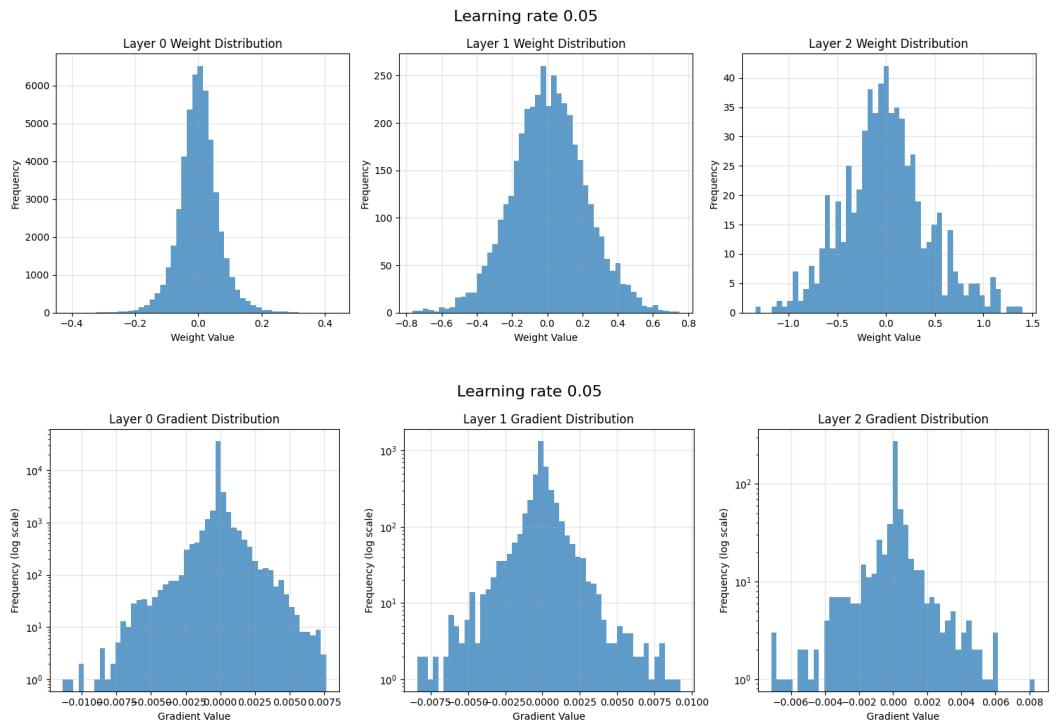
*Validation accuracy:* 0.975



### 2.2.3.2. Tes 2 - Learning rate 0.05

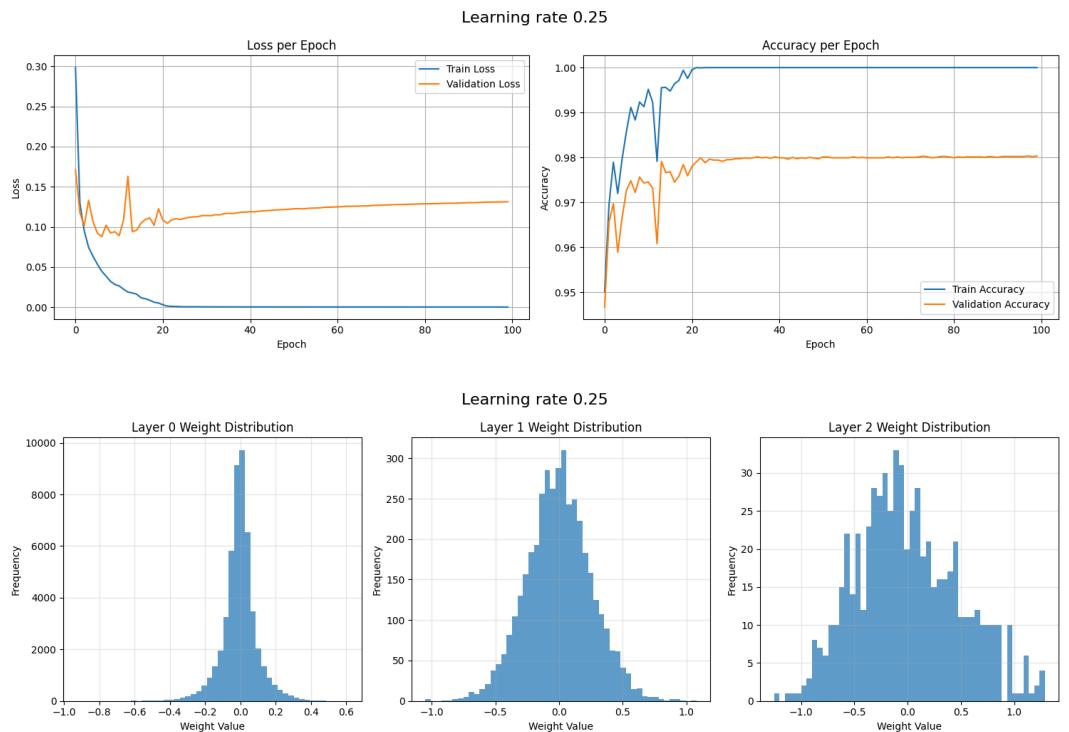
*Validation accuracy: 0.9764*

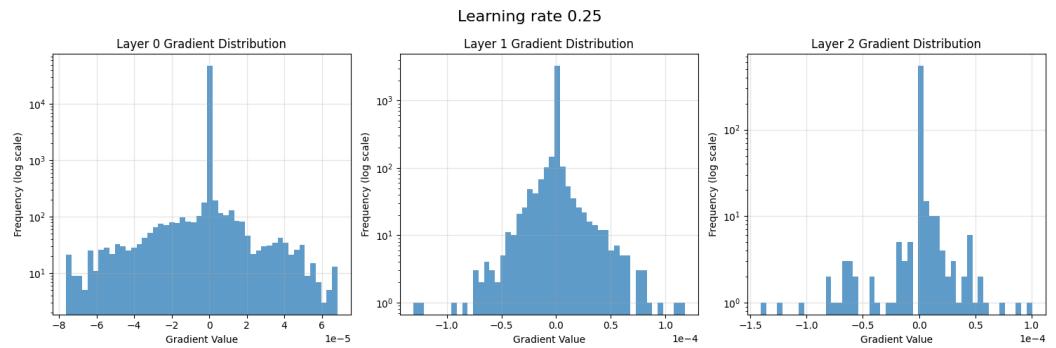




### 2.2.3.3. Tes 3 - Learning rate 0.25

*Validation accuracy:* 0.9803





#### 2.2.3.4. Kesimpulan - Pengaruh Learning Rate

Berdasarkan eksperimen kami terhadap model dengan berbagai nilai *learning rate*, kami menemukan bahwa learning rate mempengaruhi performa model, pola konvergensi, dan hasil akhir prediksi. Visualisasi kami mengungkapkan:

1. Prediksi Akhir (Akurasi Validasi):
  - a. Learning rate 0.01 : 97.50%
  - b. Learning rate 0.05 : 97.64%
  - c. Learning rate 0.25 : 98.03%
2. Pola *training loss* :
  - a. Learning rate 0.01 : Konvergensi lambat namun stabil, dengan fluktuasi yang minimum
  - b. Learning rate 0.05 : Konvergensi lebih cepat, tetapi tetap stabil
  - c. Learning rate 0.25 : Konvergensi paling cepat, awalnya tidak stabil namun akhirnya stabil
3. Distribusi Bobot (Weight):
  - a. Learning rate yang lebih tinggi menunjukkan distribusi bobot yang lebih terkonsentrasi (terlihat jelas pada learning rate 0.25 di layer input yang terkonsentrasi di sekitar 0)
  - b. Distribusi bobot menjadi lebih menyebar pada layer yang lebih dalam

4. Distribusi Gradien:

- a. Learning rate 0.01 : Gradien tinggi (-0.006 hingga 0.006)
- b. Learning rate 0.05 : Gradien tinggi (-0.006 hingga 0.008)
- c. Learning rate 0.25 : Gradien rendah (sekitar 1e-4)
- d.

Eksperimen ini menunjukkan bahwa learning rate yang lebih rendah tidak selalu menghasilkan akurasi yang lebih tinggi. Learning rate 0.25 kemungkinan memberikan hasil lebih baik karena membantu model untuk menghindari local minima secara lebih efektif, serta menjaga gradien tetap kecil.

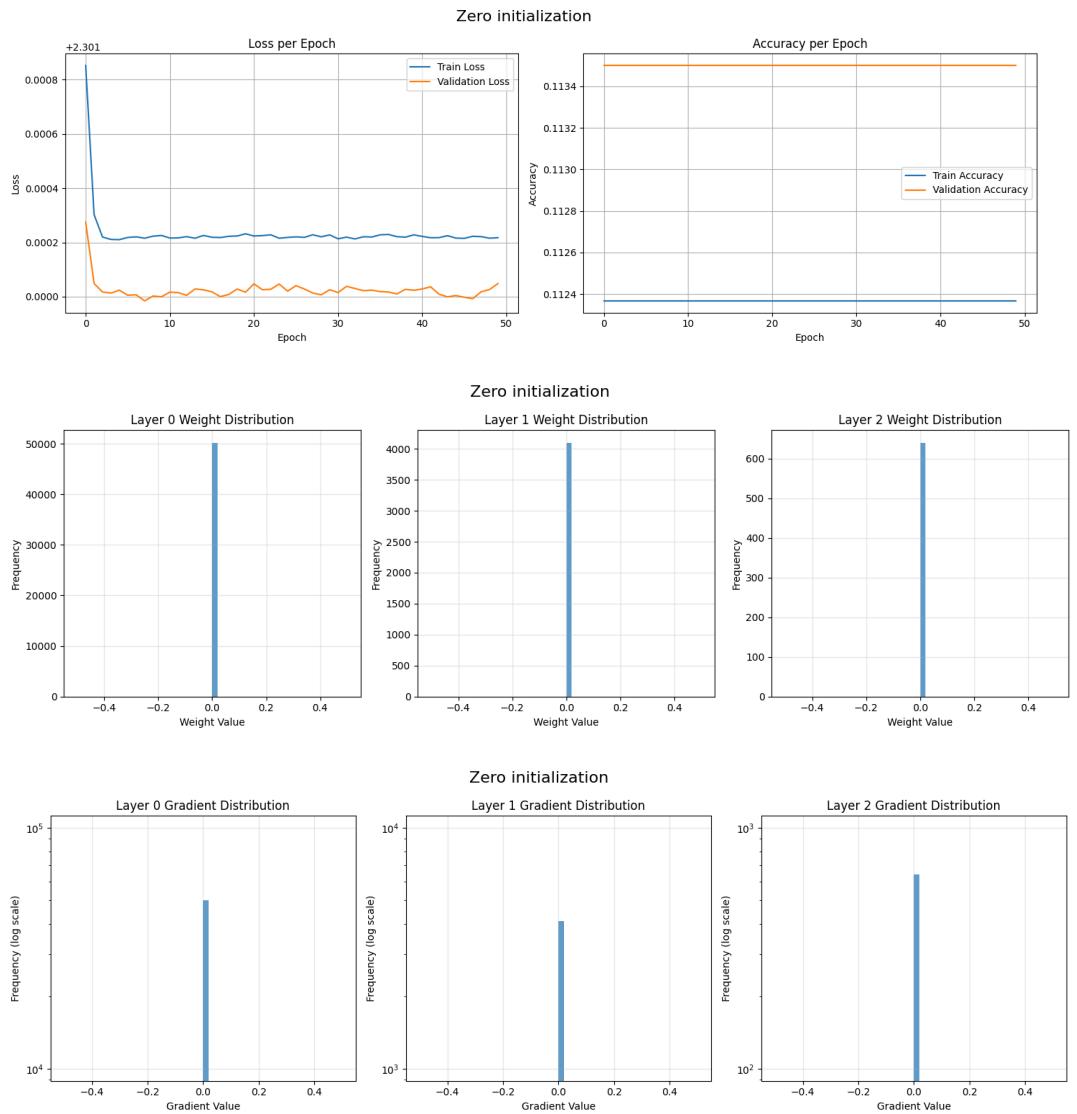
#### **2.2.4. Pengaruh Inisialisasi Bobot**

Pengujian terhadap pengaruh Inisialisasi bobot dilakukan melalui tiga tes dengan kriteria berikut :

- 2 hidden layers
- 64 neurons per layer
- Tes 1 : Inisialisasi Zero
- Tes 2 : Inisialisasi Uniform
- Tes 3 : Inisialisasi Normal
- Tes 4 : Inisialisasi Xavier
- Tes 5 : Inisialisasi He

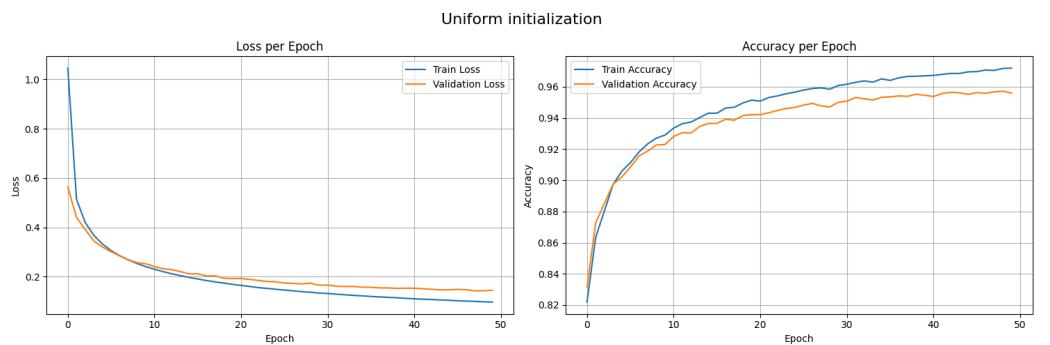
##### **2.2.4.1. Tes 1 - Inisialisasi Zero**

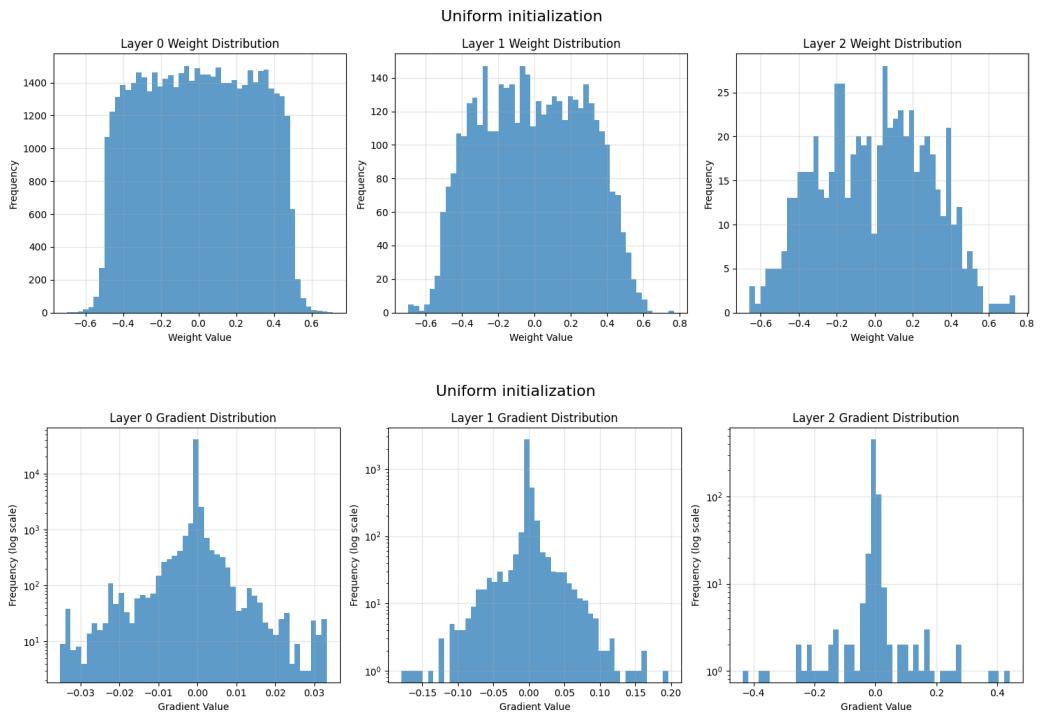
*Validation accuracy: 0.1135*



#### 2.2.4.2. Tes 2 - Inisialisasi Uniform

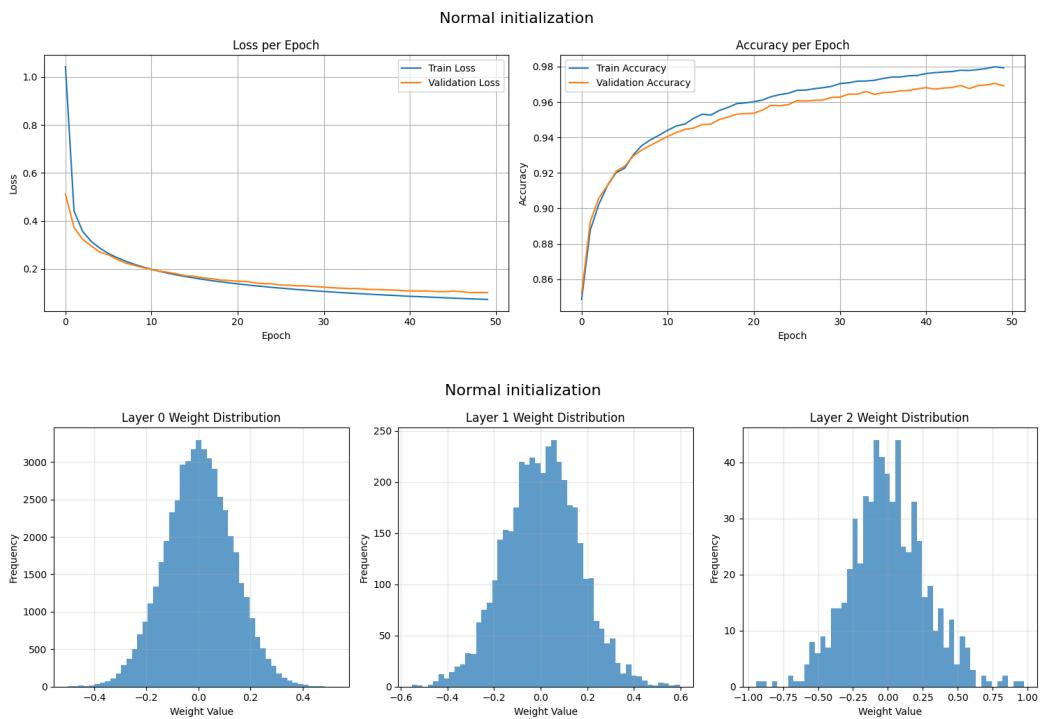
*Validation accuracy: 0.9559*

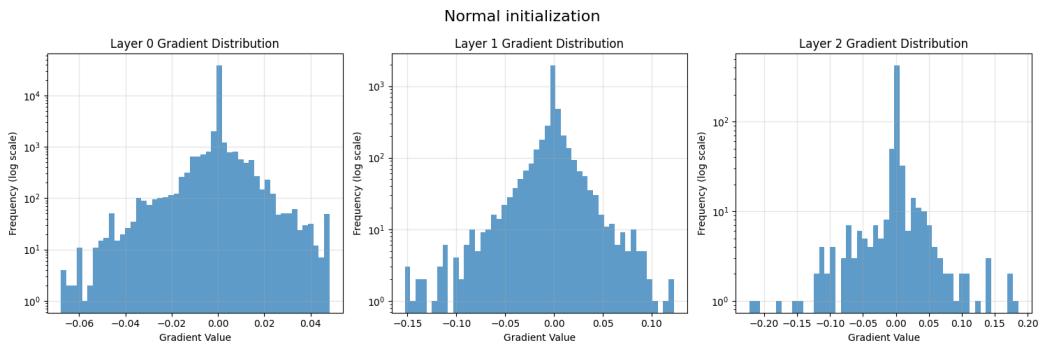




### 2.2.4.3. Tes 3 - Inisialisasi Normal

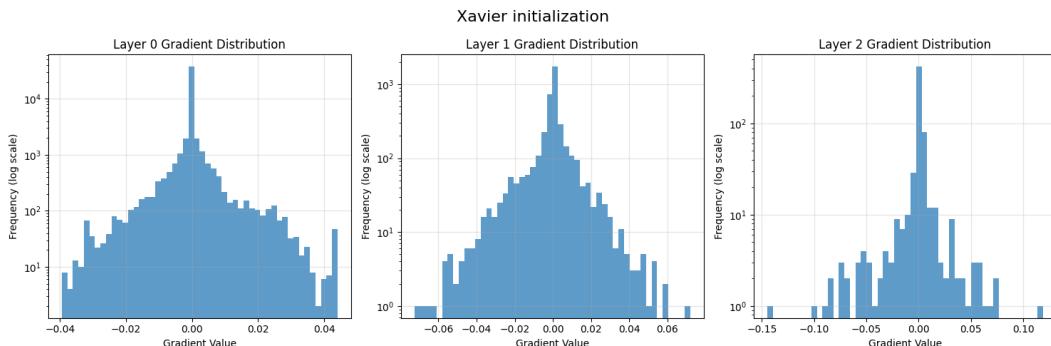
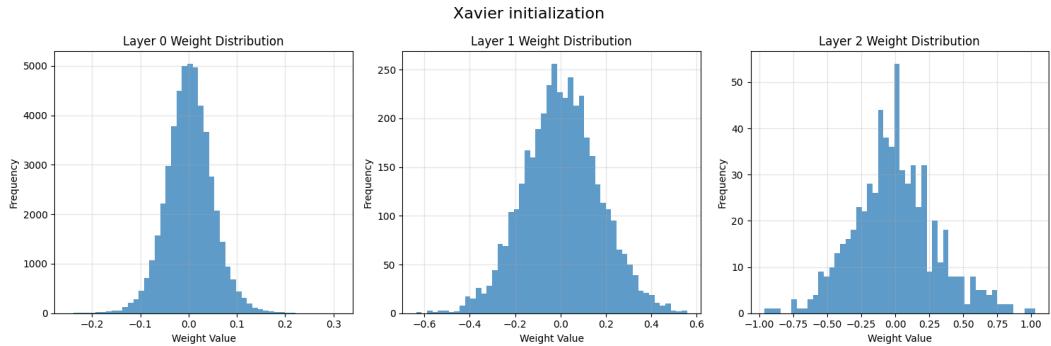
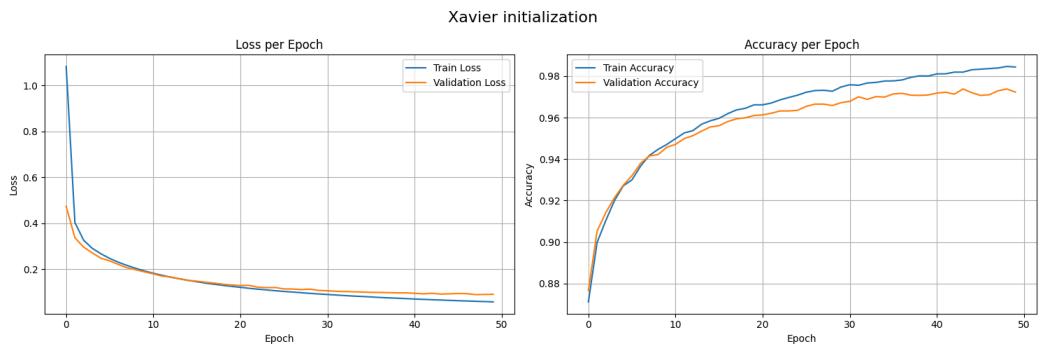
*Validation accuracy:* 0.9692





#### 2.2.4.4. Tes 4 - Inisialisasi Xavier

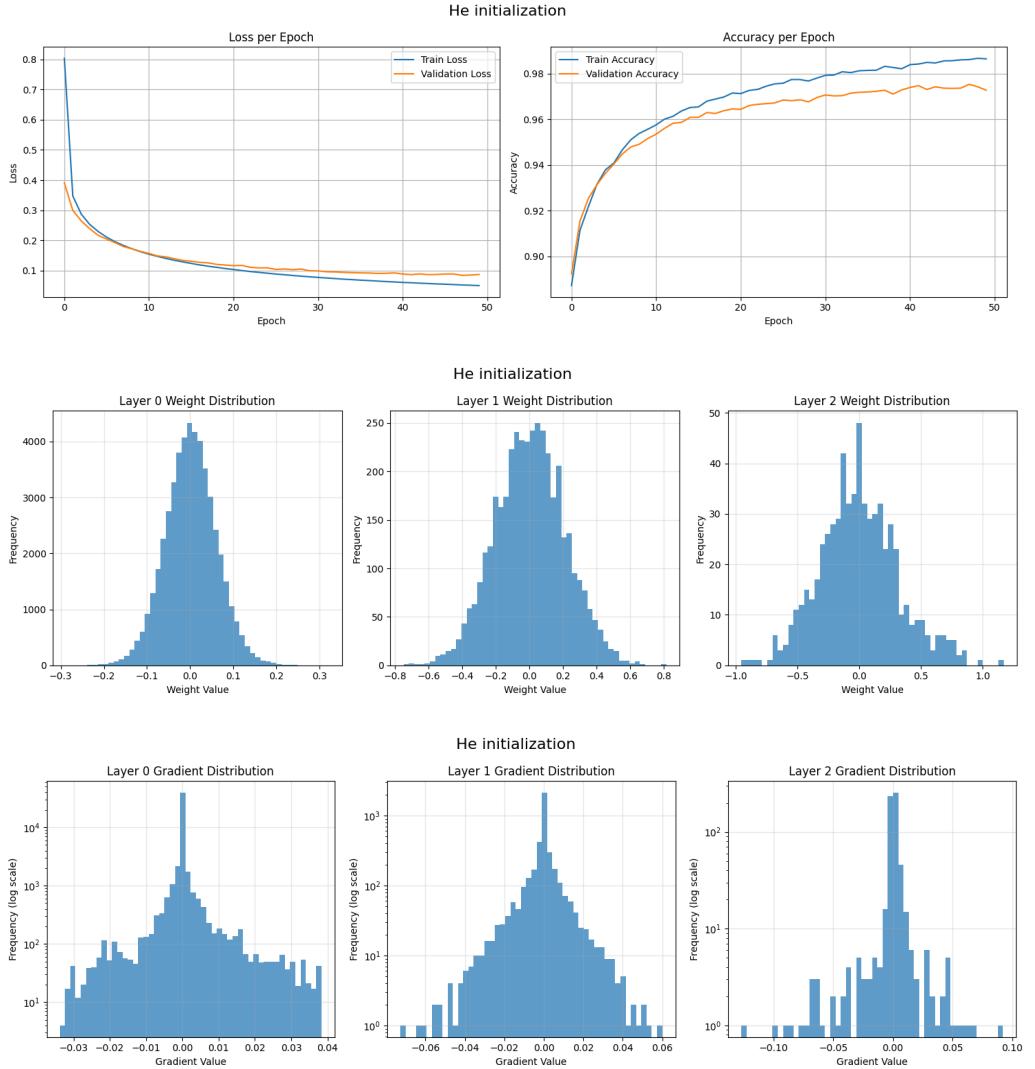
*Validation accuracy: 0.9723*



#### 2.2.4.5.

#### Tes 5 - Inisialisasi He

*Validation accuracy: 0.9728*



#### 2.2.4.6.

#### Kesimpulan - Pengaruh Inisialisasi Bobot

Berdasarkan eksperimen kami terhadap model dengan berbagai metode inisialisasi bobot, kami menemukan bahwa inisialisasi bobot memiliki pengaruh yang signifikan terhadap performa model. Visualisasi kami mengungkapkan berikut:

1. Prediksi Akhir (Akurasi Validasi):
  - a. Zero initialization : 11.35%
  - b. Uniform initialization : 95.59%

- c. Normal initialization : 96.92%
- d. Xavier initialization : 97.23%
- e. He initialization : 97.28%

2. Pola *training loss* :

- a. Zero initialization : Hampir tidak belajar sama sekali, langsung stagnan
- b. Uniform initialization : Peningkatan stabil, namun konvergensi sedikit lebih lambat dibanding Xavier
- c. Normal initialization : Konvergensi mulus, lebih cepat daripada Uniform, tapi masih lebih lambat daripada Xavier
- d. Xavier initialization : Konvergensi mulus, lebih cepat dibandingkan dengan Normal dan Uniform, namun lebih lambat daripada He
- e. He initialization : Konvergensi awal paling cepat, model dengan performa terbaik

3. Distribusi Bobot (Weight):

- a. Zero initialization : Semua bobot terpusat di nol
- b. Uniform initialization : Awalnya bobot tersebar merata, namun semakin dalam layer, distribusi bobot menjadi lebih terstruktur
- c. Normal initialization : Distribusi berbentuk *bell-shaped* dengan variansi yang meningkat di layer yang lebih dalam
- d. Xavier initialization : Distribusi *bell-shaped* yang lebih sempit di layer awal, dan lebih lebar di layer yang lebih dalam
- e. He initialization : Mirip dengan Xavier, namun distribusinya sedikit lebih lebar

4. Distribusi Gradien:

- a. Zero initialization : Distribusi gradien terkonsentrasi di nol

- b. Uniform initialization : Distribusi gradien *bell-shaped* dan tersebar dengan baik
- c. Normal initialization : Gradien *bell-shaped* dengan rentang paling lebar di layer pertama (-0.06 hingga 0.04) dibandingkan dengan layer pertama model lain, dan semakin dalam layer, distribusinya semakin melebar
- d. Xavier initialization : Gradien *bell-shaped* dengan skala yang sesuai (skala gradien tidak terlalu besar dan tidak terlalu kecil)
- e. He initialization : Mirip dengan Xavier, namun distribusinya sedikit lebih sempit di layer-layer yang lebih dalam

Eksperimen ini membuktikan bahwa inisialisasi bobot sangat mempengaruhi proses pelatihan ANN. Zero initialization gagal total karena semua bobot bernilai nol sehingga tidak ada *diversity* dalam pembelajaran. Sebaliknya, He initialization memiliki performa terbaik karena memang dirancang khusus untuk fungsi aktivasi ReLU yang digunakan dalam model tes ini. Memilih metode inisialisasi bobot yang sesuai dengan fungsi aktivasi dapat meningkatkan performa model secara signifikan.

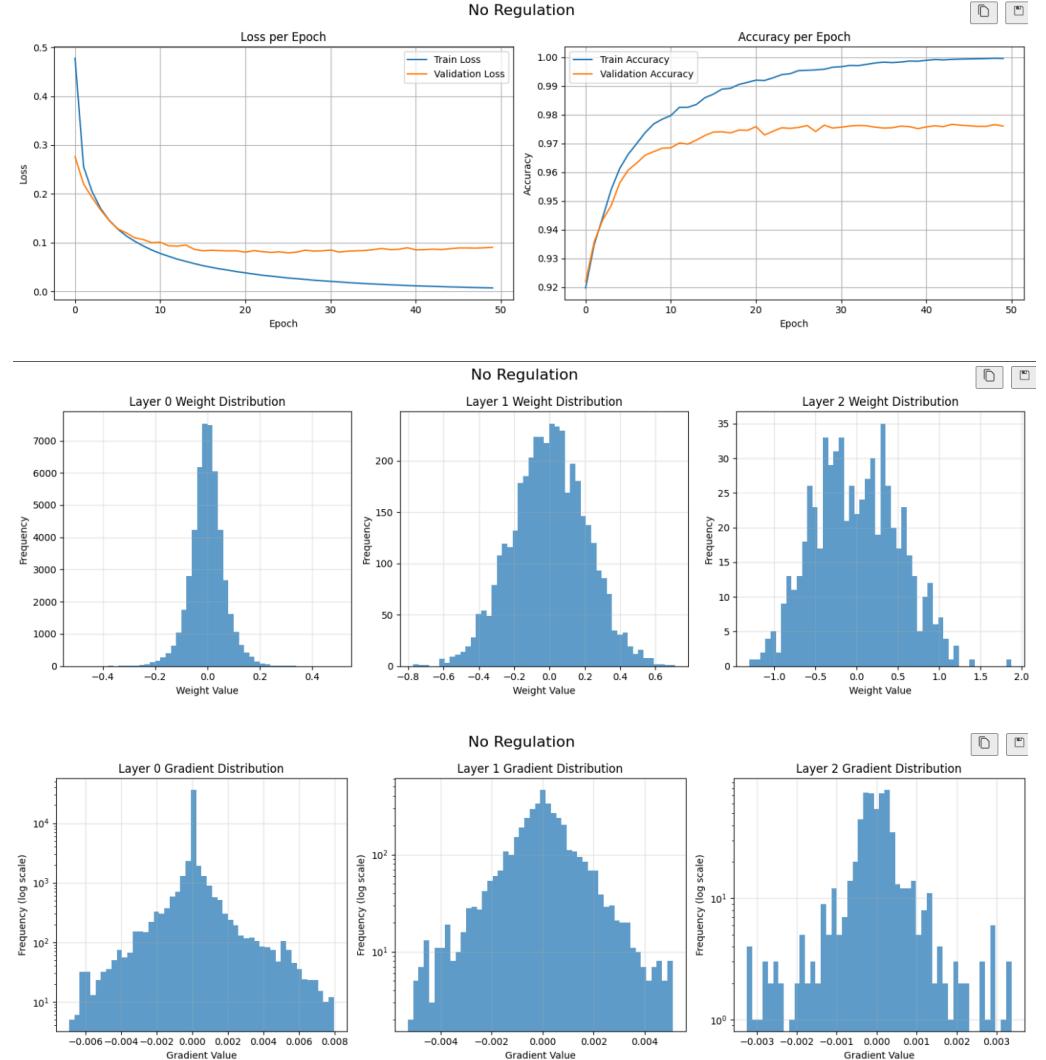
#### **2.2.5. Pengaruh Regulation Parameter**

Pengujian terhadap pengaruh *Regulation Parameter* dilakukan melalui tiga tes dengan kriteria berikut :

- 2 hidden layers
- 64 neurons per layer
- Tes 1 : Tanpa *Regulation*
- Tes 2 : *Regulation L1*
- Tes 3 : *Regulation L2*

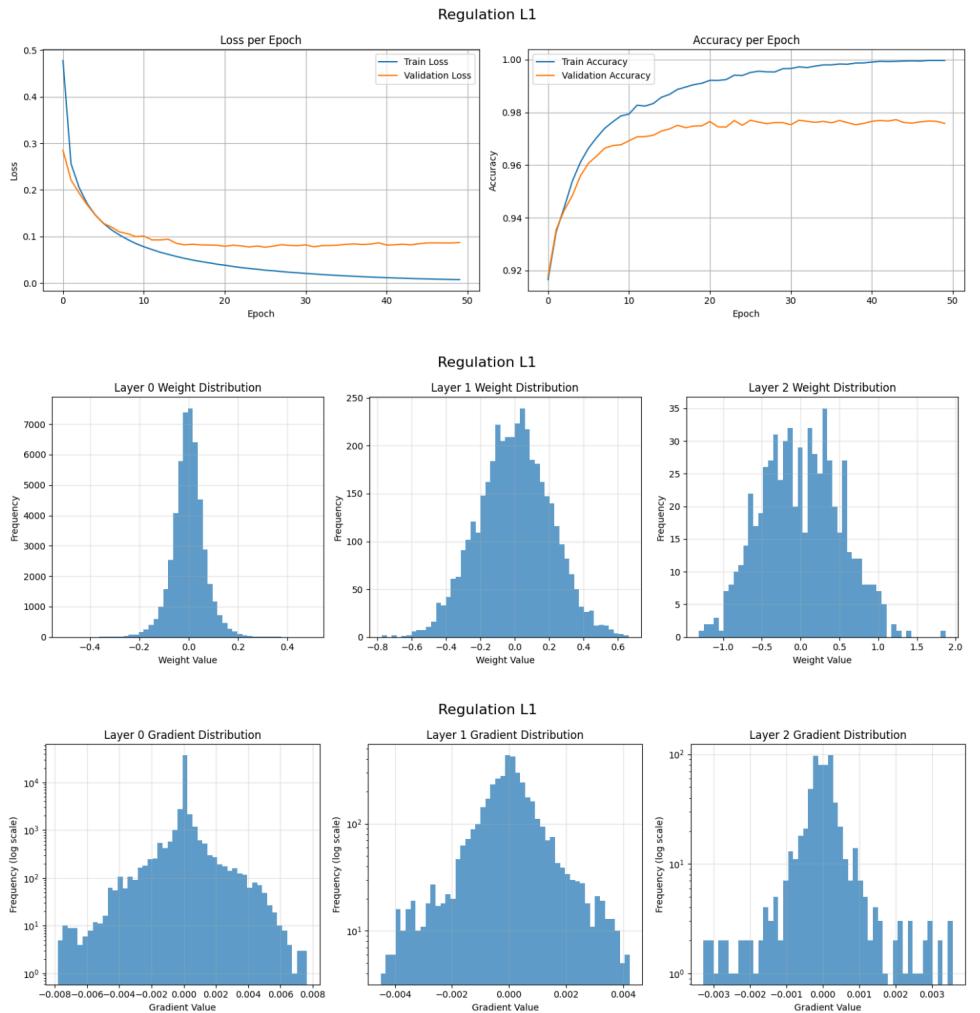
### 2.2.5.1. Tes 1 - Tanpa Regulation

*Validation accuracy:* 0.9761



### 2.2.5.2. Tes 2 - Regulation L1

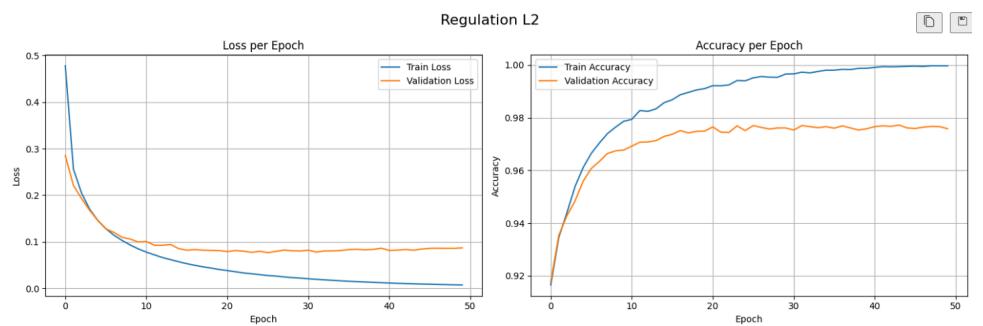
*Validation accuracy:* 0.9758

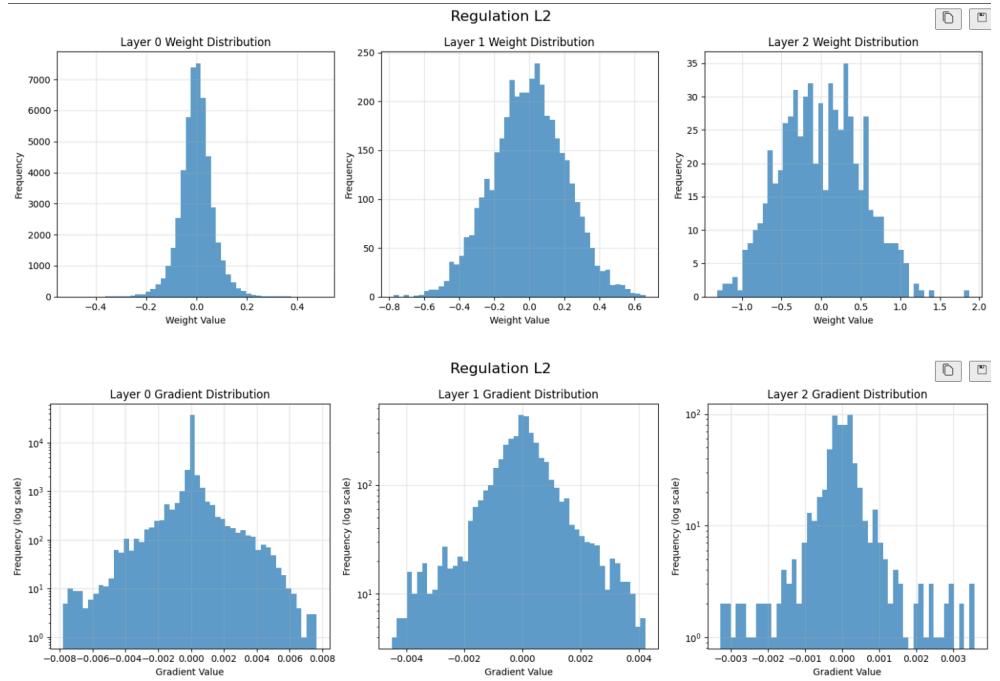


### 2.2.5.3.

### Tes 3 - Regulation L2

*Validation accuracy: 0.9758*





#### 2.2.5.4. Kesimpulan - Pengaruh Inisialisasi Bobot

Berdasarkan eksperimen kami terhadap model dengan berbagai metode inisialisasi bobot, kami menemukan bahwa inisialisasi bobot memiliki pengaruh yang signifikan terhadap performa model. Visualisasi kami mengungkapkan berikut:

1. Prediksi Akhir (Akurasi Validasi):
  - a. No Regulation : 97.61%
  - b. Regulation L1 : 97.58%
  - c. Regulation L2 : 97.58%
2. Pola *training loss* :
  - a. Terlihat sama untuk ketiga situasi.
3. Distribusi Bobot (Weight):
  - a. L1 dan L2 regularisasi mendorong beberapa bobot menuju nol.

- b. Dibandingkan dengan "Tanpa Regulation" bentuk distribusi tetap normal, tetapi memiliki kepadatan yang lebih tinggi di sekitar nol.
- c. Distribusi bobot sedikit berkurang, yang berarti bahwa L1 dan L2 regularisasi mengurangi beberapa nilai bobot sambil menjaga sebagian besar bobot mendekati nol.

4. Distribusi Gradien:

- a. Bentuk keseluruhan dari ketiganya tetap sama, tetapi gradien tampak lebih terpusat di sekitar nol pada kasus L1 dan L2.
- b. Puncak di nol lebih menonjol dalam kasus L1 dan L2, yang menunjukkan bahwa lebih banyak gradien mengalami pengurangan ukuran.

#### 2.2.6. Perbandingan dengan Library sklearn

Perbandingan antara library sklearn dan custom ANN yang kami rancang dilakukan menggunakan parameter sebagai berikut :

- 2 *hidden layers*
- 64 neuron untuk tiap layer
- fungsi aktivasi ReLU
- learning rate 0.25
- epoch 50
- random\_state 42

### 2.2.6.1. Hasil training model sklearn

Accuracy : 19.4%

```
Sklearn MLPClassifier

sklearn_mlp = MLPClassifier(
    hidden_layer_sizes=(64, 64),
    activation='relu',
    solver='sgd',
    learning_rate_init=0.25,
    batch_size=64,
    max_iter=50,
    random_state=42,
    learning_rate='constant'
)

sklearn_mlp.fit(X_train, y_train)
print(f'activation : {sklearn_mlp.activation}')
print(f'Output activation {sklearn_mlp.out_activation_}')
print(f'hidden layer {sklearn_mlp.hidden_layer_sizes}')

[69]
...
activation : relu
Output activation softmax
hidden layer (64, 64)

y_pred_mlp = sklearn_mlp.predict(X_test)
acc_mlp = accuracy_score(y_test, y_pred_mlp)
print("MLPClassifier Test Accuracy: {:.4f}")

[70]
...
MLPClassifier Test Accuracy: 0.1940
```

### 2.2.6.2. Hasil training model ANN Kelompok

Accuracy : 98.15%

```
Custom ANN

custom_ann = NeuralNetwork('categorical_crossentropy')
custom_ann.add_layer(DenseLayer(output_size=64, seed=42, activation=relu, init="Xavier"))
custom_ann.add_layer(DenseLayer(output_size=64, seed=42, activation=relu, init="Xavier"))
custom_ann.add_layer(DenseLayer(output_size=10, seed=42, activation=softmax, init="Xavier"))

custom_ann_history = custom_ann.train(
    X_train,
    y_train,
    epochs=50,
    batch_size=64,
    learning_rate=0.25,
    isOne_hot=True,
    verbose=1,
    validation_data=(X_test, y_test)
)

[71]
...
100%
```

```
[=====] 100%
Epoch 49/50 - 2.39s - loss: 0.0004 - accuracy: 1.0000 - val_loss: 0.1077 - val_accuracy: 0.9811
[=====] 100%
Epoch 50/50 - 3.05s - loss: 0.0004 - accuracy: 1.0000 - val_loss: 0.1080 - val_accuracy: 0.9815
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

▶ <
  predictions_custom_ann = custom_ann.predict(X_test)
  pred_classes_custom_ann = np.argmax(predictions_custom_ann, axis=1)
  accuracy_custom_ann = accuracy_score(pred_classes_custom_ann, y_test)
  print("Test Accuracy:", accuracy_custom_ann)

[72]
...
Test Accuracy: 0.9815
```

### 2.2.6.3.

### Kesimpulan - Perbandingan dengan Library sklearn

Dengan menggunakan hyperparameter yang sama pada kedua model, ANN buatan kami memiliki performa yang lebih baik dibandingkan dengan MLPClassifier. Model ANN kami mampu mencapai akurasi sebesar 98%, sementara MLPClassifier hanya mampu mencapai akurasi sebesar 19.4% dengan hyperparameter yang sama.

Kami menduga bahwa nilai *learning rate* menjadi penyebab utama perbedaan performa ini, model kami mampu bekerja dengan baik menggunakan *learning rate* sebesar 0.25 (yang merupakan *learning rate* terbaik berdasarkan hasil analisis kami), sedangkan MLPClassifier menunjukkan performa yang buruk dengan learning rate sebesar itu.

Kami juga telah mencoba menggunakan *learning rate* yang lebih kecil (0.05), dan dalam kasus tersebut, model kami masih mampu *outperform* MLPClassifier, meskipun tidak terlalu signifikan.

## **BAB 3**

### **Kesimpulan & Saran**

#### **3.1. Kesimpulan**

Berdasarkan eksperimen kami dapat menyimpulkan beberapa hal:

1. Untuk width yang berbeda, jaringan yang lebih besar belajar lebih baik, memiliki prediksi yang lebih baik, menghasilkan loss curve yang lebih halus, dan distribusi bobot lebih berpusat di sekitar nol
2. Untuk depth yang berbeda, menambahkan lebih banyak layer tidak selalu membuat model yang memiliki performa yang lebih baik dan penambahan jumlah layer cenderung menyebabkan fluktuasi lebih drastis pada grafik loss
3. Untuk fungsi aktivasi yang berbeda:
  - a. ReLU dan Tanh yang lebih bagus dibandingkan dengan model lainnya dapat dikaitkan dengan kemampuan mereka dalam menjaga gradien yang sehat sambil memberikan keragaman yang cukup. Distribusi gradien yang seimbang, distribusi bobot yang sesuai, serta kurva pembelajaran yang halus dan efisien berkontribusi terhadap performa yang lebih tinggi.
  - b. Performa Sigmoid yang sedikit lebih rendah disebabkan oleh masalah *vanishing gradient*, sementara performa Linear yang buruk membuktikan diperlukannya *non-linearity* dalam sebuah model ANN yang baik.
4. Untuk learning rate yang berbeda eksperimen menunjukkan bahwa learning rate yang lebih rendah tidak selalu menghasilkan akurasi yang lebih tinggi.
5. Untuk inisialisasi bobot yang berbeda eksperimen menunjukan bahwa parameter ini sangat mempengaruhi proses pelatihan ANN dimana zero init gagal total karena tidak ada diversity namun sebaliknya He memiliki performa terbaik karena sesuai dengan fungsi aktivasi yang digunakan (ReLU). Maka

memilih metode inisialisasi bobot yang sesuai dengan fungsi aktivasi dapat meningkatkan performa model secara signifikan.

6. Untuk metode regularisasi yang berbeda dapat disimpulkan bahwa penggunaan regularisasi L1 dan L2 tidak memberikan peningkatan akurasi yang signifikan dibandingkan model tanpa regularisasi, namun regularisasi berhasil mengubah karakteristik internal model dengan mendorong bobot menuju nol dan menghasilkan distribusi bobot yang lebih terkonsentrasi di sekitar nilai nol. Regularisasi juga menyebabkan gradien lebih terpusat, yang menunjukkan pengurangan ukuran gradien.
7. Perbandingan dengan library sklearn menunjukan bahwa dengan hyperparameter yang sama ANN buatan kami memiliki performa yang lebih baik dibandingkan dengan MLPClassifier. Dimana kami menduga bahwa learning rate menjadi perbedaan performa ini dimana dengan menggunakan learning rate 0.25, model kami menunjukan performa yang lebih baik namun dengan menggunakan learning rate 0.05, model kami masih mengungguli sklearn namun tidak terlalu signifikan

### **3.2. Saran**

Pada tugas besar ini, terdapat beberapa saran perbaikan untuk kelanjutan projek ini atau pun pelaksanaan projek serupa:

1. Untuk meningkatkan performa, dapat dicoba arsitektur yang lebih dalam dengan teknik seperti batch normalization dan residual connections.
2. Implementasi Teknik Augmentasi Data: Jika dataset yang digunakan terbatas, metode seperti data augmentation dapat membantu meningkatkan generalisasi model.
3. Evaluasi dengan Dataset Berbeda: Untuk menguji generalisasi model lebih lanjut, dapat dilakukan pengujian dengan dataset lain untuk melihat apakah model tetap memberikan performa yang baik.
4. Optimasi dengan GPU: Jika model yang dibangun lebih kompleks, implementasi menggunakan GPU acceleration (misalnya dengan CUDA) bisa membantu mempercepat proses pelatihan.

## BAB 4

### Pembagian Tugas

#### Pembagian Tugas

NIM	Responsibilities
13522061	AutoDiff, Activation Function, Loss Function, Laporan
13522075	DenseLayer, Analisis, Visualizer, Laporan
135220101	Base Neural Network, Regulation Bonus, Analisis, Adam Optimizer Bonus

## **LAMPIRAN**

*Repository:* <https://github.com/riyorax/Tugas-Besar-1-IF3270/tree/main>

## **DAFTAR PUSTAKA**

Institut Teknologi Bandung. (n.d.). LMS Indonesia.

<https://edunex.itb.ac.id/courses/72385/preview/323672>

Institut Teknologi Bandung. (n.d.). LMS Indonesia.

<https://edunex.itb.ac.id/courses/72385/preview/329043>

Andrej Karpathy. (2022, August 17). The spelled-out intro to neural networks and backpropagation: building micrograd [Video]. YouTube.

[https://youtu.be/VMj-3S1tku0?si=FdwzXW\\_YbsSyQTs](https://youtu.be/VMj-3S1tku0?si=FdwzXW_YbsSyQTs)