

Tugas Besar 1
IF3270 PEMBELAJARAN MESIN
Feedforward Neural Network



Disusun Oleh:

Wilson Yusda 13522019

Enrique Yanuar 13522077

Mesach Harmasendro 13522117

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

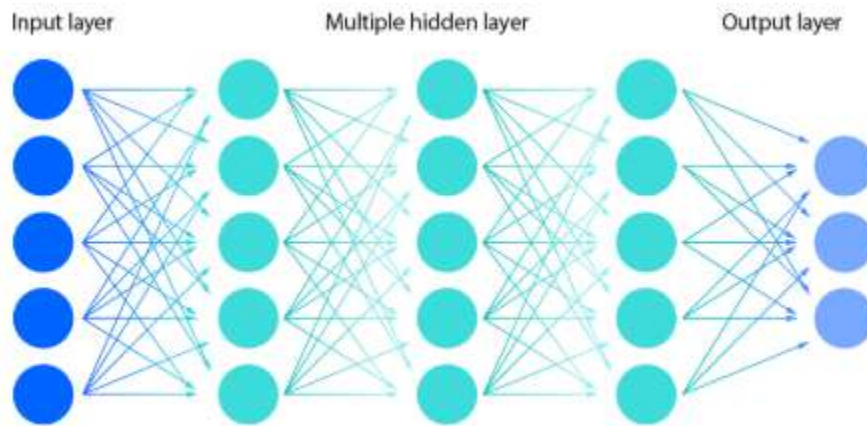
BAB I	
DESKRIPSI PERSOALAN.....	4
BAB II	
PENJELASAN IMPLEMENTASI.....	5
2.1 Deskripsi kelas beserta deskripsi atribut dan methodnya.....	5
2.1.1 Kelas Activation.....	5
2.1.2 Kelas WeightInitializer.....	8
2.1.3 Kelas Layer.....	10
2.1.4 Kelas LossFunctions.....	14
2.1.5 Kelas FFNN.....	15
2.1.6 Kelas RMSNorm.....	27
2.2 Penjelasan forward propagation.....	28
2.3 Penjelasan backward propagation dan weight update.....	32
2.4 Penjelasan Regularisasi.....	36
2.5 Penjelasan RMSNorm.....	38
BAB III	
HASIL PENGUJIAN.....	40
3.1 Pengaruh depth dan width.....	40
3.1.1 Uji coba untuk width berbeda.....	40
3.1.2 Analisis untuk width berbeda.....	46
3.1.3 Uji coba untuk depth berbeda.....	47
Untuk Validation:.....	49
3.1.4 Analisis untuk depth berbeda.....	53
3.2 Pengaruh fungsi aktivasi hidden layer.....	53
3.2.1 Uji coba linear.....	54
3.2.2 Uji coba ReLU (Menggunakan Xavier He , sebagai baseline akurasi yang baik).....	56
3.2.3 Uji coba Sigmoid.....	58
3.2.4 Uji coba tanh.....	60
3.2.5 Uji coba softsign.....	62
3.2.6 Uji coba leaky ReLU.....	64
3.2.7 Analisis untuk activation function berbeda.....	66
3.3 Pengaruh learning rate.....	66
3.3.1 Uji coba Learning Rate 0.1.....	67
3.3.2 Uji coba Learning Rate 0.01.....	69
3.3.3 Uji coba Learning Rate 0.001.....	71
3.3.4 Analisis untuk learning rate berbeda.....	73
3.4 Pengaruh inialisasi bobot.....	73
3.4.1 Uji coba inialisasi bobot 0.....	74
3.4.2 Uji coba inialisasi bobot random uniform.....	76

3.4.3 Uji coba inialisasi bobot random normal.....	78
3.4.4 Uji coba inialisasi bobot he.....	80
3.4.5 Uji coba inialisasi bobot Xavier (Sigmoid).....	82
3.4.4 Analisis untuk bobot inialisasi berbeda.....	84
3.5 Perbandingan dengan model yang di regularisasi.....	85
3.5.1 Tanpa Regularisasi.....	85
3.5.1 Regularisasi L1.....	87
3.5.2 Regularisasi L2.....	89
3.5.3 Analisis untuk pengaruh regularisasi.....	91
3.6 Perbandingan model built-in dengan model custom tugas besar.....	92
3.6.1 Model built in.....	93
3.6.2 Model custom.....	93
3.6.3 Analisis perbedaan model custom dengan built in.....	95
3.7 Perbandingan dengan model RMSNorm.....	95
3.7.1 Tanpa Normalisasi.....	96
3.7.2 Dengan Normalisasi RMSNorm.....	98
3.7.3 Analisis untuk pengaruh normalisasi.....	100
3.8 Analisis perbandingan train dan val.....	101
BAB IV	
KESIMPULAN DAN SARAN.....	102
BAB V	
LAMPIRAN.....	103
5.1 Tabel Pembagian Tugas.....	103
5.2 Repository Github.....	103
REFERENSI.....	104

BAB I

DESKRIPSI PERSOALAN

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



BAB II

PENJELASAN IMPLEMENTASI

2.1 Deskripsi kelas beserta deskripsi atribut dan methodnya

2.1.1 Kelas *Activation*

```
import torch as tc

class Activations:

    def linear(x):
        return x

    def d_linear(x):
        return tc.ones_like(x)

    def relu(x):
        # return tc.maximum(x, tc.zeros_like(x))
        return tc.max(x, tc.zeros_like(x))

    def d_relu(x):
        # return (x > 0).astype(tc.float32)
        return (x > 0).float()

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

    def d_sigmoid(x):
        s = Activations.sigmoid(x)
        return s * (1 - s)

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

    def d_tanh(x):
        return 1 - tc.tanh(x) ** 2
```

```

def softmax(x):
    # ex = tc.exp(x - tc.max(x, axis=1, keepdim=True)[0])
    # return ex / tc.sum(ex, axis=1, keepdim=True)
    ex = tc.exp(x - tc.max(x, dim=1, keepdim=True)[0])
    return ex / tc.sum(ex, dim=1, keepdim=True)

def d_softmax(x):
    s = Activations.softmax(x)
    batch_size, dim = s.shape

    diag_elements = s * (1 - s)

    off_diag_elements = -s.unsqueeze(2) * s.unsqueeze(1)

    jacobians = off_diag_elements
    diag_indices = tc.arange(dim)
    jacobians[:, diag_indices, diag_indices] =
diag_elements

    return jacobians

def d_softmax_times_vector(out, d0):
    s_dot_v = tc.sum(out * d0, dim=1, keepdim=True)
    result = out * (d0 - s_dot_v)
    return result

def softsign(x):
    return x / (1 + tc.abs(x))

def d_softsign(x):
    return 1 / (1 + tc.abs(x)) ** 2

def leaky_relu(x, alpha=0.01):
    return tc.maximum(alpha * x, x)

def d_leaky_relu(x, alpha=0.01):
    dx = tc.ones_like(x)
    dx[x < 0] = alpha
    return dx

activation_functions = {

```

```

'linear': (Activations.linear, Activations.d_linear, None),
'relu': (Activations.relu, Activations.d_relu, None),
'sigmoid': (Activations.sigmoid, Activations.d_sigmoid,
None),
'tanh': (Activations.tanh, Activations.d_tanh, None),
'softmax': (Activations.softmax, Activations.d_softmax,
Activations.d_softmax_times_vector),
'softsign': (Activations.softsign, Activations.d_softsign,
None),
'leaky_relu': (Activations.leaky_relu,
Activations.d_leaky_relu, None)
}

```

Kelas ini mendeklarasikan formula untuk berbagai fungsi aktivasi beserta turunannya, seperti:

1. Fungsi aktivasi linear

Fungsi aktivasi ini tidak melakukan transformasi sama sekali terhadap input yang diberikan.

$$f(x) = x$$

$$f'(x) = x$$

2. Fungsi aktivasi ReLU

Fungsi aktivasi ini memberikan nilai yang selalu positif. Jika input negatif, otomatis menjadi 0.

$$f(x) = \max(0, x)$$

$$f'(x) = \{1 \text{ if } x > 0, 0 \text{ if } x \leq 0$$

3. Fungsi aktivasi sigmoid

Fungsi aktivasi ini memberikan *range* output antara 0 sampai 1.

$$f(x) = \frac{1}{1+e^x}$$

$$f'(x) = f(x) \cdot (1 - f(x))$$

4. Fungsi aktivasi tanh

Fungsi aktivasi ini memberikan *range* output antara -1 sampai 1.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$f'(x) = 1 - \tanh^2(x)$$

5. Fungsi aktivasi softmax

Fungsi aktivasi ini umumnya cocok untuk prediksi *multiclass*. Fungsi ini menghasilkan probabilitas dengan menormalisasi nilai eksponensial dari input

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Untuk vector $\vec{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$,

$$\frac{d(\text{softmax}(\vec{x})_i)}{d\vec{x}} = \begin{bmatrix} \frac{\partial(\text{softmax}(\vec{x})_1)}{\partial x_1} & \dots & \frac{\partial(\text{softmax}(\vec{x})_1)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial(\text{softmax}(\vec{x})_n)}{\partial x_1} & \dots & \frac{\partial(\text{softmax}(\vec{x})_n)}{\partial x_n} \end{bmatrix}$$

Dimana untuk $i, j \in \{1, \dots, n\}$,

$$\frac{\partial(\text{softmax}(\vec{x})_i)}{\partial x_j} = \text{softmax}(\vec{x})_i (\delta_{i,j} - \text{softmax}(\vec{x})_j)$$

$$\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

2.1.2 Kelas *WeightInitializer*

```
import torch as tc

class WeightInitializer:
    def initialize(input_dim, output_dim, method=None,
init_params=None, activation=None):
        init_params = init_params or {}

        seed = init_params.get('seed', None)
        if seed is not None:
            tc.manual_seed(seed)

        if method == "he_xavier":
            if activation in ['relu', 'leaky_relu']:
                method = 'he'
            elif activation in ['sigmoid', 'tanh', 'softmax']:
                method = 'xavier'
            else:
                method = 'random_uniform'
        if method == 'zero':
            return tc.zeros((input_dim, output_dim))
        elif method == 'random_uniform':
            lower = init_params.get('lower', -0.5)
```



```

        upper = init_params.get('upper', 0.5)
        # return tc.random.uniform(lower, upper,
(input_dim, output_dim))
        return tc.empty((input_dim,
output_dim)).uniform_(lower, upper)
    elif method == 'random_normal':
        mean = init_params.get('mean', 0.0)
        variance = init_params.get('variance', 1.0)
        std = variance ** 0.5
        # return tc.random.normal(mean, std, (input_dim,
output_dim))
        return tc.randn((input_dim, output_dim)) * std +
mean
    elif method in ['xavier', 'he']:
        if activation in ['relu', 'leaky_relu']:
            std = (2 / input_dim) ** 0.5
            return tc.randn((input_dim, output_dim)) * std
        elif activation in ['sigmoid', 'tanh', 'softmax']:
            limit = (6 / (input_dim + output_dim)) ** 0.5
            return tc.empty((input_dim,
output_dim)).uniform_(-limit, limit)
        else:
            raise ValueError(f"Inisialisasi '{method}'
tidak cocok untuk aktivasi '{activation}'")
    else:
        raise ValueError("Metode inisialisasi bobot tidak
dikenali")

```

Kelas ini merepresentasikan berbagai opsi untuk memulai inisialisasi bobot dan bias pada model FFNN. Adapun berbagai metode inisialisasi yang dibuat yaitu:

1. *Zero Initialization*

Inisialisasi bobot dengan matriks yang diisi dengan angka 0, sesuai dimensi yang telah ditentukan.

2. *Random Uniform Initialization*

Menghasilkan sampel bobot dengan mempertimbangkan distribusi *uniform*. Metode ini memerlukan nilai batasan atas dan batasan bawah untuk menghasilkan distribusi yang *uniform*.

3. *Random Normal Initialization*

Menghasilkan sampel bobot dengan mempertimbangkan distribusi normal. Metode ini memerlukan rata rata, varian, dan standar deviasi untuk menghasilkan distribusi bobot.

4. *Xavier and He Initialization*

Merupakan implementasi bonus untuk tugas besar ini. Metode ini menerapkan perhitungan khusus untuk *Xavier* dan *He*. Untuk aktivasi ReLU dan leaky ReLU , umumnya digunakan *He*, dan untuk aktivasi *sigmoid*, *tanh*, dan *softmax* digunakan *Xavier*.

1. Formula *He*

$$std = \sqrt{\frac{2}{n}}$$

Dimana n merupakan jumlah neuron pada layer input, lalu dihasilkan bobot acak dengan distribusi normal menggunakan standar deviasi yang telah dihitung.

2. Formula *Xavier*

$$limit = \sqrt{\frac{6}{input+output}}$$

Dimana input dan output merupakan jumlah neuron pada layer input serta output lalu dihasilkan bobot acak dengan distribusi normal dengan batasan (-limit,limit)

2.1.3 Kelas *Layer*

```
import torch as tc
import matplotlib.pyplot as plt
from .activations import activation_functions
from .initializers import WeightInitializer
from .rms_norm import RMSNorm

class Layer:
    def __init__(self, input_dim, output_dim, activation_name,
                  weight_init='random_uniform',
                  init_params=None, use_rmsnorm=False):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.activation_name = activation_name
        self.weight_init = weight_init
        self.init_params = init_params

        self.use_rmsnorm = use_rmsnorm

        self.weights = WeightInitializer.initialize(input_dim,
```

```

output_dim,

method=weight_init, init_params=init_params,
activation=activation_name)
    self.biases = WeightInitializer.initialize(1,
output_dim,

method=weight_init, init_params=init_params,
activation=activation_name)
    self.activation, self.d_activation,
self.d_activation_times_vector =
activation_functions[activation_name]

    if self.use_rmsnorm:
        self.rmsnorm = RMSNorm(output_dim)

    self.input = None
    self.net = None
    self.out = None

    def forward(self, X):
        self.input = X
        self.net = X @ self.weights + self.biases
        if self.use_rmsnorm:
            self.normalized_net =
self.rmsnorm.forward(self.net)
            self.out = self.activation(self.normalized_net)
        else:
            self.out = self.activation(self.net)
        return self.out

    def backward(self, d0):
        m = self.input.shape[0]
        if self.activation_name == 'softmax':
            error_term =
self.d_activation_times_vector(self.out, d0)
        else:
            error_term = d0 * self.d_activation(self.net)
        if self.use_rmsnorm:
            d_net = self.rmsnorm.backward(error_term)
        else:
            d_net = error_term
        self.grad_weights = (self.input.t() @ d_net) / m
        self.grad_biases = tc.sum(d_net, dim=0, keepdims=True)
/ m

```

```

        dO_prev = d_net @ self.weights.t()
        return dO_prev

    def update(self, learning_rate):
        self.weights -= learning_rate * self.grad_weights
        self.biases -= learning_rate * self.grad_biases
        if self.use_rmsnorm:
            self.rmsnorm.update(learning_rate)

    def summary(self):
        print(f"Layer: {self.input_dim} -> {self.output_dim} |
Activation: {self.activation_name}")
        print("Weights:")
        print(self.weights)
        print("Biases:")
        print(self.biases)
        print("Gradients (Weights):")
        print(self.grad_weights)
        print("Gradients (Biases):")
        print(self.grad_biases)
        print("-" * 50)

```

Kelas ini merepresentasikan spesifikasi pada sebuah layer. Adapun komponen dalam kelas ini yaitu:

1. *input_dim*

Komponen ini merepresentasikan jumlah neuron pada layer input.

2. *output_dim*

Komponen ini merepresentasikan jumlah neuron pada layer output.

3. *activation_name*

Komponen ini menentukan jenis fungsi aktivasi yang digunakan pada layer, misalnya sigmoid, ReLU, dll.

4. *weight_init*

Komponen ini menentukan metode inisialisasi bobot, dengan default 'random_uniform'. Metode inisialisasi lain dapat digunakan sesuai kebutuhan. Namun nantinya akan diganti dengan implementasi inisialisasi lainnya.

5. *init_params*

Komponen ini merupakan parameter tambahan yang digunakan untuk metode inisialisasi bobot, jika diperlukan. Komponen ini dapat berisi nilai rata-rata dan varians untuk inisialisasi bobot dengan distribusi normal, atau dapat juga berisi batas atas dan batas bawah untuk inisialisasi bobot dengan distribusi seragam (uniform).

6. *weights*

Komponen ini menyimpan bobot yang diinisialisasi berdasarkan `input_dim` dan `output_dim` menggunakan metode yang telah ditentukan.

7. *biases*

Komponen ini menyimpan bias yang diinisialisasi ke nol dengan bentuk $(1, \text{output_dim})$.

8. *activation dan d_activation*

Komponen ini berisi fungsi aktivasi yang dipilih dan turunannya. Fungsi ini digunakan untuk menghasilkan output dan menghitung gradien selama proses backpropagation.

9. *input, Net, dan Out*

- *Input*: Menyimpan data masukan ke layer.
- *Net*: Menyimpan nilai net (hasil perkalian antara input dan bobot ditambah bias) sebelum diterapkan fungsi aktivasi.
- *Out*: Menyimpan hasil output setelah fungsi aktivasi diterapkan pada nilai net.

10. *grad_weights dan Grad_biases*

Komponen ini digunakan untuk menyimpan nilai gradien dari bobot dan bias yang dihitung selama backpropagation, guna melakukan update parameter dalam proses pembelajaran.

11. *d_activation_times_vector*

Komponen ini khusus digunakan untuk fungsi aktivasi softmax. Di dalamnya terdapat fungsi alternatif yang memungkinkan perhitungan error term tanpa perlu menghitung turunan dari fungsi softmax secara keseluruhan, sehingga tidak perlu membuat matriks Jacobian terlebih dahulu. Hal ini membuat proses perhitungan menjadi lebih efisien.

Adapun berbagai fungsi yang diimplementasikan yaitu:

1. *Forward*

Fungsi forward mengimplementasikan proses *forward propagation* dimana input diubah menjadi output melalui perkalian bobot dan penambahan bias diikuti oleh modifikasi oleh fungsi aktivasi.

2. *Backward*

Dilakukan perhitungan gradien (turunan) untuk memperbarui parameter model melalui backpropagation. Terdapat penentuan kondisi khusus karena perhitungan error term untuk fungsi aktivasi softmax berbeda dengan fungsi aktivasi lainnya. Perbedaan ini bertujuan untuk mengoptimalkan perhitungan turunan softmax, yang secara umum memerlukan sumber daya komputasi yang besar. Sementara itu, untuk fungsi aktivasi lainnya, perhitungan gradien dilakukan dengan menerapkan *chain rule*. Detail lebih lanjut akan dijelaskan pada tahapan *backward propagation*.

3. *Update*

Merupakan metode yang dibuat untuk memperbarui bobot dan bias sesuai dengan *learning rate*.

4. *Summary*

Memberikan ringkasan atas sebuah layer, termasuk bobot, bobot gradien, bias, aktivasi, serta jumlah neuron input serta output.

2.1.4 Kelas *LossFunctions*

```
class LossFunctions:

    def mse(y_true, y_pred):

        return 0.5 * tc.mean((y_true - y_pred) ** 2)

    def d_mse(y_true, y_pred):

        return -(y_true - y_pred)

    def binary_cross_entropy(y_true, y_pred, eps=1e-9):

        return -tc.mean(
            y_true * tc.log(y_pred + eps) +
            (1 - y_true) * tc.log(1 - y_pred + eps)
        )

    def d_binary_cross_entropy(y_true, y_pred, eps=1e-9):

        return (y_pred - y_true) / ( (y_pred*(1 - y_pred)) +
eps )

    def categorical_cross_entropy(y_true, y_pred, eps=1e-9):
```

```

        return -tc.mean(tc.sum(y_true * tc.log(y_pred + eps),
dim=1))

def d_categorical_cross_entropy(y_true, y_pred):

    return (y_pred - y_true)

```

Kelas ini merepresentasikan berbagai perhitungan *loss function* beserta dengan turunannya, diantaranya adalah:

1. Mean Squared Error (MSE)

MSE mengukur rata-rata kesalahan kuadrat antara nilai prediksi dan nilai asli.

$$MSE = 0.5 * mean((y_{true} - y_{pred})^2)$$

$$\frac{d}{dx} MSE = -(y_{true} - y_{pred})$$

2. Binary Cross-Entropy

Binary Cross-Entropy digunakan dalam model klasifikasi biner untuk mengukur perbedaan antara dua distribusi probabilitas.

$$BCE = - mean(y_{true} \cdot \log(y_{pred} + \epsilon) + (1 - y_{true}) \cdot \log(1 - y_{pred} + \epsilon))$$

$$\frac{d}{dx} BCE = \frac{y_{pred} - y_{true}}{y_{pred} \cdot (1 - y_{pred}) + \epsilon}$$

3. Categorical Cross-Entropy

Categorical Cross-Entropy digunakan dalam klasifikasi multi-kelas untuk menghitung seberapa baik distribusi probabilitas model cocok dengan distribusi nilai sebenarnya.

$$CCE = - mean(\sum_i y_{true,i} \cdot \log(y_{pred,i} + \epsilon))$$

$$\frac{d}{dx} CCE = y_{pred} - y_{true}$$

2.1.5 Kelas FFNN

```

import torch as tc
from .layer import Layer
from .loss_functions import loss_functions
import matplotlib.pyplot as plt

```

```

import numpy as np
import random
import math

class FFNN:
    def __init__(self, layer_sizes, activations_list,
                  loss_function='mse',
                  weight_inits='random_uniform',
                  init_params_list=None,
                  regularization='none', req_lambda=0.01,
use_rmsnorm=False):
        if loss_function not in loss_functions:
            raise NotImplementedError(f"Loss function
'{loss_function}' tidak dikenali.")
        self.loss_func, self.loss_grad_func =
loss_functions[loss_function]
        self.regularization = regularization
        self.reg_lambda = req_lambda

        self.loss_name = loss_function
        self.layers: list[Layer] = []
        for i in range(len(layer_sizes) - 1):
            print(i)
            layer = Layer(
                input_dim=layer_sizes[i],
                output_dim=layer_sizes[i + 1],
                activation_name=activations_list[i],
                weight_init=weight_inits[i] if weight_inits
else 'random_uniform',
                init_params=init_params_list[i] if
init_params_list else None,
                use_rmsnorm=use_rmsnorm
            )
            self.layers.append(layer)

        def forward(self, X):
            output = X
            for layer in self.layers:
                output = layer.forward(output)
            return output

        def compute_loss_with_regularization(self, y_true, y_pred):
            loss = self.loss_func(y_true, y_pred)
            if self.regularization == 'L1':
                reg_loss = 0.0
                for layer in self.layers:

```



```

        reg_loss += tc.sum(tc.abs(layer.weights)) /
layer.input.shape[0]
        loss += self.reg_lambda * reg_loss
    elif self.regularization == 'L2':
        reg_loss = 0.0
        for layer in self.layers:
            reg_loss += tc.sum(layer.weights ** 2) /
layer.input.shape[0]
        loss += (self.reg_lambda / 2) * reg_loss
    return loss

def backward(self, y_true, y_pred):
    d0 = self.loss_grad_func(y_true, y_pred)
    for layer in reversed(self.layers):
        d0 = layer.backward(d0)
        m = layer.input.shape[0]
        if self.regularization == 'L1':
            layer.grad_weights += self.reg_lambda *
tc.sign(layer.weights) / m
        elif self.regularization == 'L2':
            layer.grad_weights += 2*(self.reg_lambda *
layer.weights / m)

def update_weights(self, learning_rate):
    for layer in self.layers:
        layer.update(learning_rate)

def train(self, X_train, y_train, X_val=None, y_val=None,
          epochs=100, batch_size=32, learning_rate=0.01,
verbose=1, tol=1e-4, patience=10, stop_in_convergence=False):
    history = {'train_loss': [], 'val_loss': []}
    num_samples = X_train.shape[0]

    best_loss = float('inf')
    no_improve_count = 0

    for epoch in range(epochs):
        # indices = tc.random.permutation(num_samples)
        indices = tc.randperm(num_samples)

        X_train = X_train[indices]
        y_train = y_train[indices]

        epoch_loss = 0.0

        for i in range(0, num_samples, batch_size):

```

```

        X_batch = X_train[i:i+batch_size]
        y_batch = y_train[i:i+batch_size]

        y_pred = self.forward(X_batch)
        loss =
self.compute_loss_with_regularization(y_batch, y_pred)
        epoch_loss += loss.item() * X_batch.shape[0]

        self.backward(y_batch, y_pred)
        self.update_weights(learning_rate)

    epoch_loss /= num_samples
    history['train_loss'].append(epoch_loss)

    if X_val is not None and y_val is not None:
        y_val_pred = self.forward(X_val)
        val_loss =
self.compute_loss_with_regularization(y_val, y_val_pred).item()
        history['val_loss'].append(val_loss)

    if verbose:
        print(f"Epoch {epoch+1}/{epochs} "
              f"- Train Loss: {epoch_loss:.8f} -
Val Loss: {val_loss:.8f}")
    else:
        if verbose:
            print(f"Epoch {epoch+1}/{epochs} - Train
Loss: {epoch_loss:.8f}")

    # if stop_in_convergence:
    #     if abs(epoch_loss - best_loss) < tol:
    #         no_improve_count += 1
    #     else:
    #         no_improve_count = 0
    #         best_loss = epoch_loss

    #     if no_improve_count >= patience:
    #         if verbose:
    #             print(f"Early stopping triggered at
epoch {epoch+1} - loss did not improve more than {tol} for
{patience} consecutive epochs.")
    #         break

    return history

def save(self, file_path):
    layer_sizes = []

```

```

        if self.layers:
            layer_sizes.append(self.layers[0].weights.shape[0])
            for layer in self.layers:
                layer_sizes.append(layer.weights.shape[1])

        state = {
            'layer_sizes': layer_sizes,
            'activations': [layer.activation_name for layer in
self.layers],
            'loss_name': self.loss_name,
            'weight_init': [layer.weight_init for layer in
self.layers],
            'init_params': [layer.init_params for layer in
self.layers],
            'weights': [layer.weights.detach() for layer in
self.layers],
            'biases': [layer.biases.detach() for layer in
self.layers],
        }
        tc.save(state, file_path)
        print(f"Model saved to {file_path}")

    @classmethod
    def load(cls, file_path):
        checkpoint = tc.load(file_path, weights_only=False)

        model = cls(
            layer_sizes=checkpoint['layer_sizes'],
            activations_list=checkpoint['activations'],
            loss_function=checkpoint['loss_name'],
            weight_inits=checkpoint.get('weight_init',
'random_uniform'),
            init_params_list=checkpoint.get('init_params')
        )

        for i, layer in enumerate(model.layers):
            layer.weights = checkpoint['weights'][i]
            layer.biases = checkpoint['biases'][i]

        print(f"Model loaded from {file_path}")
        return model

    def summary(self):
        print("Model Summary:")
        for idx, layer in enumerate(self.layers):
            print(f"Layer {idx+1}:")

```

```

        layer.summary()
    def plot_gradients_distribution(self,
layers_to_plot,bins=100, normalized=True):
        if not layers_to_plot:
            print("Tidak ada layer yang dipilih untuk plot
distribusi bobot.")
            return
        plt.figure(figsize=(12, 8))
        for layer_index in layers_to_plot:
            if layer_index < 0 or layer_index >=
len(self.layers):
                print(f"Layer index {layer_index} berada di
luar jangkauan. Abaikan.")
                continue
            weights = self.layers[layer_index].grad_weights
            if isinstance(weights, tc.Tensor):
                weights = weights.detach().cpu().numpy()
            else:
                weights = np.array(weights)
            weights_flat = weights.flatten()
            if normalized:
                norm_weights = np.ones_like(weights_flat) /
len(weights_flat)
            plt.hist(weights_flat, bins=bins,
weights=norm_weights, alpha=0.5, label=f"Layer {layer_index}")
            else:
                plt.hist(weights_flat, bins=bins, alpha=0.5,
label=f"Layer {layer_index}")

        plt.title("Distribusi Bobot Gradien Tiap Layer")
        plt.xlabel("Nilai Bobot")
        plt.ylabel("Frekuensi (Persentase)" if normalized else
"Jumlah Instance")
        plt.legend()
        plt.grid(True)
        plt.show()
    def plot_weight_distribution(self, layers_to_plot,bins=100,
normalized=True):
        if not layers_to_plot:
            print("Tidak ada layer yang dipilih untuk plot
distribusi bobot.")
            return
        plt.figure(figsize=(12, 8))
        for layer_index in layers_to_plot:
            if layer_index < 0 or layer_index >=
len(self.layers):

```

```

        print(f"Layer index {layer_index} berada di
luar jangkauan. Abaikan.")
        continue
    weights = self.layers[layer_index].weights
    if isinstance(weights, tc.Tensor):
        weights = weights.detach().cpu().numpy()
    else:
        weights = np.array(weights)
    weights_flat = weights.flatten()
    if normalized:
        norm_weights = np.ones_like(weights_flat) /
len(weights_flat)
        plt.hist(weights_flat, bins=bins,
weights=norm_weights, alpha=0.5, label=f"Layer {layer_index}")
    else:
        plt.hist(weights_flat, bins=bins, alpha=0.5,
label=f"Layer {layer_index}")
    plt.title("Distribusi Bobot Tiap Layer")
    plt.xlabel("Nilai Bobot")
    plt.ylabel("Frekuensi (Persentase)" if normalized else
"Jumlah Instance")
    plt.legend()
    plt.grid(True)
    plt.show()
def plot_network_structure(self, limited_size):
    if not self.layers:
        print("Model tidak memiliki layer.")
        return

plt.figure(figsize=(20, 14))
light_colors = [
    '#FFD3B6',
    '#DCEDC1',
    '#A8E6CE',
    '#FFAAA5',
    '#D5E5F2',
    '#E8E1EF',
    '#FFF1BD',
    '#C7CEEA'
]

layer_weights = []
layer_grad_weights = []
layer_biases = []
layer_grad_biases = []
layer_sizes = []

```

```

        for layer in self.layers:
            if isinstance(layer.weights, tc.Tensor):
                weights = layer.weights.detach().cpu().numpy()
            else:
                weights = np.array(layer.weights)
            layer_weights.append(weights)
            if isinstance(layer.grad_weights, tc.Tensor):
                grad_weights =
layer.grad_weights.detach().cpu().numpy()
            else:
                grad_weights = np.array(layer.grad_weights)
            layer_grad_weights.append(grad_weights)
            if isinstance(layer.biases, tc.Tensor):
                biases =
layer.biases.detach().cpu().numpy().flatten()
            else:
                biases = np.array(layer.biases).flatten()
            layer_biases.append(biases)
            if isinstance(layer.grad_biases, tc.Tensor):
                grad_biases =
layer.grad_biases.detach().cpu().numpy().flatten()
            else:
                grad_biases =
np.array(layer.grad_biases).flatten()
            layer_grad_biases.append(grad_biases)
            if len(layer_sizes) == 0:
                layer_sizes.append(weights.shape[0])
                layer_sizes.append(weights.shape[1])
            else:
                layer_sizes.append(weights.shape[1])

        limited_nodes = []
        for size in layer_sizes:
            if size <= limited_size:
                limited_nodes.append(list(range(size)))
            else:
                limited_nodes.append(random.sample(range(size),
limited_size))
        pos = {}
        layer_spacing = 1.5

        for layer_idx, nodes in enumerate(limited_nodes):
            for i, node_idx in enumerate(nodes):
                x = layer_idx * layer_spacing
                y = (i - len(nodes)/2) * 0.7

```

```

        pos[(layer_idx, node_idx)] = (x, y)

    for layer_idx in range(len(limited_nodes) - 1):
        lowest_y = min([pos[(layer_idx, node_idx)][1] for
node_idx in limited_nodes[layer_idx]])
        bias_y = lowest_y - 0.8
        pos[(layer_idx, 'bias')] = (layer_idx *
layer_spacing, bias_y)

    for layer_idx, nodes in enumerate(limited_nodes):
        for node_idx in nodes:
            plt.scatter(*pos[(layer_idx, node_idx)],
s=2000, c='skyblue', zorder=2, edgecolor='black')
            plt.text(*pos[(layer_idx, node_idx)],
f"{layer_idx}-{node_idx}",
                    ha='center', va='center', fontsize=10,
fontweight='bold')

    for layer_idx in range(len(limited_nodes) - 1):
        bias_pos = pos[(layer_idx, 'bias')]
        plt.scatter(*bias_pos, s=2000, c='lightgreen',
zorder=2, edgecolor='black')
        plt.text(*bias_pos, f"Bias {layer_idx}",
ha='center', va='center', fontsize=10, fontweight='bold')
    for layer_idx in range(len(limited_nodes) - 1):
        source_nodes = limited_nodes[layer_idx]
        target_nodes = limited_nodes[layer_idx + 1]

        for s_idx in source_nodes:
            for t_idx in target_nodes:
                weight_value =
layer_weights[layer_idx][s_idx, t_idx]
                grad_weight_value =
layer_grad_weights[layer_idx][s_idx, t_idx]

                edge_color = random.choice(light_colors)
                start_pos = pos[(layer_idx, s_idx)]
                end_pos = pos[(layer_idx + 1, t_idx)]

                plt.plot([start_pos[0], end_pos[0]],
[start_pos[1], end_pos[1]],
                        color=edge_color, alpha=0.8,
zorder=1, linewidth=2)

                mid_x = (start_pos[0] + end_pos[0]) / 2
                mid_y = (start_pos[1] + end_pos[1]) / 2

```

```

        offset_x = random.uniform(-0.2, 0.2)
        offset_y = random.uniform(-0.05, 0.05)

        angle = math.degrees(math.atan2(end_pos[1]
- start_pos[1], end_pos[0] - start_pos[0]))
        if angle > 90:
            angle -= 180
        elif angle < -90:
            angle += 180

        weight_text = f"w = {weight_value:.4f} g =
{grad_weight_value:.4f}"
        plt.text(mid_x + offset_x, mid_y +
offset_y, weight_text,
                fontsize=7, ha='center',
va='center', color='black',
                bbox=dict(facecolor=edge_color,
alpha=0.9, pad=1, boxstyle='round'))
        for layer_idx in range(len(limited_nodes) - 1):
            bias_pos = pos[(layer_idx, 'bias')]
            target_nodes = limited_nodes[layer_idx + 1]

            for t_idx_pos, t_idx in enumerate(target_nodes):
                bias_value = layer_biases[layer_idx][t_idx]
                grad_bias_value =
layer_grad_biases[layer_idx][t_idx]

                edge_color = random.choice(light_colors)
                end_pos = pos[(layer_idx + 1, t_idx)]

                plt.plot([bias_pos[0], end_pos[0]],
[bias_pos[1], end_pos[1]],
                        color=edge_color, alpha=0.8, zorder=1,
linewidth=2, linestyle='--')

                mid_x = (bias_pos[0] + end_pos[0]) / 2
                mid_y = (bias_pos[1] + end_pos[1]) / 2
                offset_x = random.uniform(-0.1, 0.1)
                offset_y = random.uniform(-0.1, 0.1)

                bias_text = f"b = {bias_value:.4f} g =
{grad_bias_value:.4f}"
                plt.text(mid_x + offset_x, mid_y + offset_y,
bias_text,
                        fontsize=7, ha='center', va='center',
color='black',

```



```

        bbox=dict(facecolor=edge_color,
alpha=0.9, pad=1, boxstyle='round4'))

plt.title('Neural Network Structure with Biases')
plt.axis('off')
plt.tight_layout()
plt.show()

```

Kelas ini merupakan kelas utama dalam merepresentasikan struktur *neural network* pada tugas besar ini. Adapun cara inisialisasi fungsi ini dapat dilakukan dengan:

```

custom_model = FFNN(
    [784, 600, 10],
    activations_list=['relu','softmax'],
    loss_function='cce',
    weight_inits=['zero', 'zero'],
    init_params_list=[
        {'lower': -1, 'upper': 1},
        {'lower': -1, 'upper': 1} # for random_normal
    ]
    regularization='L2' # use regularization
    use_rmsnorm=True # true use rms norm
)

```

Parameter pertama menandakan jumlah neuron pada tiap layer, dimana untuk posisi pertama berupa *input layer* dan terakhir berupa *output layer* dan sisanya berupa *hidden layer*. Kemudian, parameter selanjutnya diisi dengan fungsi aktivasi untuk tiap *layer*. Parameter ketiga menjelaskan *loss function* pada model yang ingin dibuat. Parameter keempat merepresentasikan bagaimana cara inisialisasi distribusi bobot awal. Parameter terakhir berupa *init_params*, yang akan dipakai untuk kondisi jika permintaan distribusi berupa distribusi *uniform* atau normal. Inisialisasi kelas ini berupa inisialisasi berbagai kelas layer sesuai dengan parameter yang telah disediakan serta *loss function* beserta turunannya. Adapun fungsi dalam kelas ini yaitu:

1. *forward*

Fungsi ini memanggil fungsi *forward* pada tiap layer pada *network*.

2. *backward*

Fungsi ini memanggil fungsi *backward* pada tiap layer pada *network*, namun dari *output layer* menuju *input layer*.

3. *update_weights*

Fungsi ini memanggil fungsi *update* pada tiap layer pada *network*.

4. *train*

Fungsi ini digunakan untuk melatih model FFNN secara keseluruhan meliputi proses forward propagation, perhitungan loss, backward propagation, dan pembaruan bobot.

Pada fungsi ini juga terdapat beberapa parameter tambahan sebagai berikut:

- a. *x_val* (opsional): Data fitur untuk validasi (untuk menghitung *val_loss*).
- b. *y_val* (opsional): Label/target yang sesuai dengan *X_val* (untuk menghitung *val_loss*).
- c. *epochs*: Jumlah iterasi maksimum selama pelatihan. Default: 100.
- d. *batch_size*: Jumlah sampel yang diproses dalam satu batch. Default: 32.
- e. *learning_rate*: Tingkat pembelajaran untuk mengupdate bobot. Default: 0.01.
- f. *verbose*: Menampilkan informasi pelatihan jika diatur ke 1. Default: 1.
- g. *tol*: Toleransi perubahan loss untuk menentukan konvergensi. Default: 1e-4.
- h. *patience*: Jumlah epoch yang ditunggu sebelum early stopping jika loss tidak membaik. Default: 10.
- i. *stop_in_convergence*: Jika True, pelatihan berhenti otomatis saat loss konvergen. Default: False.

5. *summary*

Fungsi ini memberikan ringkasan tentang struktur model. Fungsi ini menampilkan informasi tentang setiap layer, seperti ukuran layer dan jumlah parameter.

6. *plot_gradients_distribution*

Fungsi ini digunakan untuk menampilkan distribusi gradien bobot dalam bentuk histogram. Fungsi ini memungkinkan pengguna memilih layer tertentu dan melihat bagaimana gradien terdistribusi di layer tersebut.

7. *plot_weight_distribution*

Fungsi *plot_weight_distribution* mirip dengan *plot_gradients_distribution*, tetapi fokusnya adalah pada distribusi bobot, bukan gradien. Fungsi ini menampilkan histogram bobot untuk layer yang dipilih.

8. *plot_network_structure*

Fungsi `plot_network_structure` menyediakan visualisasi grafis dari struktur NN. Fungsi ini menampilkan bagaimana layer-layer terhubung, termasuk bobot dan gradien antar neuron.

9. *save*

Fungsi `save` digunakan untuk menyimpan model ke dalam file. Fungsi ini menyimpan informasi penting seperti ukuran layer, fungsi aktivasi, fungsi loss, bobot, dan bias. Fungsi ini membuat model bisa digunakan lagi di lain waktu tanpa perlu melatih model ulang.

10. *load*

Fungsi `load` adalah kebalikan dari `save`. Fungsi ini memuat model yang sebelumnya disimpan ke dalam file.

2.1.6 Kelas `RMSNorm`

```
import torch as tc
class RMSNorm:
    def __init__(self, num_features, eps=1e-5):
        self.eps = eps
        self.gamma = tc.ones(num_features)
        self.beta = tc.zeros(num_features)

    def forward(self, x):
        self.input = x
        self.rms = tc.sqrt(tc.mean(x**2, dim=-1, keepdim=True)
+ self.eps)
        return self.gamma * (x / self.rms) + self.beta

    def backward(self, grad_output):
        normalized = self.input / self.rms
        self.grad_gamma = (grad_output * normalized).sum(dim=0)
        self.grad_beta = grad_output.sum(dim=0)
        dnormalized = grad_output * self.gamma
        n = self.input.shape[-1]
        grad_input = dnormalized / self.rms - self.input *
((dnormalized * self.input).sum(dim=-1, keepdim=True)) / (n *
self.rms**3)
        return grad_input

    def update(self, learning_rate):
        self.gamma -= learning_rate * self.grad_gamma
```

```
self.beta -= learning_rate * self.grad_beta
return
```

Secara keseluruhan, kelas diatas hanya sebagai diversi untuk penggunaan RMSNorm untuk *backward* , dan *forward*. Secara konsep, sebelum dimajukan ke aktivasi, nilai x input akan diproses dulu dengan formula x/RMS dimana RMS akan dihitung sebagaimana layaknya pada formula diatas. Selain itu, sewaktu *back propagation*, juga dilakukan pemrosesan *error term* sebelum dimajukan ke aktivasi untuk dijadikan *grad_weight* dan *grad_bias*. Dan juga , akan dilakukan pembaruan pada *gamma* dan *beta* pada saat perubahan *weight* terhadap *grad_weight*.

2.2 Penjelasan forward propagation

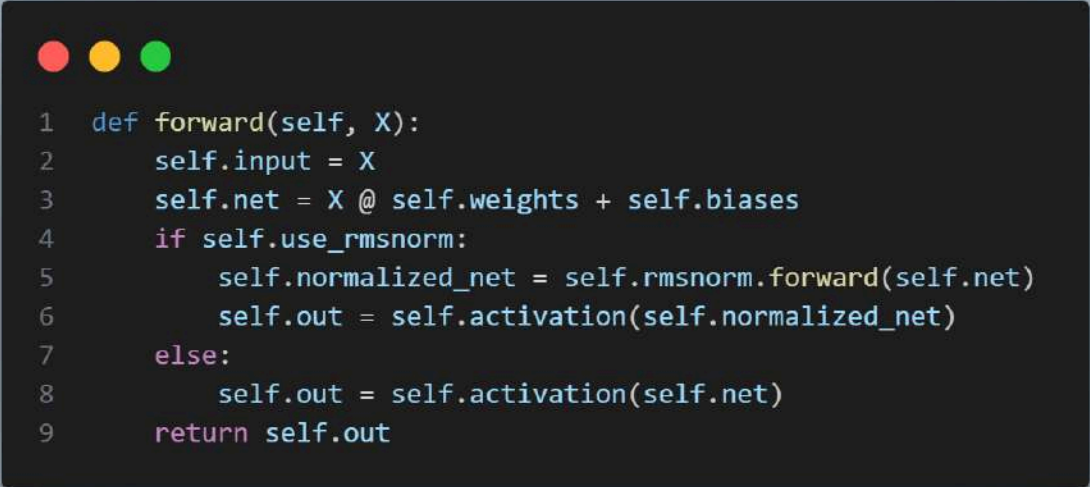
Forward propagation adalah proses inti dalam jaringan saraf yang mengubah data input menjadi prediksi melalui serangkaian perhitungan berbobot dan transformasi non-linear. Proses ini mengalirkan informasi dari lapisan input, melalui lapisan tersembunyi, hingga ke lapisan output. Dalam implementasi kami menggunakan PyTorch, forward propagation dirancang untuk menangani perhitungan vektor dan transformasi non-linear, yang membantu jaringan saraf menghasilkan prediksi yang akurat.

Implementasi kami menggunakan operasi tensor PyTorch dan kemampuan perkalian matriks untuk melakukan perhitungan secara efisien pada seluruh batch data. Dengan mengolah beberapa sampel sekaligus melalui operasi matriks, forward propagation dapat memproses batch dengan lebih cepat dibandingkan menghitung setiap sampel satu per satu. Selain itu, forward propagation menyimpan nilai-nilai sementara (seperti input dan output) di setiap layer, yang berguna untuk menghitung gradien selama backpropagation tanpa perlu menghitung ulang nilai-nilai tersebut.

Dalam implementasi kami mekanisme forward propagation diimplementasikan pada dua tingkat yaitu tingkat layer individu (pada class Layer) dan tingkat jaringan (pada class FFNN).

A. Forward Propagation pada Tingkat Layer

Unit komputasi dasar dalam FFNN kami adalah layer individu. Setiap layer mengimplementasikan forward propagation-nya sendiri melalui metode `forward()` dalam kelas Layer:



```

1  def forward(self, X):
2      self.input = X
3      self.net = X @ self.weights + self.biases
4      if self.use_rmsnorm:
5          self.normalized_net = self.rmsnorm.forward(self.net)
6          self.out = self.activation(self.normalized_net)
7      else:
8          self.out = self.activation(self.net)
9      return self.out

```

Metode ini melakukan tiga operasi penting:

1. Penyimpanan Input (`self.input = X`):

Matriks data input X memiliki bentuk $(\text{batch_size}, \text{input_dim})$, di mana batch_size merupakan jumlah sampel yang diproses secara bersamaan dan input_dim adalah jumlah fitur per sampel. Input ini disimpan dalam state layer untuk memungkinkan perhitungan gradien selama backpropagation tanpa perlu menghitung ulang nilai-nilai tersebut. Untuk lapisan pertama, X merupakan data input langsung, sedangkan untuk lapisan berikutnya, X adalah output yang dihasilkan dari lapisan sebelumnya.

2. Transformasi Linear (`self.net = X @ self.weights + self.biases`):

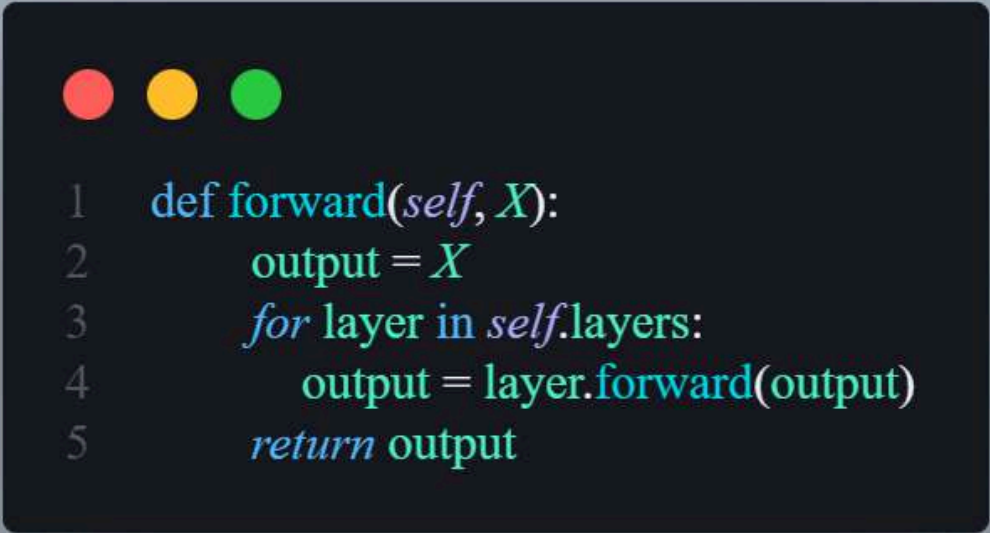
Perkalian matriks $X @ \text{self.weights}$ melakukan penjumlahan terbobot untuk setiap neuron dan setiap sampel dalam batch. Matriks `self.weights` memiliki bentuk $(\text{input_dim}, \text{output_dim})$, di mana output_dim adalah jumlah neuron pada lapisan saat ini. Untuk setiap sampel dan setiap neuron output, operasi ini menghitung $\Sigma(x_i \cdot w_{i,j})$, dengan x_i sebagai fitur input ke- i dan $w_{i,j}$ sebagai bobot yang menghubungkan input i ke neuron j . Selain itu, `self.biases` memiliki bentuk $(1, \text{output_dim})$ dan diterapkan secara seragam ke semua sampel dalam batch. Hasil dari operasi ini, `self.net`, memiliki bentuk $(\text{batch_size}, \text{output_dim})$, yang mewakili output pra-aktivasi untuk setiap neuron dan setiap sampel.

3. Aktivasi Non-Linear (`self.out = self.activation(self.net)`):

Fungsi aktivasi menerapkan transformasi non-linear pada setiap elemen dari `self.net` secara terpisah. Transformasi non-linear ini sangat penting karena memungkinkan jaringan saraf untuk mempelajari pola-pola kompleks yang tidak bisa dipahami hanya dengan transformasi linear. Fungsi aktivasi yang digunakan, seperti ReLU, sigmoid, tanh, atau softmax, dipilih saat lapisan diinisialisasi. Hasil dari transformasi ini, yaitu `self.out`, tetap memiliki bentuk (`batch_size`, `output_dim`) dan mewakili output akhir dari lapisan setelah melalui proses aktivasi.

B. Forward Propagation pada Tingkat Jaringan

Pada tingkat jaringan, metode `forward()` dalam kelas FFNN mengatur forward propagation berurutan melalui semua layer:



```
1 def forward(self, X):
2     output = X
3     for layer in self.layers:
4         output = layer.forward(output)
5     return output
```

Metode ini mengimplementasikan aliran data sekuensial di mana data input `X` diinisialisasi sebagai output awal. Setiap layer dalam jaringan kemudian memproses output dari layer sebelumnya, dan output dari layer terakhir menjadi prediksi akhir

jaringan. Proses ini dimulai dengan data input X yang memiliki bentuk $(batch_size, input_features)$, yang diatur sebagai output awal. Jaringan kemudian mengiterasi melalui setiap layer secara berurutan, mulai dari hidden layer pertama hingga output layer. Pada setiap layer, input yang diterima diubah menggunakan forward propagation yang spesifik untuk layer tersebut, memastikan bahwa data diproses secara bertahap.

Selama proses ini, dimensi data berubah sesuai dengan arsitektur jaringan yang ditentukan dalam $layer_sizes$. Sebagai contoh, dalam jaringan dengan $layer_sizes=[784, 128, 64, 10]$, data input awalnya memiliki bentuk $(batch_size, 784)$. Setelah melewati layer pertama, bentuknya berubah menjadi $(batch_size, 128)$, kemudian setelah layer kedua menjadi $(batch_size, 64)$, dan akhirnya output jaringan memiliki bentuk $(batch_size, 10)$, yang sesuai dengan jumlah kelas atau target yang ingin diprediksi.

Secara keseluruhan, jaringan saraf ini menghitung fungsi kompleks yang terdiri dari beberapa transformasi berlapis, $f(x) = f_n(f_{n-1}(\dots f_1(x)\dots))$, di mana f_i adalah transformasi yang dilakukan oleh layer ke- i . Setiap transformasi terdiri dari dua bagian: bagian linear (melibatkan bobot dan bias) dan bagian non-linear (melibatkan fungsi aktivasi). Gabungan dari transformasi linear dan non-linear ini memungkinkan jaringan saraf untuk mempelajari pola-pola rumit dalam data dan membuat prediksi yang akurat.


C. Alur Forward Propagation Secara Umum

Saat melakukan training atau inferensi, forward propagation mengikuti urutan ini:

1. Data input diteruskan ke layer pertama.
2. Setiap layer menerapkan bobot, menambahkan bias, dan mengubah hasil melalui fungsi aktivasinya.
3. Output dari setiap layer menjadi input untuk layer berikutnya.
4. Layer terakhir menghasilkan prediksi jaringan, yang kemudian dievaluasi terhadap output yang diharapkan menggunakan fungsi loss.

D. Forward Propagation dalam Proses Pelatihan

Forward propagation adalah langkah awal dalam proses pelatihan:



```
1 y_pred = self.forward(X_batch)
2 loss = self.compute_loss_with_regularization(y_batch, y_pred)
3 epoch_loss += loss.item() * X_batch.shape[0]
```

Selama pelatihan:

1. Forward propagation menghasilkan prediksi (y_{pred}) untuk batch saat ini.
2. Prediksi ini dibandingkan dengan label sebenarnya (y_{batch}) untuk menghitung loss.
3. Loss mendorong proses backward propagation untuk menghitung gradien.
4. Gradien memandu pembaruan bobot untuk meningkatkan prediksi di masa depan.

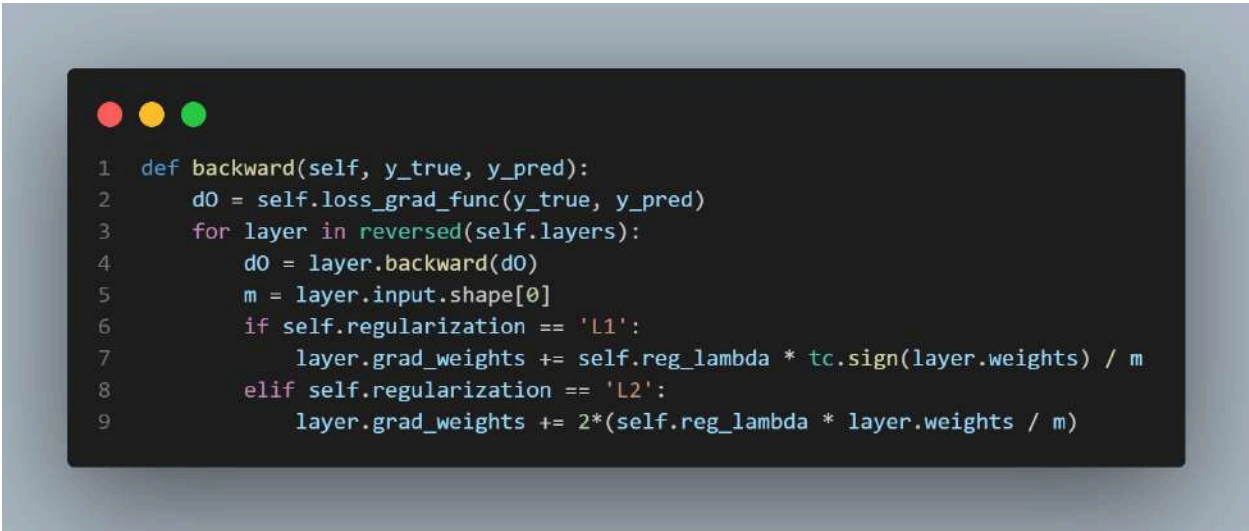
2.3 Penjelasan backward propagation dan weight update

Backward propagation (backpropagation) merupakan algoritma utama dalam proses pelatihan Feed Forward Neural Network (FFNN) yang bertujuan untuk mengoptimalkan bobot jaringan berdasarkan error yang dihasilkan. Implementasi backpropagation kami dilakukan melalui beberapa tahapan sebagai berikut:

A. Menghitung Error dari Output Layer

Proses backpropagation dimulai dari output layer dengan menghitung seberapa besar error (kesalahan) prediksi jaringan terhadap target yang sebenarnya. Dalam implementasi

yang dibuat, proses ini dimulai di method backward() pada class FFNN:



```
1 def backward(self, y_true, y_pred):
2     d0 = self.loss_grad_func(y_true, y_pred)
3     for layer in reversed(self.layers):
4         d0 = layer.backward(d0)
5         m = layer.input.shape[0]
6         if self.regularization == 'L1':
7             layer.grad_weights += self.reg_lambda * tc.sign(layer.weights) / m
8         elif self.regularization == 'L2':
9             layer.grad_weights += 2*(self.reg_lambda * layer.weights / m)
```

Perhatikan bahwa dO awal diperoleh dari turunan fungsi loss (self.loss_grad_func) yang membandingkan nilai prediksi (y_pred) dengan nilai sebenarnya (y_true). Nilai dO ini merepresentasikan gradien error terhadap output jaringan.

B. Backpropagation pada Setiap Layer

Setelah mendapatkan gradien error pada output, proses berlanjut ke tiap layer secara terbalik (dari output menuju input) menggunakan method backward() pada class Layer:

```

1  def backward(self, dO):
2      m = self.input.shape[0]
3      if self.activation_name == 'softmax':
4          error_term = self.d_activation_times_vector(self.out, dO)
5      else:
6          error_term = dO * self.d_activation(self.net)
7      if self.use_rmsnorm:
8          d_net = self.rmsnorm.backward(error_term)
9      else:
10         d_net = error_term
11     self.grad_weights = (self.input.t() @ d_net) / m
12     self.grad_biases = tc.sum(d_net, dim=0, keepdims=True) / m
13     dO_prev = d_net @ self.weights.t()
14     return dO_prev

```

Pada setiap layer, beberapa operasi penting dilakukan:

1. Penanganan Khusus untuk Softmax:


Karena sifat softmax yang menghasilkan matriks Jacobian, implementasi ini memberikan penanganan khusus. Untuk softmax, digunakan optimasi khusus dengan fungsi `d_activation_times_vector()` yang melakukan operasi vektor Jacobian tanpa perlu menghitung seluruh matriks Jacobian secara lengkap. Hal ini membuat perhitungan menjadi lebih efisien dan mengurangi kompleksitas komputasi.

2. Perhitungan Gradien Bobot dan Bias

Setelah mendapatkan `error_term`, gradien untuk bobot dan bias dihitung. Gradien bobot diperoleh dari operasi perkalian matriks antara transpose input dan error term, kemudian dibagi dengan banyak sampel dalam satu batch (m) untuk mendapatkan nilai rata-rata gradien. Sedangkan gradien bias diperoleh dari penjumlahan error term pada dimensi batch, lalu dibagi banyak sampel dalam satu batch. Pembagian dengan jumlah sampel (m) dilakukan untuk menormalisasi gradien, sehingga gradien yang dihasilkan mewakili rata-rata kesalahan dari


seluruh batch, bukan hanya dari satu sampel. Hal ini membantu menjaga stabilitas proses pembelajaran dan memastikan bahwa pembaruan bobot dan bias tidak terlalu besar atau terlalu kecil, sehingga model dapat belajar secara lebih konsisten.

Setelah gradien untuk setiap bobot dan bias dihitung melalui backpropagation, langkah selanjutnya adalah memperbaharui bobot-bobot tersebut untuk meminimalkan error. Proses ini diimplementasikan dalam method `update_weights()` pada class FFNN:



```
1 def update_weights(self, learning_rate):  
2     for layer in self.layers:  
3         layer.update(learning_rate)
```

Method ini memanggil `update()` pada setiap layer, yang implementasinya sebagai berikut:



```
1 def update(self, learning_rate):  
2     self.weights -= learning_rate * self.grad_weights  
3     self.biases -= learning_rate * self.grad_biases  
4     if self.use_rmsnorm:  
5         self.rmsnorm.update(learning_rate)
```

Pembaruan bobot dilakukan dengan formula:

Bobot baru = Bobot lama - (learning rate \times gradien bobot)

Bias baru = Bias lama - (learning rate \times gradien bias)

Note: nilai gradient pasti bernilai negatif

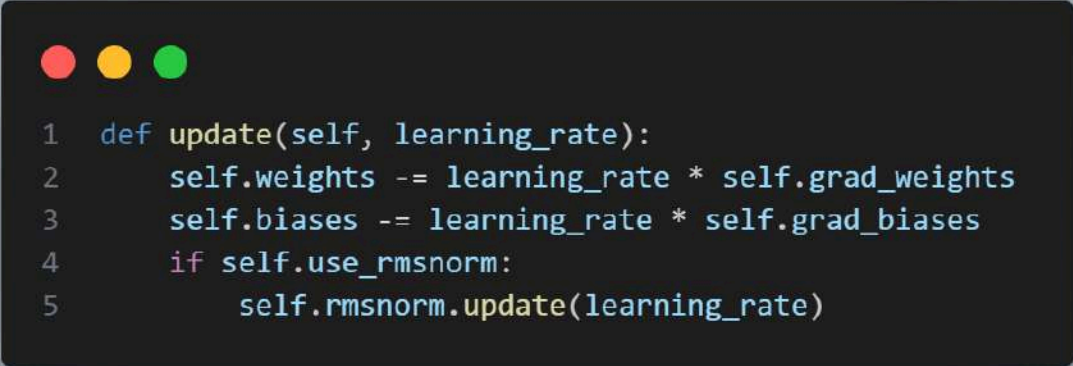
Parameter `learning_rate` berperan penting dalam mengontrol seberapa besar perubahan yang terjadi pada setiap langkah pembaruan. Nilai yang terlalu besar dapat menyebabkan algoritma melewati solusi optimal, sementara nilai yang terlalu kecil membuat proses konvergensi menjadi sangat lambat.

Dalam implementasi kami, kami menambahkan early stopping untuk mencegah overfitting dan menghemat waktu komputasi. Mekanisme ini akan menghentikan proses pelatihan jika tidak ada perbaikan signifikan pada nilai loss selama beberapa epoch berturut-turut. Dengan cara ini, pelatihan dapat dihentikan lebih awal ketika model sudah mencapai performa yang optimal, sehingga menghindari pelatihan yang tidak perlu dan menghemat sumber daya komputasi.

```
1  if stop_in_convergence:
2      if abs(epoch_loss - best_loss) < tol:
3          no_improve_count += 1
4      else:
5          no_improve_count = 0
6          best_loss = epoch_loss
7
8      if no_improve_count >= patience:
9          if verbose:
10             print(f"Early stopping triggered at epoch {epoch+1} - loss did not improve more than {tol} for {patience} consecutive epochs.")
11             break
```

2.4 Penjelasan Regularisasi

Pada tugas besar ini, dilakukan 2 metode regularisasi, L1 dan L2. Secara singkat L1 berfokus pada mengurangi *sparsity* dari partisipasi *weight* pada sebuah output, dimana kemungkinan bobot gradien diinisialisasi menjadi 0 dan dianggap hilang dari posisi *neural*.

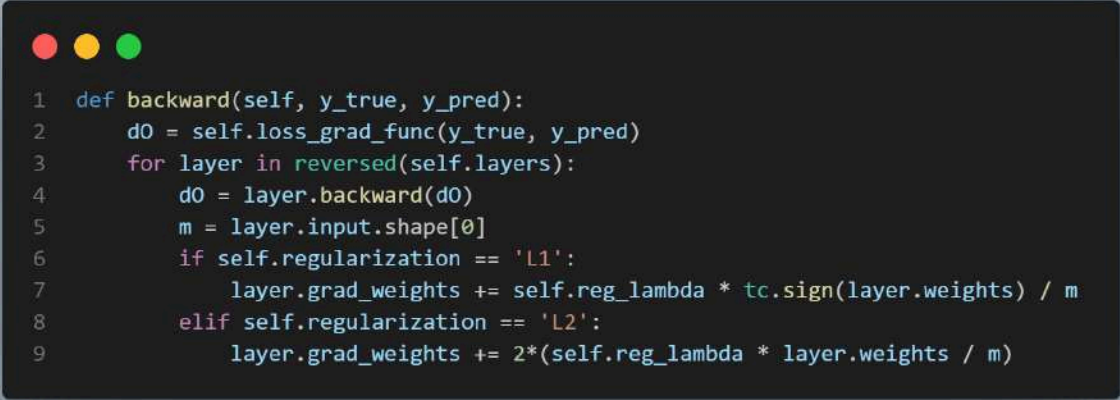


```

1 def update(self, learning_rate):
2     self.weights -= learning_rate * self.grad_weights
3     self.biases -= learning_rate * self.grad_biases
4     if self.use_rmsnorm:
5         self.rmsnorm.update(learning_rate)

```

Perubahan dilakukan pada bobot gradien , dimana pada konsep ini, kita secara tidak langsung mendorong bobot untuk menjadi 0. Hal yang sama terjadi pada L2 , dimana L2 melakukan hal yang sama , hanya saja, daripada mendekatkan ke 0 dengan melakukan *sign*, rumus L2 menyesuaikan pengurangan bobot sesuai dengan nilai bobot itu sendiri. Semakin besar bobot , semakin besar pula pengurangannya. Dari gambar diatas terlihat bahwa fungsi perubahan bobot akan mengurangi bobot sesuai bobot gradien. Untuk nilai positif akan dikurangi dengan positif pula sehingga lebih menuju ke 0 , sedangkan untuk negatif akan menjadi penjumlahan yang juga menuju 0.



```

1 def backward(self, y_true, y_pred):
2     d0 = self.loss_grad_func(y_true, y_pred)
3     for layer in reversed(self.layers):
4         d0 = layer.backward(d0)
5         m = layer.input.shape[0]
6         if self.regularization == 'L1':
7             layer.grad_weights += self.reg_lambda * tc.sign(layer.weights) / m
8         elif self.regularization == 'L2':
9             layer.grad_weights += 2*(self.reg_lambda * layer.weights / m)

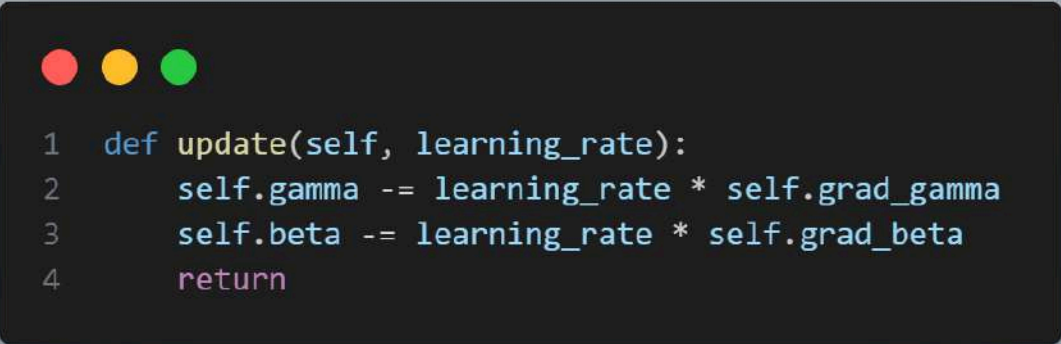
```

Untuk L2 , lebih nyaman untuk melihat *snippet* diatas, dimana dari sana kita bisa dengan mudah melihat bahwa semakin besar bobot , semakin besar pula penalti pengurangan , dimana nilai dari

bobot gradien berbanding lurus dengan nilai bobot. Hasil eksperimen akan terlihat pada percobaan dibawah nantinya. Tindakan ini tidak mengurangi tenaga komputasi, bahkan dalam implementasinya di tugas besar ini, memakan lebih banyak tenaga komputasi. Namun , akan terasa perubahan pada saat *inference* karena adanya regularisasi bobot.

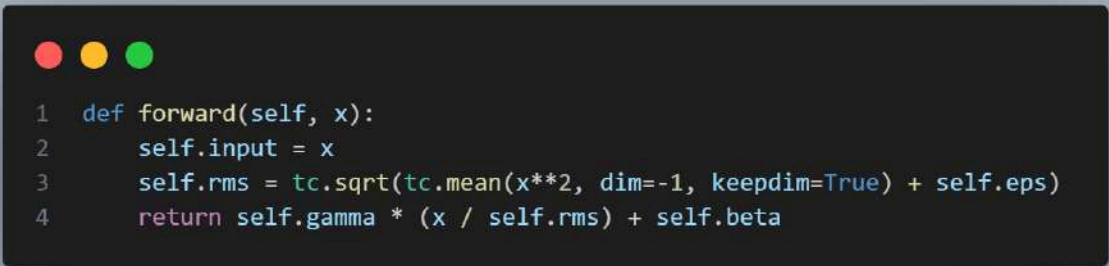
2.5 Penjelasan RMSNorm

Berbeda dengan regularisasi yang mengandalkan penalti pada saat pembangunan model *neural network*. RMSNorm berfokus pada perubahan pada saat *forward* , mengubah hasil aktivasinya. Salah satu hal menarik dari RMSNorm adalah kemampuan untuk menyesuaikan normalisasi terhadap aktivasi.



```
1 def update(self, learning_rate):
2     self.gamma -= learning_rate * self.grad_gamma
3     self.beta -= learning_rate * self.grad_beta
4     return
```

Terdapat komponen gamma dan beta, yang sebenarnya memiliki karakteristik yang cukup mirip dengan Adam. Gamma bertugas melakukan perubahan skalar sedangkan beta bertanggung jawab atas *shifting*. Secara keseluruhan , keduanya terus diperbarui untuk setiap pembelajaran.



```
1 def forward(self, x):
2     self.input = x
3     self.rms = tc.sqrt(tc.mean(x**2, dim=-1, keepdim=True) + self.eps)
4     return self.gamma * (x / self.rms) + self.beta
```

Fungsi forward khusus RMSNorm hanya menerapkan normalisasi sesuai formula dari RMS norm.

$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i, \quad \text{where } \text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n} \sum_{i=1}^n a_i^2}.$$

Dan setelah didapat RMS dihitung dengan:

$$y = \frac{x}{\sqrt{\text{RMS}[x] + \epsilon}} * \gamma$$

Untuk *backward propagation* juga dilakukan perubahan:

```
1 def backward(self, grad_output):
2     normalized = self.input / self.rms
3     self.grad_gamma = (grad_output * normalized).sum(dim=0)
4     self.grad_beta = grad_output.sum(dim=0)
5     dnormalized = grad_output * self.gamma
6     n = self.input.shape[-1]
7     grad_input = dnormalized / self.rms - self.input * ((dnormalized * self.input).sum(dim=-1, keepdim=True)) / (n * self.rms**3)
8     return grad_input
```

Implementasinya mirip dengan ketika dilakukan *forward propagation*, dimana urutannya dibalik dan mengandalkan turunan. Pemanggilan kedua fungsi diatas dilakukan pada nilai yang seharusnya lgsg *di return* sewaktu training dengan dalih untuk dilakukan normalisasi.

Efek dari hasilnya yaitu nilai akhir yang lebih tidak terdistribusi secara luas. Meskipun sebenarnya berbalik dengan beberapa konsep yang mengatakan distribusi merata lebih baik, maksud dari normalisasi ini yaitu menghilangkan *gap* yang terlalu berlebihan antar 2 bobot, namun masih mempertahankan jarak yang seharusnya. Normalisasi RMS menjadi metrik yang cocok untuk direpresentasikan sebagai cara untuk mencegah pertumbuhan bobot yang berlebihan.

BAB III

HASIL PENGUJIAN

Pada pengujian ini, dilakukan inisialisasi bobot yang sama untuk semua *layer* untuk memudahkan dan menyetarakan pengujian. Selain itu, perlu dicatat bahwa data masukan telah dilakukan proses normalisasi dari rentang awal 0–255 menjadi 0–1, sehingga input yang diterima oleh model sudah berada dalam skala yang seragam.

3.1 Pengaruh *depth* dan *width*

Percobaan ini akan menggunakan berbagai spesifikasi umum seperti:

1. *Epoch* = 1000
2. *Learning rate* = 0.001
3. Fungsi aktivasi :
 - ReLU untuk *hidden layer*
 - *Softmax* untuk *output layer*
4. *Loss Function*: *Categorical Cross-Entropy*
5. Inisialisasi bobot : Xavier dan He

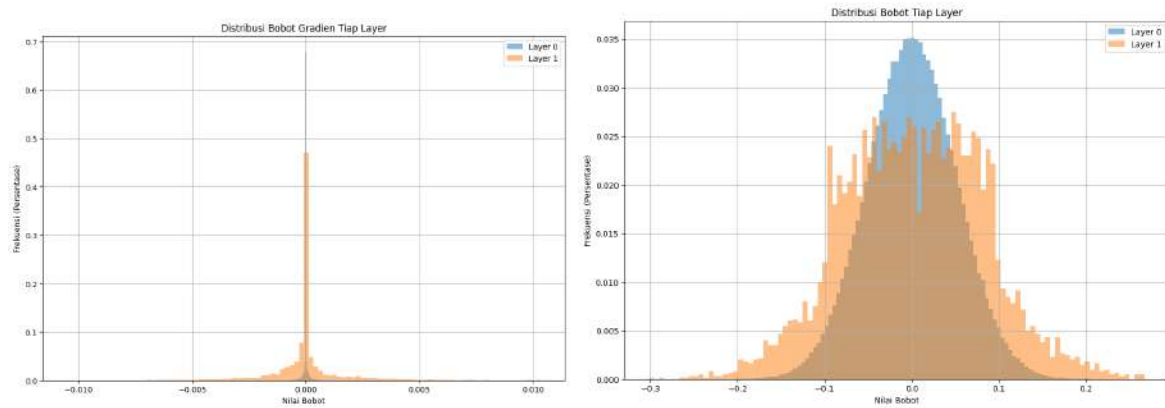
3.1.1 Uji coba untuk *width* berbeda

Uji coba pertama dengan spesifikasi

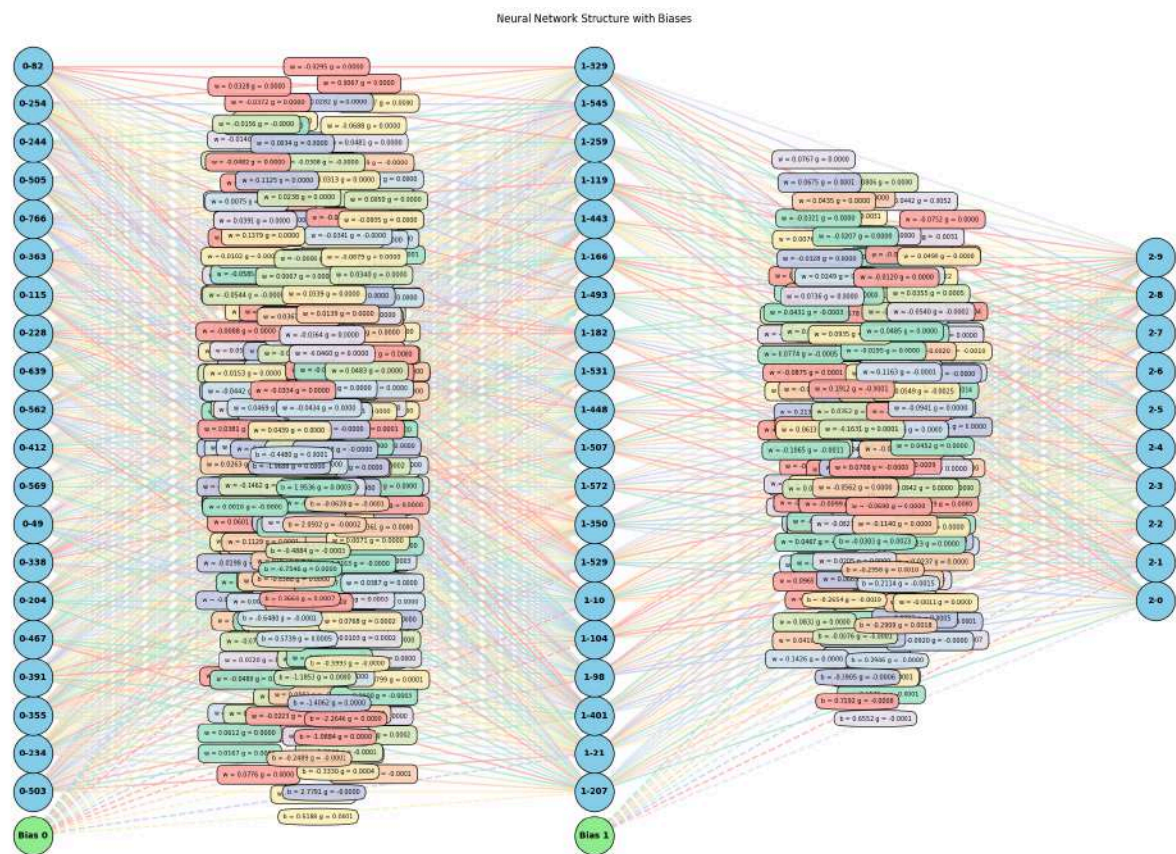
1. Input Layer = 784 neuron
2. Hidden Layer 1 = 600 neuron
3. Output Layer = 10 neuron

Hasil yang didapat:

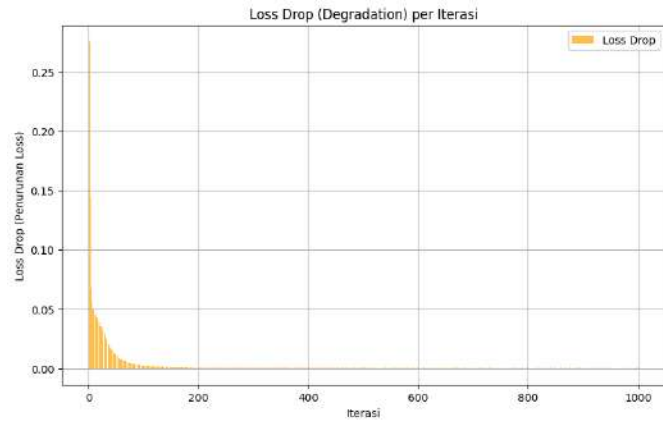
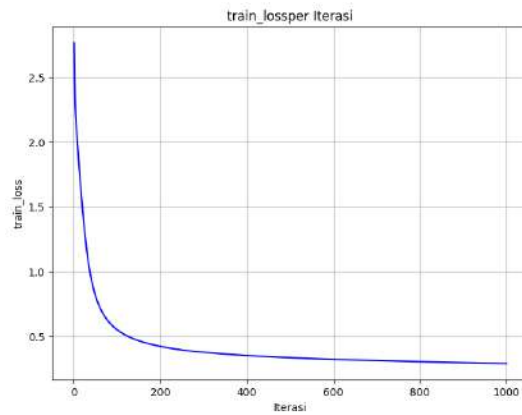
Akurasi model yang didapat yaitu 0.9214285612106323. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 533.78 detik.. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



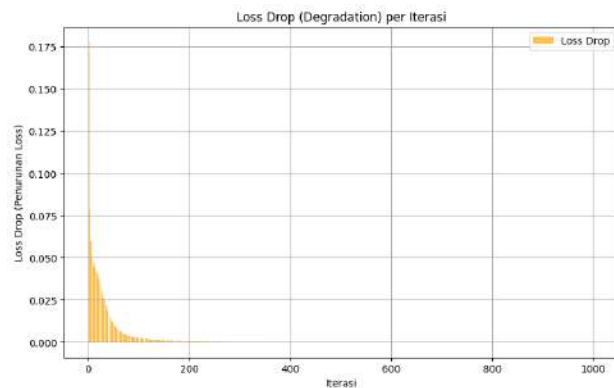
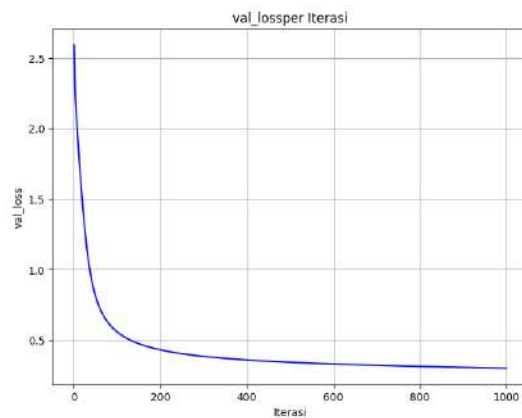
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validasi:

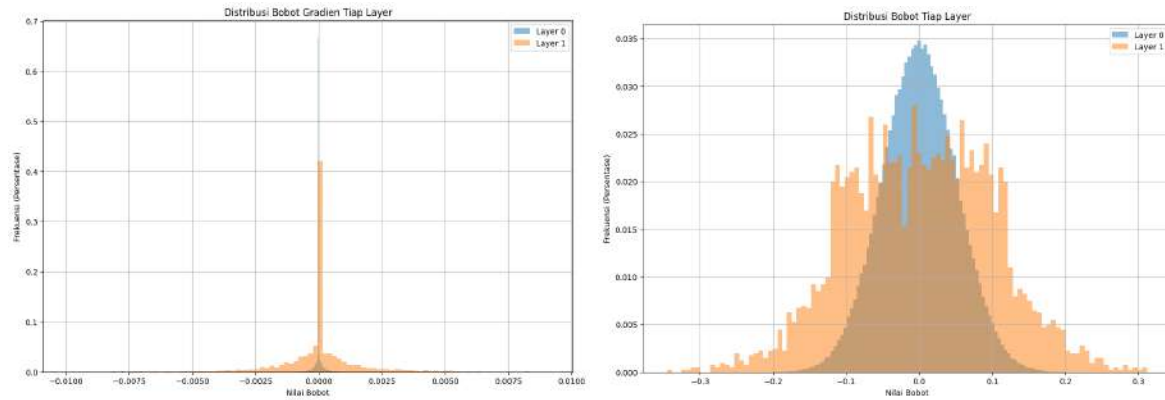


Uji coba kedua dengan spesifikasi:

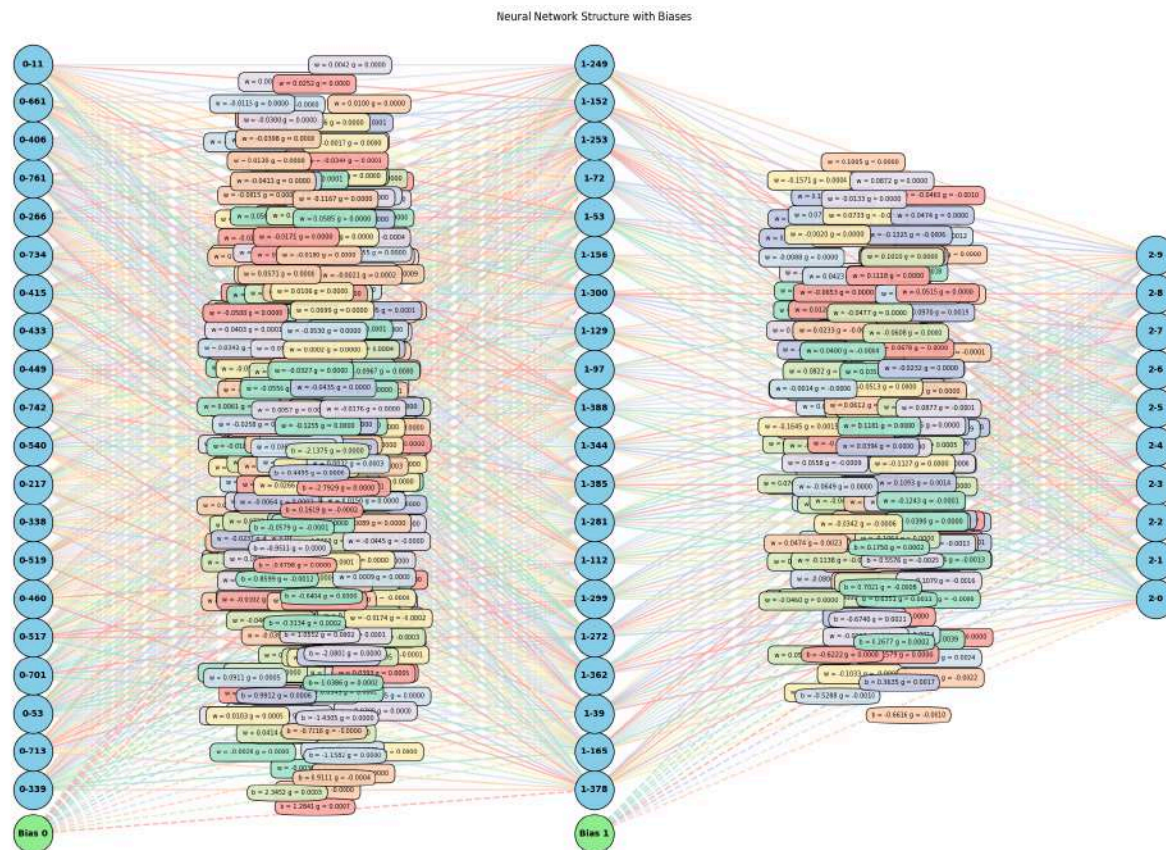
1. Input Layer = 784 neuron
2. Hidden Layer 1 = 400 neuron
3. Output Layer = 10 neuron

Hasil yang didapat:

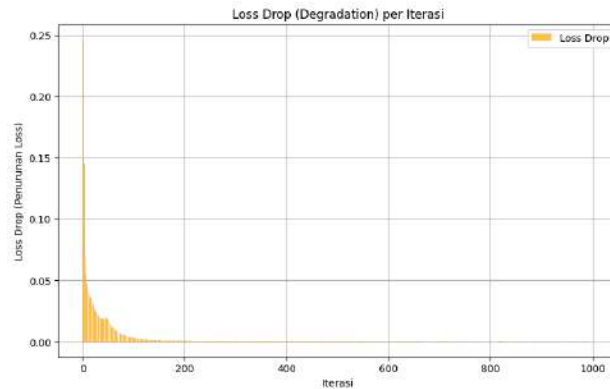
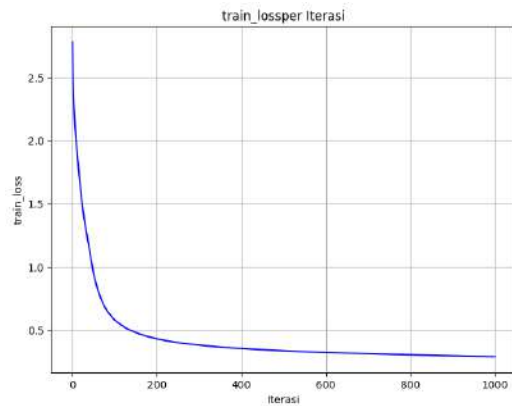
Akurasi model yang didapat yaitu 0.92149996757507323. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 538.46 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



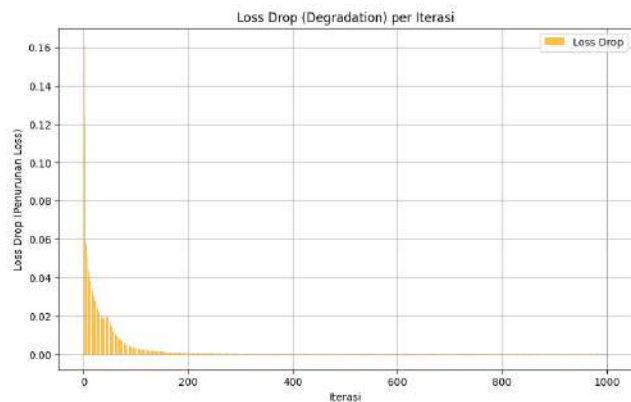
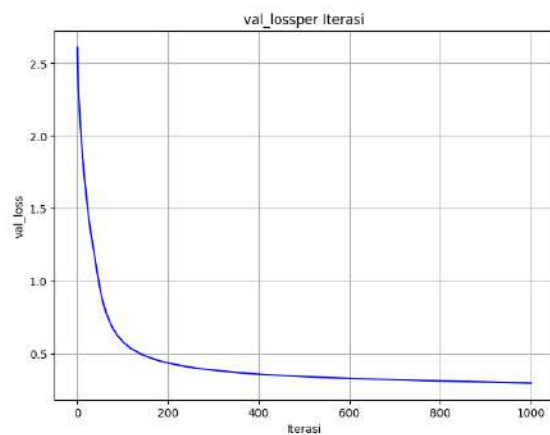
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validation:

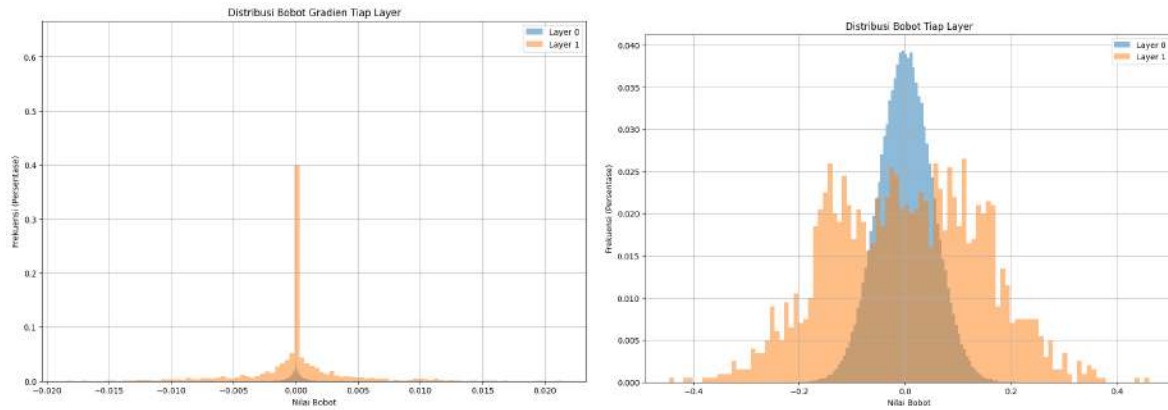


Uji coba ketiga dengan spesifikasi:

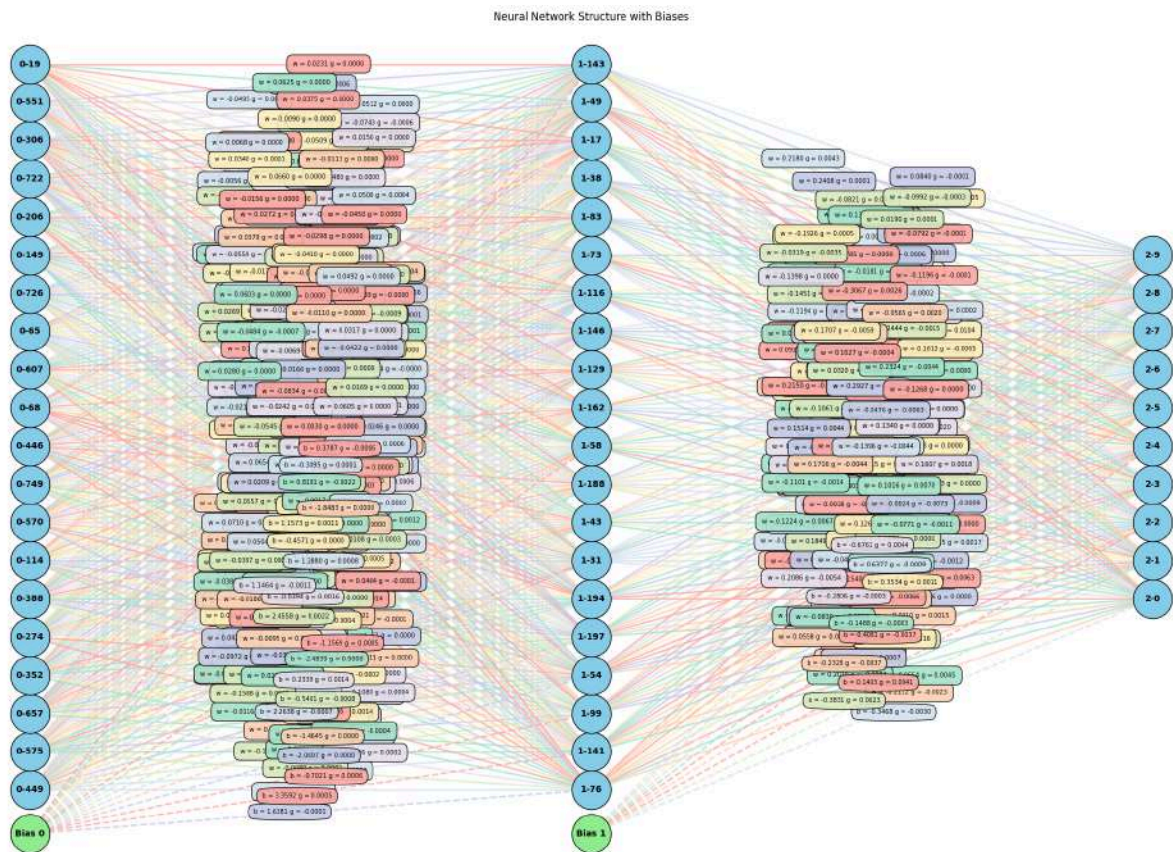
1. Input Layer = 784 neuron
2. Hidden Layer 1 = 200 neuron
3. Output Layer = 10 neuron

Hasil yang didapat:

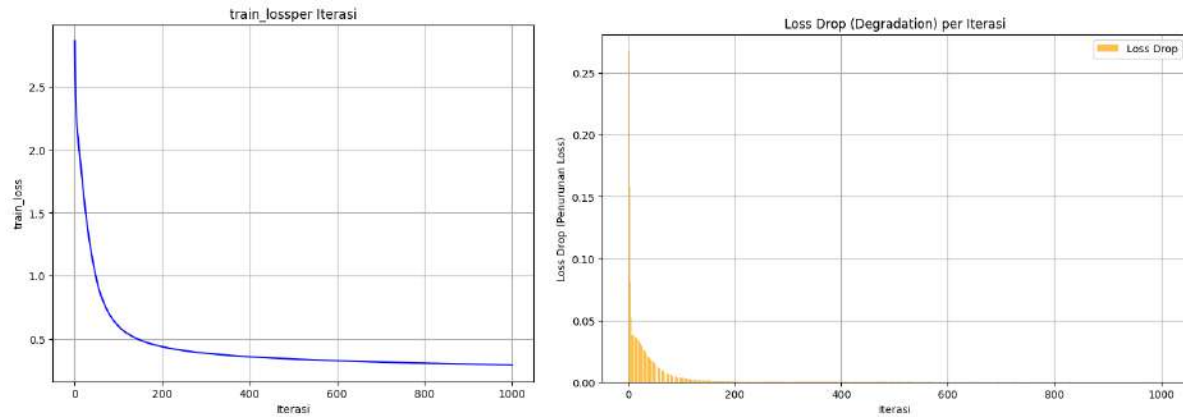
Akurasi model yang didapat yaitu 0.9204285740852356. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 553.50 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



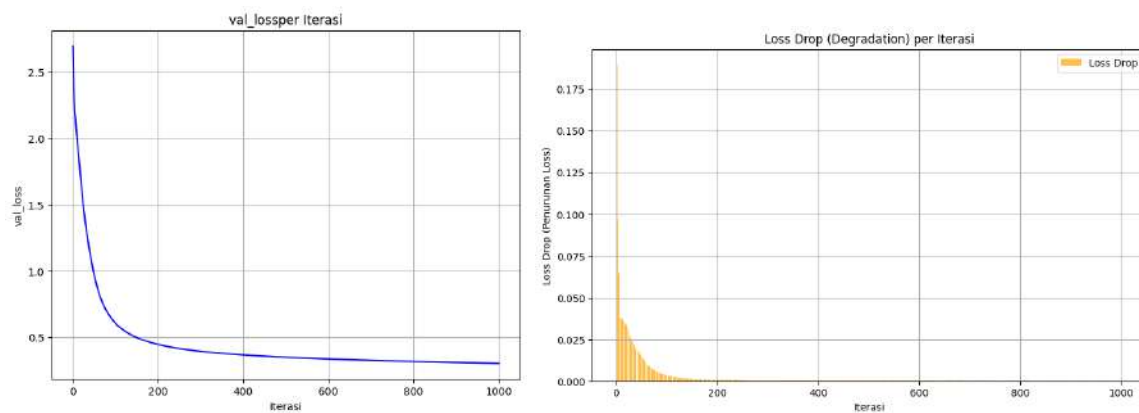
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validation:



3.1.2 Analisis untuk *width* berbeda

Hasil dari percobaan antara *width* yang berbeda tidak memberikan sebuah *insight* khusus terkait apakah lebih baik dilakukan pengujian pada *width* lebih luas atau tidak. Perbedaan akurasi terlihat acak dan tidak ada pola tertentu dalam kenaikan ataupun penurunan. Selain itu, hasilnya juga berbeda secara tidak signifikan. Adapun perbedaan utama dapat dilihat pada pembagian jumlah konsistensi yang berada pada posisi 0. Ketiga percobaan menggunakan ReLU, dan semakin besar *width*, jumlah weight yang bernilai 0 lebih banyak dibandingkan, dimana semakin membesar persentasenya dengan tiap *width*.

Karena percobaan menggunakan ReLU, untuk semua nilai negatif akan dianggap menjadi 0. Karena inisialisasi dilakukan dengan He (pada kasus ini), nilai dari bobot awal sangat lah kecil, bahkan ada kemungkinan dibawah dari nilai *learning rate*. Oleh karena itu, ketika dilakukan *backward propagation*, banyak bobot yang bertumpu pada nilai 0. Jika dimodifikasi nilai

learning rate menjadi lebih besar, maka akan semakin terlihat *range* distribusi akan semakin luas pula. (Akan dilihat di perbedaan percobaan *learning rate* dibawah nantinya)

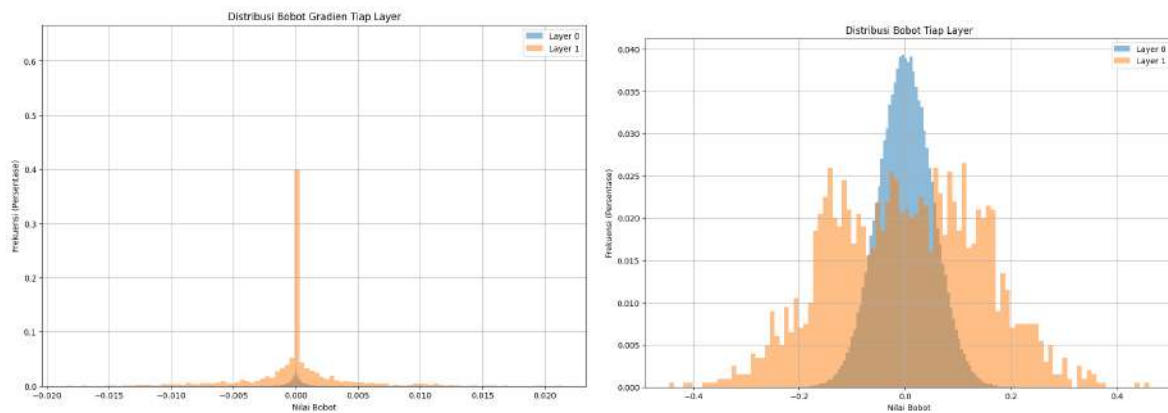
3.1.3 Uji coba untuk *depth* berbeda

Uji coba ketiga dengan spesifikasi:

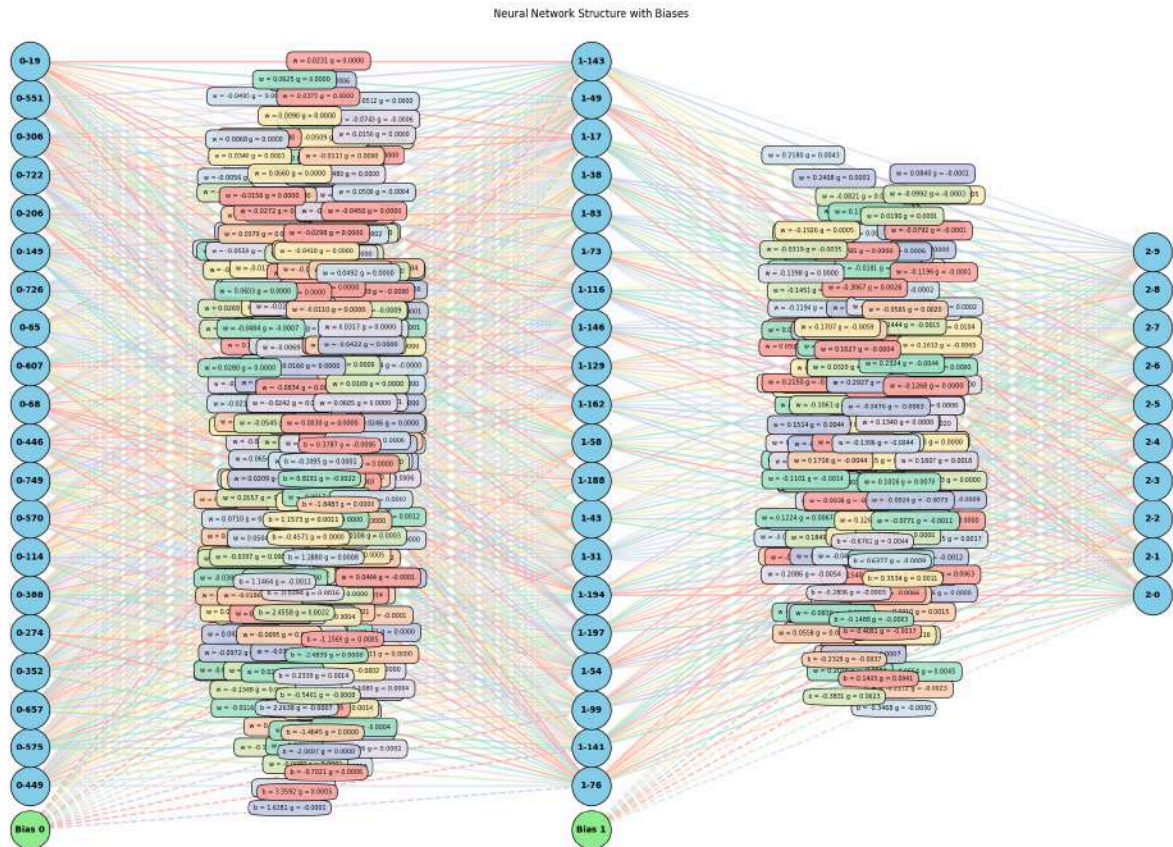
4. Input Layer = 784 neuron
5. Hidden Layer 1 = 200 neuron
6. Output Layer = 10 neuron

Hasil yang didapat:

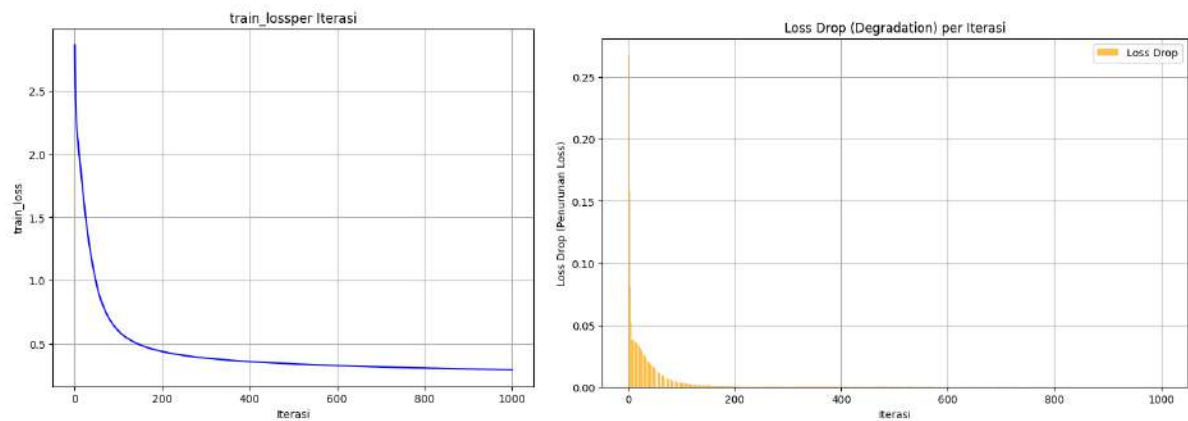
Akurasi model yang didapat yaitu 0.9204285740852356. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 553.50 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



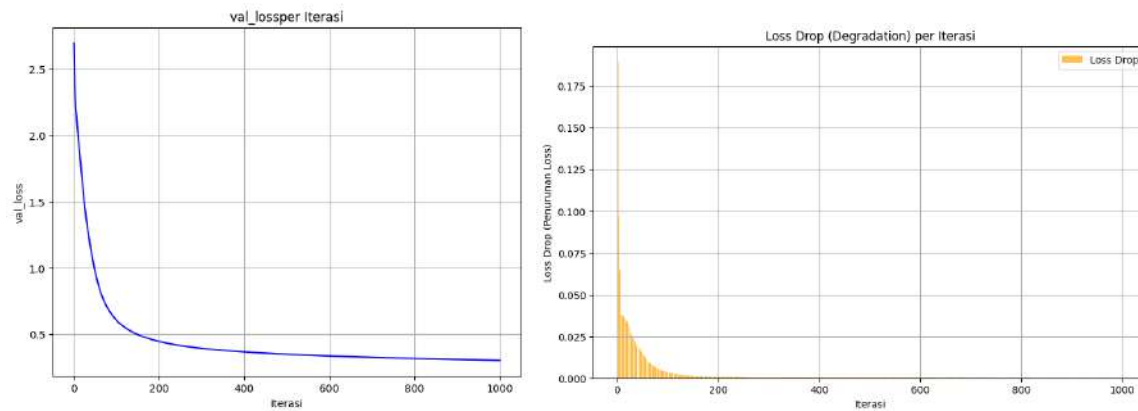
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validation:

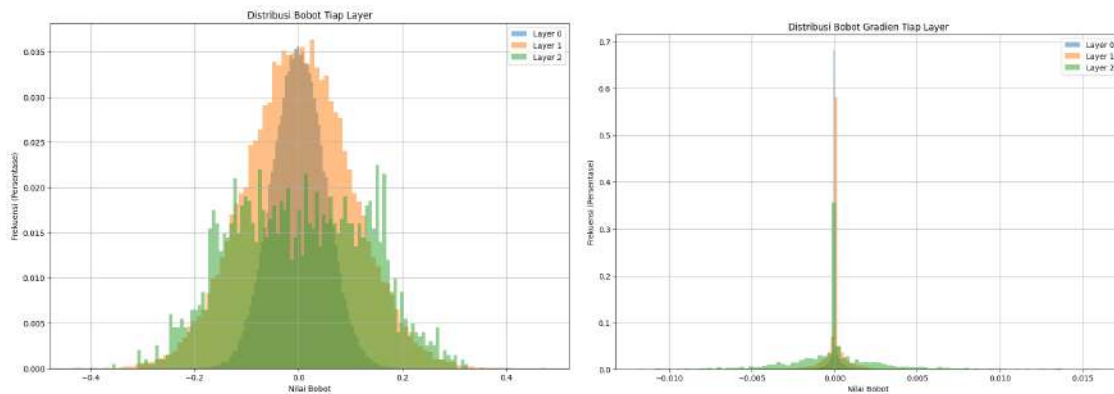


Uji coba kedua dengan spesifikasi

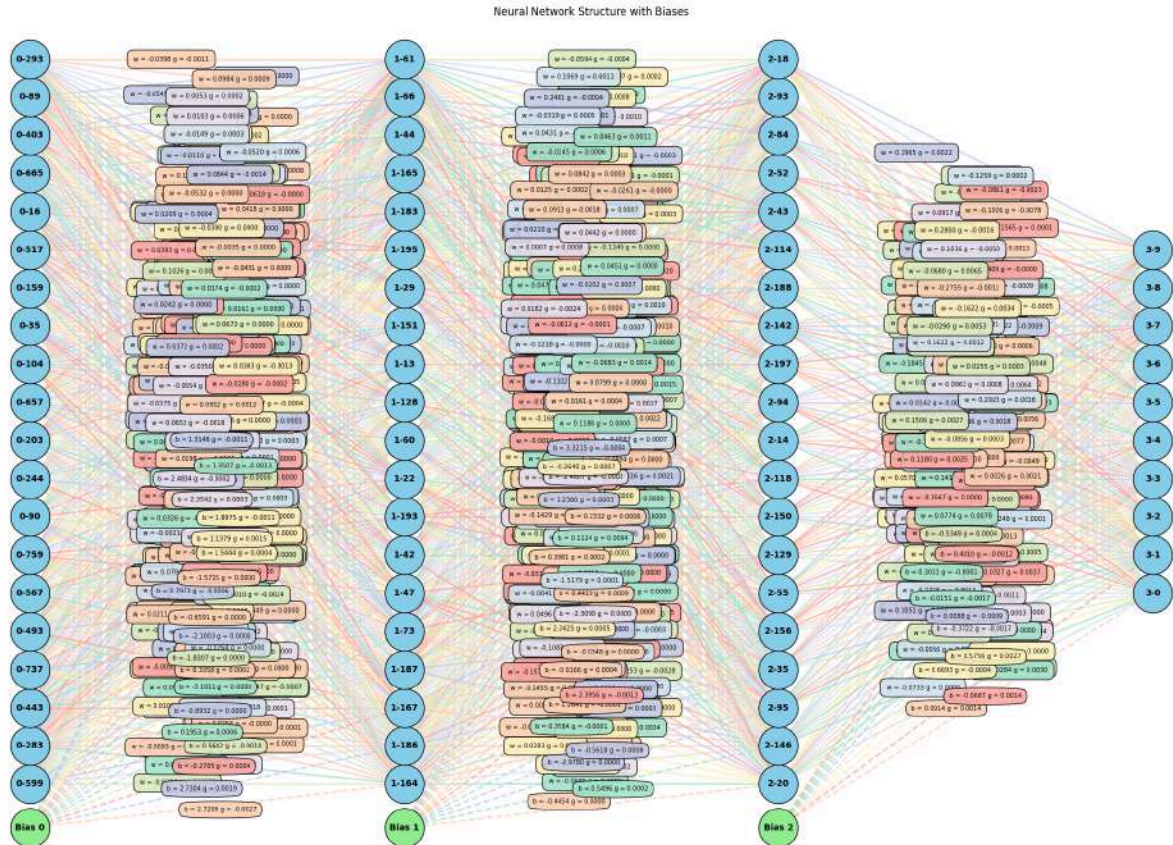
1. Input Layer = 784 neuron
2. Hidden Layer 1 = 200 neuron
3. Hidden Layer 2 = 200 neuron
4. Output Layer = 10 neuron

Hasil yang didapat:

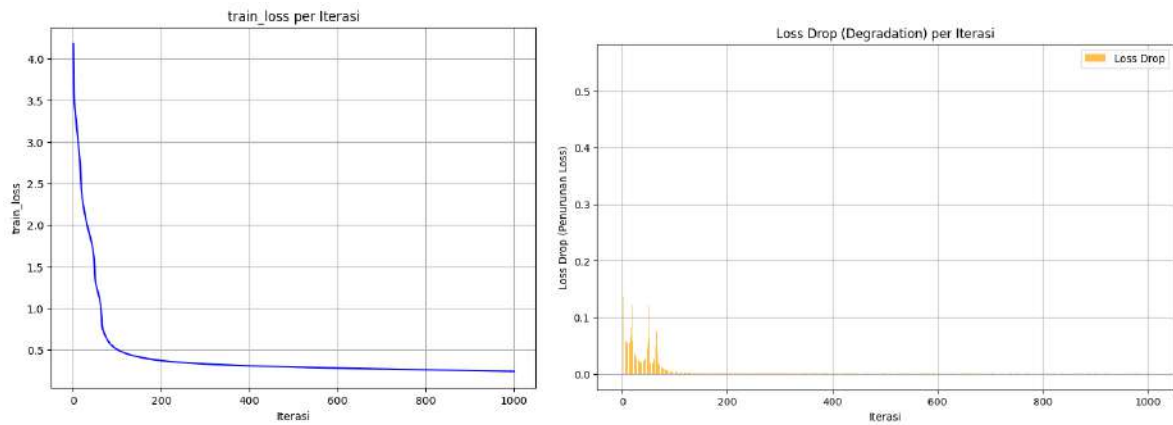
Akurasi model yang didapat yaitu 0.95921. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 350.08 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



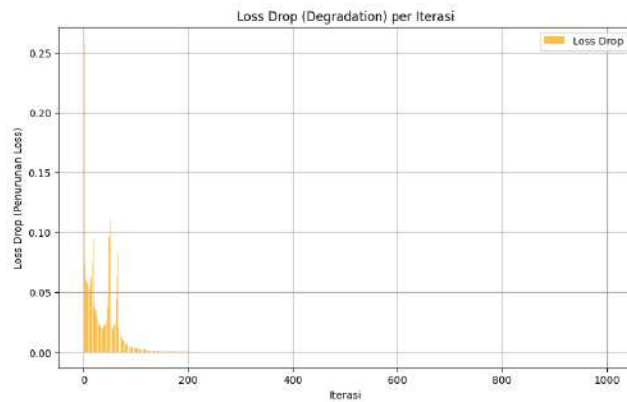
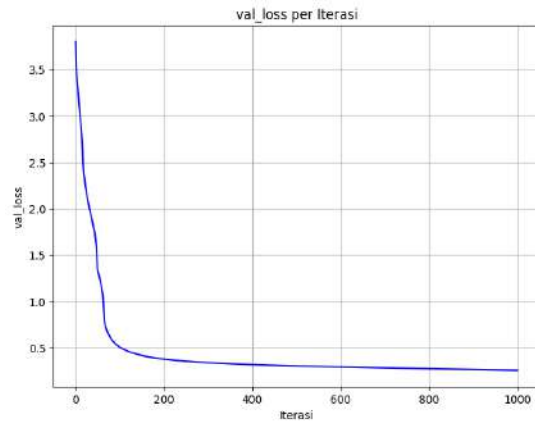
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validasi:

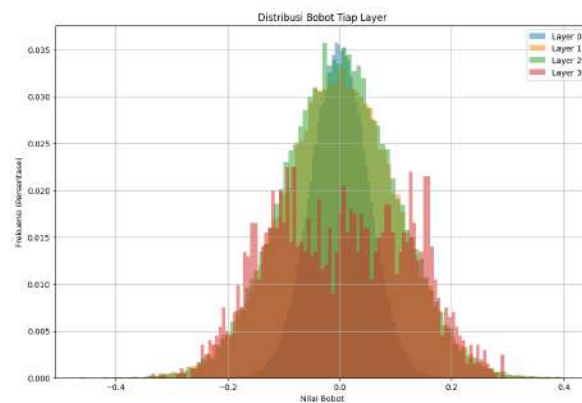
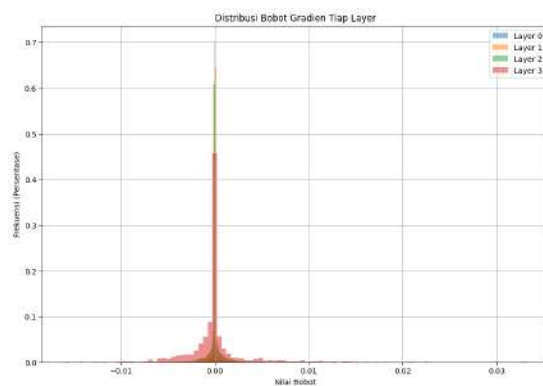


Uji coba pertama dengan spesifikasi

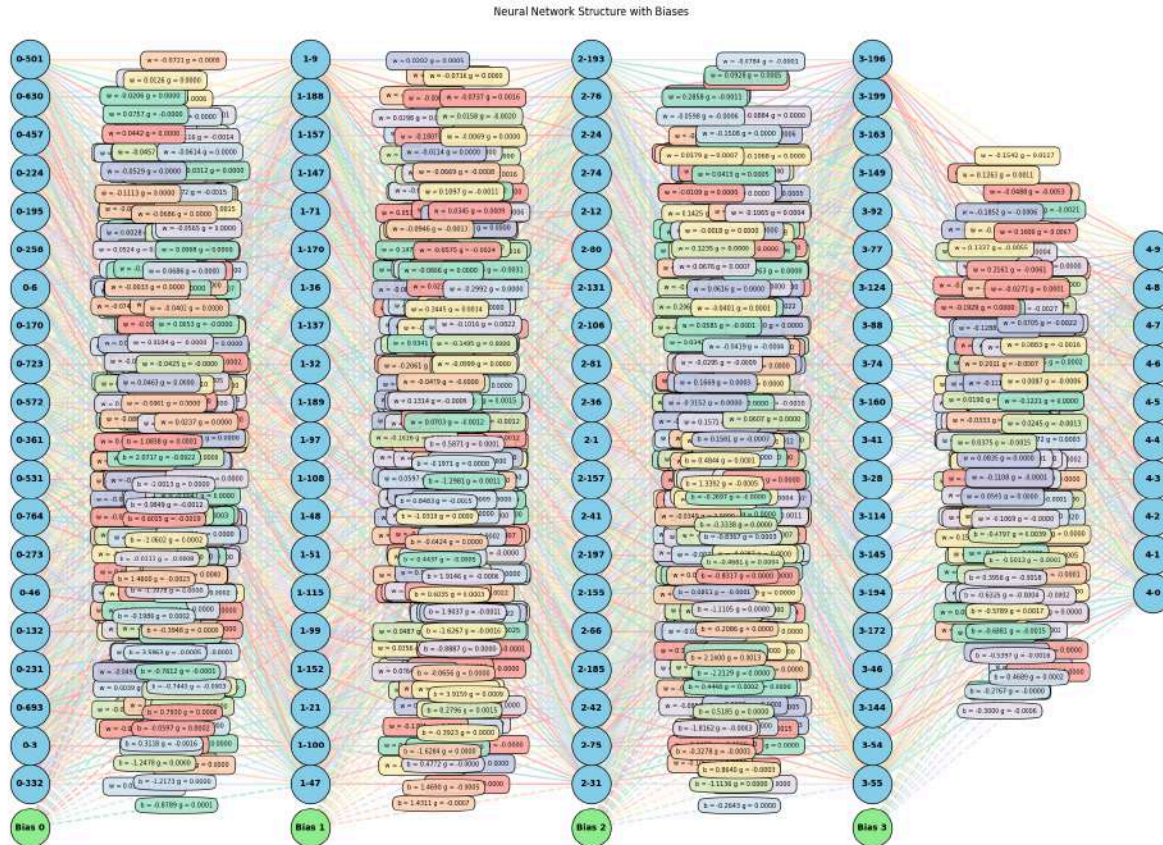
1. Input Layer = 784 neuron
2. Hidden Layer 1 = 200 neuron
3. Hidden Layer 2 = 200 neuron
4. Hidden Layer 3 = 200 neuron
5. Output Layer = 10 neuron

Hasil yang didapat:

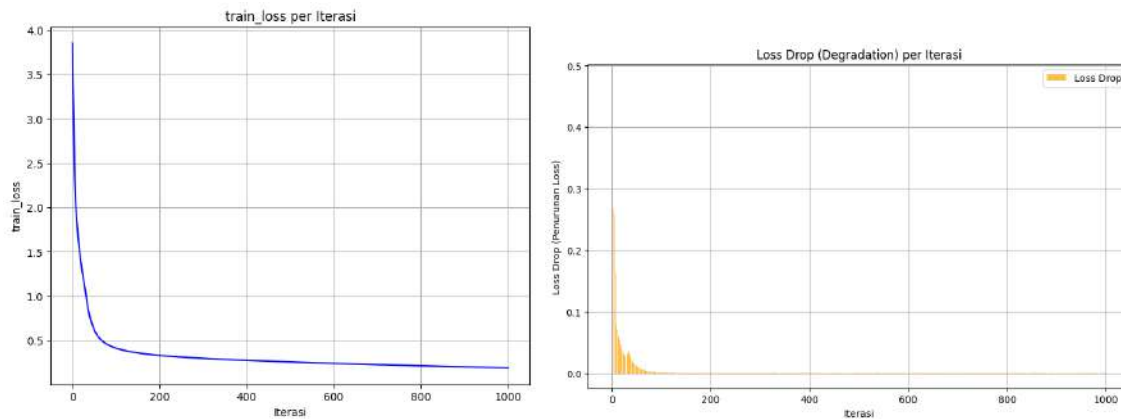
Akurasi model yang didapat yaitu 0.9440714120864868. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 924.73 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



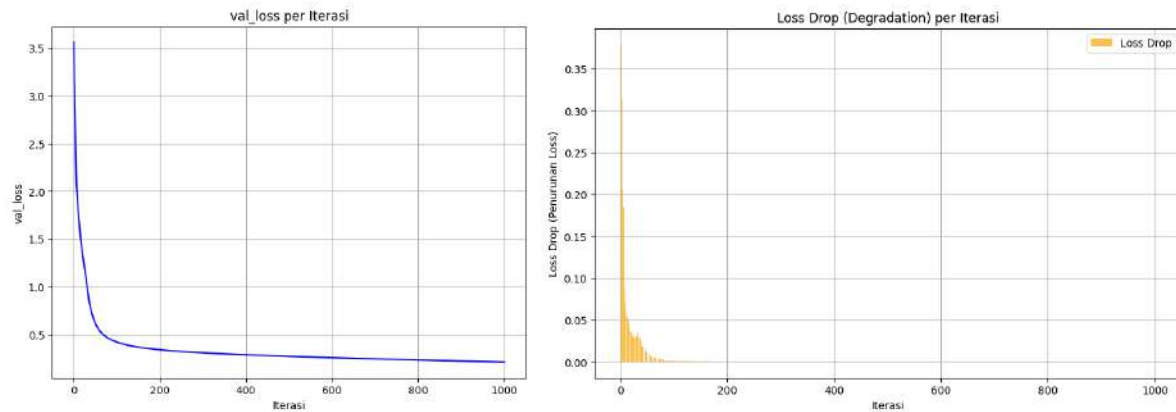
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validasi:



3.1.4 Analisis untuk *depth* berbeda

Perbedaan utama ada pada akurasi , dimana dalam kasus ini , semakin dalam *depth*, menghasilkan akurasi yang semakin baik pula. Dalam susunan ini, terlihat bahwa untuk layer terakhir, bobotnya selalu terdistribusi dengan baik. Perbedaannya ada pada transisi antar bobot, dimana untuk model dengan *depth* yang rendah, transisi dari *weight* layer ke n dengan $n+1$ semakin luas. Hal ini tentu dapat mempengaruhi cara kerja model, terutama dari sisi generalisasi. Permasalahan yang dihadapi oleh dataset kemungkinan berhubungan dengan generalisasi, dimana terlihat bahwa model dengan *depth* yang lebih besar memiliki permulaan *train loss* yang lebih kecil, serta memiliki *loss degradation* yang lebih signifikan untuk setiap iterasinya. Kita tidak seharusnya hanya melihat dari akurasi karena adanya *snippet* ataupun potongan data yang juga berbeda sehingga salah satu metrik yang paling cocok yaitu dengan melihat *loss start* serta *loss drop* nya. Ketiga eksperimen membuktikan performa yang sebenarnya cukup baik, melihat bahwa meskipun dengan *loss start* yang tinggi , tetap memberikan sebuah hasil yang baik dikarenakan *loss drop* yang besar pula. Terdapat sedikit lonjakan pada *loss drop* percobaan kedua. Hal ini perlu diperhatikan dimana dalam implementasi nyata, terkadang diberikan sebuah kondisi jika *loss* tidak berubah atau malah membesar pada *patience* tertentu , model akan memberikan *early stopping*.

3.2 Pengaruh fungsi aktivasi hidden layer

Percobaan ini akan menggunakan berbagai spesifikasi umum seperti:

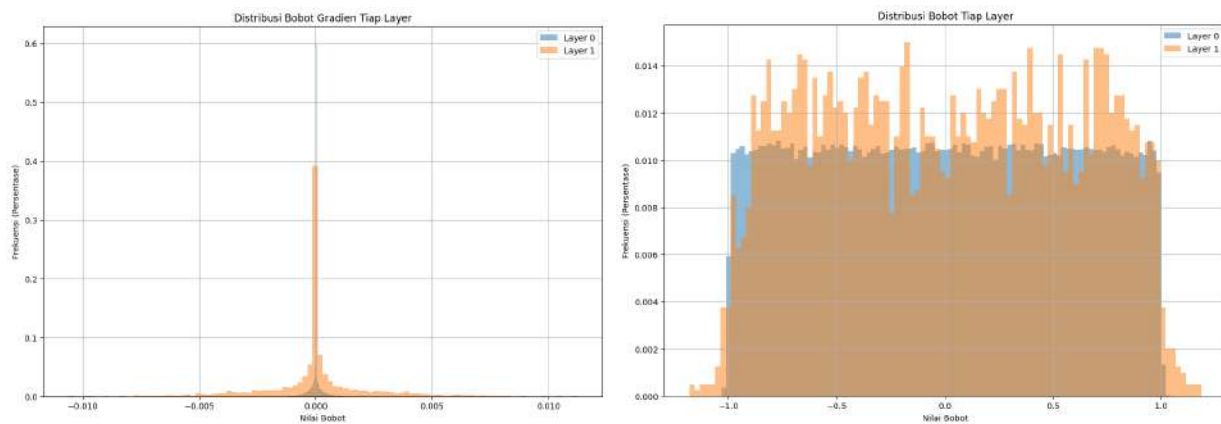
1. *Epoch* = 1000
2. *Input Layer* = 784
3. *Hidden Layer* = 400
4. *Output Layer* = 10

5. *Learning Rate* = 0.001
6. *Loss Function: Categorical Cross-Entropy*
7. Inisialisasi bobot : random uniform (-1,1)

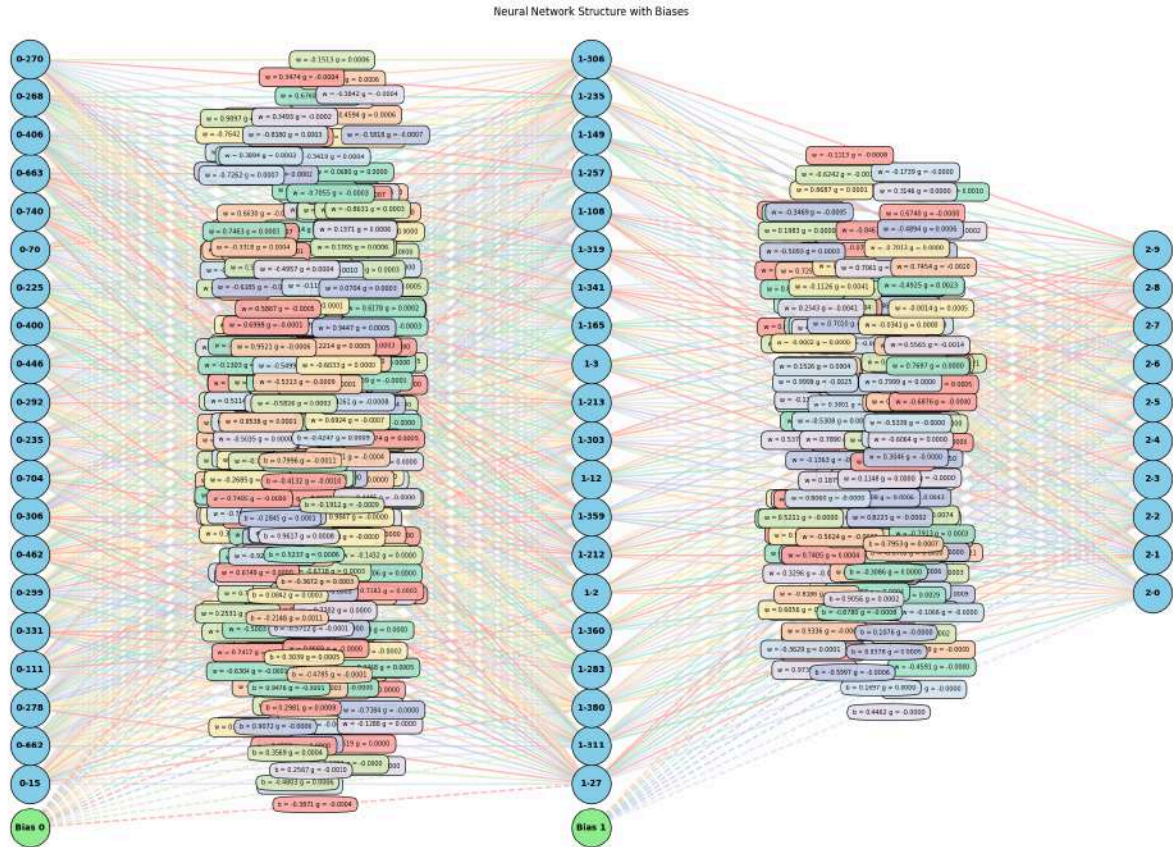
3.2.1 Uji coba *linear*

Hasil yang didapat:

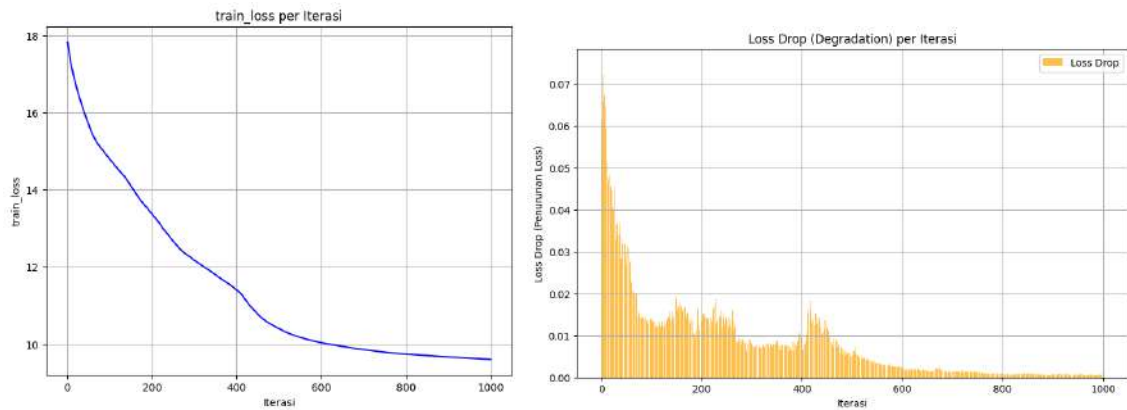
Akurasi model yang didapat yaitu 0.5145000219345093. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 491.92 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



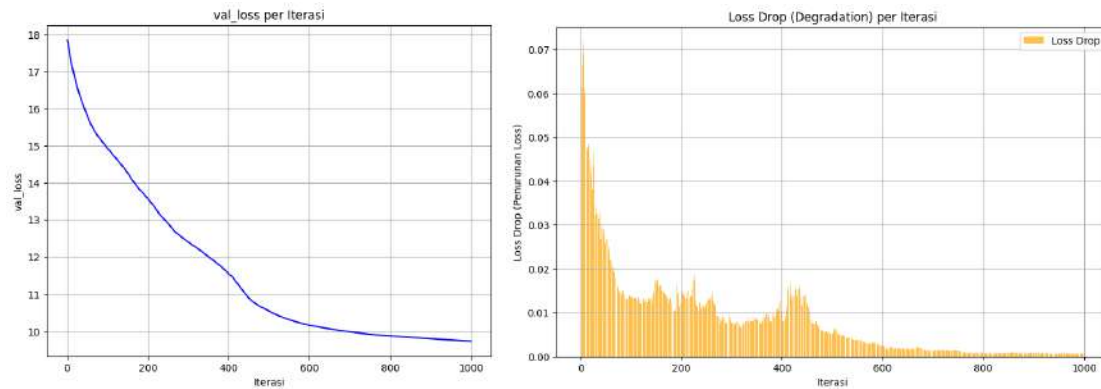
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



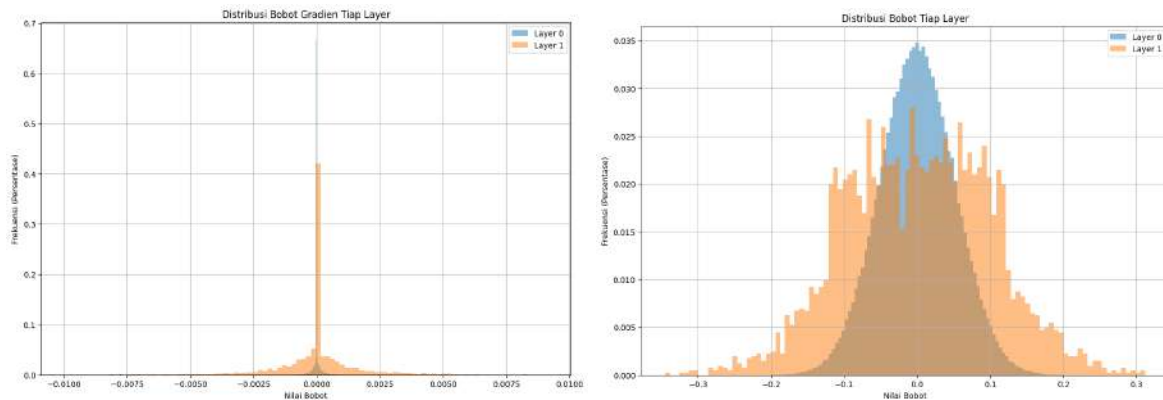
Untuk validasi:



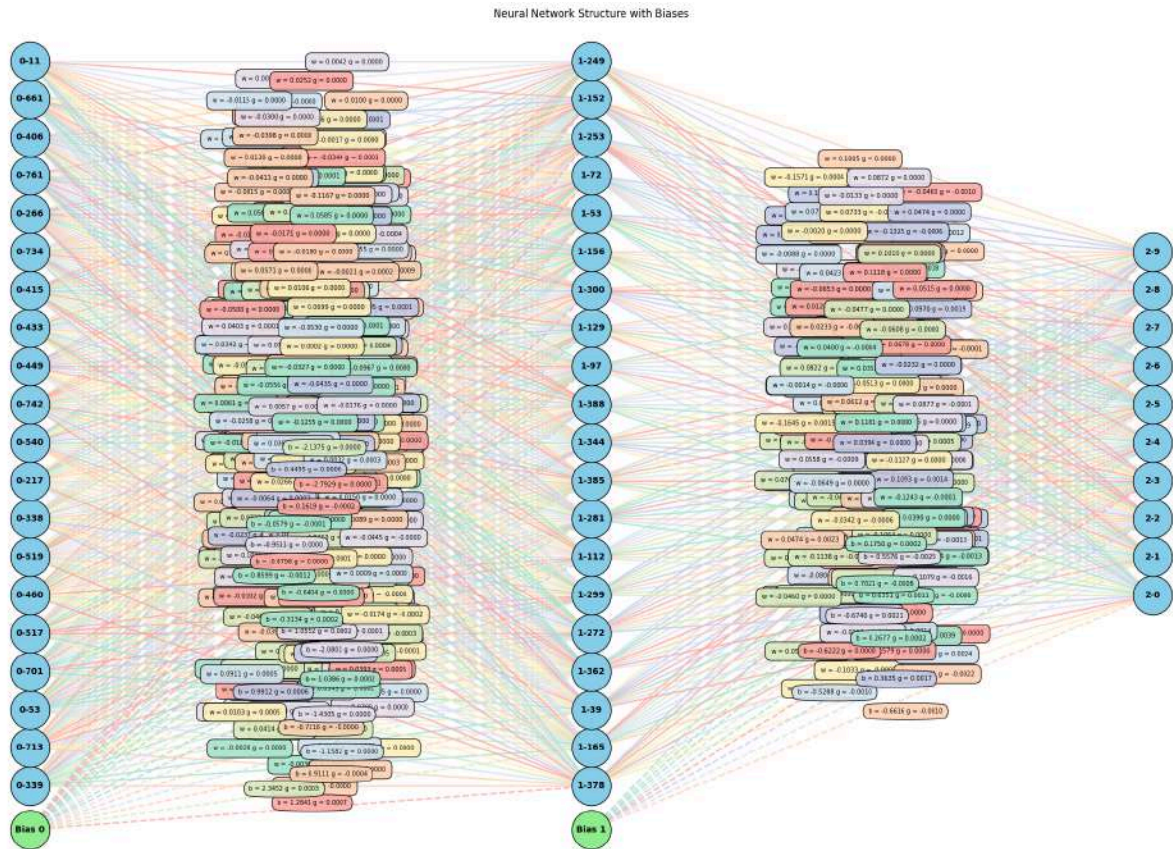
3.2.2 Uji coba ReLU (Menggunakan Xavier He , sebagai baseline akurasi yang baik)

Hasil yang didapat:

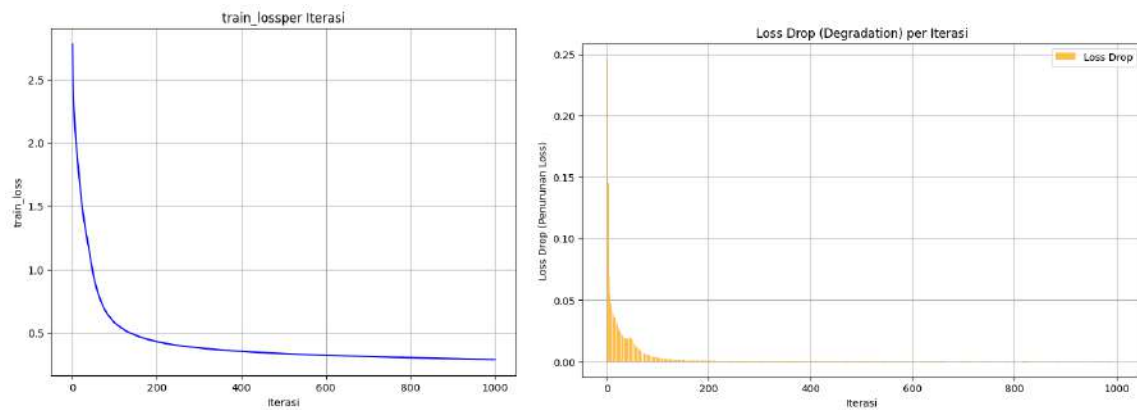
Akurasi model yang didapat yaitu 0.92149996757507323. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 538.46 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



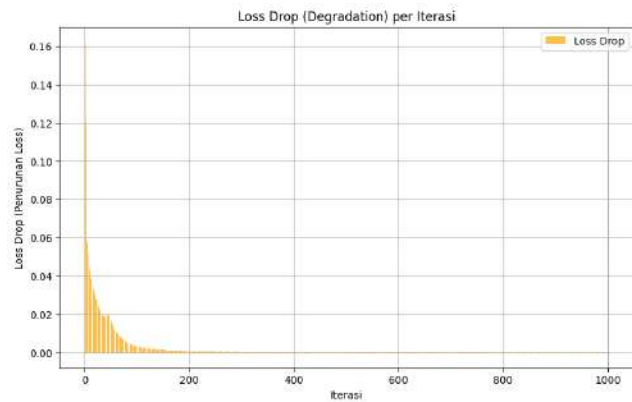
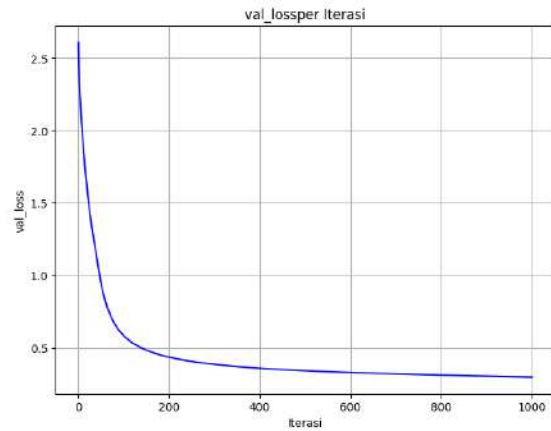
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



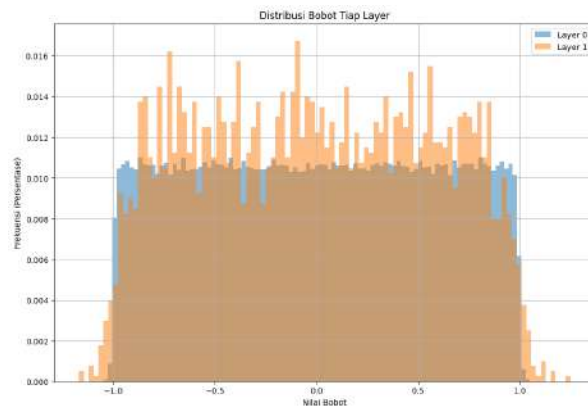
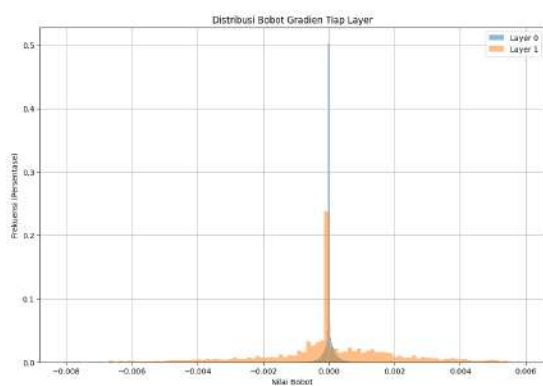
Untuk Validation:



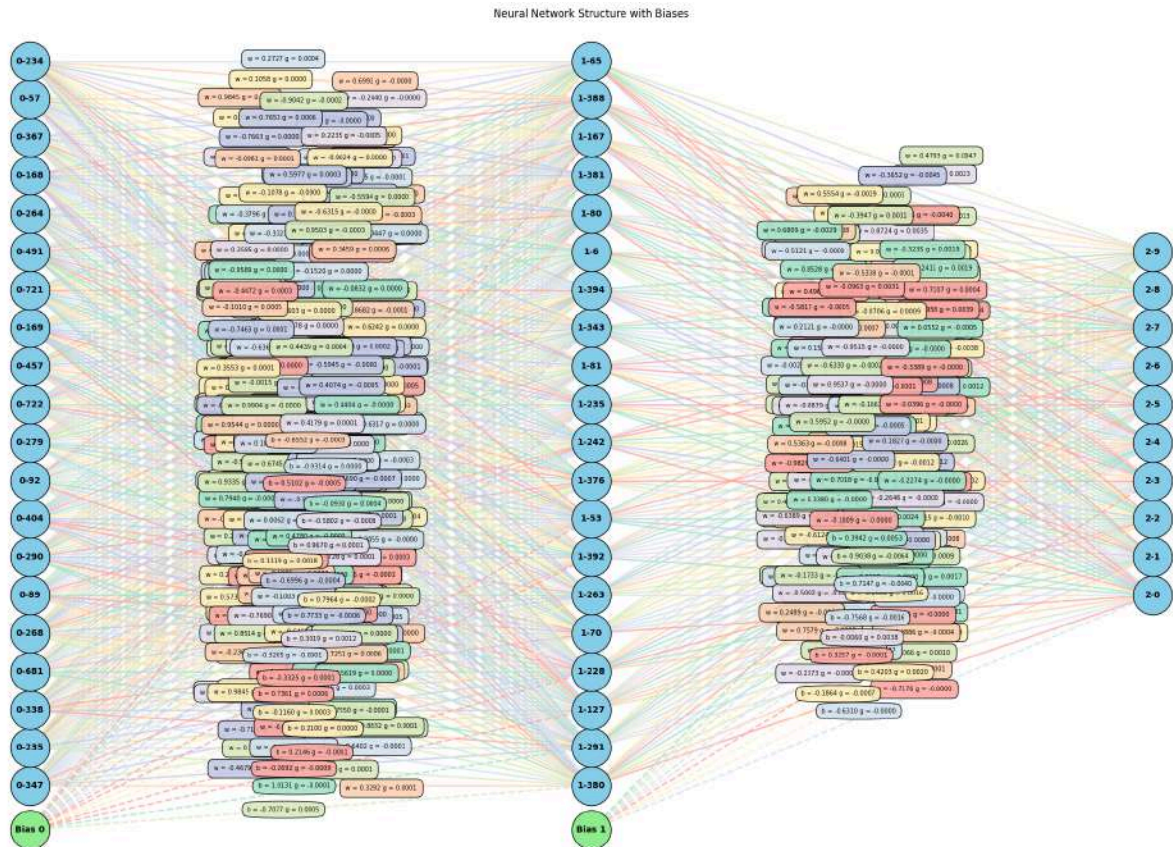
3.2.3 Uji coba Sigmoid

Hasil yang didapat:

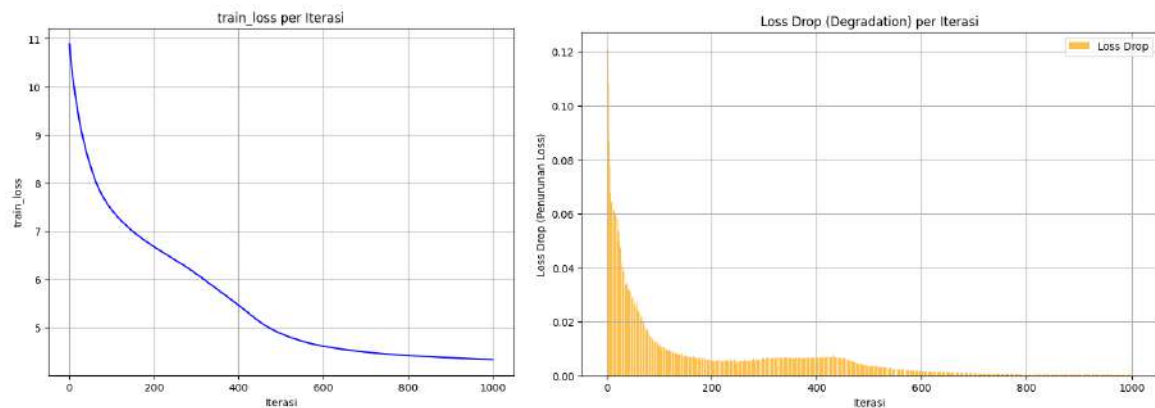
Akurasi model yang didapat yaitu 0.656071. Waktu yang dihabiskan oleh *device* pengujian dalam melaksanakan uji coba ini yaitu: 716.61 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



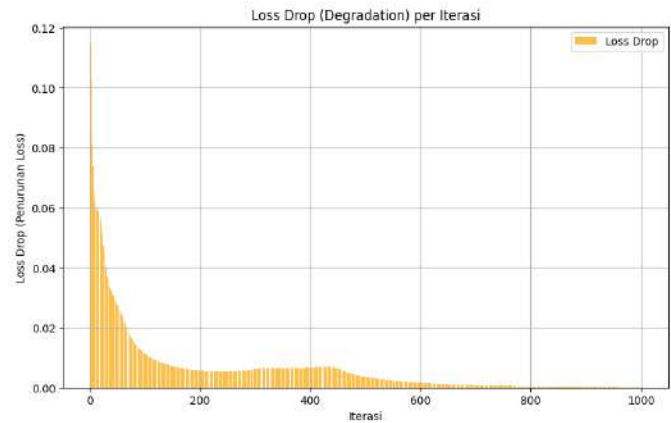
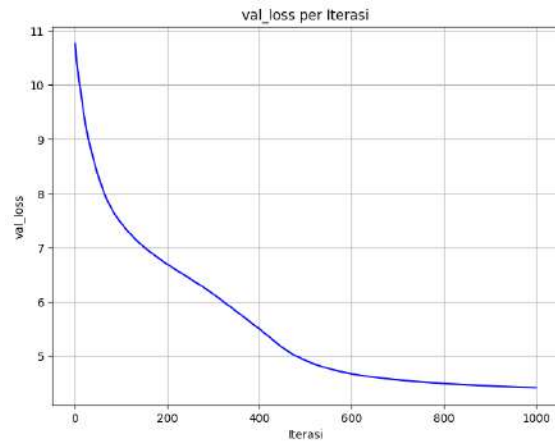
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



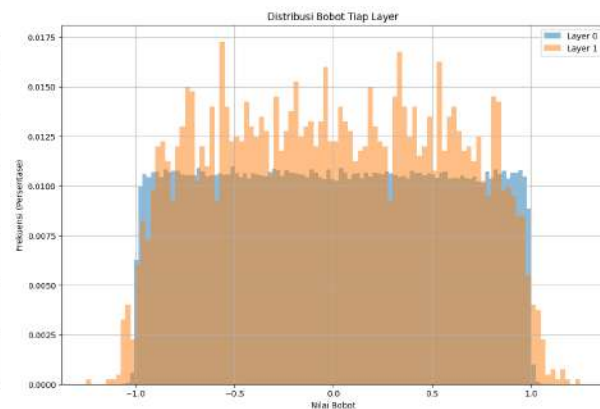
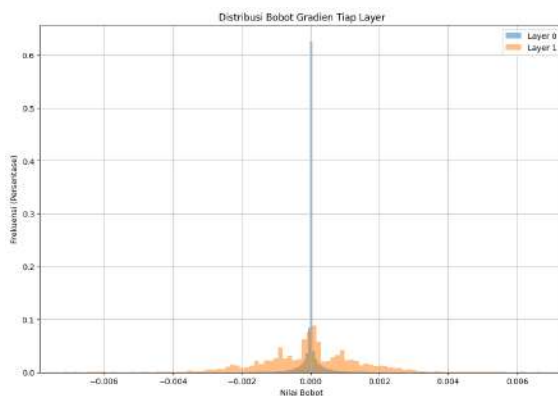
Untuk validasi:



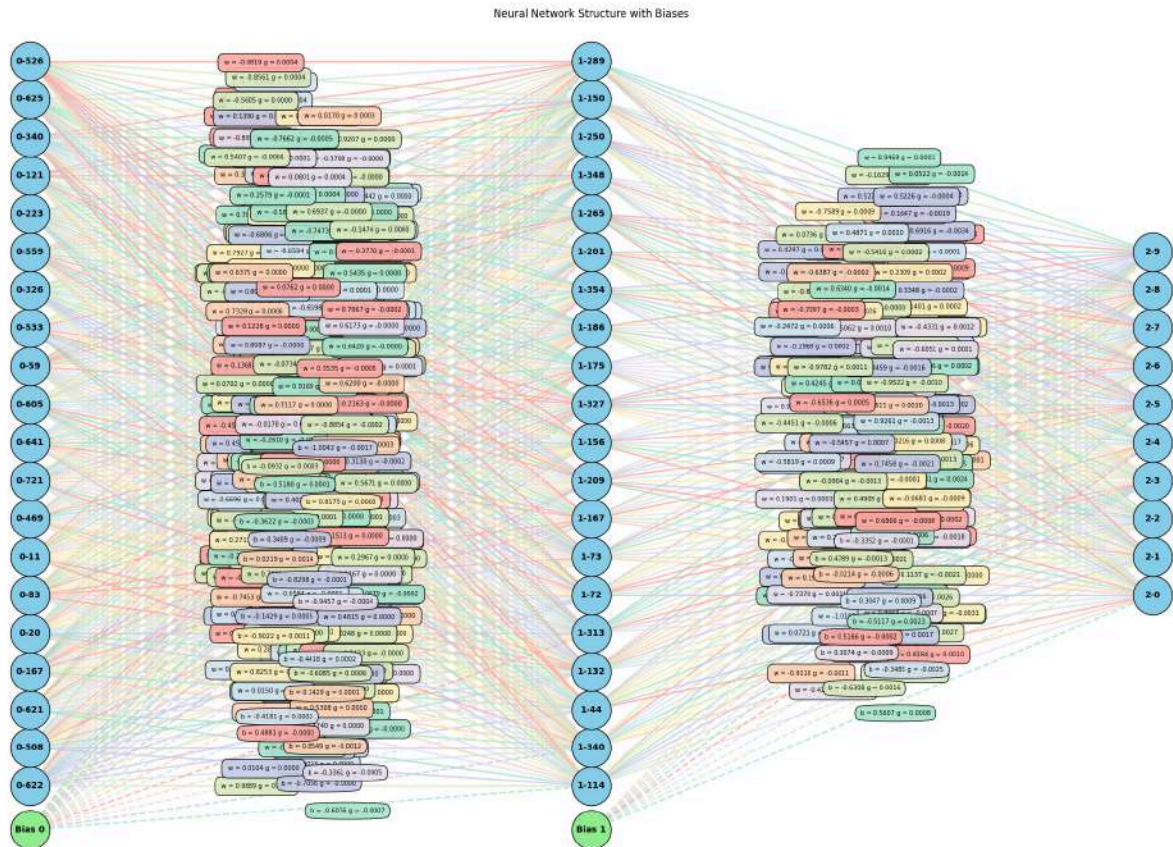
3.2.4 Uji coba tanh

Hasil yang didapat:

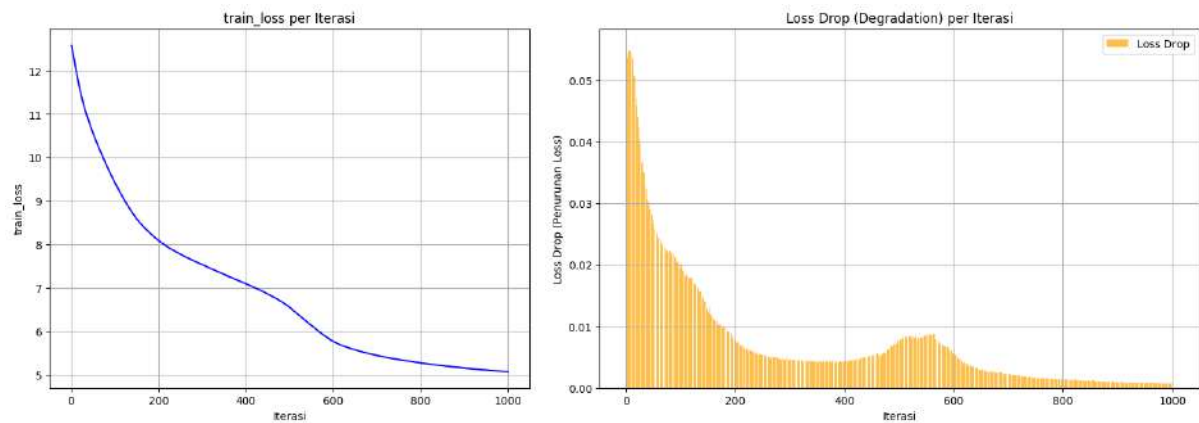
Akurasi model yang didapat yaitu 0.6695713996887207. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 536.62 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



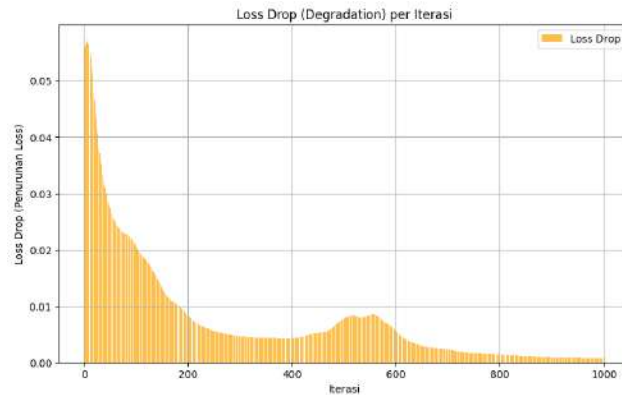
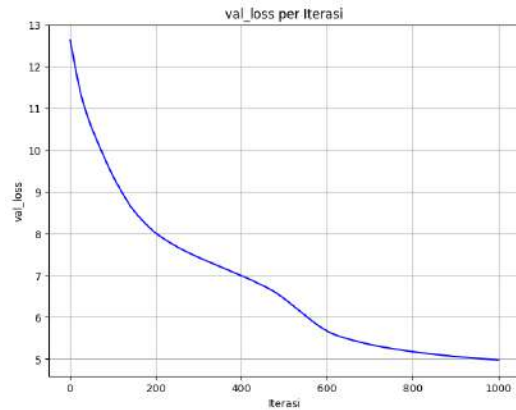
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



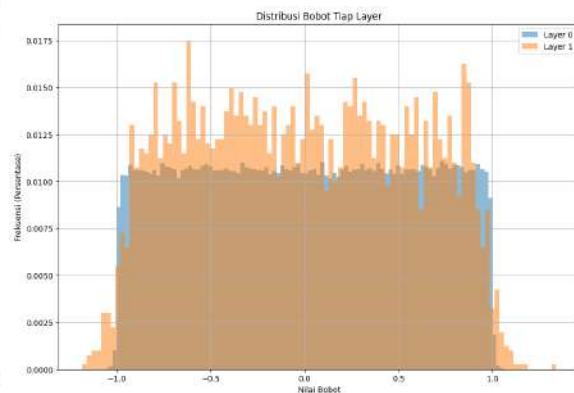
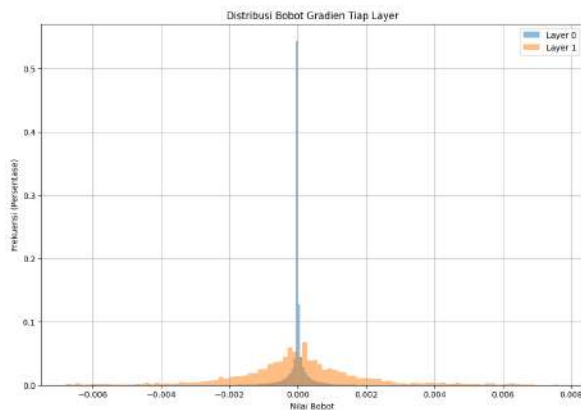
Untuk validasi:



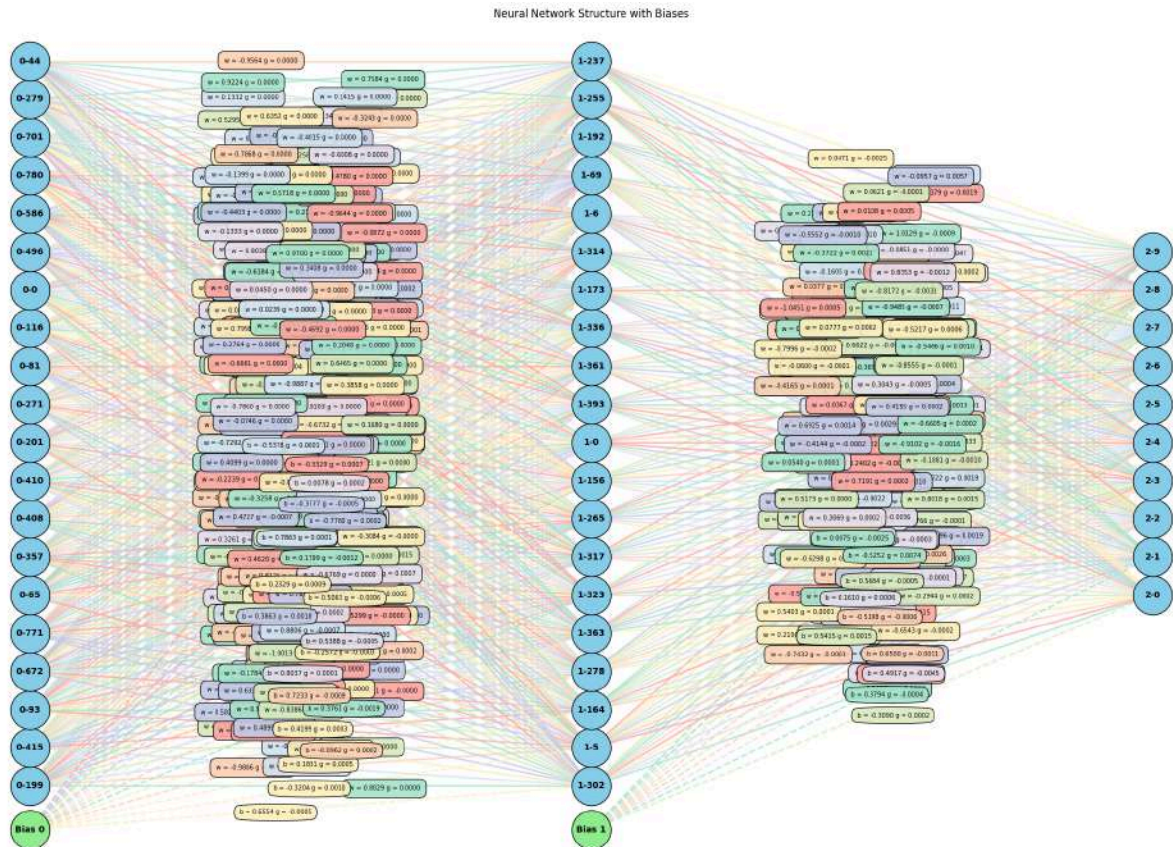
3.2.5 Uji coba *softsign*

Hasil yang didapat:

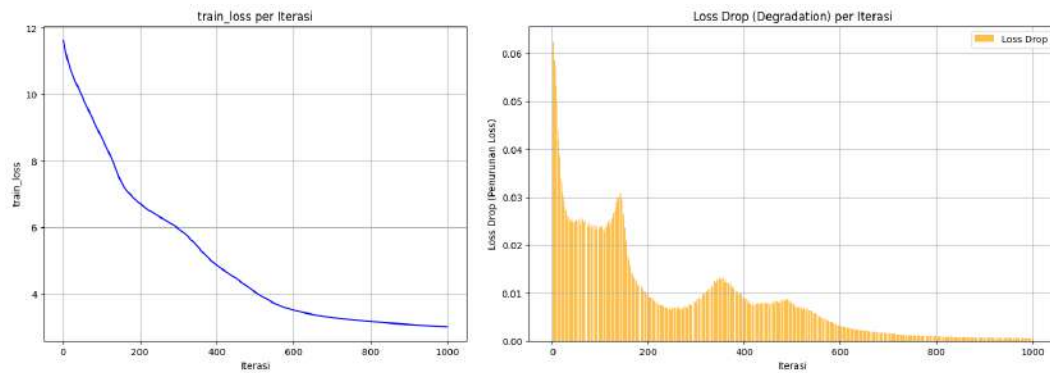
Akurasi model yang didapat yaitu 0.7355714440345764. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 582.20 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



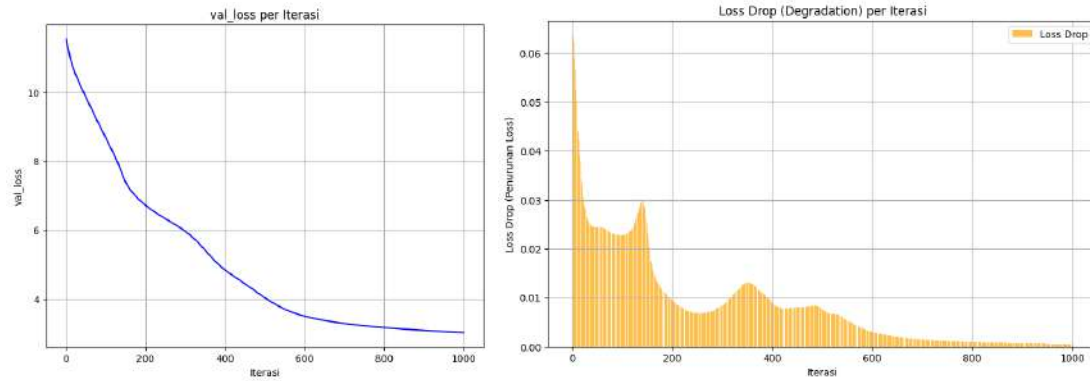
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



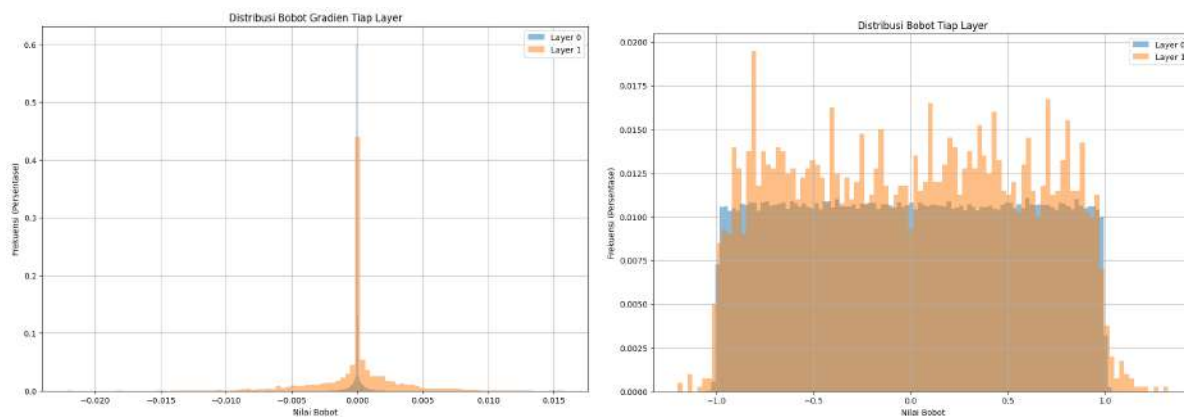
Untuk Validasi:



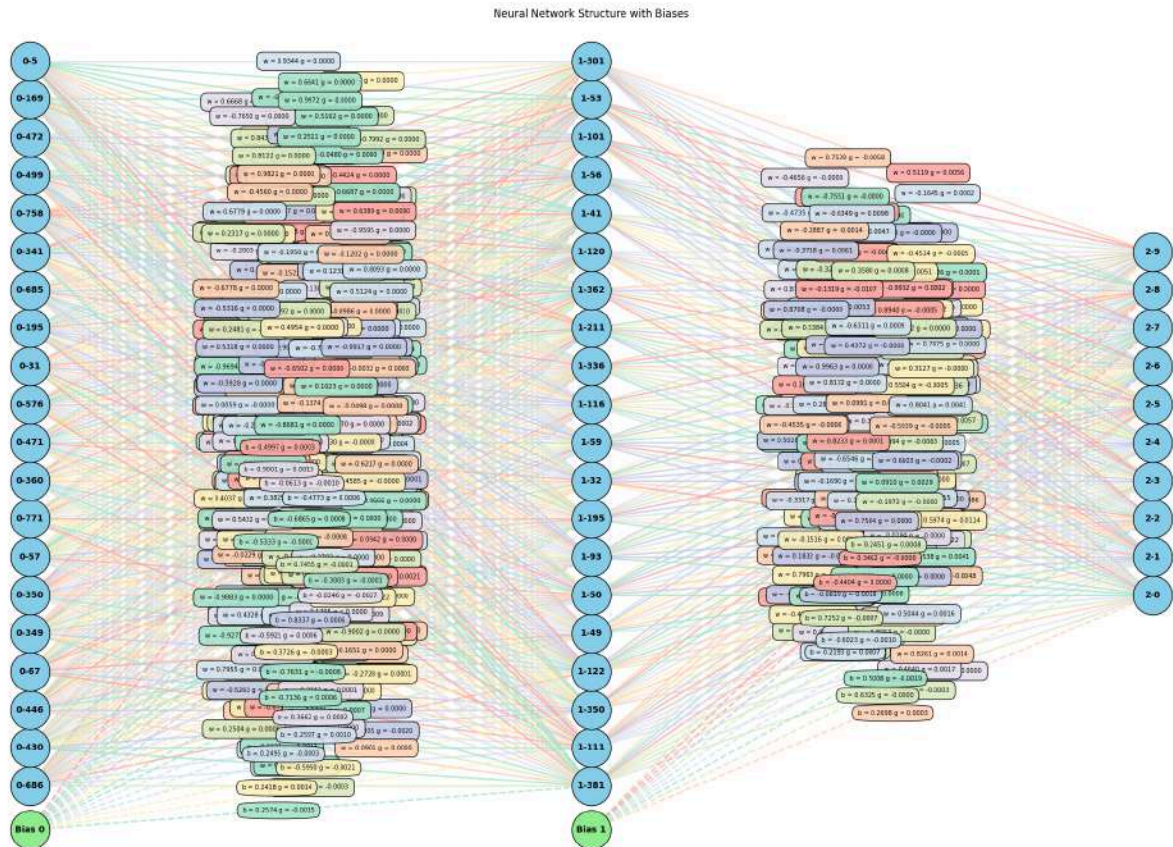
3.2.6 Uji coba *leaky ReLU*

Hasil yang didapat:

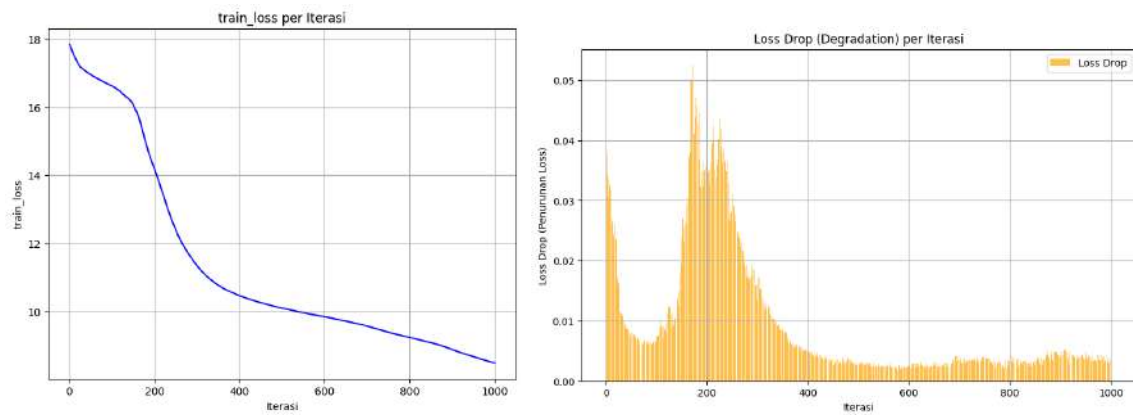
Akurasi model yang didapat yaitu 0.5453571081161499. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 542.30 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



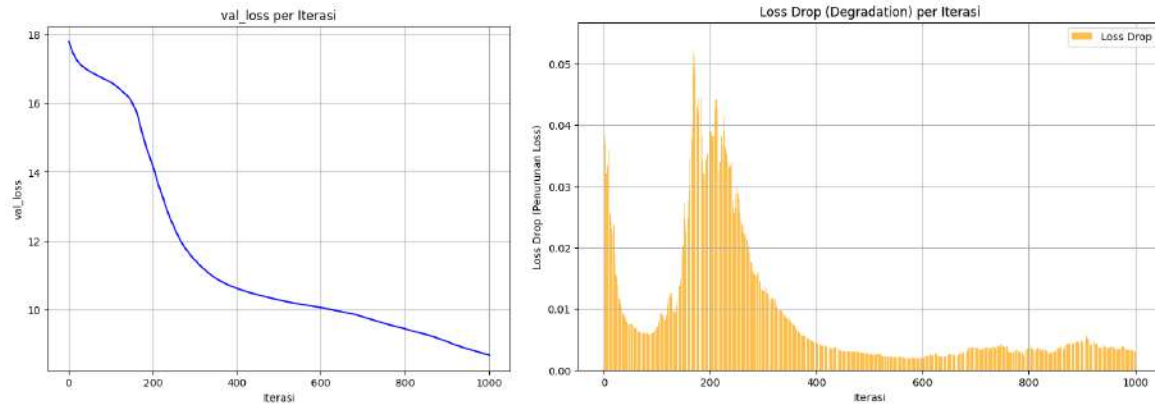
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validation:



3.2.7 Analisis untuk *activation function* berbeda

Secara keseluruhan, penulis merasa fungsi aktivasi lebih baik disesuaikan sesuai dengan kecocokan fungsi inisialisasi bobot. Namun, demi kepentingan analisis, semua inisialisasi bobot dilakukan dengan *random uniform*. Dari perbandingan akurasi, fungsi sigmoid memberikan akurasi yang paling baik. Namun tentu saja, argumen ini dapat dipatahkan dengan ungkapan bahwa fungsi lain tidak cocok dengan fungsi aktivasi spesifik.

Ketika *backpropagation*, fungsi *tanh* lebih *zero-centered*, karena turunannya yang bersifat lebih mendekati 0. Maka dari itu, terlihat bahwa pada grafik distribusi *tanh*, proporsi 0 lebih tinggi, meskipun distribusinya tetap merata.

Untuk *train* dan *val loss*, tidak terdapat *pattern* khusus. Hanya saja, kami ingin meng-*highlight* bahwa ReLU cocok dijadikan dengan xavier dan He. Variasi untuk distribusi bobot dan gradien bobot lebih dibutuhkan pada dataset ini.

3.3 Pengaruh *learning rate*

Percobaan ini akan menggunakan berbagai spesifikasi umum seperti:

1. *Epoch* = 1000
2. *Input Layer* = 784
3. *Hidden Layer* = 600
4. *Output Layer* = 10
5. Fungsi aktivasi :
 - ReLU untuk *hidden layer*
 - *Softmax* untuk *output layer*

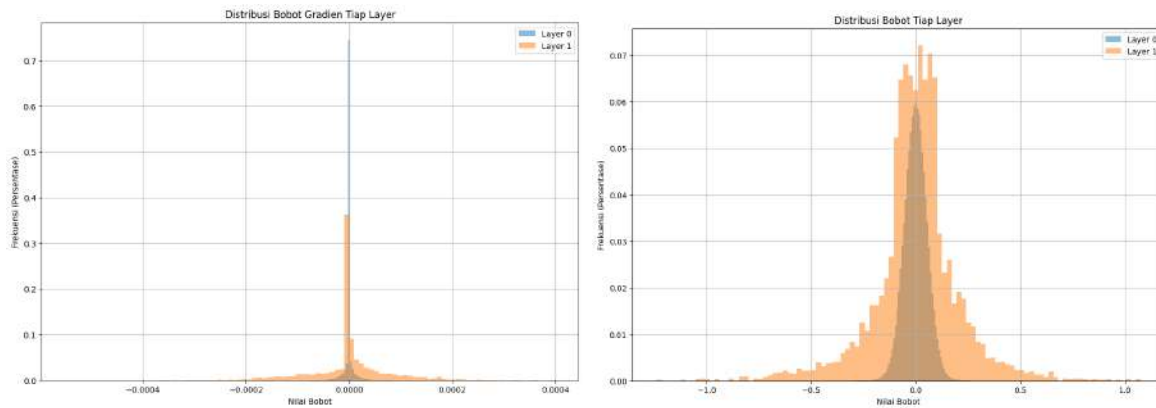
6. *Loss Function: Categorical Cross-Entropy*

7. Inisialisasi bobot : Xavier dan He

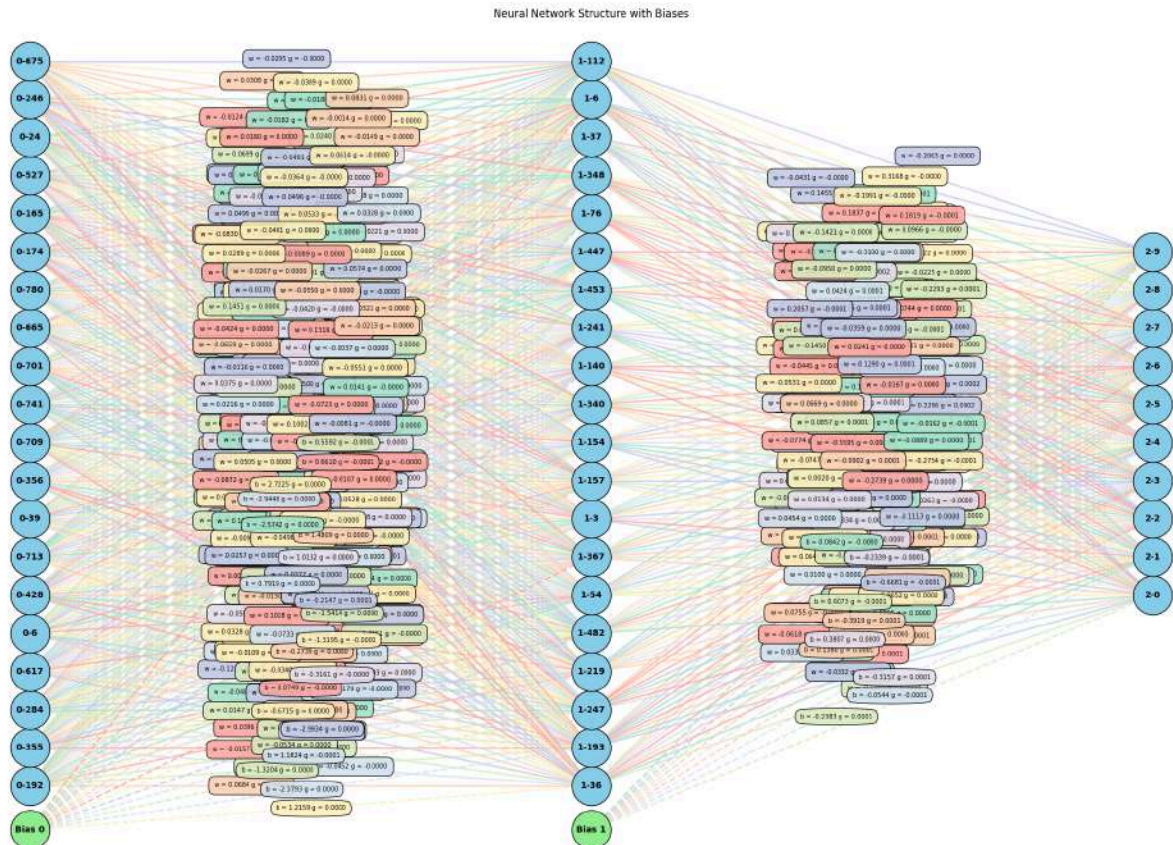
3.3.1 Uji coba *Learning Rate* 0.1

Hasil yang didapat:

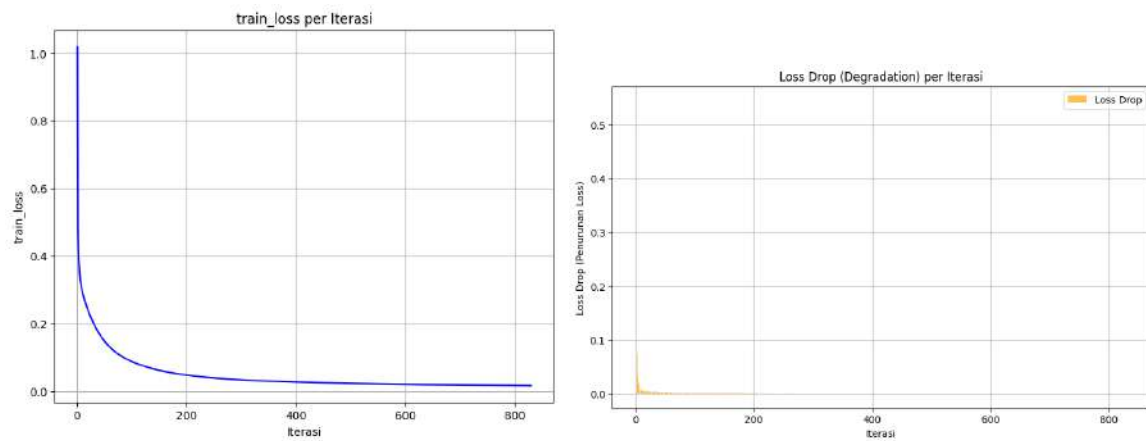
Akurasi model yang didapat yaitu 0.9784285426139832. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 434.92 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



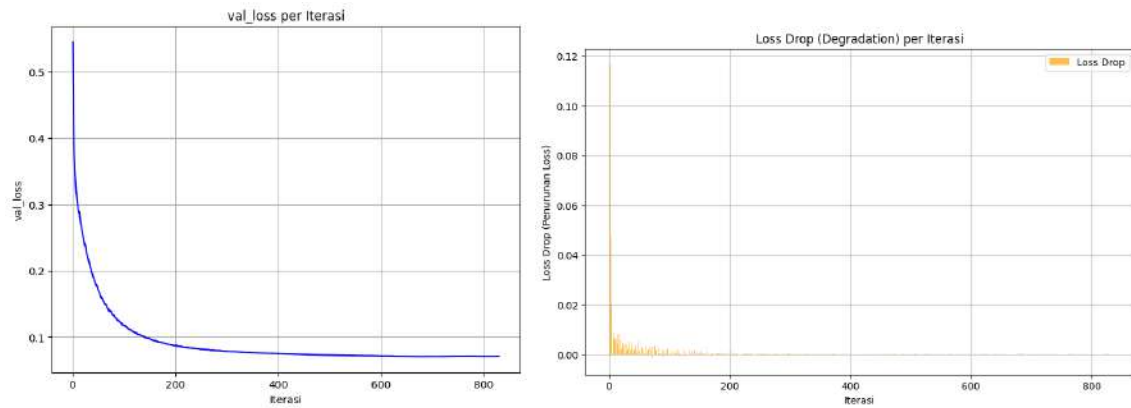
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



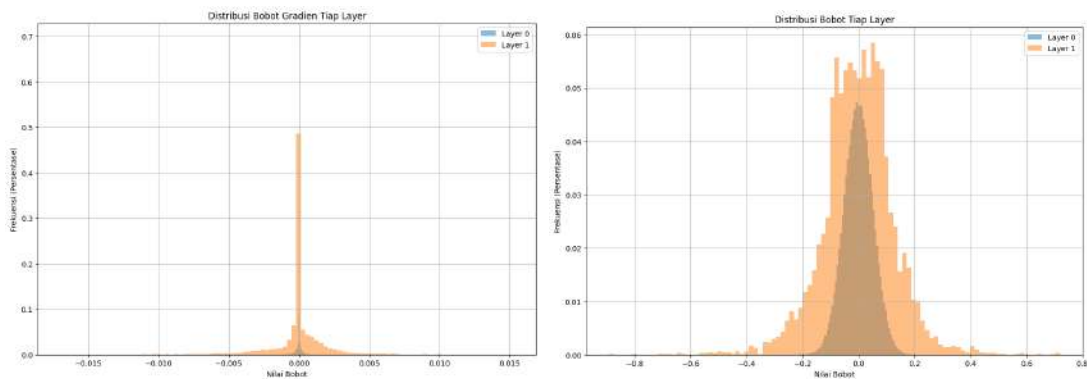
Untuk Validasi:



3.3.2 Uji coba *Learning Rate* 0.01

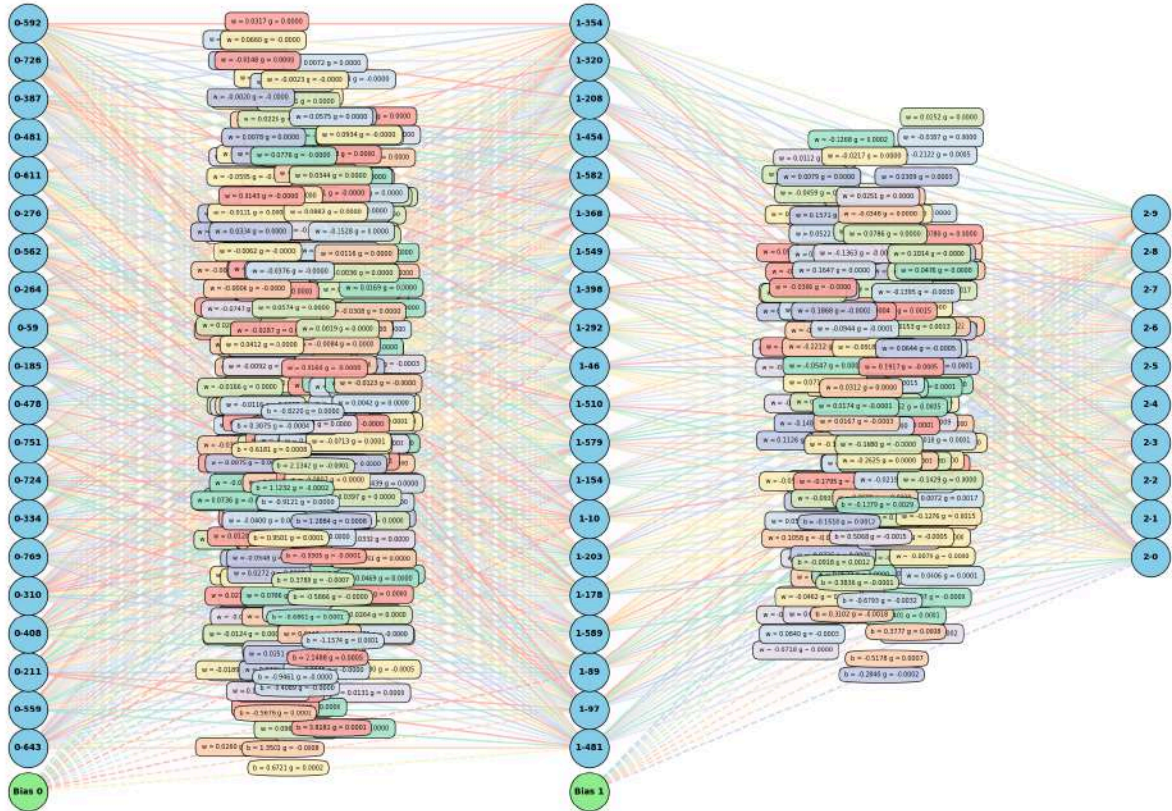
Hasil yang didapat:

Akurasi model yang didapat yaitu 0.9693571329116821. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 528.87 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:

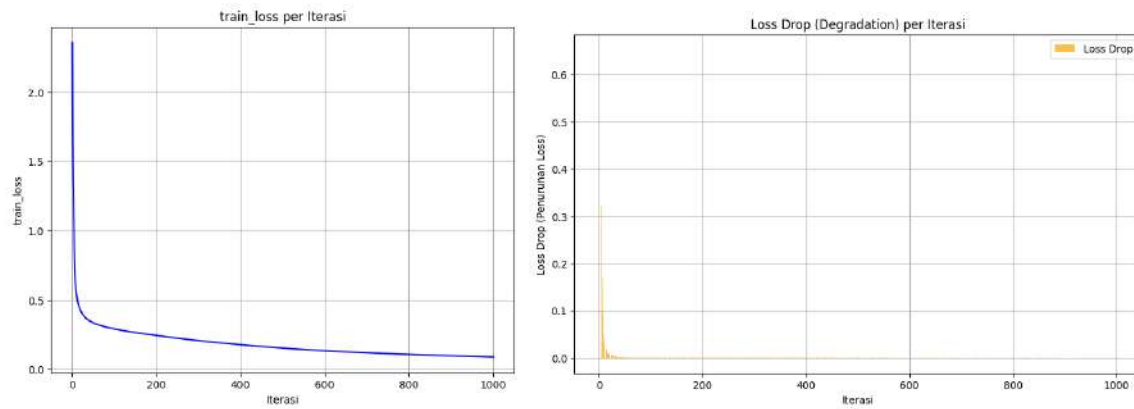


Struktur *network* dapat dilihat pada gambar dibawah:

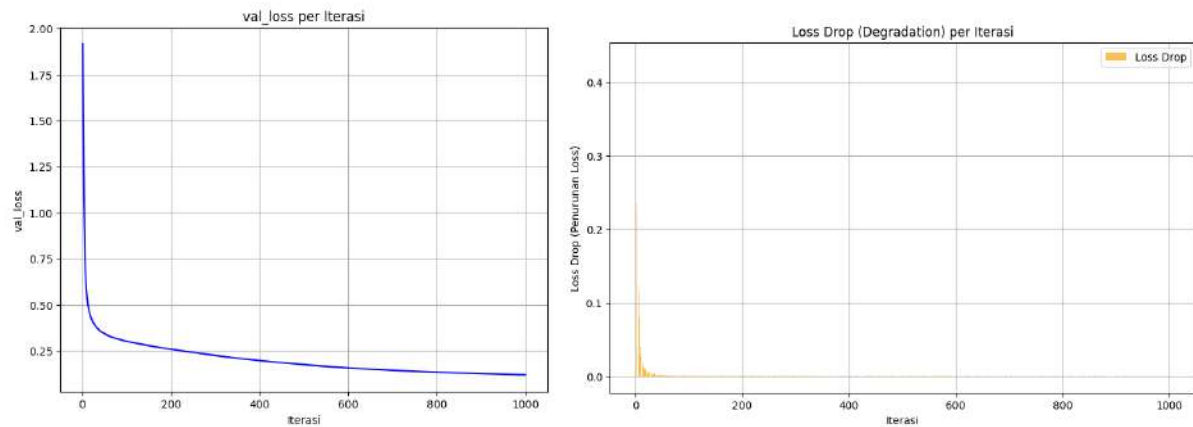
Neural Network Structure with Biases



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



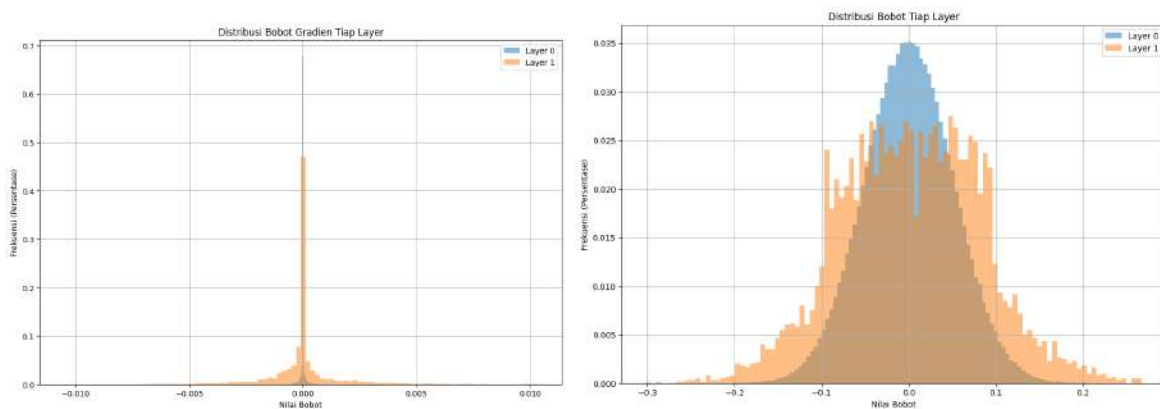
Untuk Validasi:



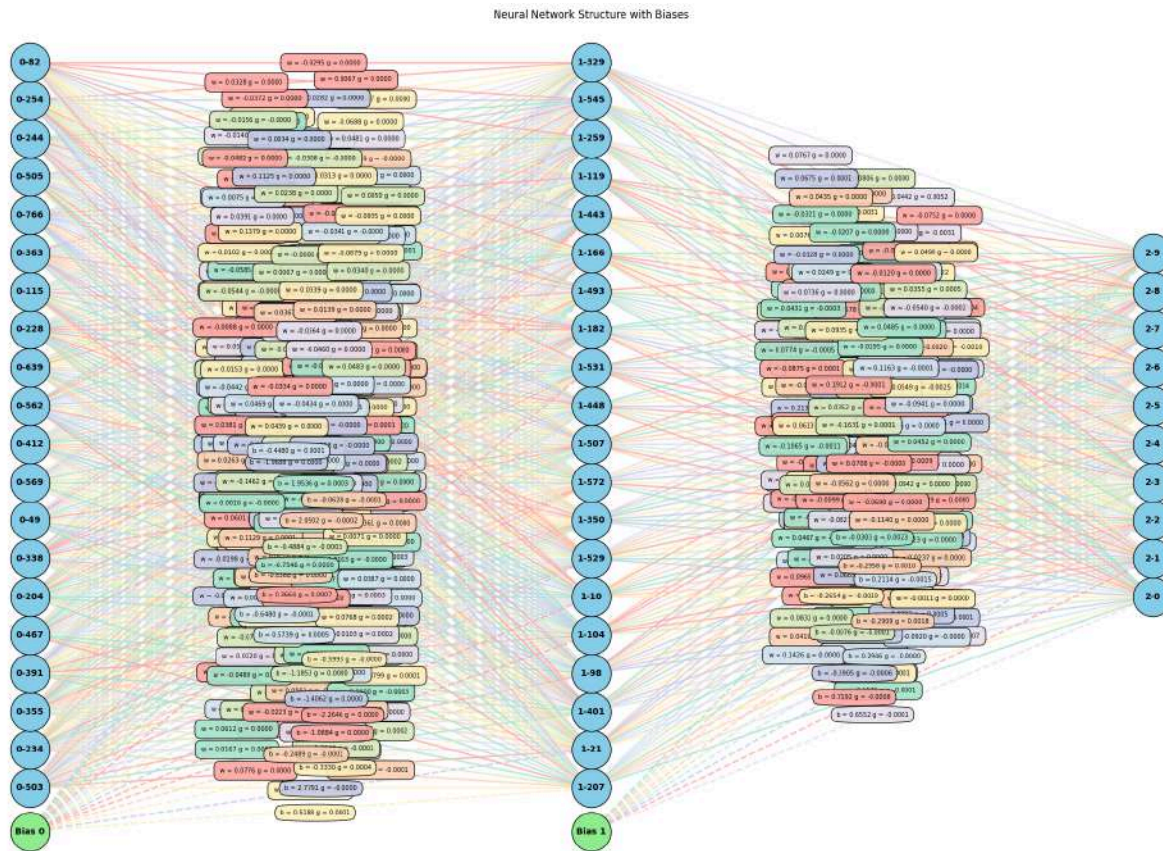
3.3.3 Uji coba *Learning Rate* 0.001

Hasil yang didapat:

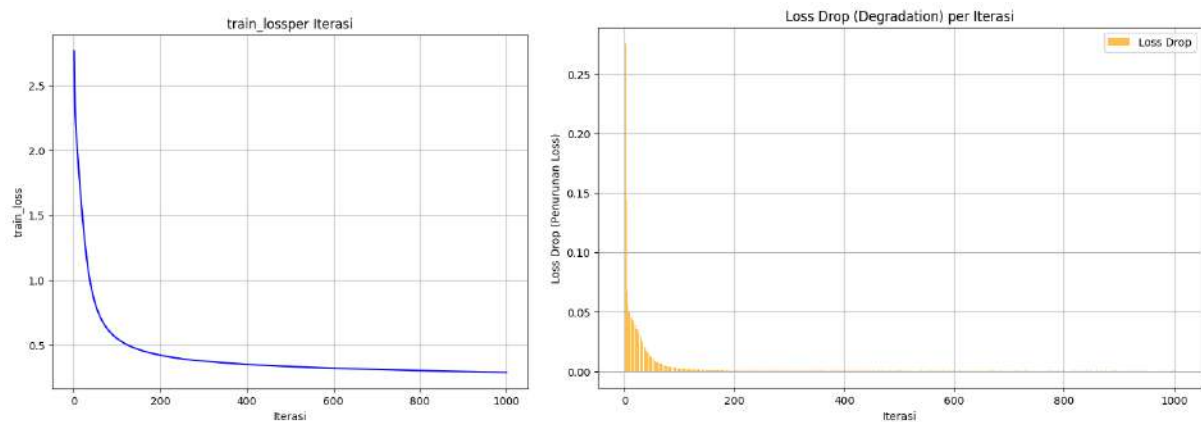
Akurasi model yang didapat yaitu 0.9214285612106323. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 533.78 detik.. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



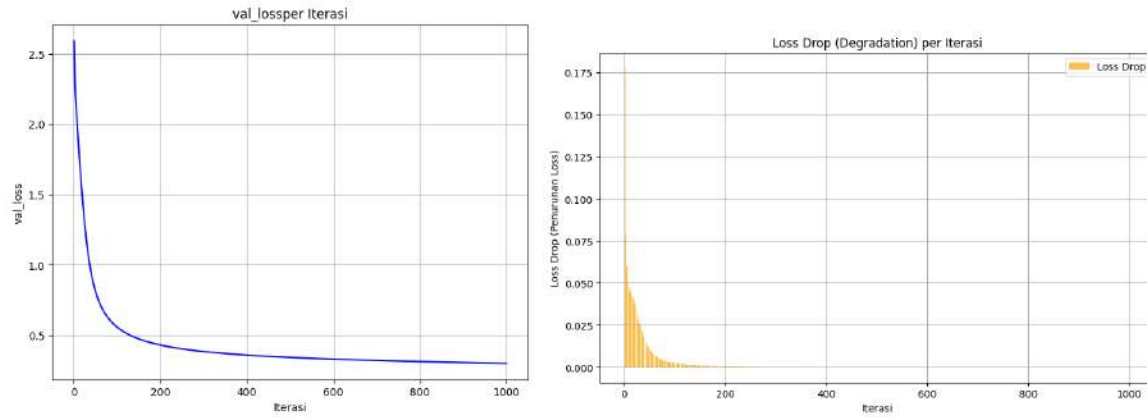
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validasi:



3.3.4 Analisis untuk *learning rate* berbeda

Dari analisis diatas, kita dapat mengasumsikan *learning rate* 0.1 memberikan hasil paling optimal pada skenario ini .Kita dapat melihat bagaimana *learning rate* yang berbeda mempengaruhi distribusi bobot akhir. *Learning rate* yang besar membuat *range* bobot menjadi lebih luas (dapat dilihat pada perbedaan distribusi bobot pada layer 0). Hal ini disebabkan oleh perubahan bobot oleh *learning rate* pada saat *backward propagation*. Semakin besar bobot, maka akan menyebabkan pengurangan yang lebih besar, sehingga membuat *range* distribusi menjadi besar pula. Jika perubahan bobot terasa kurang signifikan, maka disarankan untuk menambah *learning rate*, namun perlu diperhatikan untuk tidak menambah terlalu banyak karena dapat membuat model salah dalam memprediksi target.

Seperti yang dijelaskan diatas, karena penggunaan ReLU, *learning rate* sangat mempengaruhi distribusi akhir dari penyebaran bobot. Dari eksperimen sebelumnya , kita juga meng-asumsikan bahwa generalisasi berperan penting pada dataset ini. Terlihat pada percobaan ini bahwa dengan *learning rate* yang besar , terlihat persebaran bobot yang semakin bervariasi sehingga mendukung generalisasi.

3.4 Pengaruh inisialisasi bobot

Percobaan ini akan menggunakan berbagai spesifikasi umum seperti:

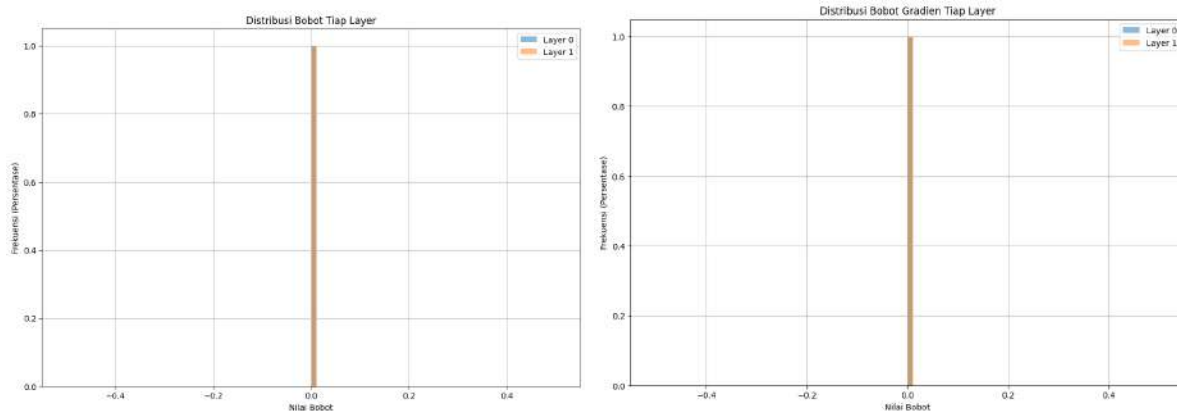
1. *Epoch* = 1000
2. *Input Layer* = 784
3. *Hidden Layer* = 600
4. *Output Layer* = 10

5. *Learning Rate* = 0.001
6. *Loss Function* = *Categorical Cross-Entropy*
7. *Activation Function Hidden Layer* = ReLU

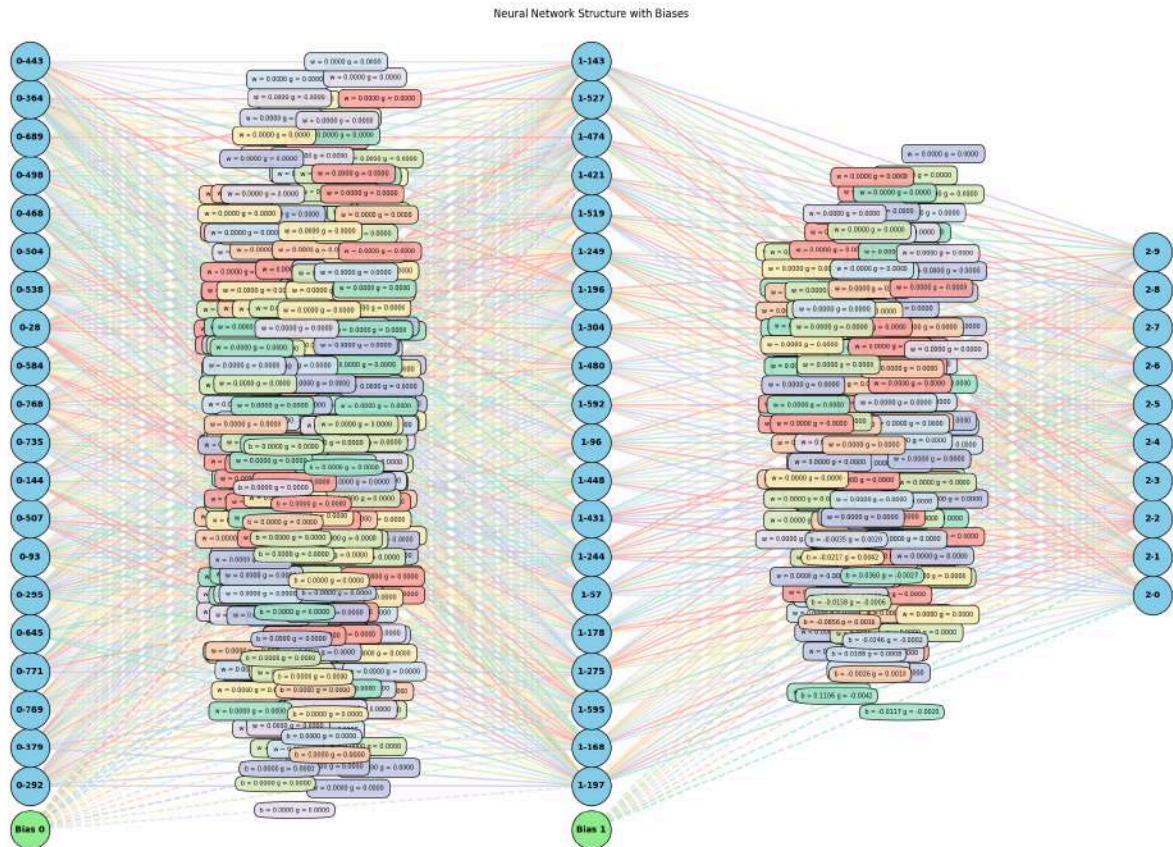
3.4.1 Uji coba inisialisasi bobot 0

Hasil yang didapat:

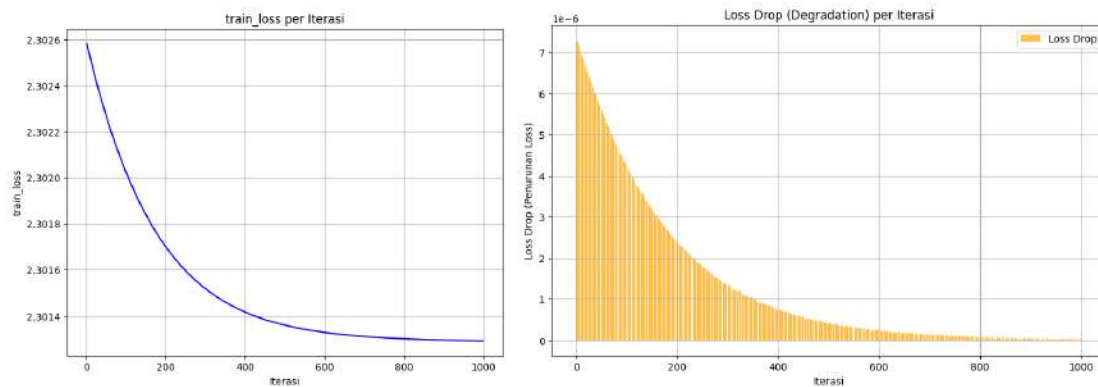
Akurasi model yang didapat yaitu 0.11435714364051819. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 573.79 detik.. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



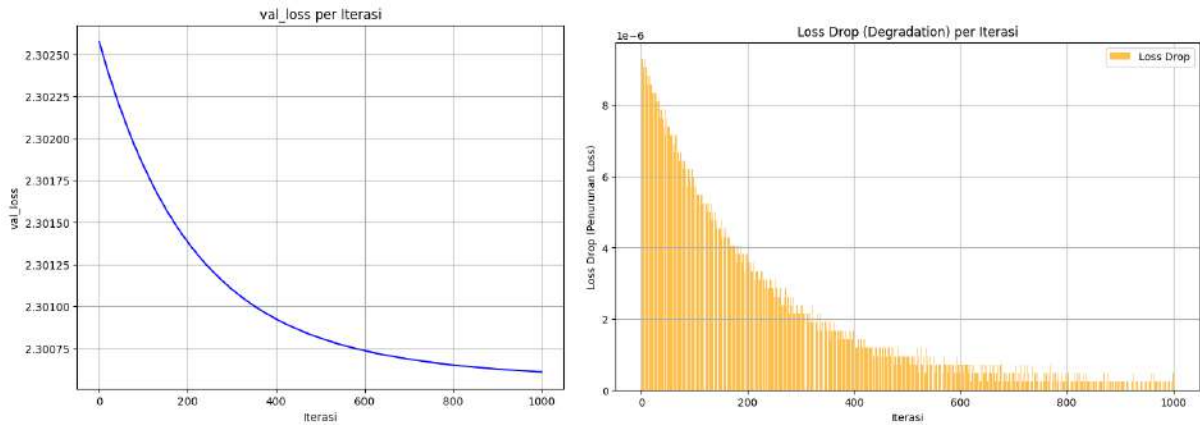
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



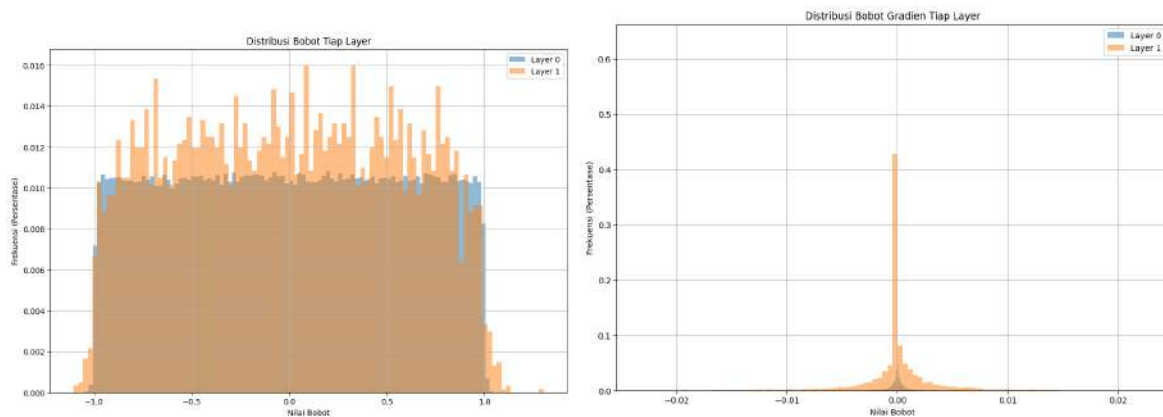
Untuk Validasi:



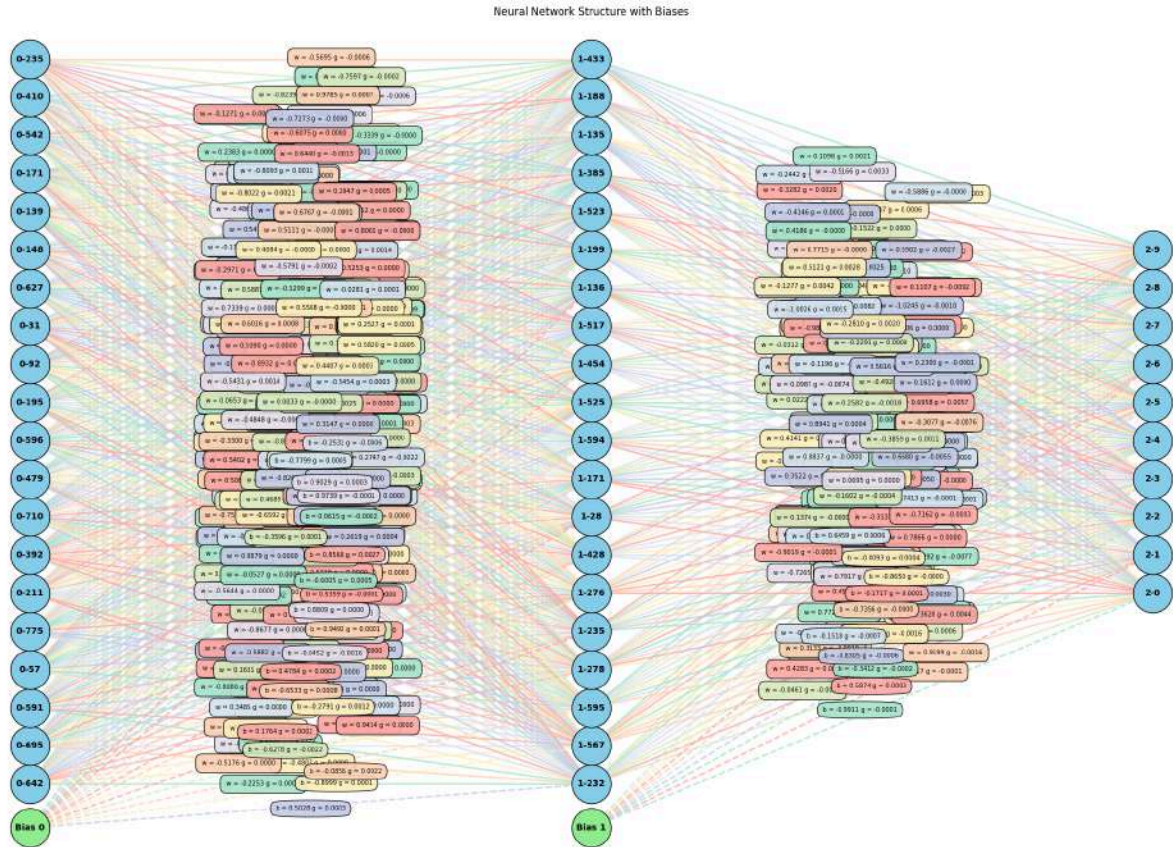
3.4.2 Uji coba inisialisasi bobot *random uniform*

Hasil yang didapat:

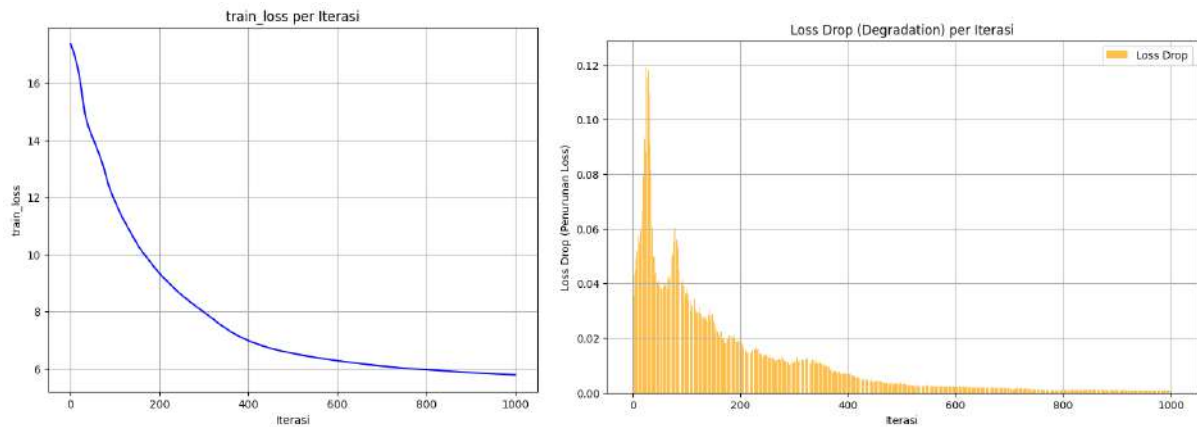
Akurasi model yang didapat yaitu 0.6876428723335266. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 619.18 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



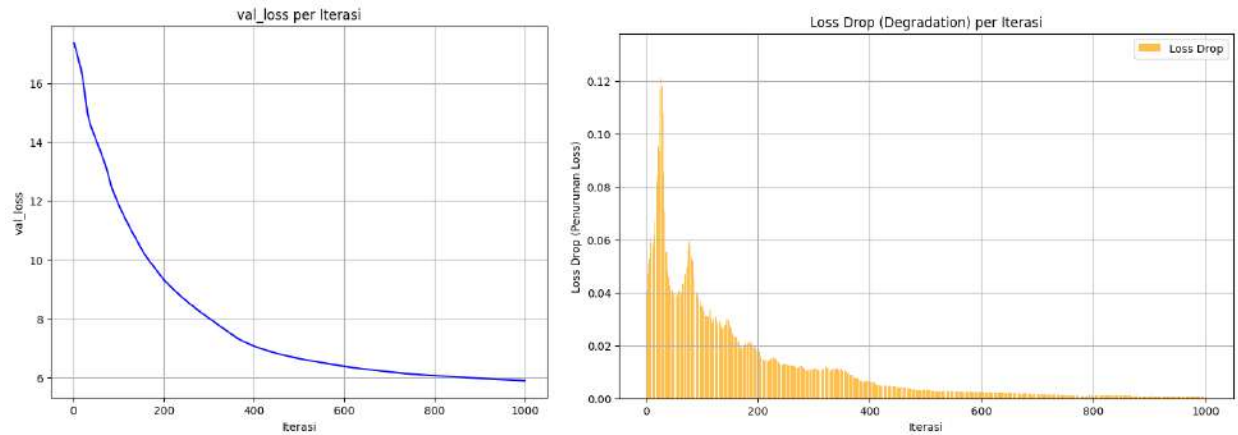
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



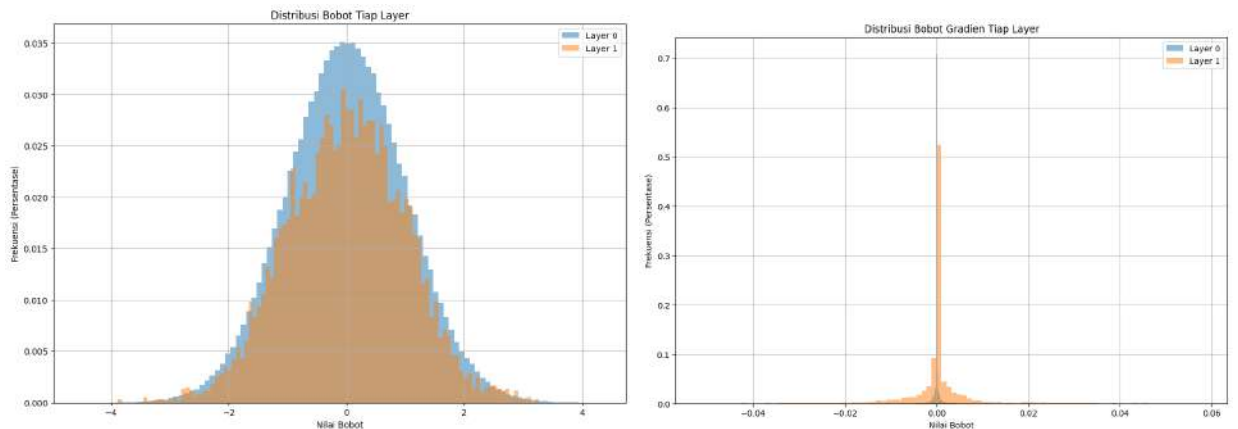
Untuk Validation:



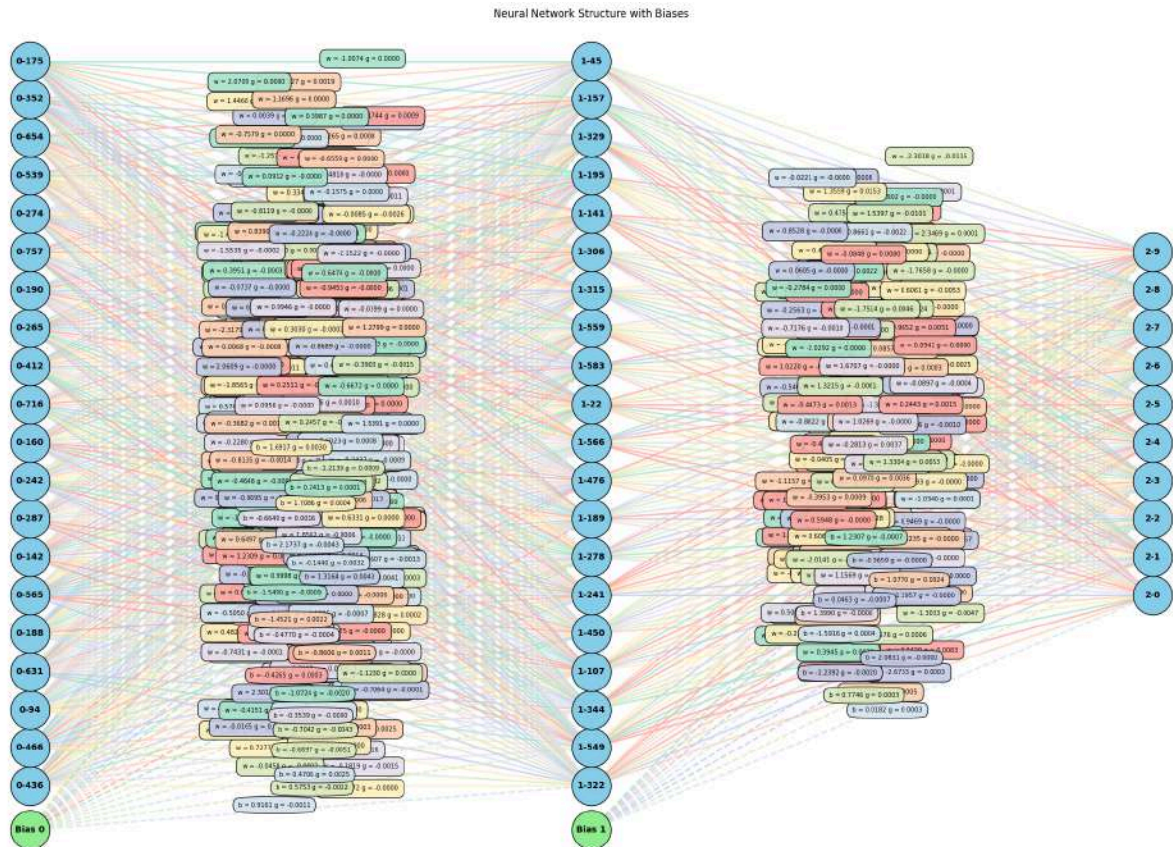
3.4.3 Uji coba inisialisasi bobot *random normal*

Hasil yang didapat:

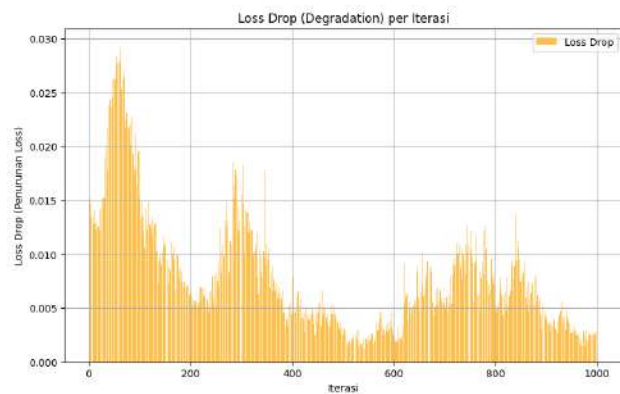
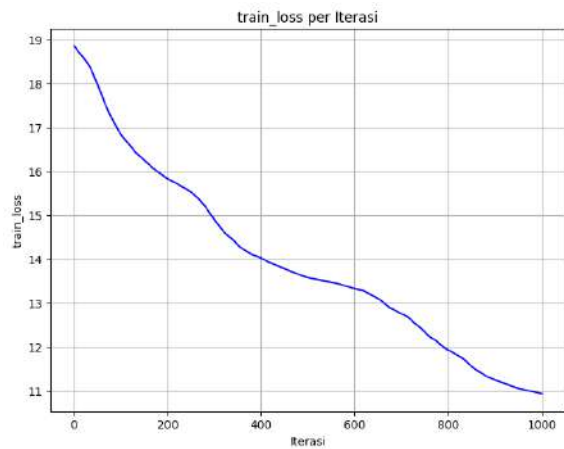
Akurasi model yang didapat yaitu 0.45728570222854614. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 601.37 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



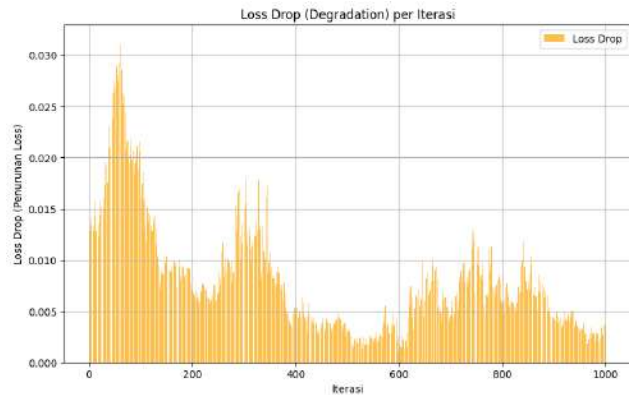
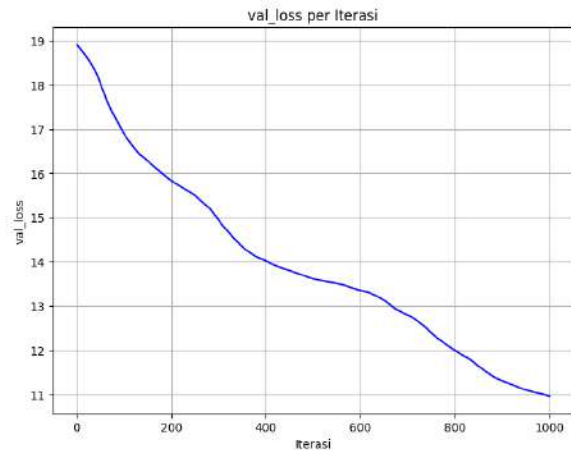
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



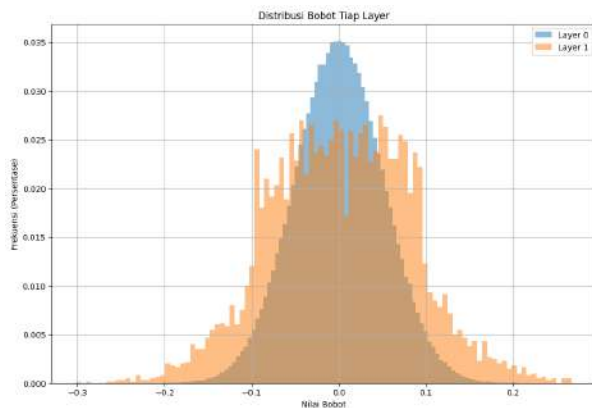
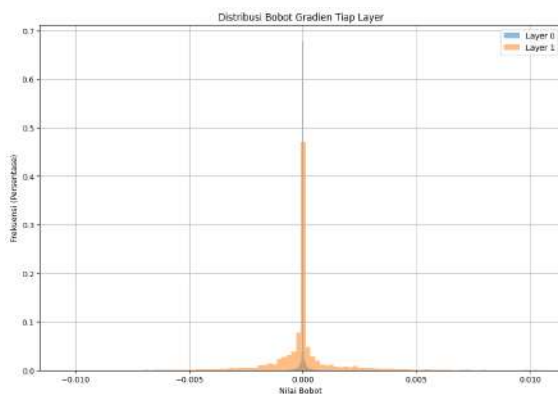
Untuk Validasi:



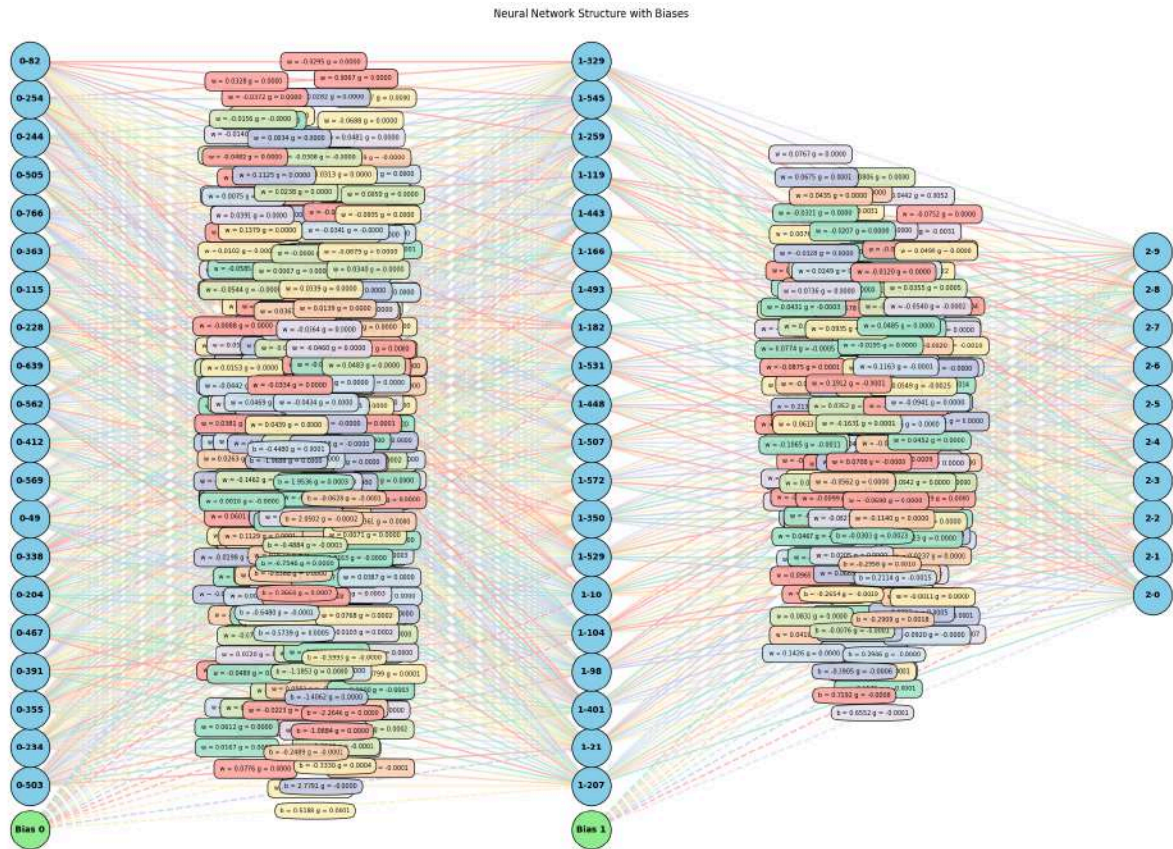
3.4.4 Uji coba inisialisasi bobot *he*

Hasil yang didapat:

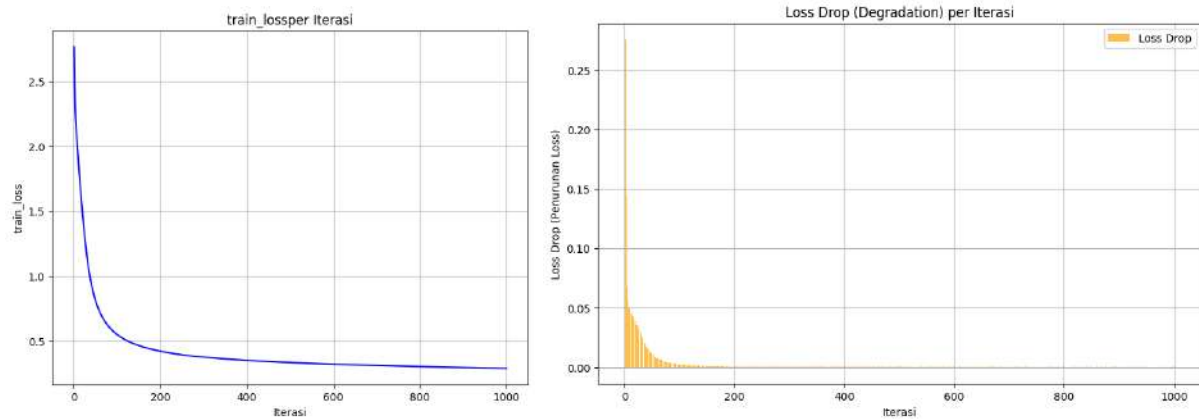
Akurasi model yang didapat yaitu 0.9214285612106323. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 533.78 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



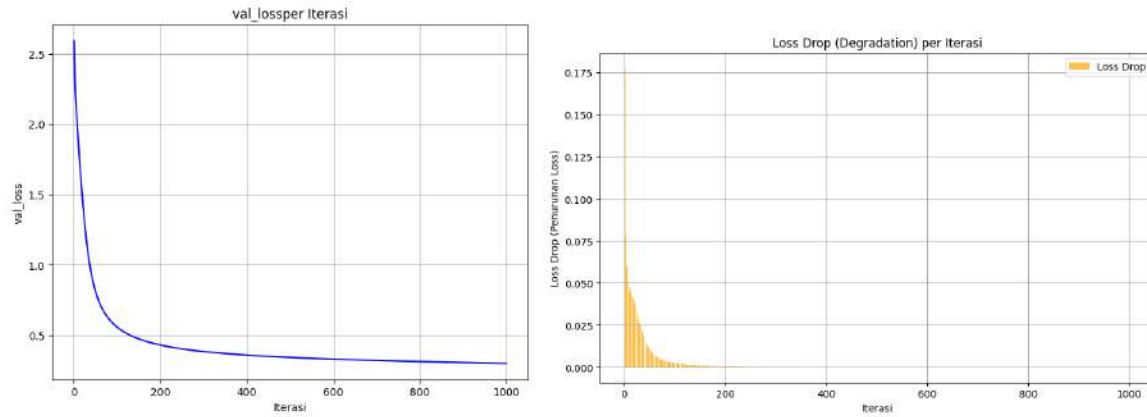
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



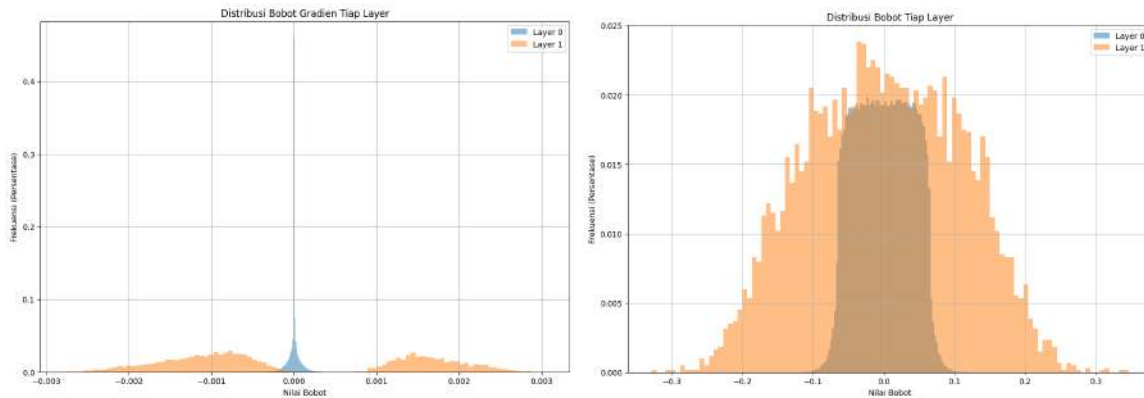
Untuk Validasi:



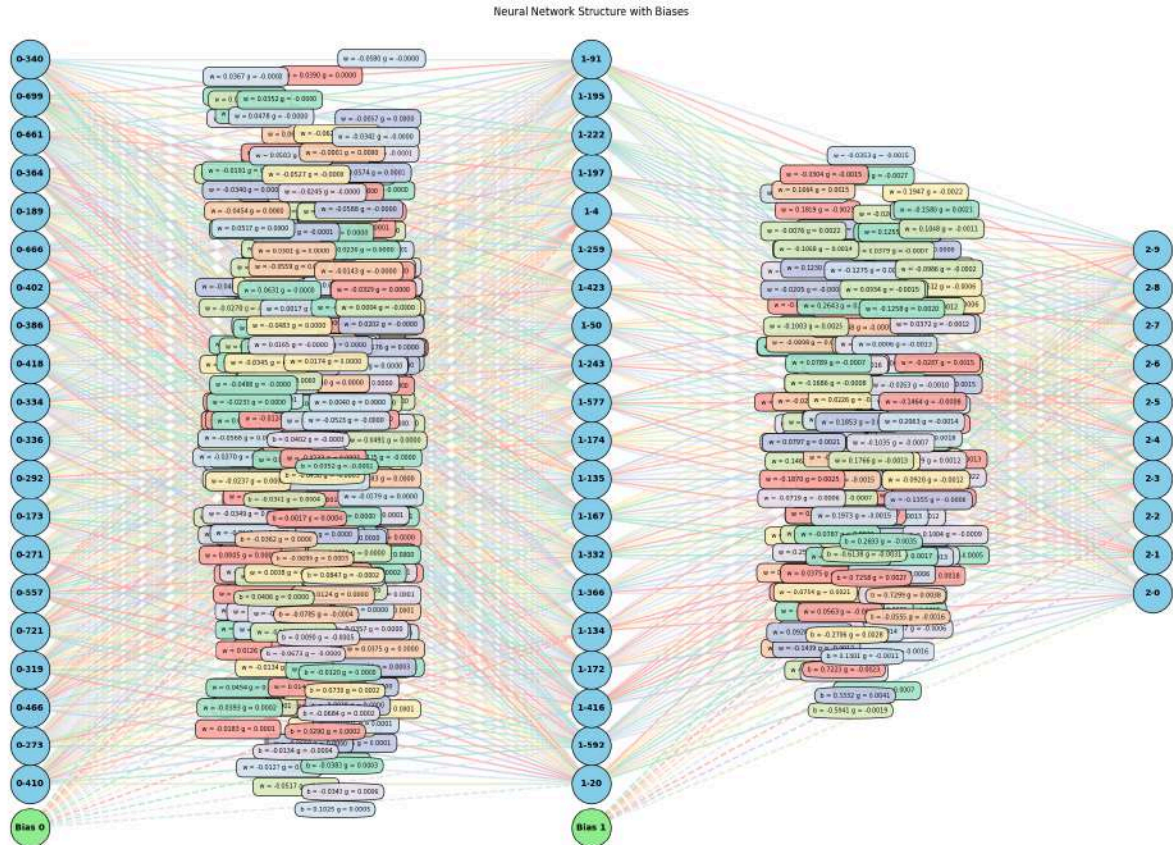
3.4.5 Uji coba inisialisasi bobot *Xavier (Sigmoid)*

Hasil yang didapat:

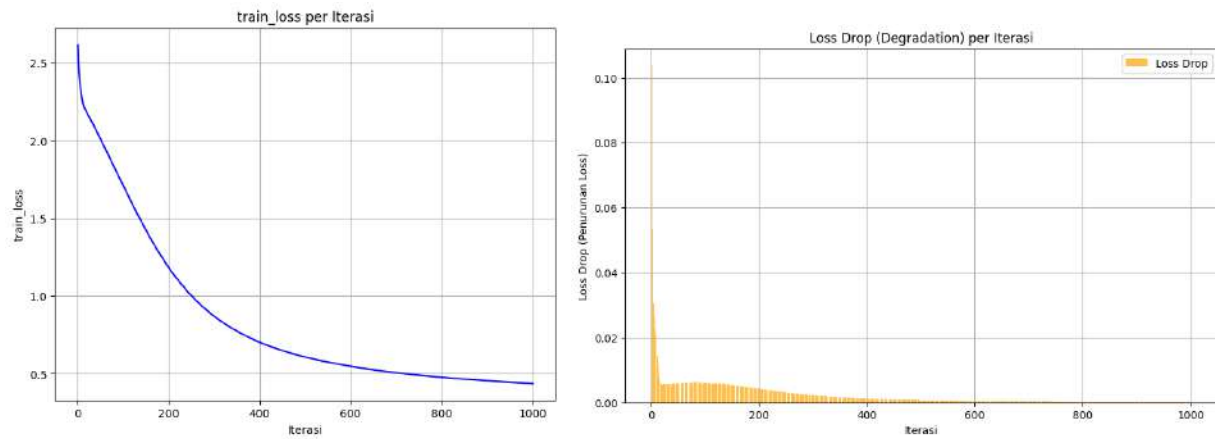
Akurasi model yang didapat yaitu 0.8902857303619385 Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 701.72 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



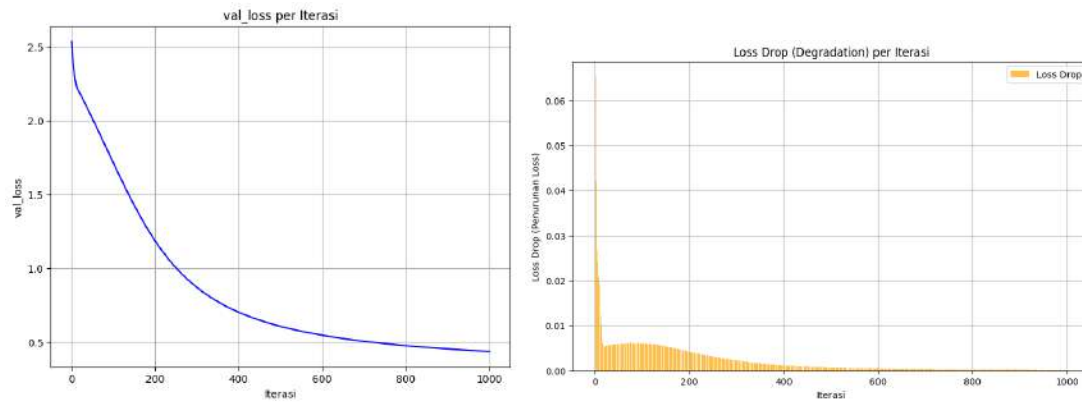
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validasi:



3.4.4 Analisis untuk bobot inisialisasi berbeda

Sebuah pernyataan yang dapat kita ambil yaitu bahwa selain dari inisialisasi bobot bernilai 0, *train loss* pertama sangat mempengaruhi akurasi akhir. Seiring berjalannya *epoch*, meskipun *train loss* akan turun secara perlahan, diperlukan lebih banyak iterasi untuk mencapai *train loss* yang kecil. Oleh karena itu, inisialisasi bobot penting untuk memulai pelatihan data dengan *train loss* yang rendah, dimana pada kondisi ini, Xavier dan He (tepatnya He) memiliki penerapan yang paling baik pada komposisi ini.

Terlihat pada inisialisasi *random normal* dan *random uniform* memiliki *train loss* awal yang sangat tinggi dan dan diperlukan lebih banyak iterasi untuk mencapai nilai kecil. Selain itu, *random normal* memiliki lonjakan pada *train loss* sehingga membuat *random normal* memiliki akurasi yang sedikit lebih buruk dari *random uniform*. Distribusi 0 memiliki inisialisasi yang baik, namun terlihat dari grafik *train loss* bahwa distribusi 0 memiliki perubahan *train loss* yang cukup lambat, mungkin disebabkan karena penggunaan fungsi aktivasi ReLU yang tidak begitu cocok dengan inisialisasi bobot 0.

Secara keseluruhan, inisialisasi bobot berperan penting dalam menentukan *range* dari *train loss* yang dihasilkan. Selama grafik *train loss* tidak mengalami lonjakan, inisialisasi bobot yang baik yaitu inisialisasi yang dapat memberikan *train loss* yang rendah pada iterasi awal dan konsisten dalam penurunannya selama iterasi berlangsung.

Sebagai informasi tambahan, inisialisasi bobot pada tugas besar ini juga mempengaruhi inisialisasi bobot awal. Penulis awalnya tidak tahu dan selalu menginisialisasi bobot bias menjadi 0. Alhasil, penulis merasa bahwa inisialisasi tersebut memberikan hasil yang lebih baik. Meskipun begitu, karena kebutuhan spesifikasi, inisialisasi bobot bias disamakan dengan inisialisasi bobot berat, sehingga informasi diatas hanya sebuah *insight* bahwa kecocokan antara inisialisasi bobot dengan bias diperlukan untuk memberikan hasil yang bagus.

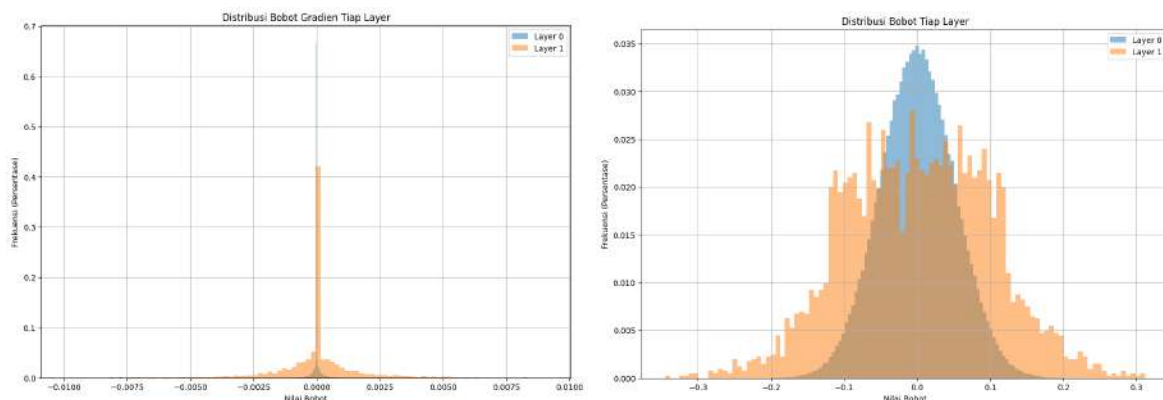
3.5 Perbandingan dengan model yang di regularisasi

Parameter uji coba:

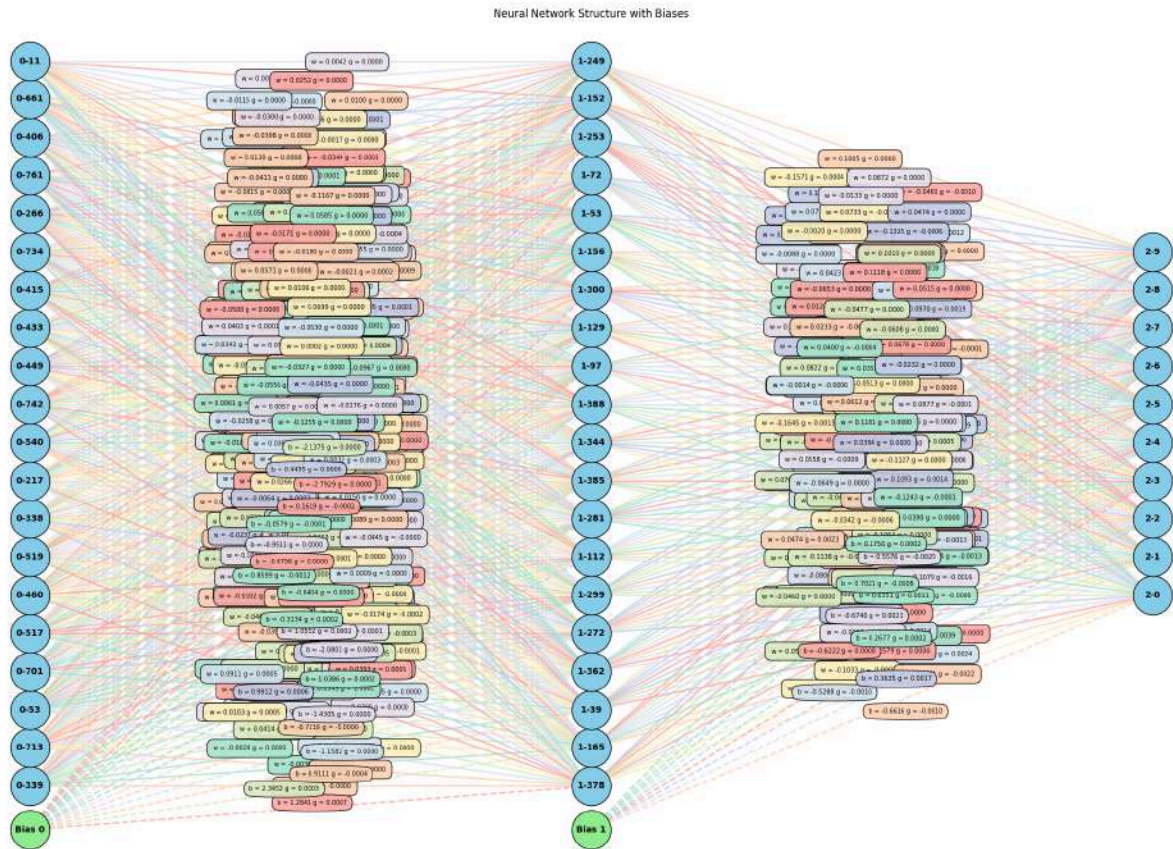
1. *Epoch* = 1000
2. *Input Layer* = 784
3. *Hidden Layer* = 400
4. *Output Layer* = 10
5. *Learning Rate* = 0.001
6. *Loss Function* = *Categorical Cross-Entropy*
7. *Activation Function Hidden Layer* = ReLU
8. Inisialisasi bobot = Xavier dan He

3.5.1 Tanpa Regularisasi

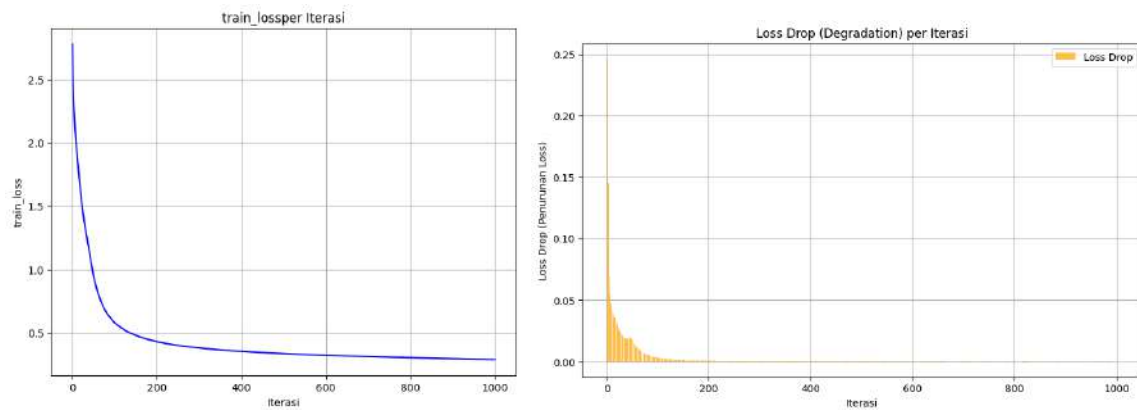
Akurasi model yang didapat yaitu 0.92149996757507323. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 538.46 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



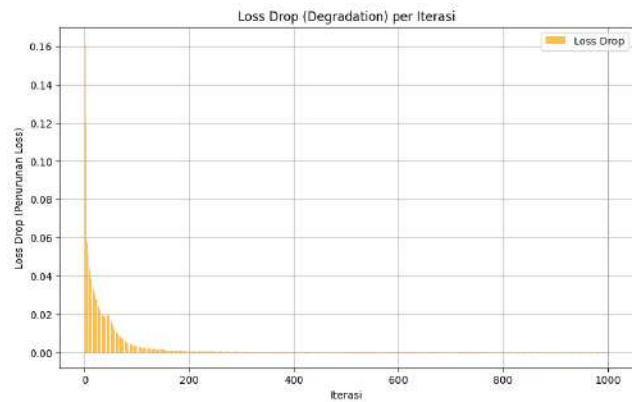
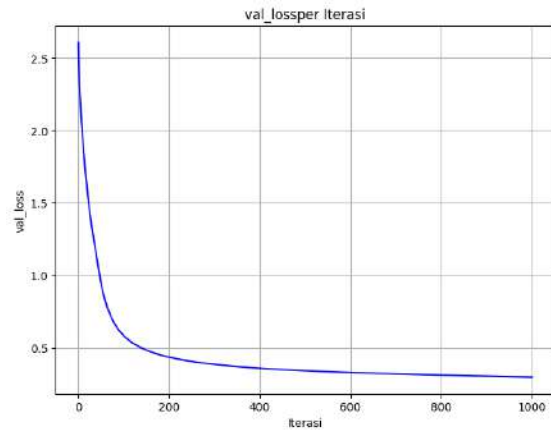
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



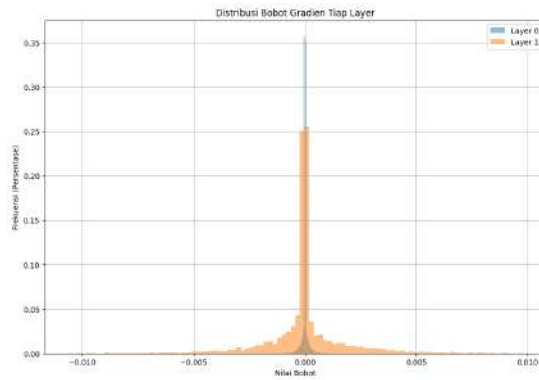
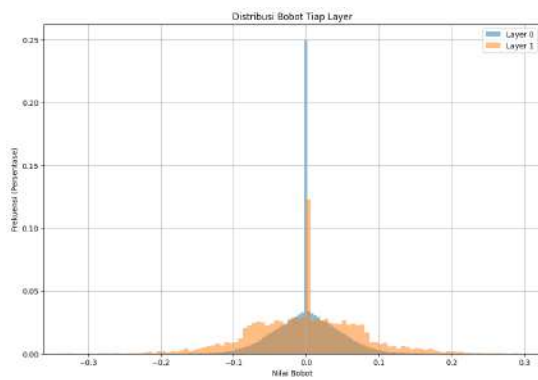
Untuk Validation:



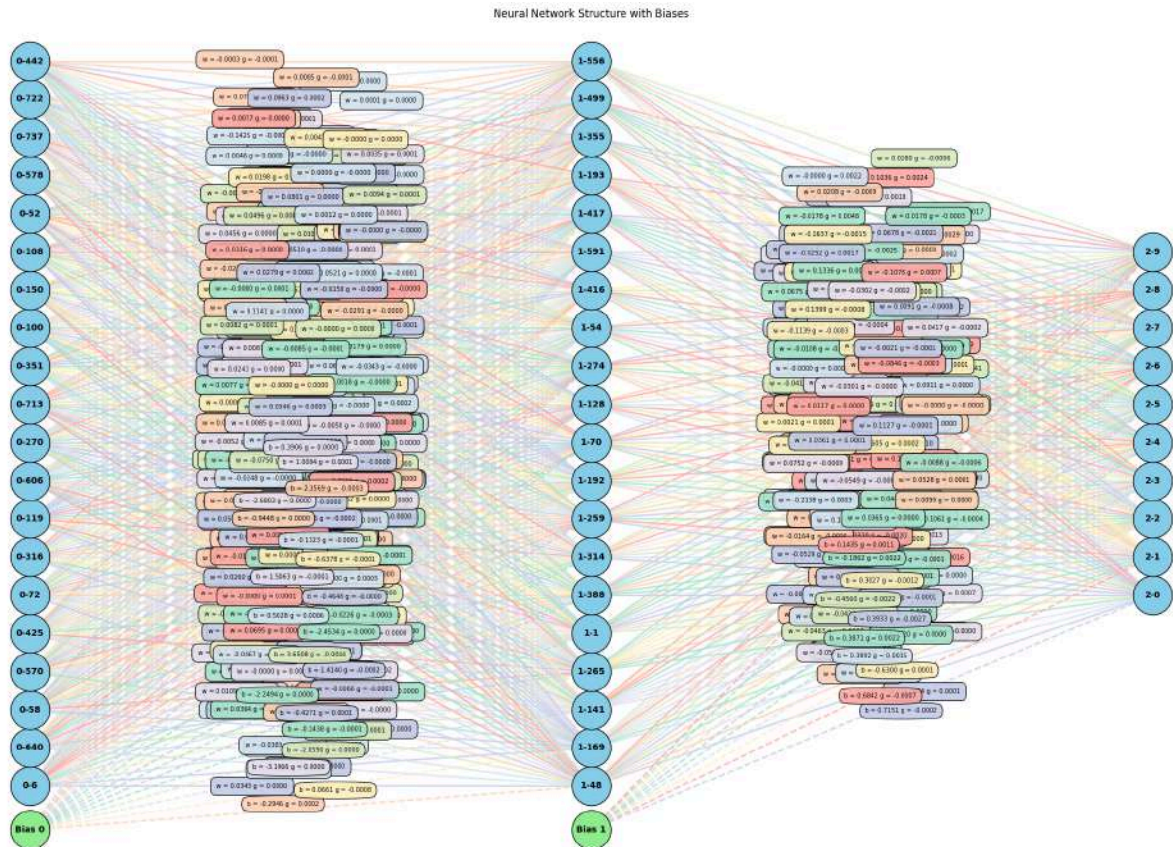
3.5.1 Regularisasi L1

Hasil yang didapat:

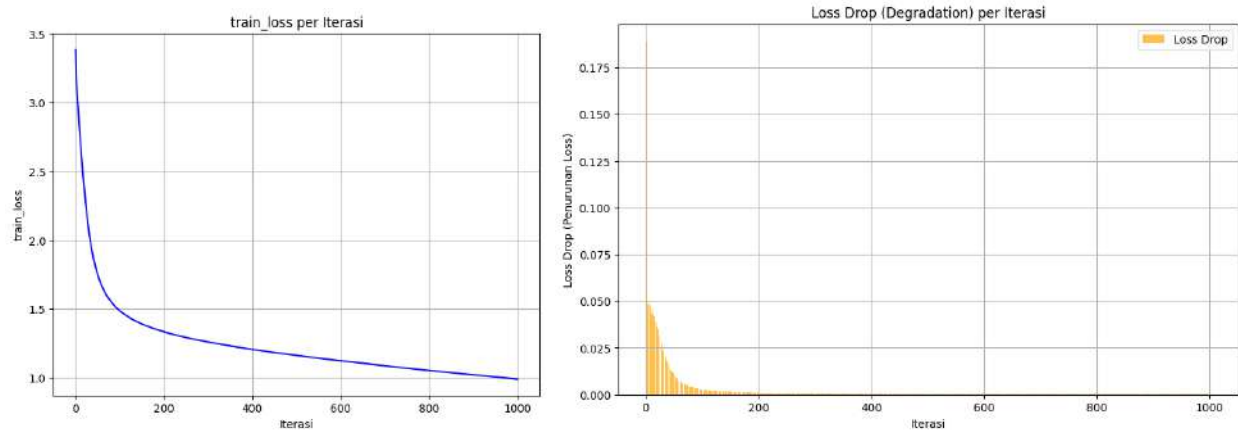
Akurasi model yang didapat yaitu 0.9195713996887207. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 690.76 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



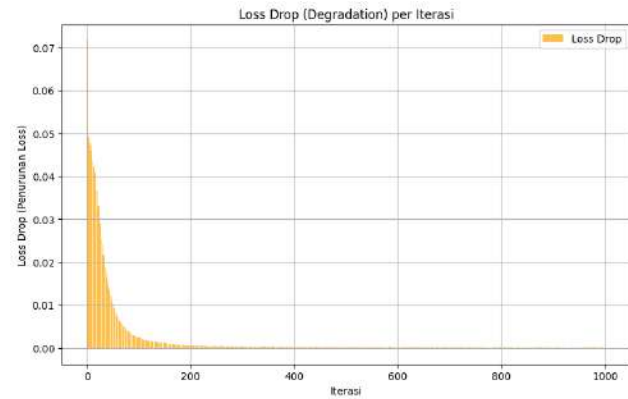
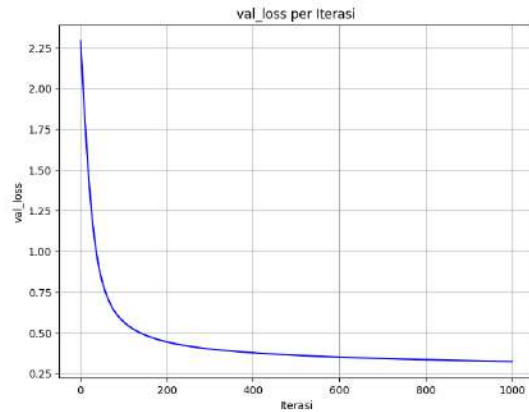
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training loss* dan *loss drop* dapat dilihat pada gambar dibawah:



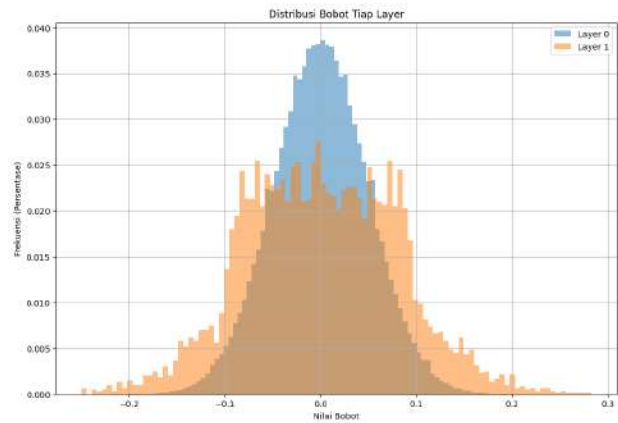
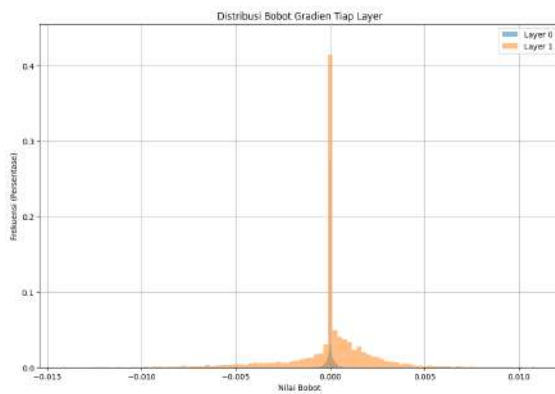
Untuk Validasi:



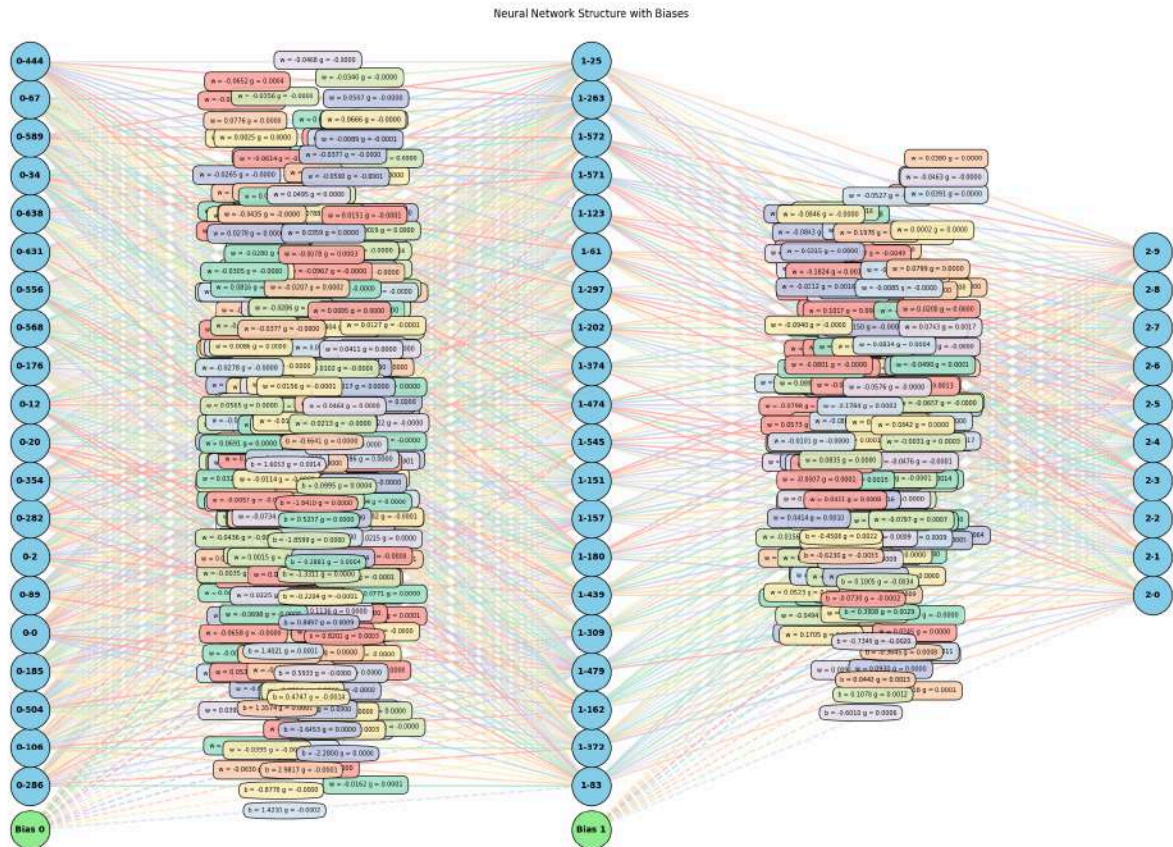
3.5.2 Regularisasi L2

Hasil yang didapat:

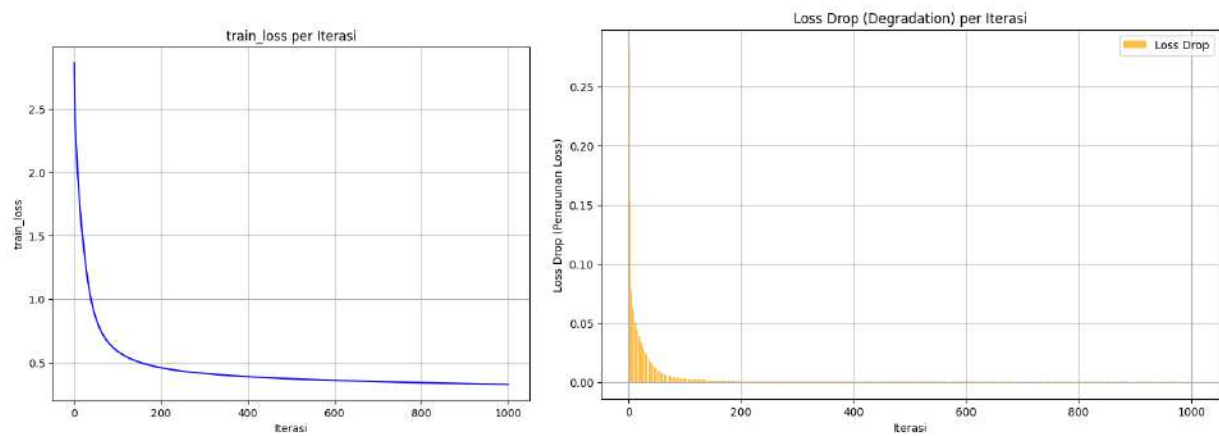
Akurasi model yang didapat yaitu 0.9210714101791382. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 732.82 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



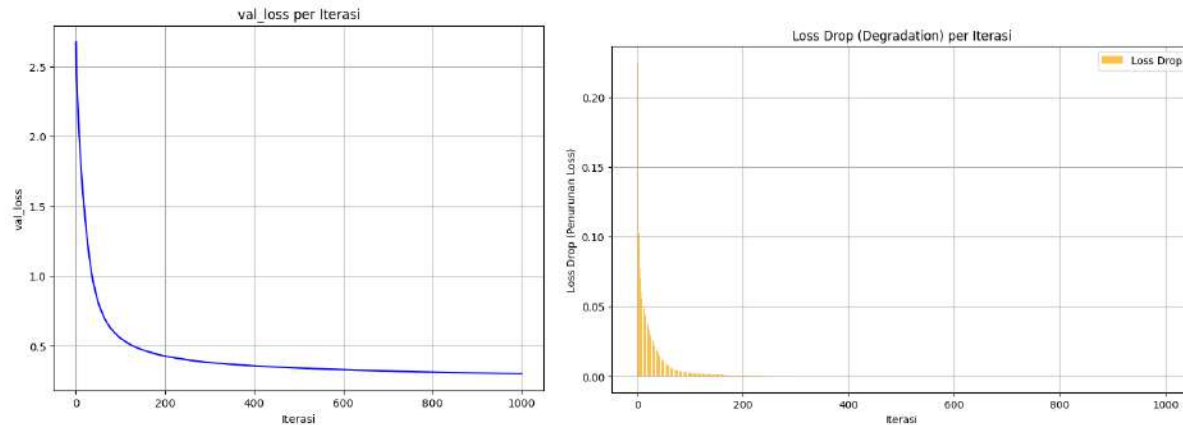
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training loss* dan *loss drop* dapat dilihat pada gambar dibawah:



Untuk Validasi:



3.5.3 Analisis untuk pengaruh regularisasi

Dalam dataset ini, penentuan nilai alpha pada uji coba regularisasi di atas mungkin tidak tepat dijadikan standar tunggal untuk meningkatkan akurasi, mengingat akurasi justru menurun setelah dilakukan regularisasi. Menurut penulis, fenomena ini cukup lumrah terjadi karena tidak semua dataset maupun parameter model akan otomatis cocok dengan metode regularisasi tertentu. Hal ini dapat dipahami dari fungsi dasar regularisasi itu sendiri, yang memang bertujuan untuk mencegah overfitting. Akibatnya, wajar jika terkadang akurasi pada data pengujian terlihat menurun, tetapi secara praktik, model yang di regularisasi akan memiliki daya generalisasi yang lebih baik ketika diterapkan pada data dunia nyata.

Pada regularization L1 (Lasso), penalti diberikan berdasarkan nilai mutlak dari bobot yang menyebabkan banyak bobot menjadi persis nol atau mendekati nol, seperti yang tergambar jelas pada distribusi bobot dalam grafik pengujian. Kondisi ini dikenal sebagai efek sparsity, di mana model dipaksa untuk memilih fitur-fitur yang benar-benar signifikan saja. Akibatnya, meskipun akurasi menurun, L1 secara tidak langsung menghasilkan model yang lebih sederhana dan efisien.

Sementara itu, regularization L2 (Ridge) memberikan penalti yang lebih besar pada bobot-bobot dengan nilai tinggi. Hal ini menyebabkan model cenderung mendistribusikan bobot secara lebih merata, dan bobot dengan kesalahan yang besar akan mendapatkan penalti lebih signifikan. Konsepnya adalah, fitur yang memiliki korelasi tinggi dengan target akan memperoleh bobot yang lebih besar secara proporsional, namun apabila terjadi kesalahan prediksi, penalti yang

dikenakan juga besar. Dalam grafik distribusi bobot gradien dengan L2 terlihat bahwa nilai bobot cenderung lebih homogen dan dominan pada rentang tertentu.

Dengan demikian, meskipun dalam uji coba ini regularisasi menyebabkan penurunan akurasi pengujian, penggunaan regularisasi tetap penting terutama untuk menjaga kestabilan model dalam menghadapi data yang lebih variatif di dunia nyata. Penurunan akurasi yang teramati pada data uji sebenarnya merupakan hal yang wajar terjadi sebagai trade-off dalam mencapai kestabilan dan ketahanan model terhadap kondisi baru, di luar data pelatihan. Terlihat dari perbandingan *train* dan *val loss*, tanpa regularisasi memberikan performa lebih baik.

3.6 Perbandingan model *built-in* dengan model *custom* tugas besar

Komparasi ini menggunakan parameter sebagai berikut:

1. Hidden layer = *depth*:1 , *width*: 400
2. Epoch / max-epoch = 2000
3. *Learning_rate* = 0.001
4. *Batch_size* = 200
5. *Activation Function* = default hidden: ReLU , dan output *softmax* untuk *multiclass classification*
6. *Weight Initialization* = Xavier dan He

3.6.1 Model *built in*

```
Iteration 1, loss = 0.32980190
Iteration 2, loss = 0.14267594
Iteration 3, loss = 0.09653205
Iteration 4, loss = 0.07009054
Iteration 5, loss = 0.05395898
Iteration 6, loss = 0.04154504
Iteration 7, loss = 0.03377707
Iteration 8, loss = 0.02591062
Iteration 9, loss = 0.02117890
Iteration 10, loss = 0.01685890
Iteration 11, loss = 0.01333001
Iteration 12, loss = 0.01140906
Iteration 13, loss = 0.00844530
Iteration 14, loss = 0.00688647
Iteration 15, loss = 0.00550894
Iteration 16, loss = 0.00395972
Iteration 17, loss = 0.00330826
Iteration 18, loss = 0.00525504
Iteration 19, loss = 0.00670997
Iteration 20, loss = 0.00963354
Iteration 21, loss = 0.00561977
Iteration 22, loss = 0.00206197
Iteration 23, loss = 0.00135769
Iteration 24, loss = 0.00121231
Iteration 25, loss = 0.00115183
...
Iteration 35, loss = 0.00120618
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
Akurasi MLPClassifier: 0.9811428571428571
Waktu training MLPClassifier: 157.46 detik
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

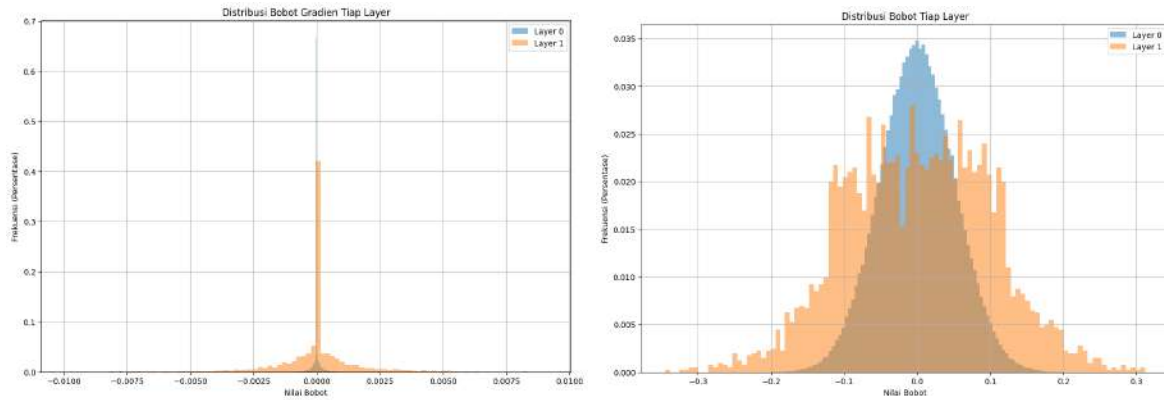
3.6.2 Model custom

Uji coba kedua dengan spesifikasi:

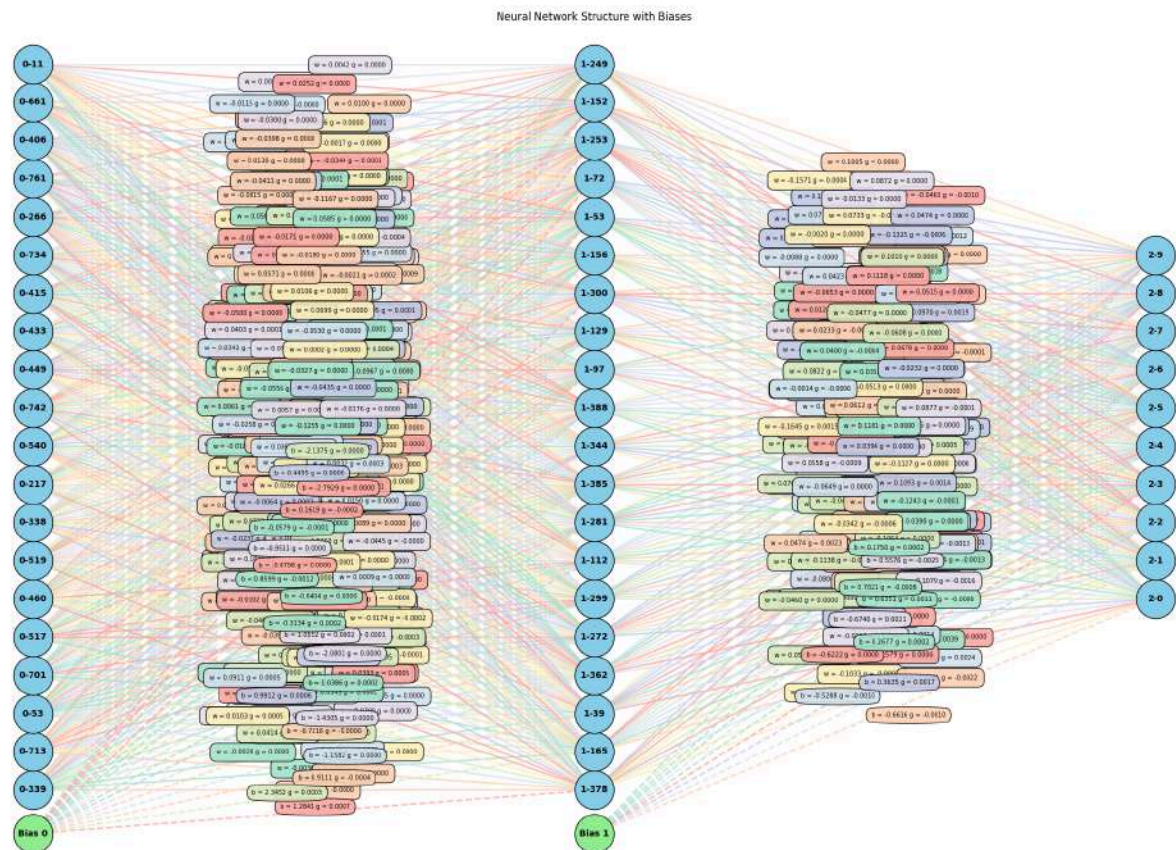
4. Input Layer = 784 neuron
5. Hidden Layer 1 = 400 neuron
6. Output Layer = 10 neuron

Hasil yang didapat:

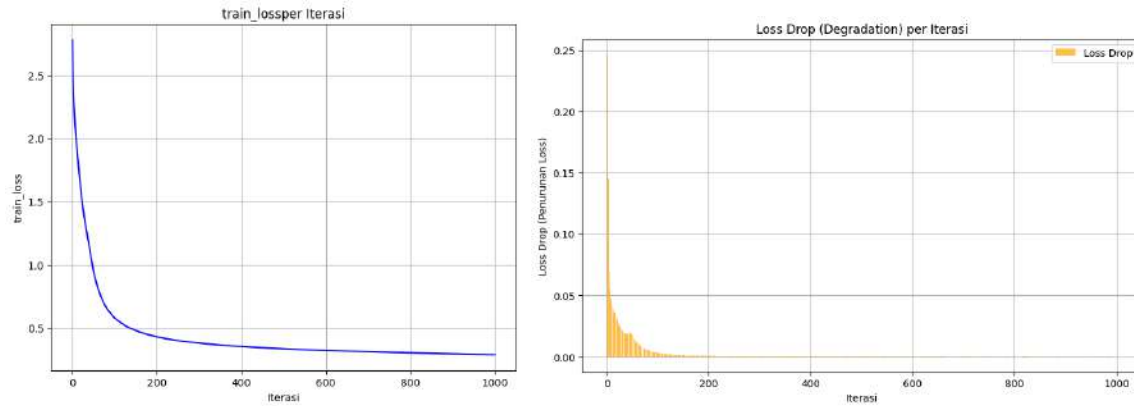
Akurasi model yang didapat yaitu 0.92149996757507323. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 538.46 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



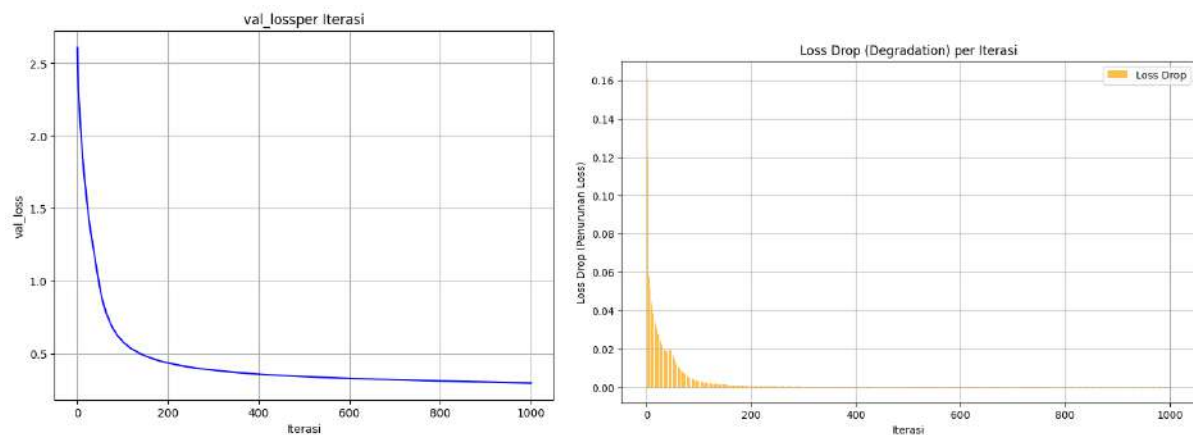
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



Untuk Validation:



3.6.3 Analisis perbedaan model *custom* dengan *built in*

Menurut kami, kami berhasil mendapatkan hasil yang optimal dalam pembangunan model kami. Disini, alasan terjadinya perbedaan pada hasil model ada pada bahwa model *built in* menerapkan *adam* sebagai sarana perubahan *learning rate*. Adam disini berpengaruh dalam menyesuaikan *learning rate* yang efektif.

Pada analisis *learning rate*, kami berhasil mendapatkan akurasi mirip dengan *learning rate* statis 0.1, sehingga kami percaya bahwa jikalau kami mampu dalam membuat *adam* sendiri, akan dapat menghasilkan akurasi yang lebih baik dari yang kami dapati saat ini.

3.7 Perbandingan dengan model RMSNorm

Parameter uji coba:

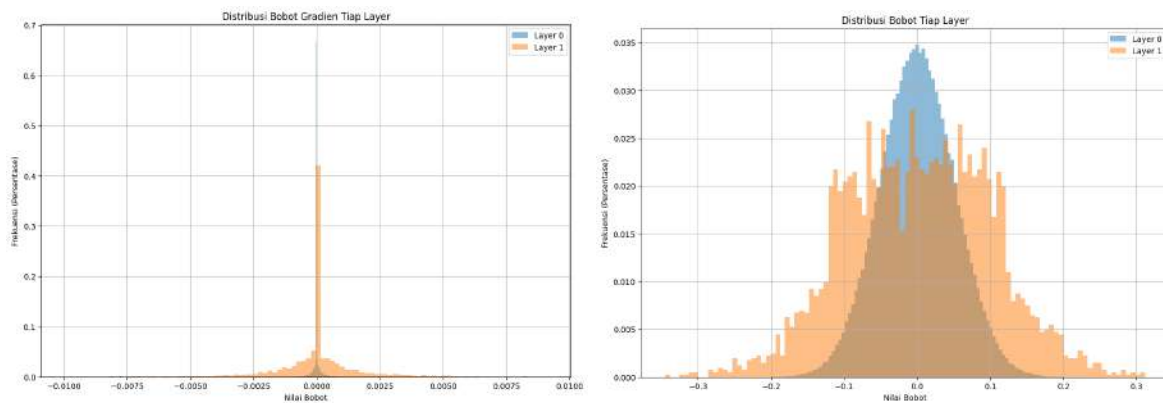
1. *Epoch* = 1000

2. *Input Layer* = 784
3. *Hidden Layer* = 400
4. *Output Layer* = 10
5. *Learning Rate* = 0.001
6. *Loss Function* = *Categorical Cross-Entropy*
7. *Activation Function Hidden Layer* = ReLU
8. Inisialisasi bobot = Xavier dan He

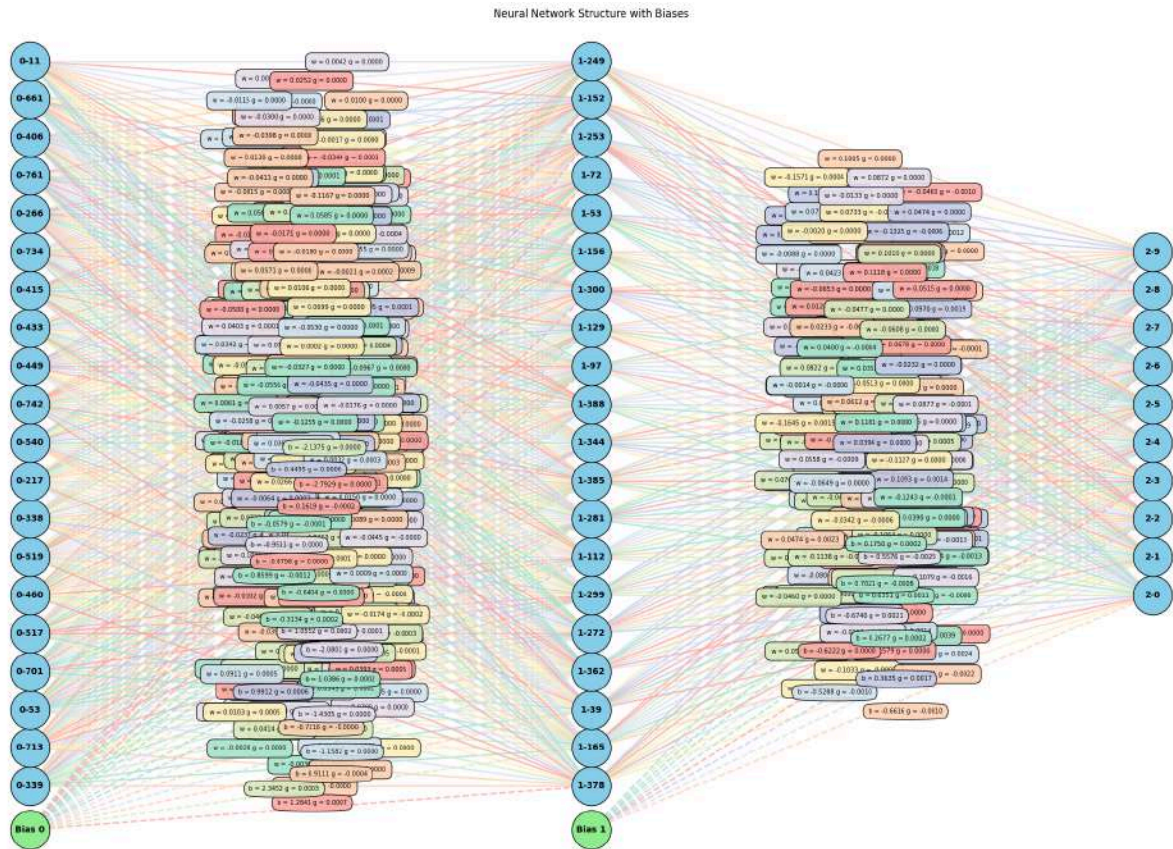
3.7.1 Tanpa Normalisasi

Hasil yang didapat:

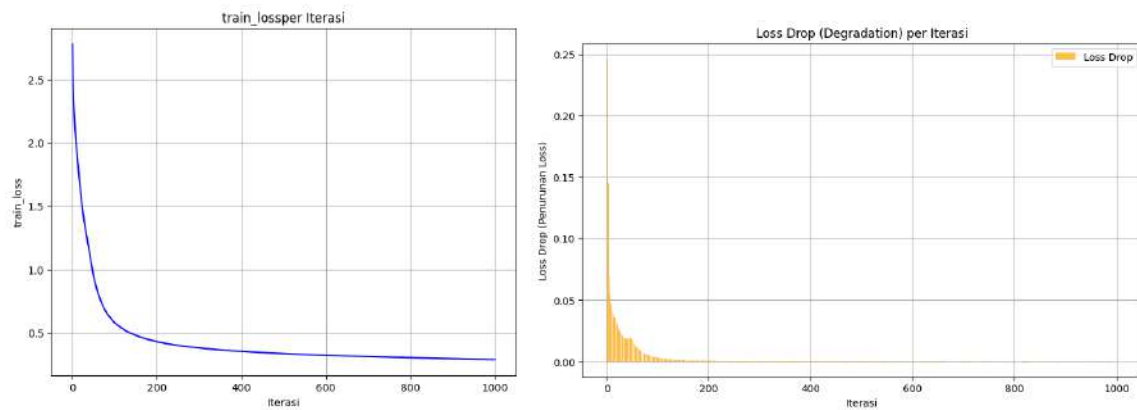
Akurasi model yang didapat yaitu 0.92149996757507323. Waktu yang dihabiskan oleh *device* penguji dalam melaksanakan uji coba ini yaitu: 538.46 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



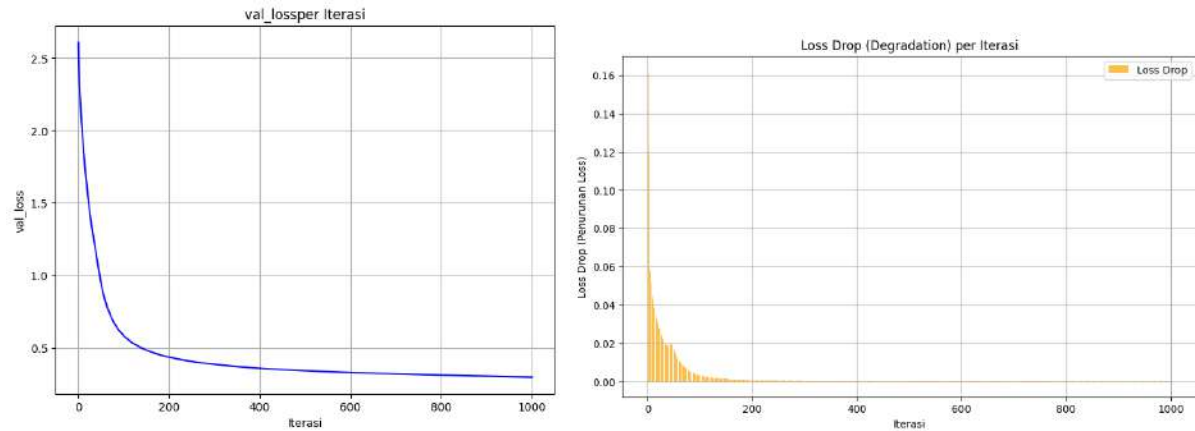
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training drop* dan *loss drop* dapat dilihat pada grafik dibawah:



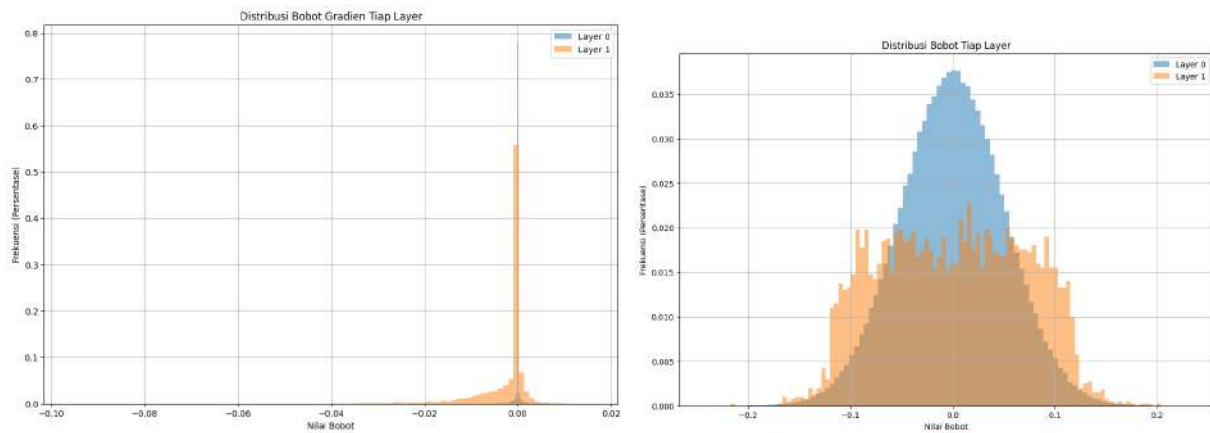
Untuk Validation:



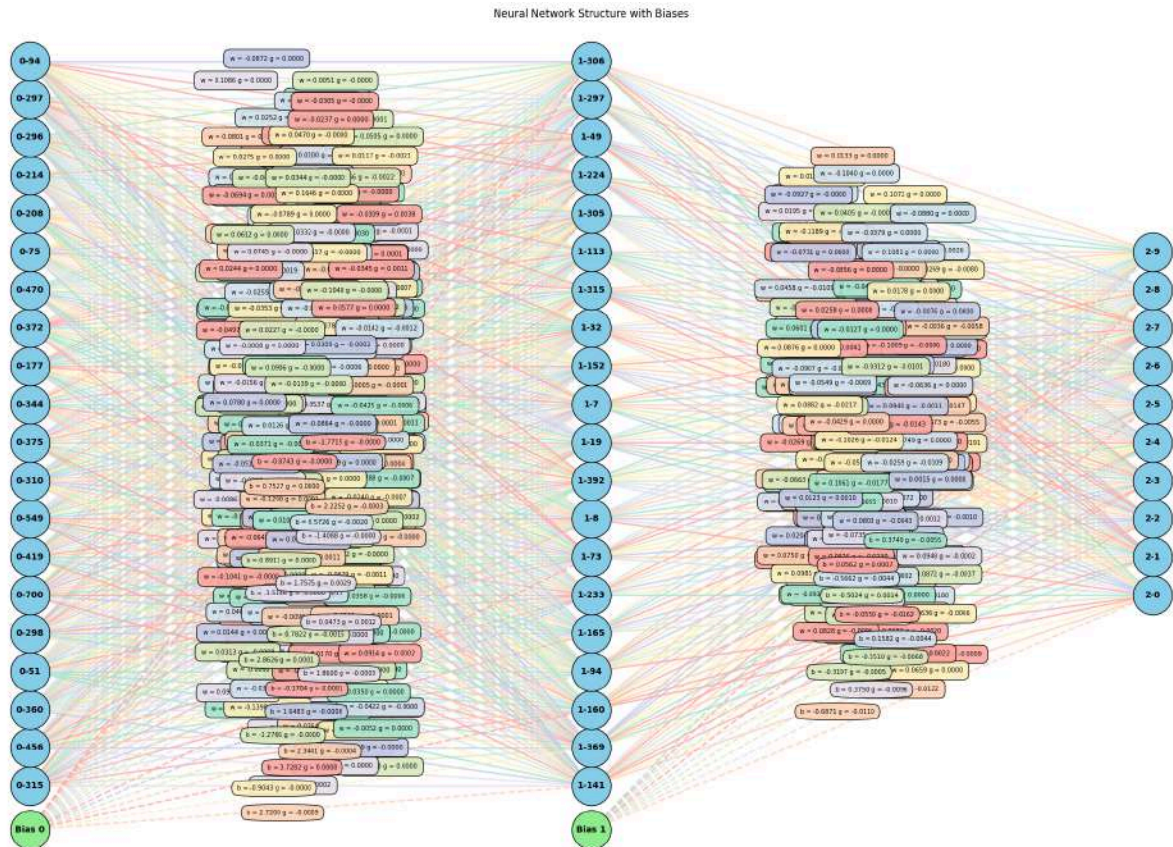
3.7.2 Dengan Normalisasi RMSNorm

Hasil yang didapat:

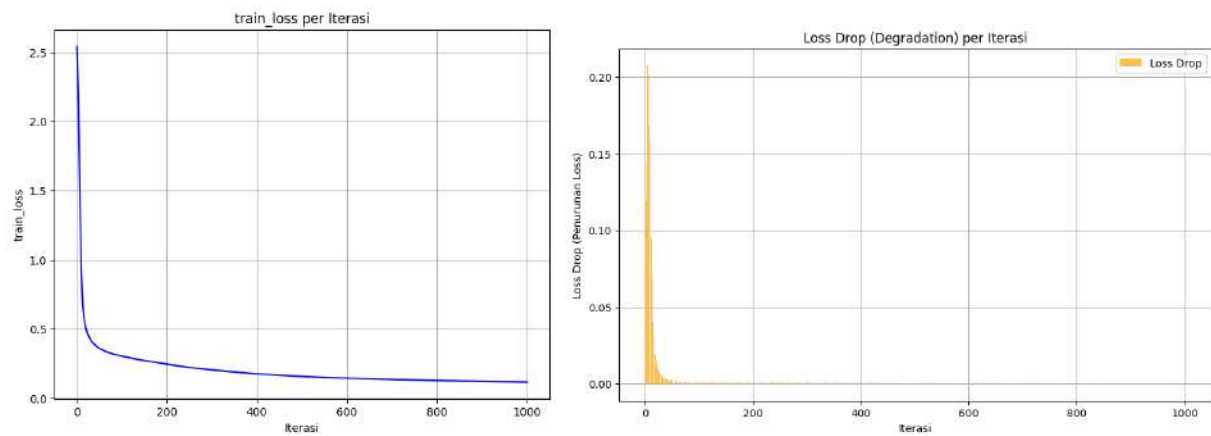
Akurasi model yang didapat yaitu 0.9575714468955994. Waktu yang dihabiskan oleh *device* pengujian dalam melaksanakan uji coba ini yaitu: 1085.13 detik. Grafik distribusi bobot dan bobot gradien dapat dilihat pada grafik dibawah:



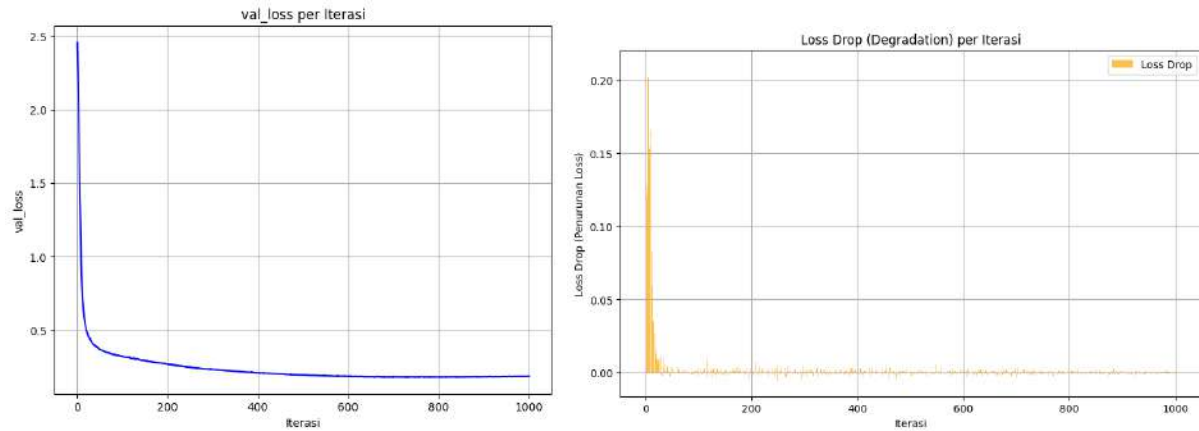
Struktur *network* dapat dilihat pada gambar dibawah:



Perubahan *training loss* dan *loss drop* dapat dilihat pada gambar dibawah:



Untuk validasi:



3.7.3 Analisis untuk pengaruh normalisasi

Menghitung performa *base* , RMSNorm memberikan hasil yang cukup baik pada pengujian dataset ini. Salah satu performa terbaik yang telah kami uji yakni berupa gabungan antara RMSNorm dengan *learning rate* 0,1. Menurut kami gabungan ini cukup baik, karena hasil *loss drop* yang cukup cepat dimana pada iterasi kedua telah mencapai *train* dan *val* loss dibawah 0.5.

Karakteristik dari normalisasi tentu ada pada skala yang lebih konsisten. Dibandingkan dengan tanpa normalisasi, distribusi bobot pada RMSNorm lebih terstruktur , dimana komposisi-nya terlihat lebih tinggi (berfrekuensi tinggi). Secara umum, RMSNorm digunakan untuk mencegah adanya *outlier*, layaknya normalisasi lainnya. Dengan adanya normalisasi, ibaratnya seperti mengurangi *gap* antara bobot.

Bobot dapat bertumbuh secara tidak terkontrol tanpa ternormalisasi , dan hal ini dapat menyebabkan *overfitting*. Normalisasi membatasi pertumbuhan bobot sehingga selalu ada pada *range* yang lebih kecil, seperti yang terlihat pada grafik diatas.

Train loss serta *val loss* juga mengalami penurunan yang lebih curam (secara grafik), dimana terlihat pada grafik , bahkan sebelum iterasi ke 50 saja sudah terlihat terjadinya lengkungan.

Secara keseluruhan, RMSNorm membatasi pertumbuhan bobot yang berlebihan. Batasan yang diberikan memang tidak selalu cocok untuk setiap dataset, namun setidaknya pada dataset ini memberikan hasil yang menurut kami cukup baik.

3.8 Analisis perbandingan train dan val

Dari semua percobaan diatas , terlihat bahwa tidak ada perbedaan signifikan antara *train* dan *val* jika dilakukan perbandingan terhadap *loss* nya. Dari *verbose* sendiri juga terlihat bedanya sangat kecil. Hal ini membuktikan bahwa model dapat bekerja pada potongan data manapun dan merupakan sebuah parameter kesuksesan.

BAB IV

KESIMPULAN DAN SARAN

Pada tugas besar ini , lebih ditekankan pada pembelajaran mengenai beberapa cara dalam memodifikasi sebuah model *neural network*. Tidak ada sebuah parameter khusus yang membuat sebuah model dapat memprediksi “lebih baik” . Semua hasil *tuning* model bergantung kepada data serta kecocokannya dengan berbagai parameter.

Pada hasil uji coba , perubahan yang paling mempengaruhi ada pada perubahan *learning rate* dan juga penentuan bobot awal, dimana kita melihat semakin besar *learning rate* memberikan hasil yang lebih baik pada dataset ini dan juga penggunaan Xavier/He juga memberikan hasil yang baik pula.

Konten dataset sebenarnya cukup general , mengingat dataset MNIST adalah dataset yang umumnya diberikan pada pemula sehingga tidak memiliki kompleksitas tinggi. Oleh sebab itu, implementasi model secara umum sudah dapat merepresentasikan data dengan baik. Penambahan kompleksitas seperti regularisasi atau normalisasi dapat menjadi senjata makan tuan bagi penguji. Untuk sekarang , hasil terbaik kami ada pada *learning rate* 0,1, menggunakan RMSNorm, dan juga mengandalkan *depth*.

Kesimpulannya, model sudah bekerja dengan baik , dimana jika kita menggunakan pembandingan model *built in*, model kami mencapai akurasi yang cukup serupa, hanya saja lebih lambat. Konsiderasi diperlukan dimana model bawaan menerapkan Adam serta *early stopping*. Kombinasi dua fitur tersebut membuat model bawaan dapat menghasilkan model yang merepresentasi data dengan baik namun masih menghemat waktu , karena Adam menyesuaikan dengan data dan jika sudah terlewat tidak sesuai, *early stopping* akan berhenti. Namun, kami berhasil mendapatkan hasil yang serupa , hanya saja dengan mengorbankan waktu yang cukup lama. Secara keseluruhan , kami merasa bahwa pengujian ini bersifat sukses. Saran untuk tugas besar ini cukup dengan melakukan percobaan untuk mencari parameter terbaik.

BAB V

LAMPIRAN

5.1 Tabel Pembagian Tugas

Nama	Pembagian Tugas
Wilson Yusda (13522019)	Code , Laporan
Enrique Yanuar (13522077)	Code , Laporan
Mesach Harmasendro (13522117)	Code , Laporan

5.2 Repository Github

<https://github.com/mybajwk/ML-49>

REFERENSI

https://edunexcontentproshot.blob.core.windows.net/edunex/nullfile/1741577891298_IF3270-Mgg03-FFNN-print?sv=2024-11-04&spr=https&st=2025-03-10T03%3A32%3A33Z&se=2027-03-10T03%3A32%3A33Z&sr=b&sp=r&sig=KdLqCzzo5Z3lQwFtxt%2BEoNGBpLkPPB2YMdYxbAAqYzQ%3D&rsct=application%2Fpdf

<https://medium.com/@alejandro.itoaramendia/l1-and-l2-regularization-part-1-a-complete-guide-51cf45bb4ade>

<https://www.v7labs.com/blog/neural-networks-activation-functions>

https://docs.google.com/document/d/1ygwQ-vVzynPJG2KwMVpqhCozwrB-zVfjU_hvdhoE2mg/edit?tab=t.0