

Laporan Tugas Besar 1 IF3270 Pembelajaran Mesin

Feedforward Neural Network



Disusun oleh:

Kelompok 26

Muhammad Yusuf Rafi (13522009)

Debrina Veisha Rashika W (13522025)

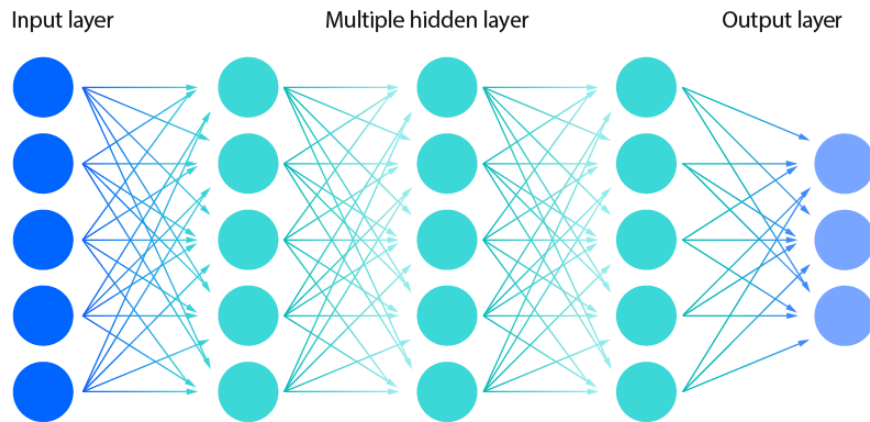
Melati Anggraini (13522035)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2025

DAFTAR ISI

DAFTAR ISI.....	1
BAB I DESKRIPSI PERSOALAN.....	2
BAB II IMPLEMENTASI.....	5
2.1 Implementasi Kelas dan Fungsi.....	5
2.1.1 Kelas FFNN.....	5
2.1.1.1 Atribut Kelas FFNN.....	5
2.1.1.2 Method Kelas FFNN.....	6
2.1.2 Kelas Layers.....	7
2.1.2.1 Atribut Kelas Layers.....	8
2.1.3 Fungsi Aktivasi.....	8
2.2 Penjelasan Forward Propagation.....	10
2.2.1 Penjelasan Implementasi Algoritma Forward Propagation.....	10
2.2.2 Implementasi Kode FeedForward.....	13
2.3 Penjelasan BackPropagation.....	15
2.3.1 Penjelasan Implementasi Algoritma BackPropagation.....	15
2.3.2 Implementasi Kode BackPropagation.....	17
BAB III PENGUJIAN DAN ANALISIS.....	19
3.1 Pengaruh Depth dan Width.....	19
3.1.1 Pengaruh Width.....	19
3.1.2 Pengaruh Depth.....	24
3.2 Pengaruh Fungsi Aktivasi.....	29
3.3 Pengaruh Learning Rate.....	39
3.4 Pengaruh Inisialisasi Bobot.....	44
3.5 Perbandingan Regularisasi.....	52
3.6 Perbandingan Normalisasi RMSNorm.....	57
3.7 Perbandingan dengan library sklearn.....	60
BAB IV KESIMPULAN DAN SARAN.....	62
4.1 Kesimpulan.....	62
4.2 Saran.....	62
PEMBAGIAN TUGAS.....	63
LINK REPOSITORY.....	63
DAFTAR PUSTAKA.....	63

BAB I DESKRIPSI PERSOALAN



Gambar 1.1 Ilustrasi Feedforward Neural Network

Feedforward Neural Network (FFNN) adalah salah satu jenis ANN yang terdiri dari beberapa lapisan yang menghubungkan neuron-neuron secara searah, dari *input layer* → *hidden layer* → *output layer*. Dalam tugas ini, permasalahan yang diselesaikan adalah bagaimana membangun dan mengimplementasikan Feedforward Neural Network (FFNN) dari awal (from *scratch*) tanpa menggunakan library deep learning. FFNN merupakan salah satu jenis Artificial Neural Network (ANN) yang digunakan untuk berbagai tugas seperti klasifikasi, regresi, dan prediksi data.

Model ANN yang dibuat dapat menerima jumlah neuron pada setiap layer, memilih fungsi aktivasi yang sesuai, serta menggunakan berbagai metode inisialisasi bobot. Selain itu, model mendukung berbagai fungsi loss yang umum digunakan dalam supervised learning, seperti Mean Squared Error (MSE), Binary Cross-Entropy, dan Categorical Cross-Entropy, dengan rumus sebagai berikut.

Nama Fungsi Loss	Definisi Fungsi
MSE	$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
Binary Cross-Entropy	$\mathcal{L}_{BCE} = -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$ <p> y_i = Actual binary label (0 or 1) \hat{y}_i = Predicted value of y_i n = Batch size </p>
Categorical Cross-Entropy	$\mathcal{L}_{CCE} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^C (y_{ij} \log \hat{y}_{ij})$ <p> y_{ij} = Actual value of instance i for class j \hat{y}_{ij} = Predicted value of y_{ij} C = Number of classes n = Batch size </p>

Selain membangun struktur model FFNN, tugas ini juga melakukan implementasi forward propagation untuk memproses input data dan menghasilkan output prediksi, serta backward propagation untuk menghitung gradien guna memperbarui bobot menggunakan gradient descent. Model yang dibuat mendukung batch processing, serta memungkinkan pengguna mengatur parameter seperti batch size, learning rate, jumlah epoch, dan opsi verbose untuk memantau proses training. Fungsi aktivasi yang harus diimplementasikan pada tugas ini, sebagai berikut.

Nama Fungsi Aktivasi	Definisi Fungsi
Linear	$Linear(x) = x$
ReLU	$ReLU(x) = \max(0, x)$
Sigmoid	$\sigma(x) = \frac{1}{1 + e^{-x}}$
Hyperbolic Tangent (tanh)	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Softmax	Untuk vector $\vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$, $softmax(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$
Bonus	
Elu	$ELU(x, \alpha) = \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$
Swish	$Swish(x, \beta) = x\sigma(\beta x)$

Untuk melakukan analisis, model yang dikembangkan dapat menampilkan struktur *graph* beserta bobot dan gradien, serta menyajikan distribusi bobot dan gradien dari layer tertentu dalam bentuk visualisasi. Selain itu, model memiliki kemampuan untuk *load* dan *save* model yang telah dilatih agar dapat digunakan di lain waktu.

BAB II IMPLEMENTASI

2.1 Implementasi Kelas dan Fungsi

2.1.1 Kelas FFNN

Kelas utama yang menangani dari inisialisasi, forward, dan backward propagation. Inisialisasi dari bobot dan gradien, serta berbagai *method* perhitungan untuk membantu menyelesaikan perhitungan FFNN. Berikut merupakan atribut dan *method* dari kelas ini:

2.1.1.1 Atribut Kelas FFNN

Nama Atribut	Deskripsi
batch_size	Ukuran batch input.
learning_rate	Learning rate untuk perhitungan <i>gradient descent</i>
epoch	Jumlah iterasi training.
verbose	Pilihan untuk menampilkan progress bar beserta dengan kondisi training loss dan validation loss saat itu atau tidak.
loss_func	Fungsi loss yang digunakan (mse, binary, categorical).
weight_init	Metode inisialisasi bobot (zero, uniform, normal).
seed	Nilai seed untuk untuk <i>reproducibility</i> .
input_train	Daftar data Input untuk training.
target_train	Daftar train target dari <i>layer</i> .
input_val	Daftar data Input untuk validasi.
target_val	Daftar target validasi dari <i>layer</i> .

layers	Daftar objek <i>layers</i> yang membentuk jaringan.
delta_gradien	Daftar gradien untuk pembaruan bobot.
loss_train_history	Menyimpan nilai histori loss pada data train untuk setiap epoch.
loss_val_history	Menyimpan nilai histori loss pada data validasi untuk setiap epoch.

2.1.1.2 Method Kelas FFNN

Nama Method	Deskripsi
initDeltaGradien()	Menginisialisasi matriks gradien dengan nol.
initWeight()	Menginisialisasi bobot berdasarkan metode tertentu.
calcLoss(output, target)	Menghitung error berdasarkan fungsi loss yang dipilih.
updateWeight()	Memperbarui bobot berdasarkan gradien yang dihitung.
updateGradien(layer_idx, delta, input)	Menghitung gradien untuk setiap layer berdasarkan error.
def addInputTarget(self, input_train, input_val, target_train, target_val)	Menambahkan data input dan target untuk training dan validasi.
addHiddenLayer(layer)	Menambahkan <i>hidden layer</i> ke jaringan.
plot_weight_distribution()	Menampilkan histogram distribusi bobot.
plot_gradient_distribution()	Menampilkan histogram distribusi gradien.
feedForward()	Fungsi untuk melakukan propagasi maju dalam

	Feedforward Neural Network (FFNN) dan melakukan prediksi serta menghitung loss.
<code>backPropagation(inputs, netsLayer, target)</code>	Melakukan propagasi mundur untuk mendapat nilai gradien pada tiap bobot dan memperbarui nilai bobot.
<code>visualize_network(ffnn)</code>	Melakukan visualisasi terhadap nilai bobot dan gradien dari model yang telah dibuat dalam bentuk graf.
<code>save_model(filename)</code>	Menyimpan model ke dalam file dengan nama yang diberikan, sehingga model dapat digunakan kembali tanpa perlu dilatih ulang.
<code>load_model(filename)</code>	Memuat model dari file yang telah disimpan sebelumnya.
<code>plot_loss()</code>	Menampilkan grafik loss selama proses training, baik untuk training loss maupun validation loss.
<code>predict(X)</code>	Melakukan prediksi terhadap input X menggunakan model yang telah dilatih.

2.1.2 Kelas Layers

Kelas ini merepresentasikan komponen layer dalam jaringan, Kelas ini menerima dan menyimpan beberapa masukan seperti, jumlah neuron input, neuron dalam layer, fungsi aktivasi, dan juga nilai dan bobot biasanya. Berikut merupakan atribut dari kelas ini:

2.1.2.1 Atribut Kelas Layers

Nama Atribut	Deskripsi
n_inputs	Jumlah neuron input ke layer ini.
n_neurons	Jumlah neuron dalam layer.
activ_func	Fungsi aktivasi yang digunakan.
use_bias	Apakah layer menggunakan bias atau tidak.
weight	Matriks bobot yang menghubungkan input dengan neuron.

2.1.3 Fungsi Aktivasi

Nama Method	Deskripsi
tanh(x)	Mengubah input menjadi nilai dalam rentang (-1, 1).
softmax(x)	Mengubah input menjadi probabilitas (jumlah totalnya = 1)
linear(x)	Mengembalikan nilai inputnya tanpa transformasi.
relu(x)	Mengubah semua nilai negatif menjadi nol, sementara nilai positif tetap sama.
sigmoid(x)	Mengubah input menjadi nilai dalam rentang (0,1)

2.1.3 Fungsi Turunan Aktivasi

Nama Method	Deskripsi
linearDerivative(net)	Menghitung turunan dari fungsi aktivasi linear terhadap x
tanDerivative(net)	Menghitung turunan dari fungsi aktivasi tan terhadap x
softmaxDerivative(net, threshold)	Menghitung turunan dari fungsi aktivasi softmax terhadap x
sigmoidDerivative(net)	Menghitung turunan dari fungsi aktivasi sigmoid terhadap x
reluDerivative(net)	Menghitung turunan dari fungsi aktivasi relu terhadap x
eluDerivative(net)	Menghitung turunan dari fungsi aktivasi elu terhadap x
swishDerivative(net)	Menghitung turunan dari fungsi aktivasi swish terhadap x

2.1.4 Fungsi Turunan Loss

Nama Method	Deskripsi
mseDerivative(o, target)	Menghitung turunan dari fungsi loss MSE terhadap x
binaryDerivative(o, target)	Menghitung turunan dari fungsi loss binary cross-entropy terhadap x
categoricalDerivative(o, target)	Menghitung turunan dari fungsi loss categorical cross-entropy terhadap x

2.1.5 Fungsi Menghitung Gradien

Nama Method	Deskripsi
outputLayer(o, net, target, activFunc, LossFunc)	Menghitung gradien bobot pada output layer
hiddenLayer(w, net, delta, activFunc)	Menghitung gradien bobot pada hidden layer

2.2 Penjelasan Forward Propagation

2.2.1 Penjelasan Implementasi Algoritma Forward Propagation

Algoritma pada fungsi `feedForward()` digunakan untuk melakukan propagasi maju (forward propagation) dalam Feedforward Neural Network (FFNN). Tahapan proses yang dilakukan pada algoritma ini, sebagai berikut.

1. Proses Feed Forward diawali dengan menginisialisasi bobot pada tiap layer dengan metode yang telah dipilih sebelumnya. Proses inisialisasi bobot menggunakan fungsi `initWeight()` yang akan mengembalikan bobot pada tiap layer termasuk bias. Terdapat 5 pilihan inisialisasi bobot yang bisa dipilih pengguna sebagai berikut.
 - a. Zero initialization
 - b. Random dengan distribusi uniform.
 - c. Random dengan distribusi normal.
 - d. Random dengan Xavier
 - e. Random dengan He
2. Setelah melakukan inisialisasi pada bobot tiap layer, dilakukan proses iterasi pada masing-masing epoch yang diawali dengan menginisialisasi nilai gradien tiap layer menjadi 0 dan juga menginisialisasi nilai error menjadi 0.
3. Selanjutnya dilakukan iterasi untuk setiap batchnya dan menambah nilai 1 pada matriks input sebagai nilai bias yang akan digunakan untuk mengalikan dengan bobot pada tiap layer nantinya dan nilai matriks ini disimpan sebagai *current input*.

4. Setelah menambah nilai bias pada input awal dilakukan iterasi pada setiap layer dimulai dari input layer hingga output layer. Pada setiap layer akan dihitung nilai net dengan mengalikan matriks input dengan bobot.
5. Selanjutnya nilai net akan dimasukkan pada fungsi aktivasi yang dipilih untuk menghasilkan output dari layer tersebut. Nilai output akan disimpan sebagai nilai *current* saat ini untuk digunakan pada layer selanjutnya. Fungsi aktivasi yang digunakan pada bagian ini sebagai berikut:

Nama Fungsi Aktivasi	Definisi Fungsi
Linear	$Linear(x) = x$
ReLU	$ReLU(x) = \max(0, x)$
Sigmoid	$\sigma(x) = \frac{1}{1 + e^{-x}}$
Hyperbolic Tangent (tanh)	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Softmax	Untuk vector $\vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$, $softmax(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$
Elu	$ELU(x, \alpha) = \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$
Swish	$Swish(x, \beta) = x\sigma(\beta x)$

6. Jika layer bukan layer terakhir, maka dilakukan penambahan nilai bias 1 untuk menjadi nilai pertama dari matriks *current* dan proses 4 - 6 terus dilakukan hingga mencapai layer terakhir.
7. Setelah proses iterasi selesai, akan dihitung nilai loss pada output dengan nilai target menggunakan fungsi calcLoss(). Terdapat tiga fungsi Loss yang diterapkan pada tugas ini, sebagai berikut.

Nama Fungsi Loss	Definisi Fungsi
MSE	$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
Binary Cross-Entropy	$\mathcal{L}_{BCE} = -\frac{1}{n} \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$ <p>y_i = Actual binary label (0 or 1)</p> <p>\hat{y}_i = Predicted value of y_i</p> <p>n = Batch size</p>
Categorical Cross-Entropy	$\mathcal{L}_{CCE} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^C (y_{ij} \log \hat{y}_{ij})$ <p>y_{ij} = Actual value of instance i for class j</p> <p>\hat{y}_{ij} = Predicted value of y_{ij}</p> <p>C = Number of classes</p> <p>n = Batch size</p>

8. Setelah menghitung nilai loss, akan dimulai untuk proses backpropagation untuk mengupdate nilai bobot melalui fungsi backPropagation(). Hasil output dari fungsi ini

berupa nilai gradien yang selanjutnya digunakan untuk memperbarui nilai bobot melalui fungsi `updateWeight()` sehingga pada epoch selanjutnya bobot yang digunakan sudah merupakan bobot terbaru.

2.2.2 Implementasi Kode FeedForward

```
def feedForward(self):
    self.initWeight()
    start_time = time.time()
    self.loss_train_history = []
    self.loss_val_history = []

    for j in range(self.epoch):
        self.initDeltaGradien()
        train_error = 0
        epoch_start_time = time.time()

        for i, batch in enumerate(self.input_train):
            inputs: list[list[float]] = []
            nets: list[list[float]] = []
            self.initDeltaGradien()

            batch = np.array(batch)
            batch_size = batch.shape[0]
            bias = np.ones((batch_size, 1))
            current = np.hstack((bias, batch))
            inputs.append(current.copy().transpose().tolist())

            for layer in self.layers:
                net = np.dot(current, layer.weight)
                current = layer.activ_func(net)
                nets.append(net.copy().transpose().tolist())
                if layer != self.layers[-1]:
                    bias = np.ones((batch_size, 1))
```

```
        current = np.hstack((bias, current))
        inputs.append(current.copy().transpose().tolist())

        train_error += self.calcLoss(current,
self.target_train[i])
        self.backPropagation(inputs, nets,
self.target_train[i])
        self.updateWeight()

        self.loss_train_history.append(train_error)
        val_pred = self.predict(self.input_val)
        val_error = self.calcLoss(val_pred, self.target_val)
        self.loss_val_history.append(val_error)

        epoch_duration = time.time() - epoch_start_time
        total_elapsed = time.time() - start_time
        estimated_total_time = (total_elapsed / (j + 1)) *
self.epoch
        eta = estimated_total_time - total_elapsed

        if self.verbose == 1:
            bar_length = 30
            progress = (j + 1) / self.epoch
            bar = "=" * int(bar_length * progress) + "-" *
(bar_length - int(bar_length * progress))
            sys.stdout.write(f"\rEpoch {j+1}/{self.epoch} [{bar}]
{progress*100:.1f}% - Training Loss: {train_error:.4f} - Validation
Loss: {val_error:.4f} - {epoch_duration:.2f}s/epoch - ETA:
{eta:.2f}s")
            sys.stdout.flush()

        print("\n          Training History          ")
        print("=====")
        print("Epoch | Train Loss | Val Loss")
```

```
print("-----")
for epoch, (train_loss, val_loss) in
enumerate(zip(self.loss_train_history, self.loss_val_history)):
    print(f"{epoch+1:5d} | {train_loss:.6f} |
{val_loss:.6f}")
print("=====")
```

2.3 Penjelasan BackPropagation

2.3.1 Penjelasan Implementasi Algoritma BackPropagation

Algoritma pada fungsi backPropagation digunakan untuk melakukan propagasi mundur (Backpropagation) dalam Feedforward Neural Network (FFNN) untuk melakukan mencari nilai gradien dan melakukan update bobot pada tiap layer. Tahapan proses yang dilakukan pada algoritma ini, sebagai berikut.

1. Backpropagation dilakukan dari menghitung gradien dari layer terakhir atau output layer menuju layer pertama.
2. Saat berada di output layer nilai delta akan dihitung dengan fungsi `outputLayer()` yang menerima masukan nilai output, net, target, fungsi aktivasi yang digunakan dan loss yang digunakan.
3. Delta pada output layer dihitung dengan *chain rule* berikut.

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

Sehingga untuk menghitung nilai delta pada output dilakukan dengan mengalikan turunan fungsi loss dengan turunan fungsi aktivasi yang digunakan pada layer tersebut. Hal ini dilakukan pada fungsi `outputLayer()` yang didalamnya akan menghitung nilai turunan dari fungsi loss dan aktivasi lalu mengalikannya. Hasil delta akan disimpan pada variabel `delta1` untuk digunakan dalam perhitungan gradien hidden layer nantinya.

4. Setelah mendapat nilai delta pada *output layer*, proses perhitungan gradien dilanjutkan pada layer selanjutnya dengan menggunakan fungsi `hiddenLayer()` yang menerima input nilai output, bobot, nilai x, delta, dan fungsi aktivasi yang digunakan pada layer tersebut.

5. Delta pada hidden layer dihitung dengan *chain rule* berikut.

$$\frac{\partial \hat{E}_d}{\partial net_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

Sehingga untuk menghitung nilai gradien pada hidden layer dilakukan dengan melakukan dot product antara bobot dan delta pada layer setelahnya dikali dengan hasil turunan fungsi aktivasi pada layer saat ini. Hal ini dilakukan pada fungsi `hiddenLayer()` yang didalamnya akan menghitung dot product dari bobot dan delta layer setelahnya dan turunan aktivasi lalu mengalikannya. Hasil delta akan disimpan pada variabel `delta1` untuk digunakan dalam perhitungan gradien hidden layer sebelumnya.

6. Selanjutnya setelah mendapat nilai delta pada hidden layer proses perhitungan gradien bisa dilakukan dengan menggunakan fungsi `updateGradien()`, sebagai berikut.

```
def updateGradien(self, layer_idx: int, delta: np.ndarray, input: np.ndarray):
    grad = self.learning_rate * (np.array(input) @ np.array(delta).T)
    self.delta_gradien[layer_idx] = grad
```

7. Langkah 5-6 terus dilakukan hingga telah mencapai layer pertama atau sudah mendapat nilai gradien untuk setiap bobot pada setiap layernya.
8. Setelah fungsi `backPropagation()` selesai, dilanjutkan dengan pemanggilan fungsi `updateWeight()` yang berisikan pengurangan weight pada tiap layer dengan daftar nilai gradien hasil perhitungan sebelumnya, sebagai berikut.

```
def updateWeight(self):
    for idx, layer in enumerate(self.layers):
        print(f"Layer {idx} before: {layer.weight}")
        layer.weight -= self.delta_gradien[idx]
        print(f"Layer {idx} after: {layer.weight}")
```

2.3.2 Implementasi Kode BackPropagation

```
def backPropagation(self, inputs, netsLayer, target):
    # print(f"Total layers : {len(self.layers)}")
    i = len(self.layers) - 1
    delta1: np.ndarray = None

    while i >= -1:
        nets = np.array(netsLayer[i]).T
        if i == len(self.layers) - 1: # Output layer
            if self.layers[i].activ_func == activations.softmax:
                delta1 = outputLayer(inputs[i+1], nets,
target,"softmax",self.loss_func)
            elif self.layers[i].activ_func == activations.tanh:
                delta1 = outputLayer(inputs[i+1], nets,
target,"tanh",self.loss_func)
            elif self.layers[i].activ_func == activations.sigmoid:
                delta1 = outputLayer(inputs[i+1], nets,
target,"sigmoid",self.loss_func)
            elif self.layers[i].activ_func == activations.relu:
                delta1 = outputLayer(inputs[i+1], nets,
target,"relu",self.loss_func)
            else: # linear
                delta1 = outputLayer(inputs[i+1], nets,
target,"linear",self.loss_func)

        else: # Hidden layer
            if self.layers[i].activ_func == activations.softmax:
                delta2 = hiddenLayer(self.layers[i + 1].weight[1:],
inputs[i+1][1:], delta1, "softmax")
            elif self.layers[i].activ_func == activations.tanh:
                delta2 = hiddenLayer(self.layers[i + 1].weight[1:],
```

```
inputs[i+1][1:], delta1, "tanh")
        elif self.layers[i].activ_func == activations.sigmoid:
            delta2 = hiddenLayer(self.layers[i + 1].weight[1:],
inputs[i+1][1:], delta1, "sigmoid")
        elif self.layers[i].activ_func == activations.relu:
            delta2 = hiddenLayer(self.layers[i + 1].weight[1:],
inputs[i+1][1:], delta1, "relu")
        else: # linear
            delta2 = hiddenLayer(self.layers[i + 1].weight[1:],
inputs[i+1][1:], delta1, "linear")

        self.updateGradien(i + 1, delta1,inputs[i+1])

        delta1 = delta2
    i -= 1
```

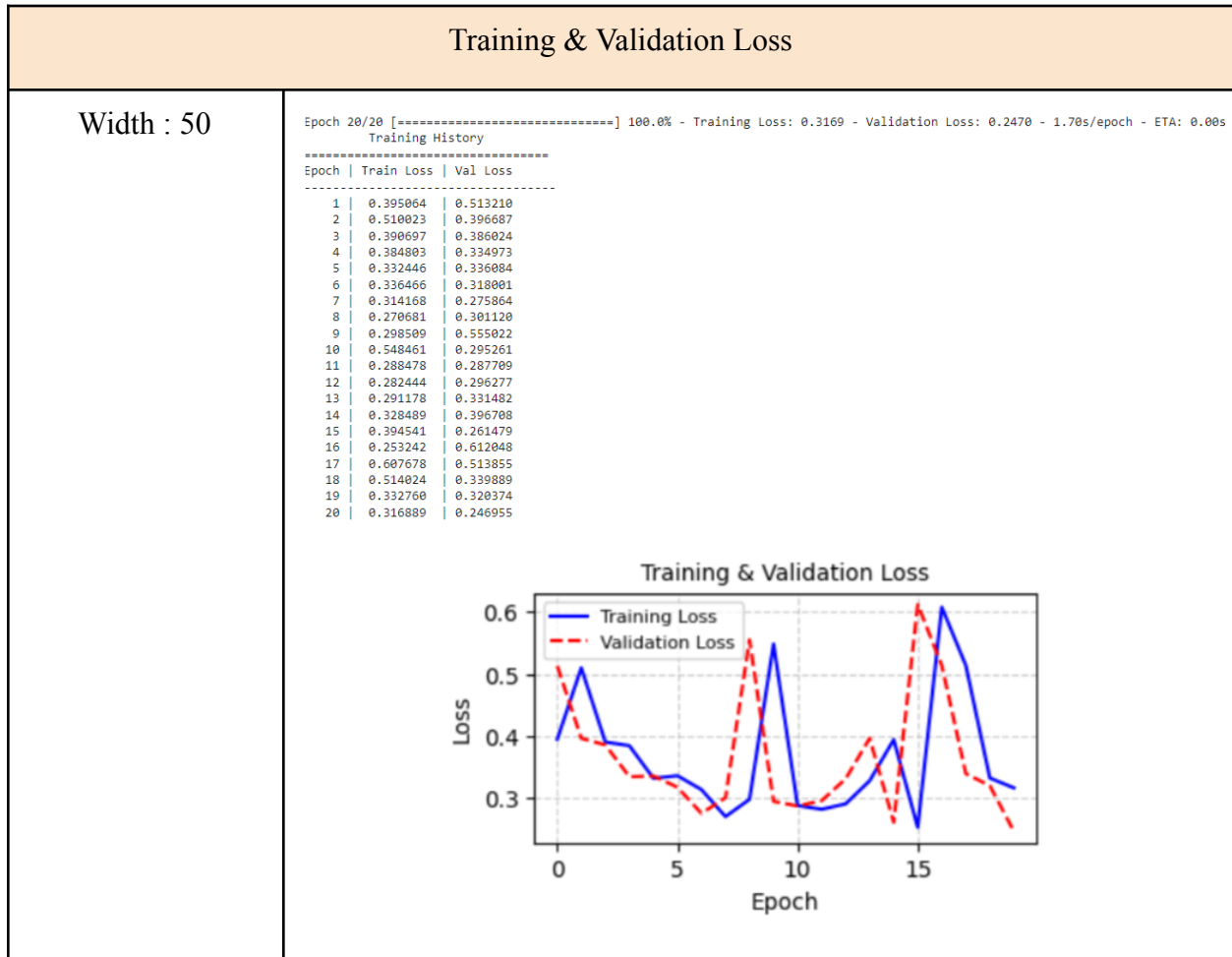
BAB III

PENGUJIAN DAN ANALISIS

3.1 Pengaruh Depth dan Width

3.1.1 Pengaruh Width

Berikut merupakan hasil pengujian terhadap width dengan peninjauan pengaruhnya terhadap training & validation loss, distribusi bobot per layer, distribusi gradien per layer.



Width : 100

Width 100 Result

Epoch 20/20 [=====] 100.0% - Training Loss: 0.5905 - Validation Loss: 0.8100 - 2.10s/epoch - ETA: 0.00s

Training History

Epoch	Train Loss	Val Loss
1	0.407086	0.578735
2	0.580890	0.480137
3	0.483065	0.476330
4	0.474644	0.397306
5	0.391444	0.482596
6	0.478558	0.337691
7	0.336816	0.413317
8	0.409401	0.309833
9	0.299379	0.230789
10	0.227175	0.314609
11	0.307912	0.320288
12	0.315270	0.398744
13	0.395243	0.389515
14	0.384899	0.591205
15	0.586423	0.287938
16	0.285562	0.253697
17	0.249547	0.244736
18	0.240981	0.287785
19	0.290667	0.587207
20	0.590510	0.810000

Training & Validation Loss

Epoch	Train Loss	Val Loss
1	0.407086	0.578735
2	0.580890	0.480137
3	0.483065	0.476330
4	0.474644	0.397306
5	0.391444	0.482596
6	0.478558	0.337691
7	0.336816	0.413317
8	0.409401	0.309833
9	0.299379	0.230789
10	0.227175	0.314609
11	0.307912	0.320288
12	0.315270	0.398744
13	0.395243	0.389515
14	0.384899	0.591205
15	0.586423	0.287938
16	0.285562	0.253697
17	0.249547	0.244736
18	0.240981	0.287785
19	0.290667	0.587207
20	0.590510	0.810000

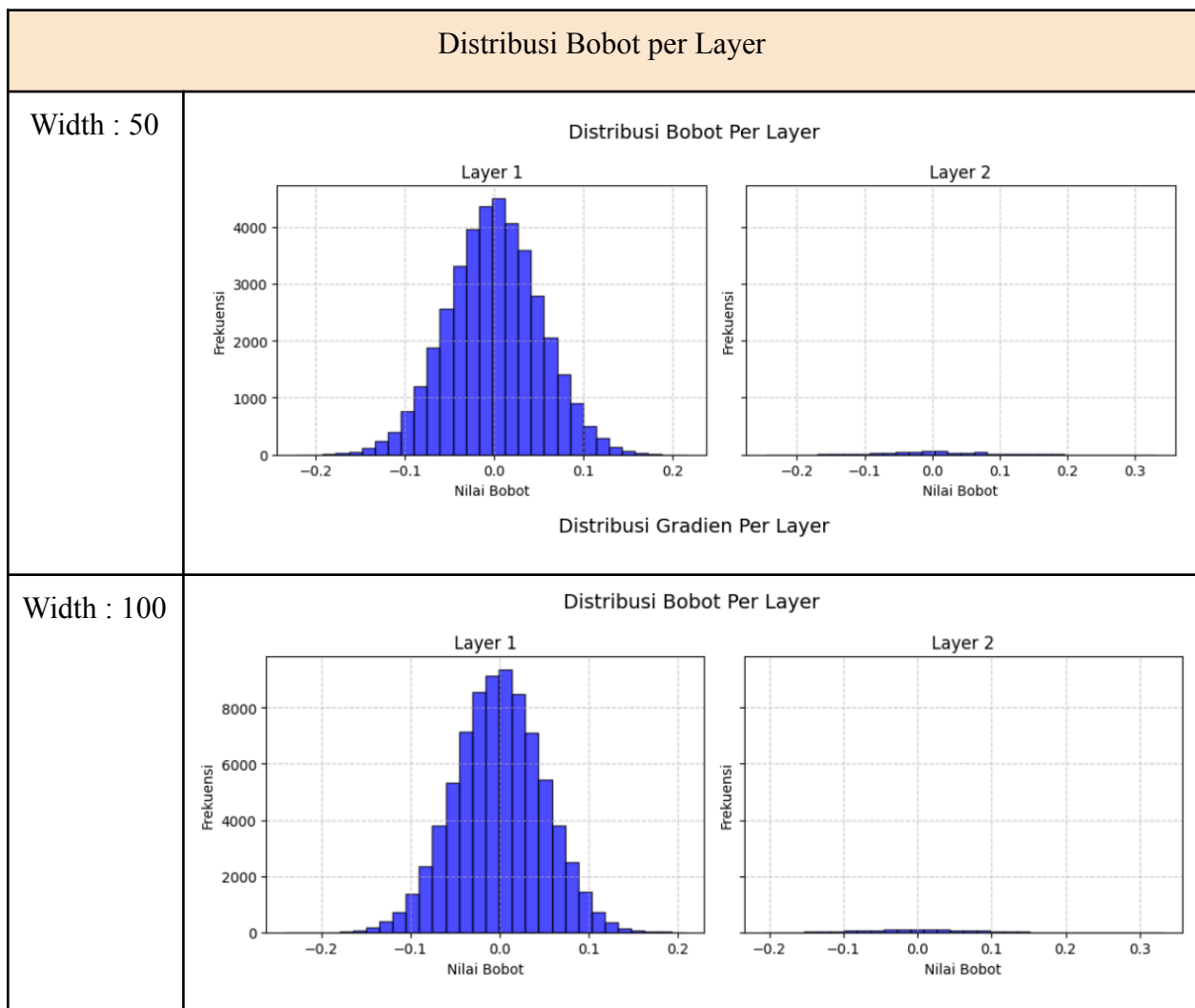
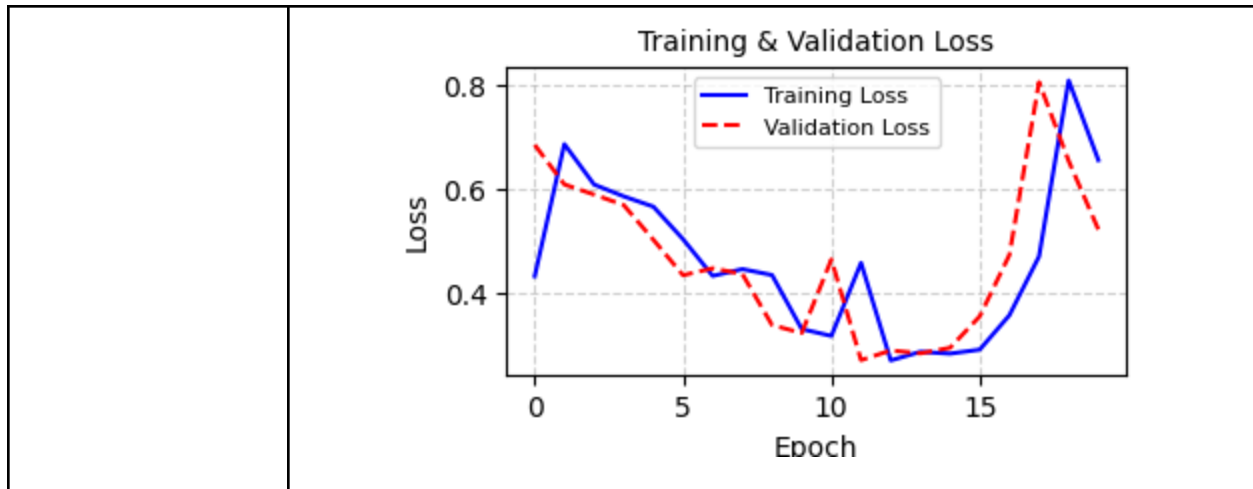
Width : 200

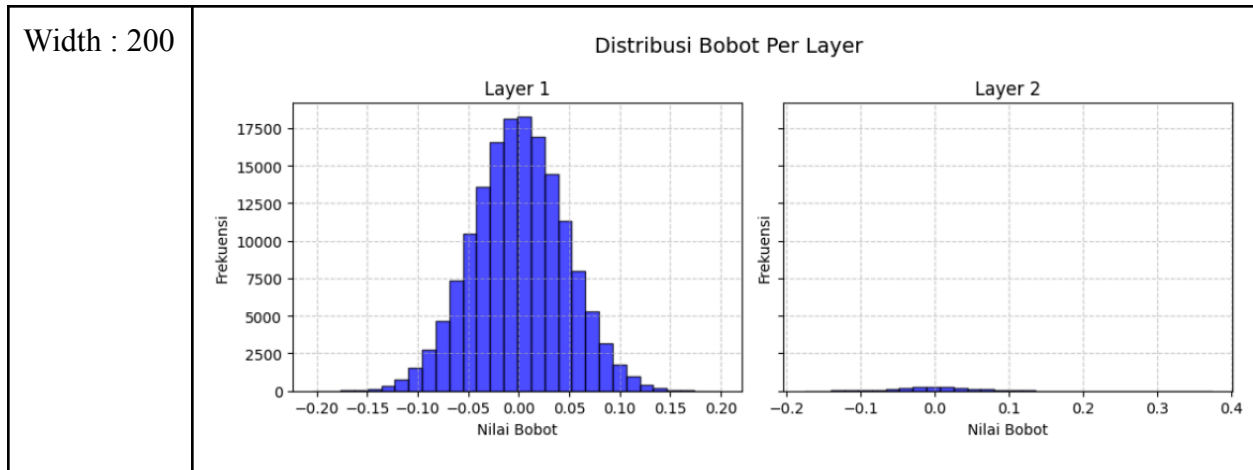
Width 200 Result

Epoch 20/20 [=====] 100.0% - Training Loss: 0.6540 - Validation Loss: 0.5205 - 2.20s/epoch - ETA: 0.00s

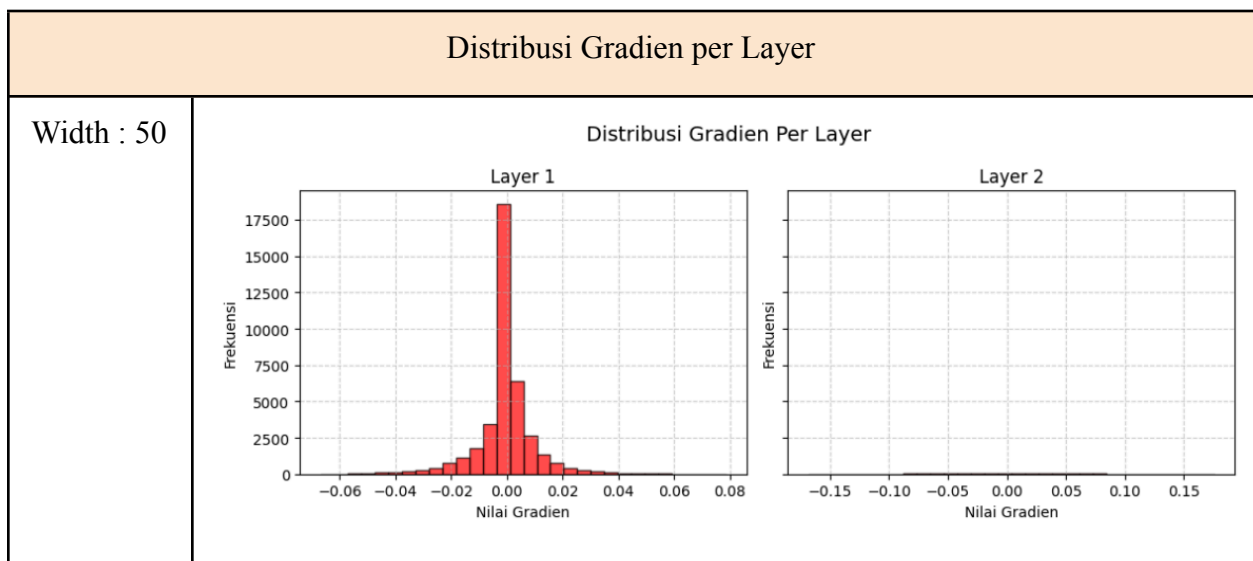
Training History

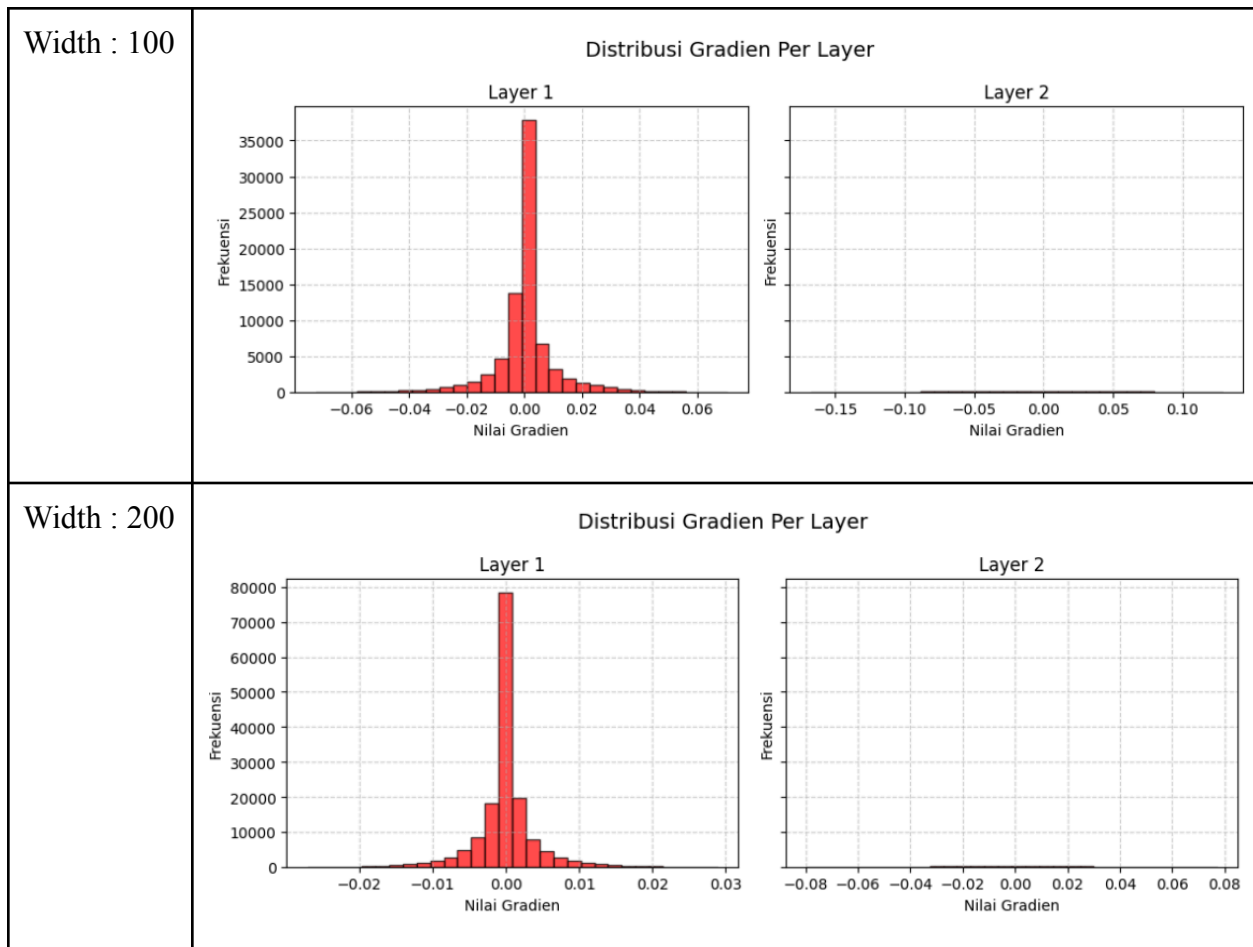
Epoch	Train Loss	Val Loss
1	0.430902	0.683296
2	0.684724	0.607212
3	0.607060	0.587789
4	0.584319	0.567520
5	0.564497	0.500867
6	0.501931	0.432617
7	0.431302	0.445923
8	0.444384	0.434910
9	0.433011	0.336787
10	0.329124	0.321488
11	0.315907	0.462862
12	0.456440	0.269352
13	0.268769	0.287952
14	0.285326	0.282422
15	0.281884	0.293057
16	0.289536	0.353542
17	0.355597	0.471530
18	0.468765	0.803821
19	0.807219	0.652293
20	0.654019	0.520536





Width	50	100	200
Akurasi	41.23%	18.75%	29.97%





Dari hasil pengujian di atas, terlihat bahwa semakin banyak jumlah width-nya (jumlah neuron) maka nilai training loss & validation loss cenderung akan meningkat yang menandakan bahwa model semakin buruk dalam memahami pola pada data. Dengan hasil nilai loss akhir dimulai dari 0.3169, 0.5905 dan 0.6550 untuk percobaan pada width 50, 100, dan 200. Selain itu, width 50 memberikan nilai terbaik, karena ketika width ditingkatkan menjadi 100 dan 200 terlihat bahwa penambahan width memberikan penurunan pada nilai akurasi. Kecenderungan ini membuktikan bahwa terdapat suatu batas efektivitas ketika memperkecil width.

Peninjauan kedua dilakukan terhadap distribusi bobot per layer. Pada width 50, bobot cenderung lebih merata, menunjukkan pembelajaran yang lebih sederhana dan memberikan distribusi bobot yang normal. Sementara itu, pada width 100, distribusi bobot normal, menandakan keseimbangan antara kapasitas model dan stabilitas pembelajaran. Di sisi lain, pada width 200, distribusi bobot semakin tajam dengan variansi besar, yang dapat berisiko menyebabkan exploding gradients, yaitu saat nilai gradien selama backpropagation menjadi

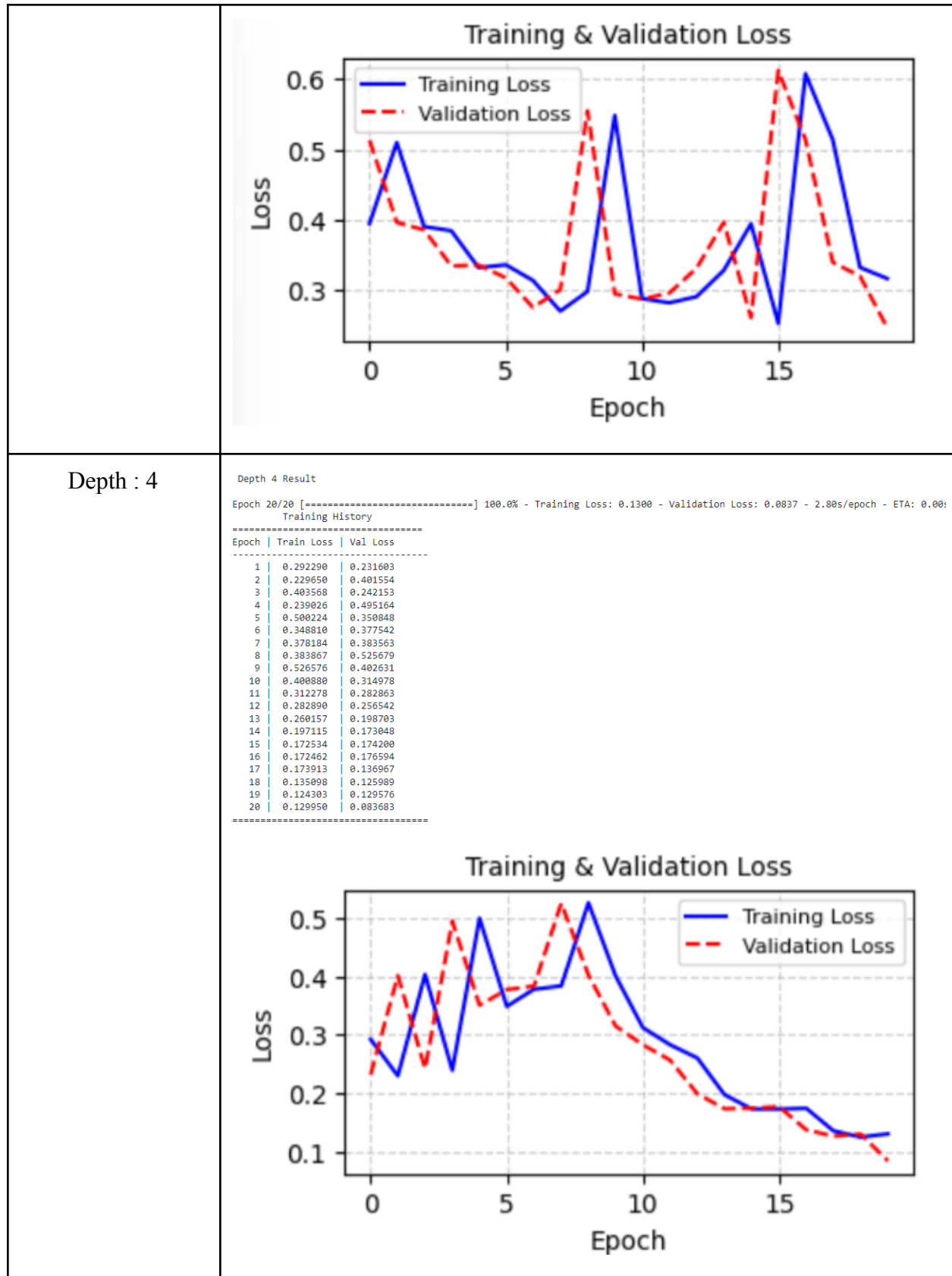
sangat besar yang menyebabkan perubahan bobot yang ekstrem dan ketidakstabilan dalam pelatihan. Sehingga perlu dikendalikan dengan teknik seperti normalisasi bobot atau regulasi yang baik. Oleh karena itu, width 50 memberikan keseimbangan yang baik antara generalisasi dan stabilitas dalam proses pembelajaran.

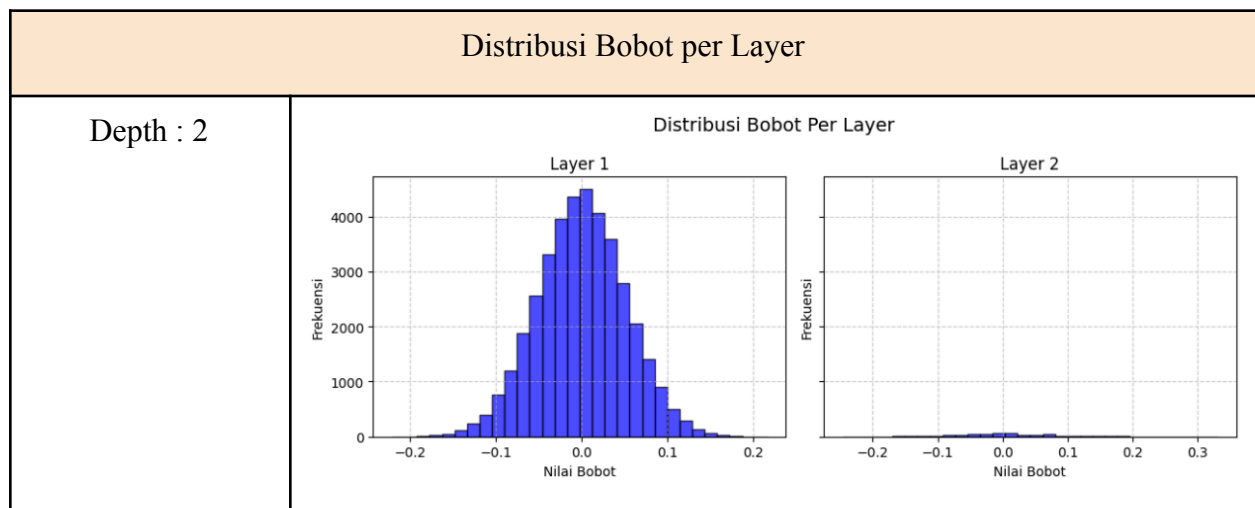
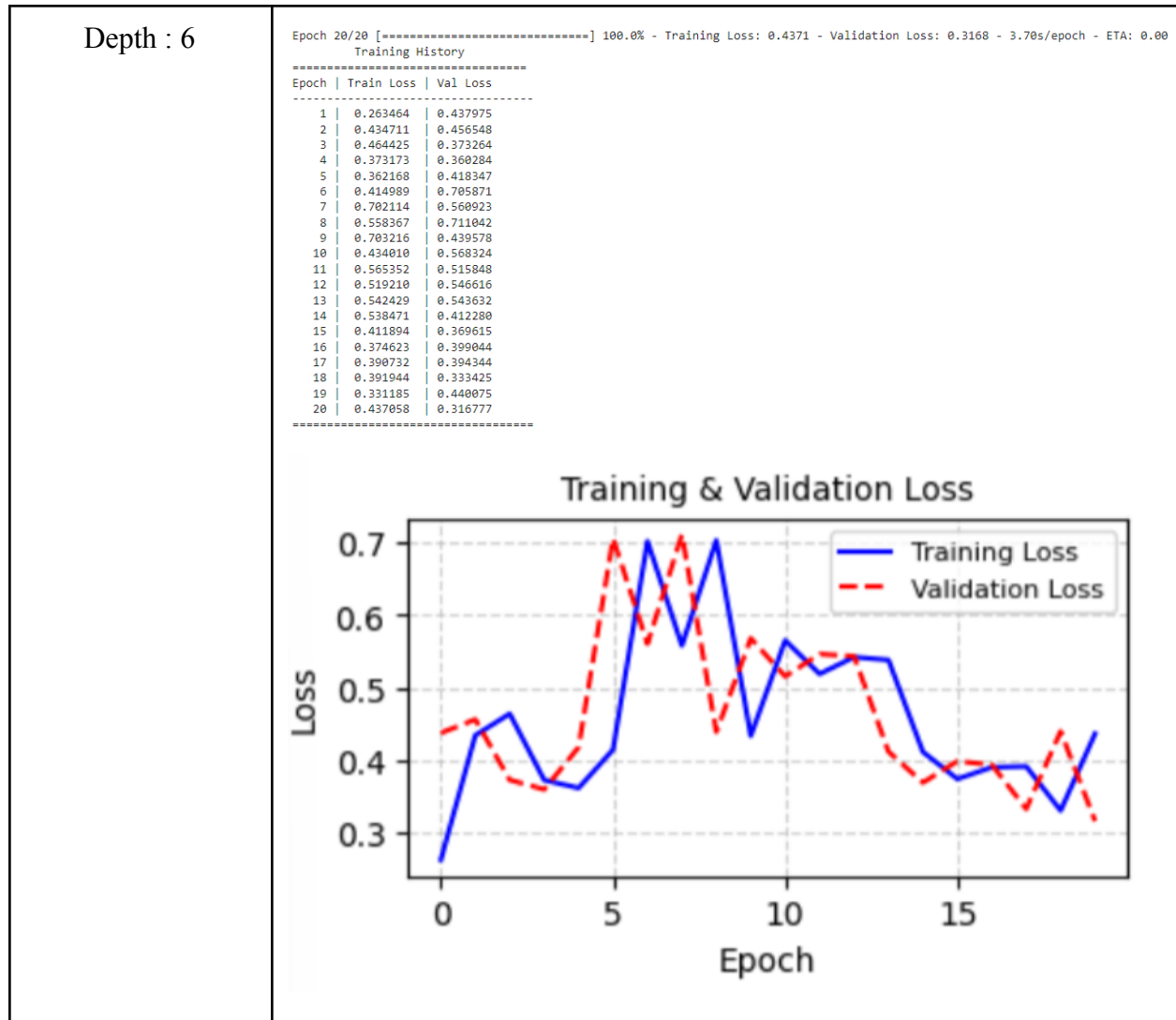
Peninjauan terakhir dilakukan terhadap distribusi gradien per layer. Pada width 50, distribusi gradien menunjukkan nilai yang relatif kecil dan terpusat di sekitar nol, menandakan bahwa pembelajaran berjalan stabil. Pada width 100, distribusi gradien menjadi lebih tajam dengan variansi yang sangat besar, yang dapat menyebabkan exploding gradients dan membuat pembelajaran menjadi tidak stabil. Sementara itu, pada width 200, distribusi gradien juga menunjukkan nilai yang relatif kecil dan terpusat di sekitar nol. Oleh karena itu, width 50 menjadi pilihan yang optimal untuk melakukan training pada model.

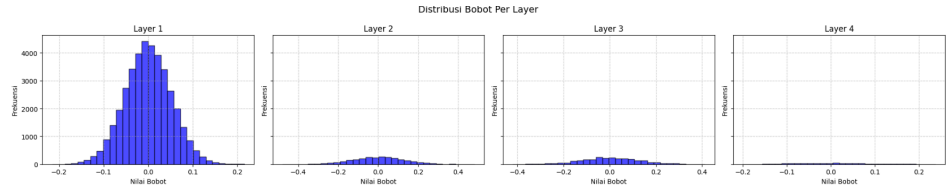
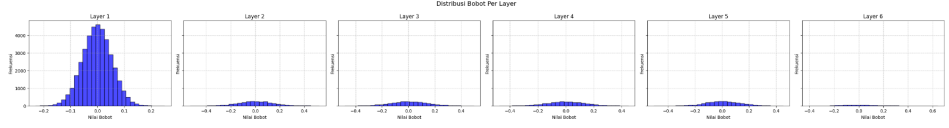
3.1.2 Pengaruh Depth

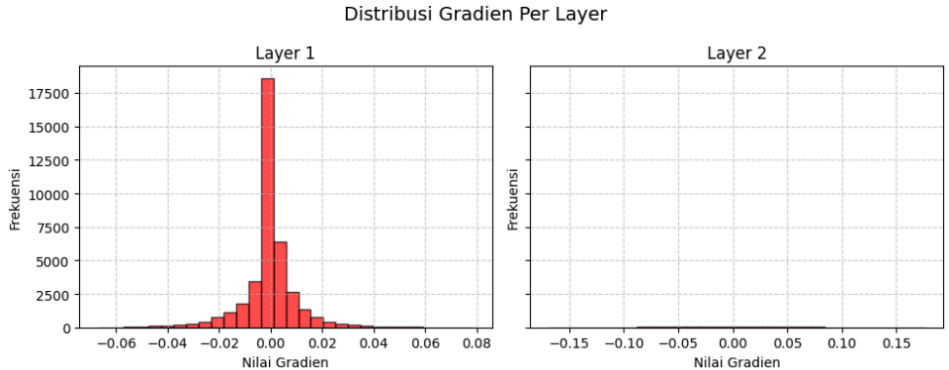
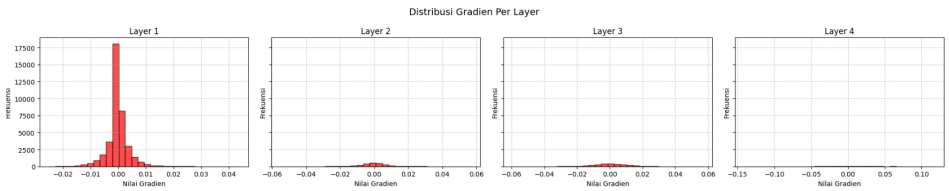
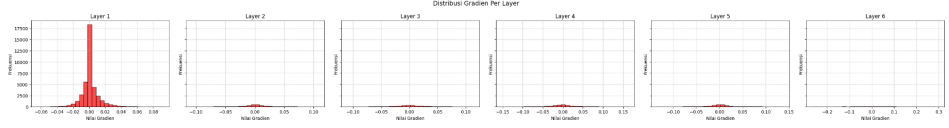
Berikut merupakan hasil pengujian terhadap depth dengan peninjauan pengaruhnya terhadap training & validation loss, distribusi bobot per layer, distribusi gradien per layer.

Training & Validation Loss																																																																
Depth : 2	Depth 2 Result																																																															
	Epoch 20/20 [=====] 100.0% - Training Loss: 0.3169 - Validation Loss: 0.2470 - 1.80s/epoch - ETA: 0.00: Training History ===== <table> <thead> <tr> <th>Epoch</th><th>Train Loss</th><th>Val Loss</th></tr> </thead> <tbody> <tr><td>1</td><td>0.395064</td><td>0.513210</td></tr> <tr><td>2</td><td>0.510023</td><td>0.396687</td></tr> <tr><td>3</td><td>0.390697</td><td>0.386024</td></tr> <tr><td>4</td><td>0.384803</td><td>0.334973</td></tr> <tr><td>5</td><td>0.332446</td><td>0.336084</td></tr> <tr><td>6</td><td>0.336466</td><td>0.318001</td></tr> <tr><td>7</td><td>0.314168</td><td>0.275864</td></tr> <tr><td>8</td><td>0.270681</td><td>0.301120</td></tr> <tr><td>9</td><td>0.298509</td><td>0.555022</td></tr> <tr><td>10</td><td>0.548461</td><td>0.295261</td></tr> <tr><td>11</td><td>0.288478</td><td>0.287709</td></tr> <tr><td>12</td><td>0.282444</td><td>0.296277</td></tr> <tr><td>13</td><td>0.291178</td><td>0.331482</td></tr> <tr><td>14</td><td>0.328489</td><td>0.396708</td></tr> <tr><td>15</td><td>0.394541</td><td>0.261479</td></tr> <tr><td>16</td><td>0.253242</td><td>0.612048</td></tr> <tr><td>17</td><td>0.607678</td><td>0.513855</td></tr> <tr><td>18</td><td>0.514024</td><td>0.339889</td></tr> <tr><td>19</td><td>0.332760</td><td>0.320374</td></tr> <tr><td>20</td><td>0.316889</td><td>0.246955</td></tr> </tbody> </table> =====		Epoch	Train Loss	Val Loss	1	0.395064	0.513210	2	0.510023	0.396687	3	0.390697	0.386024	4	0.384803	0.334973	5	0.332446	0.336084	6	0.336466	0.318001	7	0.314168	0.275864	8	0.270681	0.301120	9	0.298509	0.555022	10	0.548461	0.295261	11	0.288478	0.287709	12	0.282444	0.296277	13	0.291178	0.331482	14	0.328489	0.396708	15	0.394541	0.261479	16	0.253242	0.612048	17	0.607678	0.513855	18	0.514024	0.339889	19	0.332760	0.320374	20	0.316889
Epoch	Train Loss	Val Loss																																																														
1	0.395064	0.513210																																																														
2	0.510023	0.396687																																																														
3	0.390697	0.386024																																																														
4	0.384803	0.334973																																																														
5	0.332446	0.336084																																																														
6	0.336466	0.318001																																																														
7	0.314168	0.275864																																																														
8	0.270681	0.301120																																																														
9	0.298509	0.555022																																																														
10	0.548461	0.295261																																																														
11	0.288478	0.287709																																																														
12	0.282444	0.296277																																																														
13	0.291178	0.331482																																																														
14	0.328489	0.396708																																																														
15	0.394541	0.261479																																																														
16	0.253242	0.612048																																																														
17	0.607678	0.513855																																																														
18	0.514024	0.339889																																																														
19	0.332760	0.320374																																																														
20	0.316889	0.246955																																																														





Depth : 4			
Depth : 6			
Width	2	4	6
Akurasi	41.23%	59.29%	15.97%

Distribusi Gradien per Layer	
Depth : 2	
Depth : 4	
Depth : 6	

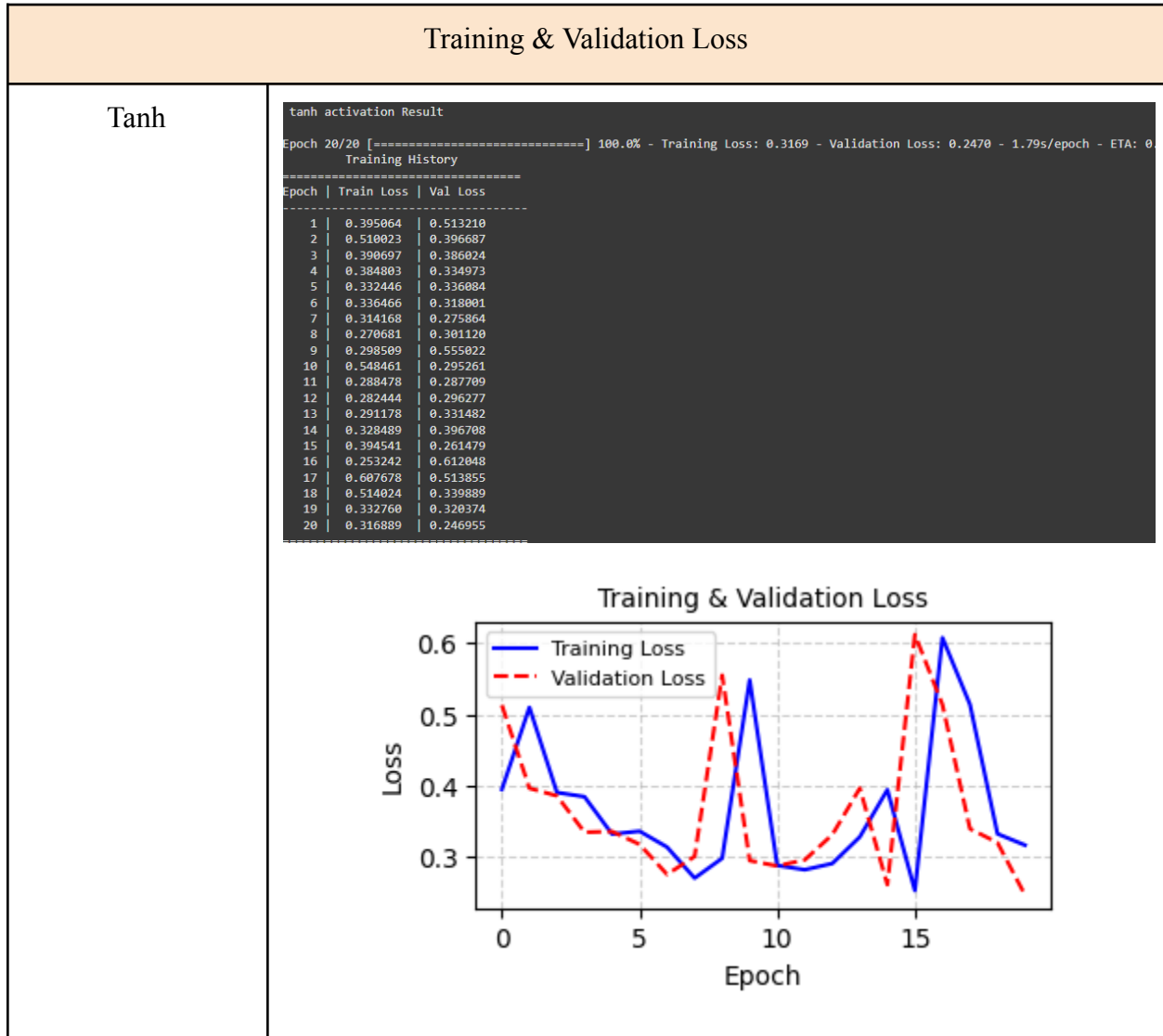
Dari hasil pengujian, terlihat bahwa semakin dalam *depth* nya nilai training loss dan validation loss nya semakin meningkat. Namun, *depth* 4 memberikan nilai loss yang paling optimum yaitu 0.08 dengan nilai akurasi yang diberikan adalah 59.29%. Semua *depth* memberikan nilai validation loss yang lebih rendah dari training loss yang mengartikan bahwa model dengan *depth* tersebut tidak overfitting.

Selanjutnya peninjauan kedua dilakukan terhadap distribusi bobot per layer-nya. Pada *depth* 2, distribusi bobot pada layer pertama tampak merata, menunjukkan bahwa pembelajaran terjadi dengan cukup baik, tetapi layer kedua memiliki bobot yang hampir tidak bervariasi cenderung berada pada sekitar nilai nol. Pada *depth* 4, bobot mulai terdistribusi dengan pola lebih tajam di sekitar nol, dan beberapa layer memiliki bobot dengan dengan nilai berskala besar. Pada *depth* 6, distribusi bobot semakin ekstrim dan nilai distribusi bobot yang meningkat, yang berpotensi menyebabkan *exploding gradients*. Secara keseluruhan, model dengan *depth* yang lebih besar cenderung mengalami ketidakstabilan dalam pembelajaran jika tidak dikontrol dengan baik menggunakan normalisasi atau regularisasi.

Terakhir peninjauan dilakukan terhadap distribusi gradien per layer-nya. Distribusi gradien menunjukkan bahwa semakin dalam arsitektur jaringan, semakin besar masalah *vanishing gradient* yang terjadi. Pada *depth* 2, gradien terkonsentrasi di layer pertama, sementara layer kedua hampir tidak belajar. Pada *depth* 4 dan 6, hanya layer awal yang menerima gradien signifikan, sedangkan layer-layer selanjutnya mengalami penurunan tajam nilai gradien, menandakan pembelajaran yang tidak efektif di lapisan terdalam. Selain itu, skala yang semakin besar, menandakan kemungkinan *exploding gradients*. Secara keseluruhan, semakin dalam model maka akan semakin rentan terhadap masalah *vanishing* atau *exploding gradients*. Masalah *vanishing* dapat menyebabkan model sulit untuk mengupdate bobot secara efektif karena tidak ada perubahan selama pembelajaran yang berdampak pada performansi yang semakin menurun. Selain itu, untuk masalah *exploding gradient* menyebabkan pembaruan bobot sulit dikendalikan. Oleh karena itu, model ini masih memerlukan teknik lebih lanjut seperti normalisasi, regulasi, atau optimisasi yang lebih baik untuk menjaga stabilitas pembelajaran.

3.2 Pengaruh Fungsi Aktivasi

Berikut merupakan hasil pengujian terhadap fungsi aktivasi dengan peninjauan pengaruhnya terhadap training & validation loss, distribusi bobot per layer, distribusi gradien per layer.



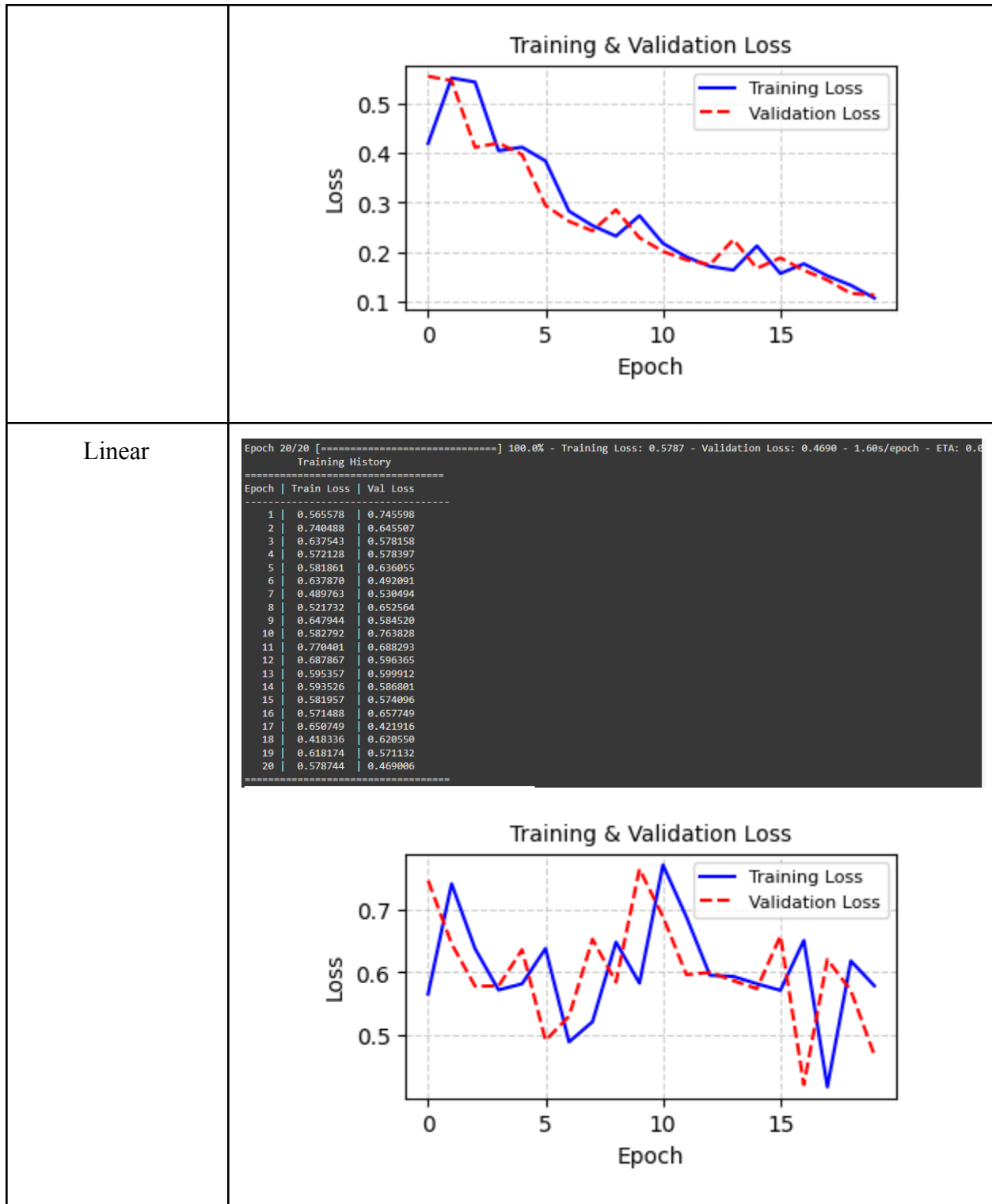
Sigmoid

```
Epoch 20/20 [=====] 100.0% - Training Loss: 0.5432 - Validation Loss: 0.4181 - 1.60s/epoch - ETA: 0.6s
Training History
-----
Epoch | Train Loss | Val Loss
-----
1 | 0.291421 | 1.029281
2 | 1.023088 | 0.958172
3 | 0.951565 | 0.947750
4 | 0.945704 | 0.859608
5 | 0.860628 | 0.793784
6 | 0.791922 | 0.697048
7 | 0.695060 | 0.814231
8 | 0.810883 | 0.742141
9 | 0.742029 | 0.832297
10 | 0.829511 | 0.697361
11 | 0.695637 | 0.657493
12 | 0.655019 | 0.716314
13 | 0.715832 | 0.683855
14 | 0.682732 | 0.719207
15 | 0.720462 | 0.634667
16 | 0.635264 | 0.628234
17 | 0.626159 | 0.546602
18 | 0.543960 | 0.615588
19 | 0.615417 | 0.543279
20 | 0.543249 | 0.418126
-----
```



ReLU

```
Epoch 20/20 [=====] 100.0% - Training Loss: 0.1085 - Validation Loss: 0.1146 - 1.50s/epoch - ETA: 0.6s
Training History
-----
Epoch | Train Loss | Val Loss
-----
1 | 0.418706 | 0.553434
2 | 0.550169 | 0.545016
3 | 0.542005 | 0.410753
4 | 0.403984 | 0.419646
5 | 0.411318 | 0.396224
6 | 0.383545 | 0.294141
7 | 0.282781 | 0.262323
8 | 0.253784 | 0.243227
9 | 0.232677 | 0.285663
10 | 0.273796 | 0.229359
11 | 0.218131 | 0.201876
12 | 0.191046 | 0.185091
13 | 0.171893 | 0.175395
14 | 0.164507 | 0.225823
15 | 0.213101 | 0.168015
16 | 0.157590 | 0.189069
17 | 0.177186 | 0.163920
18 | 0.153009 | 0.144631
19 | 0.134037 | 0.117531
20 | 0.108475 | 0.114596
-----
```



elu

Epoch 20/20 [=====] 100.0% - Training Loss: 0.4686 - Validation Loss: 0.3756 - 1.80s/epoch - ETA: 0.6

Training History

Epoch	Train Loss	Val Loss
1	0.483019	0.644125
2	0.641855	0.557178
3	0.550319	0.472023
4	0.467451	0.516658
5	0.513886	0.530688
6	0.528863	0.371128
7	0.364979	0.379094
8	0.375103	0.477592
9	0.472931	0.484651
10	0.479124	0.496236
11	0.486225	0.584835
12	0.582887	0.445855
13	0.440358	0.363351
14	0.354616	0.371065
15	0.360156	0.320149
16	0.315522	0.424054
17	0.419793	0.362640
18	0.355548	0.495623
19	0.491837	0.472079
20	0.468576	0.375592

Training & Validation Loss

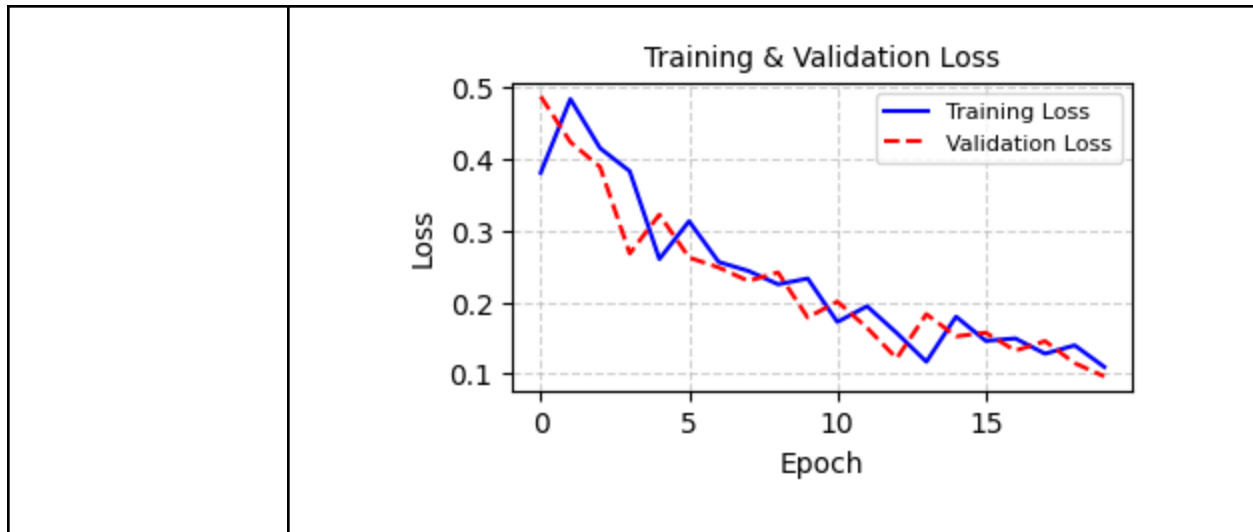
Epoch	Training Loss	Validation Loss
1	0.483019	0.644125
2	0.641855	0.557178
3	0.550319	0.472023
4	0.467451	0.516658
5	0.513886	0.530688
6	0.528863	0.371128
7	0.364979	0.379094
8	0.375103	0.477592
9	0.472931	0.484651
10	0.479124	0.496236
11	0.486225	0.584835
12	0.582887	0.445855
13	0.440358	0.363351
14	0.354616	0.371065
15	0.360156	0.320149
16	0.315522	0.424054
17	0.419793	0.362640
18	0.355548	0.495623
19	0.491837	0.472079
20	0.468576	0.375592

swish

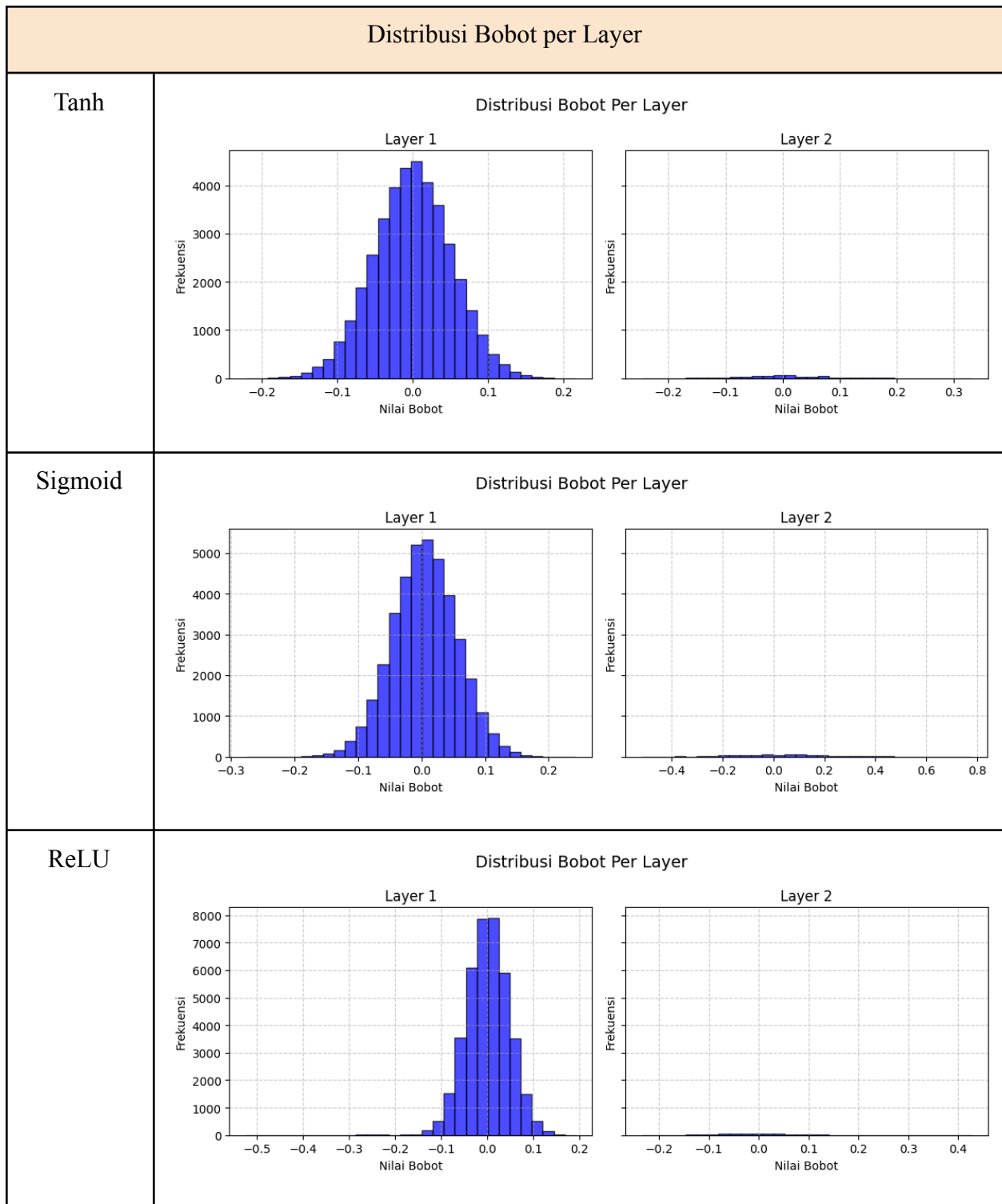
Epoch 20/20 [=====] 100.0% - Training Loss: 0.1095 - Validation Loss: 0.0963 - 1.60s/epoch - ETA: 0.6

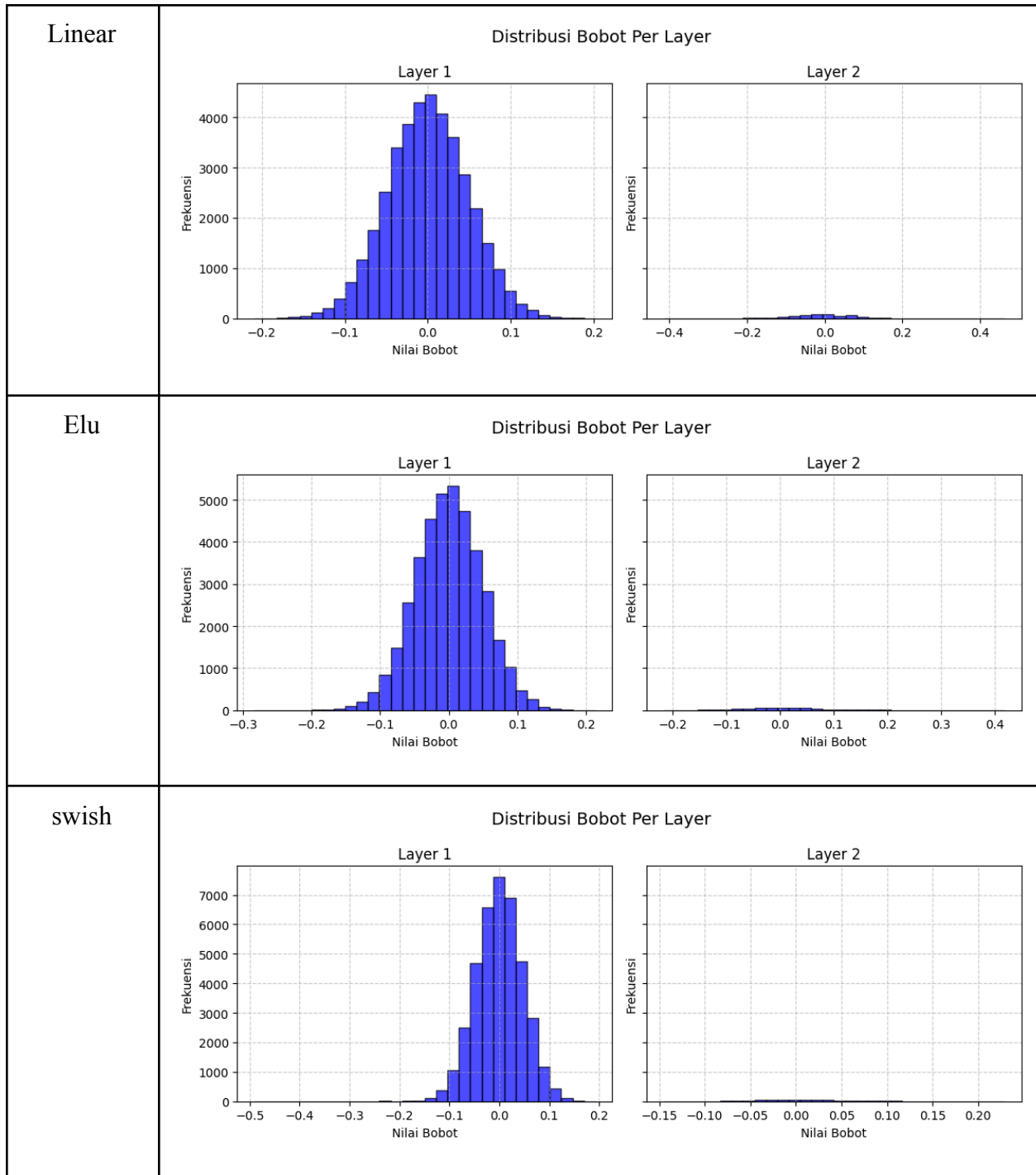
Training History

Epoch	Train Loss	Val Loss
1	0.380719	0.487166
2	0.483428	0.423460
3	0.414932	0.388756
4	0.382799	0.267915
5	0.260181	0.322138
6	0.312983	0.262267
7	0.256057	0.248355
8	0.243596	0.229517
9	0.224880	0.241434
10	0.233000	0.178460
11	0.172276	0.200823
12	0.194289	0.164213
13	0.157040	0.121546
14	0.116716	0.183221
15	0.179758	0.152263
16	0.146026	0.157269
17	0.149294	0.132206
18	0.128019	0.145555
19	0.139693	0.114886
20	0.109485	0.096268



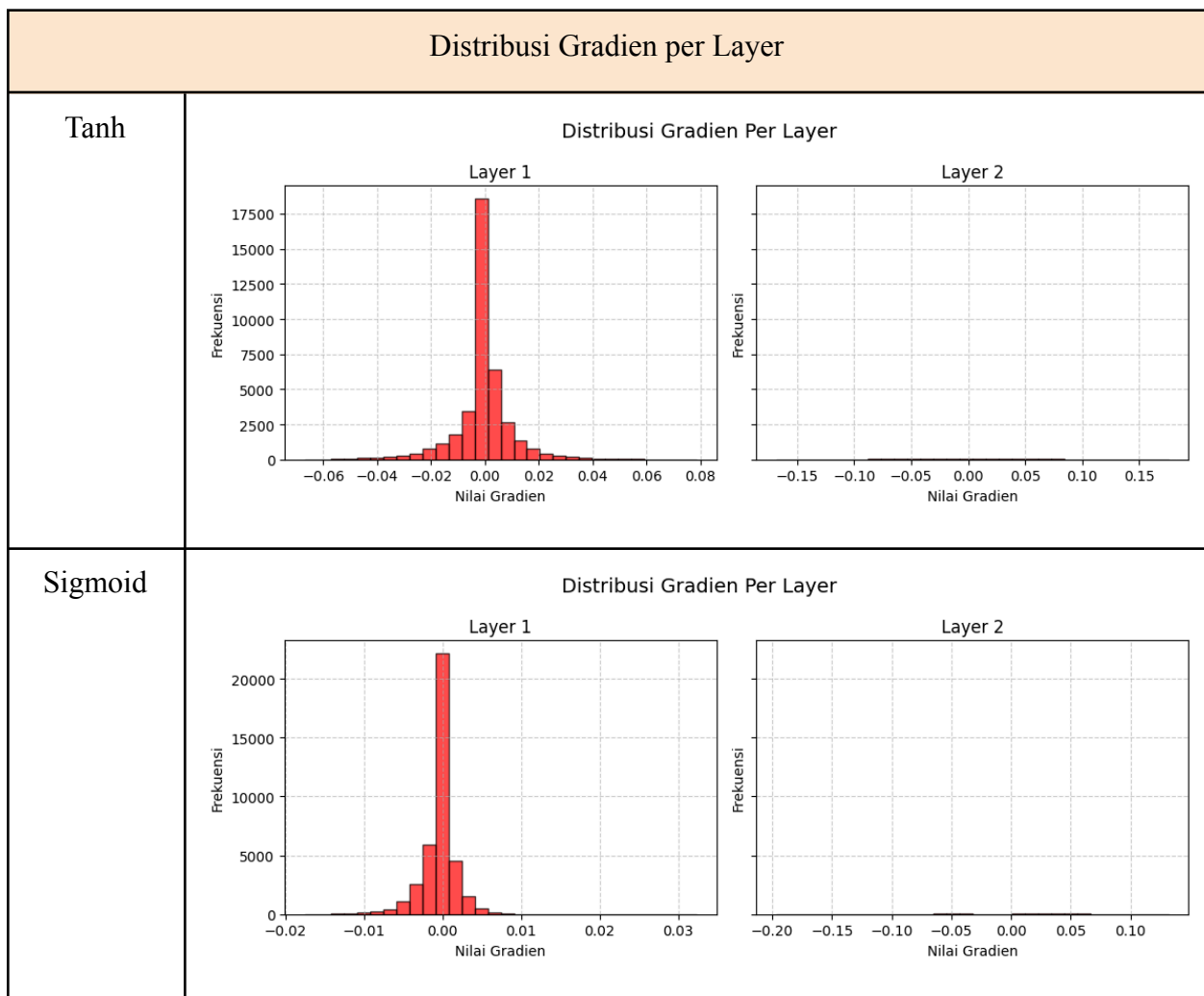
Fungsi aktivasi tanh memberikan nilai training loss yang meningkat dan menurun akan tetapi nilai loss semakin menurun setiap terjadi penurunan pada epoch yang lebih besar. Selanjutnya, fungsi aktivasi sigmoid memberikan nilai training loss yang meningkat tajam pada awal epoch dan terus menurun pada epoch selanjutnya. Hal ini mengindikasikan model ini mengalami kesulitan dalam konvergensi kemungkinan karena terjadi *vanishing gradient* pada lapisan yang lebih dalam. Selanjutnya fungsi aktivasi ReLU memberikan nilai training loss turun yang mengindikasikan model tidak overfit. Pada fungsi aktivasi Linear memberikan nilai training loss yang naik turun secara tidak beraturan. Hal ini juga terjadi pada fungsi aktivasi elu yang memberikan hasil loss yang turun naik secara tidak beraturan. Terakhir, pada fungsi aktivasi swish pada grafik loss terlihat penurunan nilai loss pada epoch yang lebih tinggi walaupun terjadi kenaikan loss pada beberapa epoch, kenaikannya tidak tajam. Sehingga bisa disimpulkan ReLU dan Swish memberikan model yang optimal karena memberikan model yang tidak overfit dan memberikan nilai validation loss paling kecil.

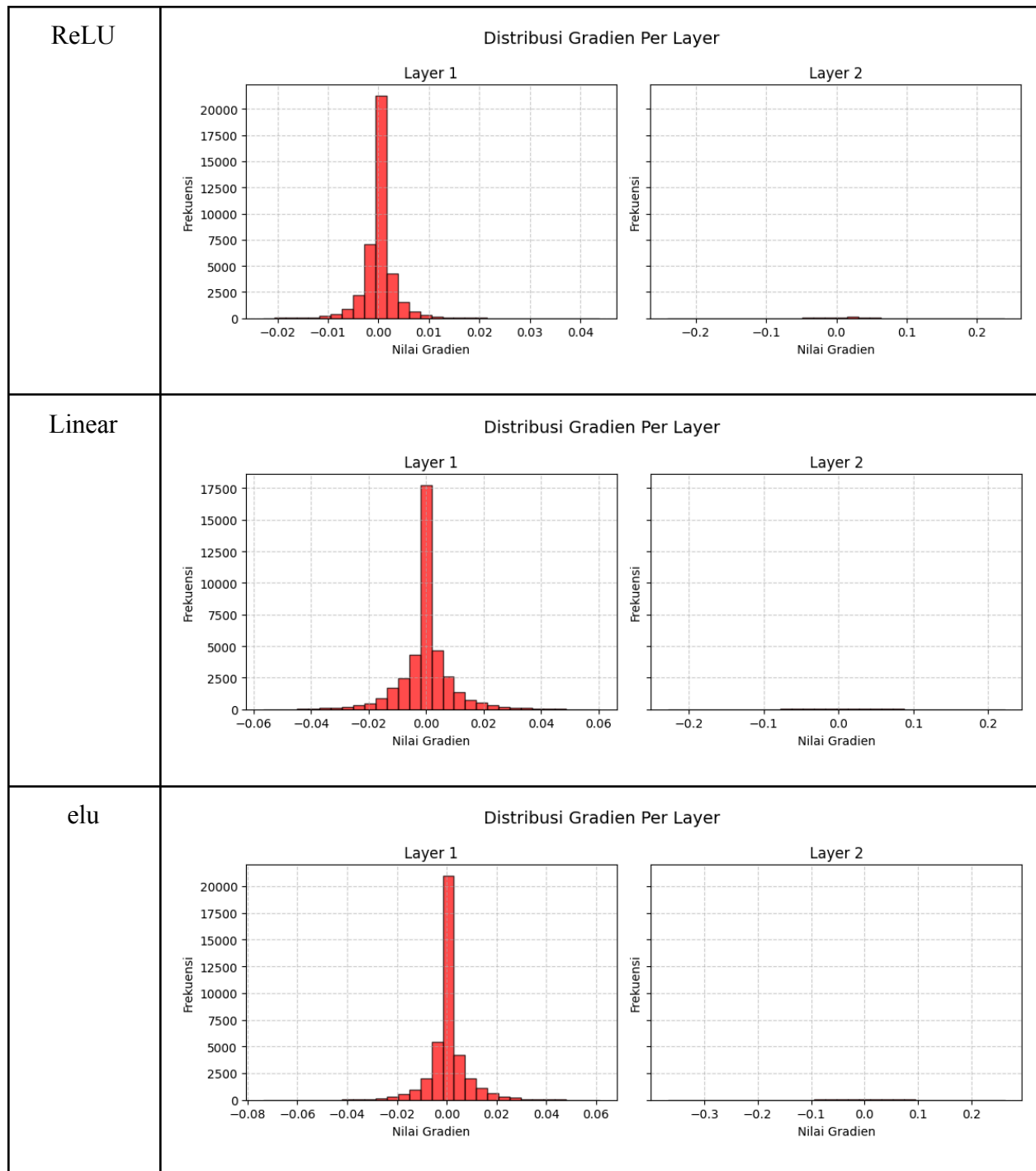


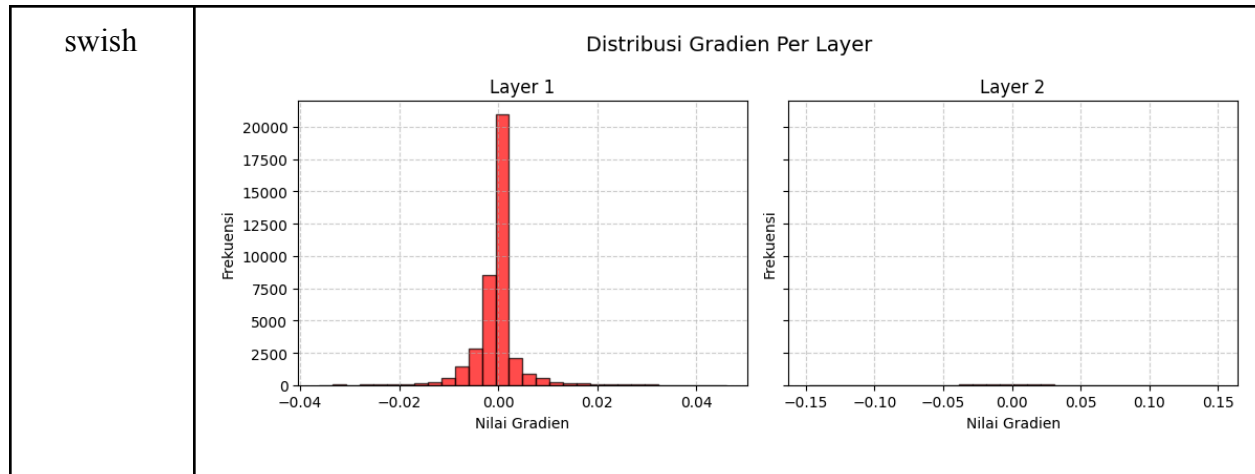


Activations	tanh	sigmoid	relu	linear	elu	swish
Akurasi	41.23%	42.56%	29.65%	11.39%	34.86%	64.59%

Hasil distribusi bobot menunjukkan bahwa fungsi aktivasi tanh mengalami distribusi bobot yang cenderung stabil dari layer pertama dan layer kedua. Sedangkan pada Sigmoid dan linear mengalami vanishing gradient, di mana nilai bobot mendekati nol seiring bertambahnya kedalaman layer dengan *range* yang semakin besar, menyebabkan sulit untuk melakukan pembaruan bobot dan training model menjadi lambat. Pada, ReLU dan elu grafik cenderung skewed ke kiri pada layer ke satu dan skewed ke kanan pada layer selanjutnya. Sementara itu, swish cenderung mengalami *exploding gradient*, dengan distribusi bobot yang lebih luas di layer lebih dalam. Oleh karena itu, tanh menjadi model yang paling optimal daripada fungsi aktivasi lainnya karena perubahan bobotnya lebih stabil.



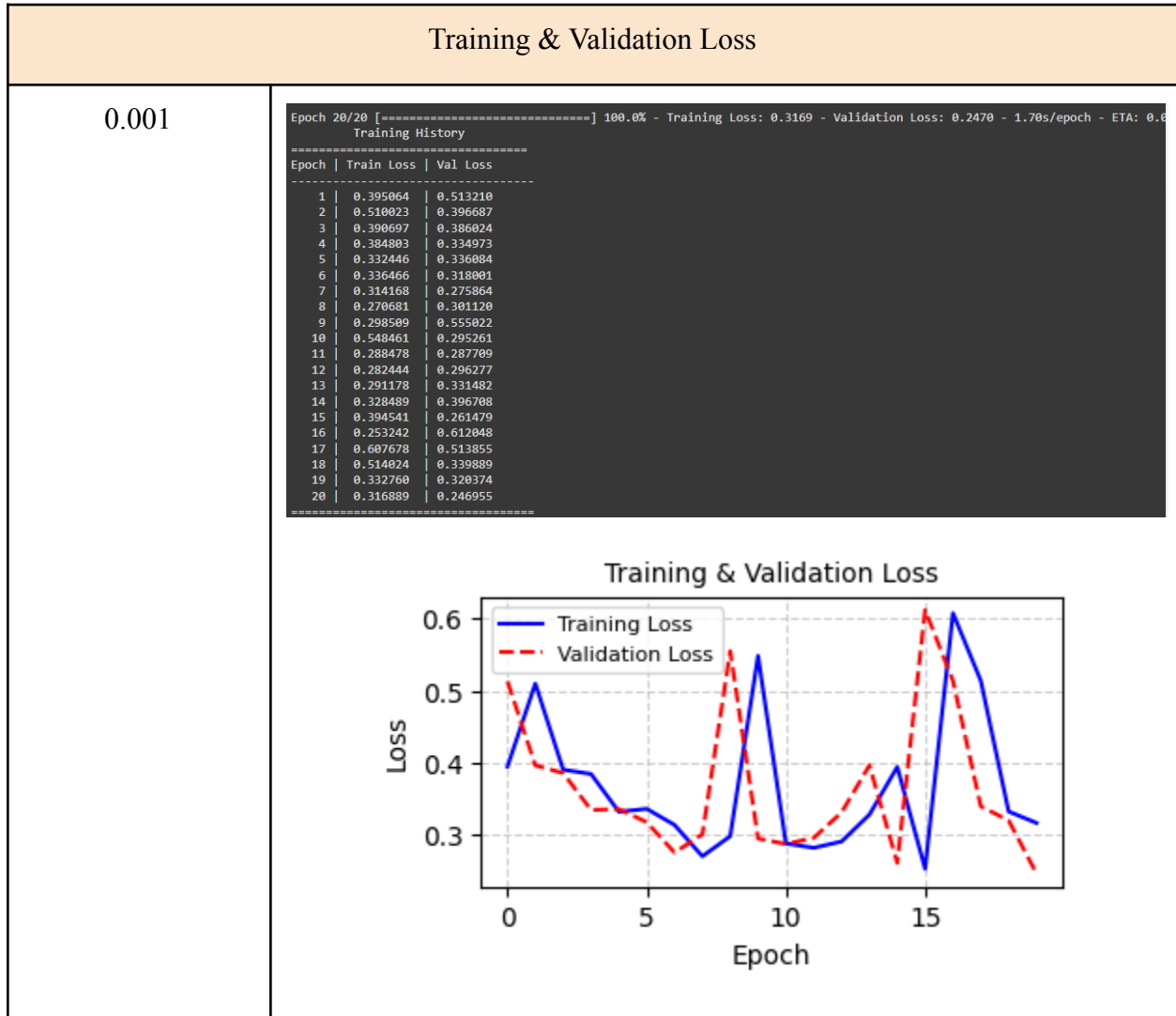




Hasil distribusi gradien menunjukkan fungsi aktivasi Tanh dan Sigmoid mengalami pelemahan dengan gradien yang mendekati nol seiring bertambahnya kedalaman layer, menyebabkan masalah *vanishing gradient* yang menghambat proses pembaruan bobot dan memperlambat *training*. Sementara itu, ReLU, swish dan linear cenderung mengalami peningkatan distribusi nilai gradien di layer lebih dalam, yang juga dapat menyebabkan *exploding gradient*, tetapi tetap lebih terkendali dibandingkan elu, yang menunjukkan distribusi gradien yang lebih dalam. Akibatnya, model dengan aktivasi elu berisiko mengalami ketidakstabilan selama training, dengan perubahan yang menurun dan meningkat pada training loss dan validation loss yang terus meningkat.

3.3 Pengaruh Learning Rate

Berikut merupakan hasil pengujian terhadap learning rate dengan peninjauan pengaruhnya terhadap training & validation loss, distribusi bobot per layer, distribusi gradien per layer.

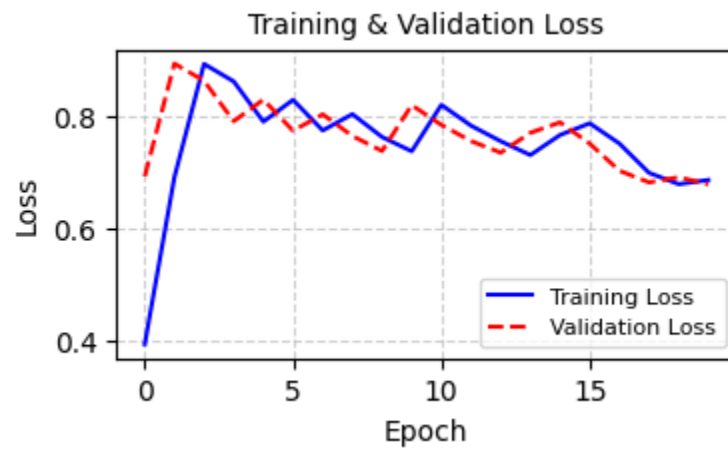


0.01

Epoch 20/20 [=====] 100.0% - Training Loss: 0.6862 - Validation Loss: 0.6786 - 1.69s/epoch - ETA: 0.6

Training History

Epoch	Train Loss	Val Loss
1	0.395064	0.692550
2	0.690989	0.892515
3	0.891871	0.861959
4	0.860824	0.790322
5	0.789595	0.828638
6	0.828194	0.773453
7	0.773970	0.802584
8	0.802881	0.764213
9	0.762368	0.737575
10	0.737058	0.818962
11	0.819354	0.783764
12	0.782454	0.755220
13	0.754903	0.734554
14	0.730362	0.769866
15	0.765922	0.788384
16	0.786652	0.750618
17	0.750879	0.702674
18	0.698841	0.681954
19	0.678791	0.690998
20	0.686167	0.678645

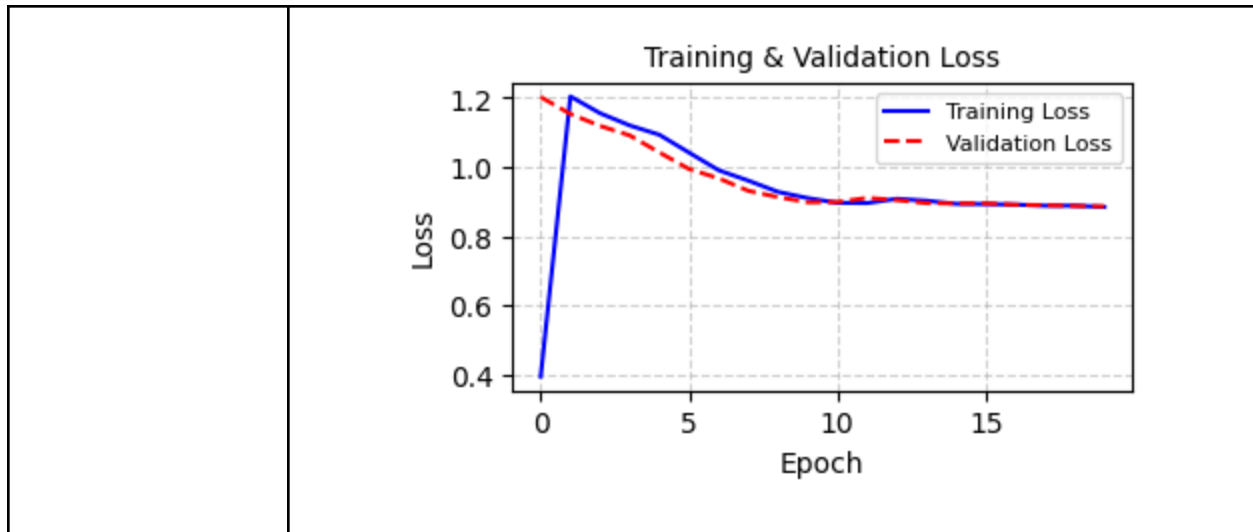


0.1

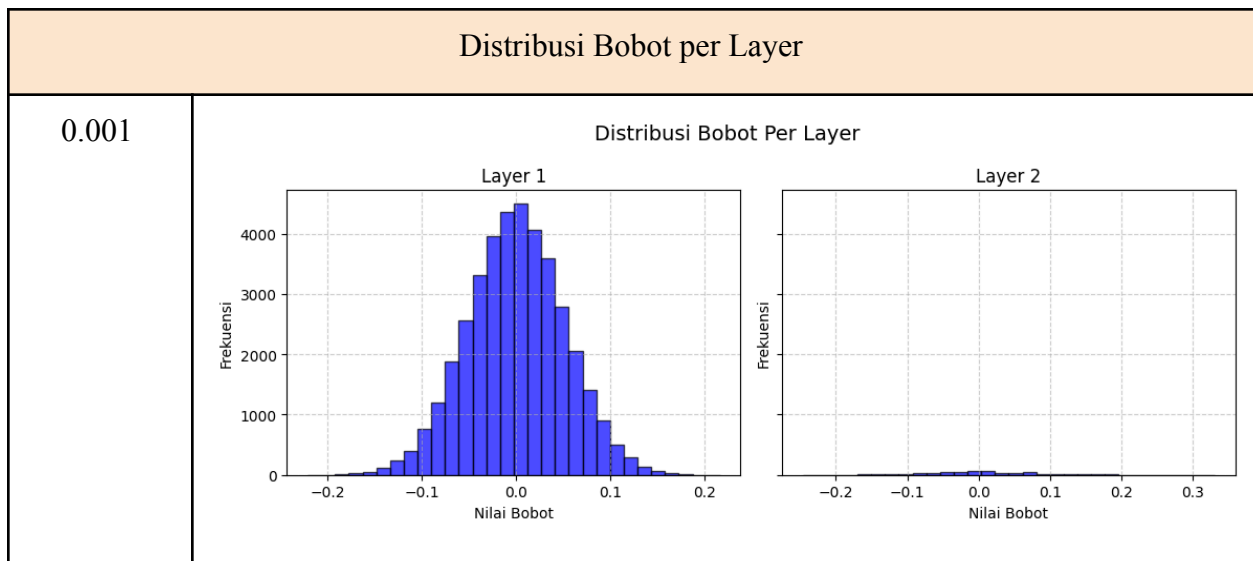
Epoch 20/20 [=====] 100.0% - Training Loss: 0.8848 - Validation Loss: 0.8866 - 1.60s/epoch - ETA: 0.6

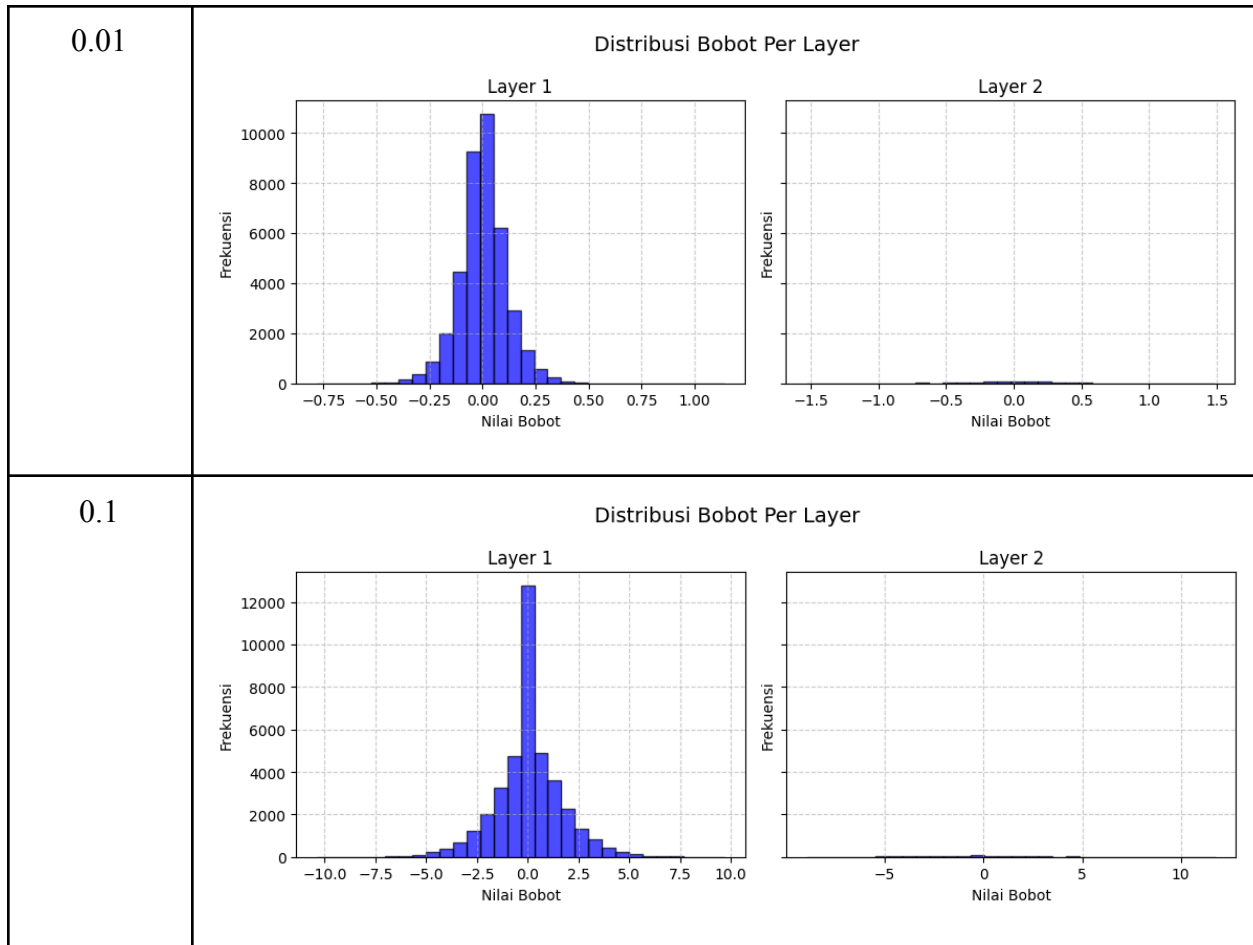
Training History

Epoch	Train Loss	Val Loss
1	0.395064	1.202531
2	1.204248	1.153647
3	1.155787	1.119233
4	1.120407	1.091614
5	1.093477	1.042414
6	1.041905	0.994528
7	0.990881	0.967611
8	0.960876	0.931502
9	0.928420	0.913359
10	0.910502	0.898608
11	0.896813	0.899105
12	0.896107	0.910023
13	0.908409	0.904843
14	0.902985	0.896030
15	0.893883	0.894897
16	0.893218	0.895254
17	0.891679	0.891330
18	0.888547	0.889909
19	0.888531	0.887858
20	0.884845	0.886602



Dari hasil uji coba, pengaruh learning rate berturut-turut dari percobaan learning rate 0,001; 0,01; 0,1 berturut-turut memberikan nilai 0,246; 0,979; dan 0,887. Hal ini menunjukkan bahwa semakin kecil learning rate maka nilai validation lossnya akan semakin mengecil. Terakhir, Pengujian dengan learning rate 0.001 memberikan nilai yang optimal.

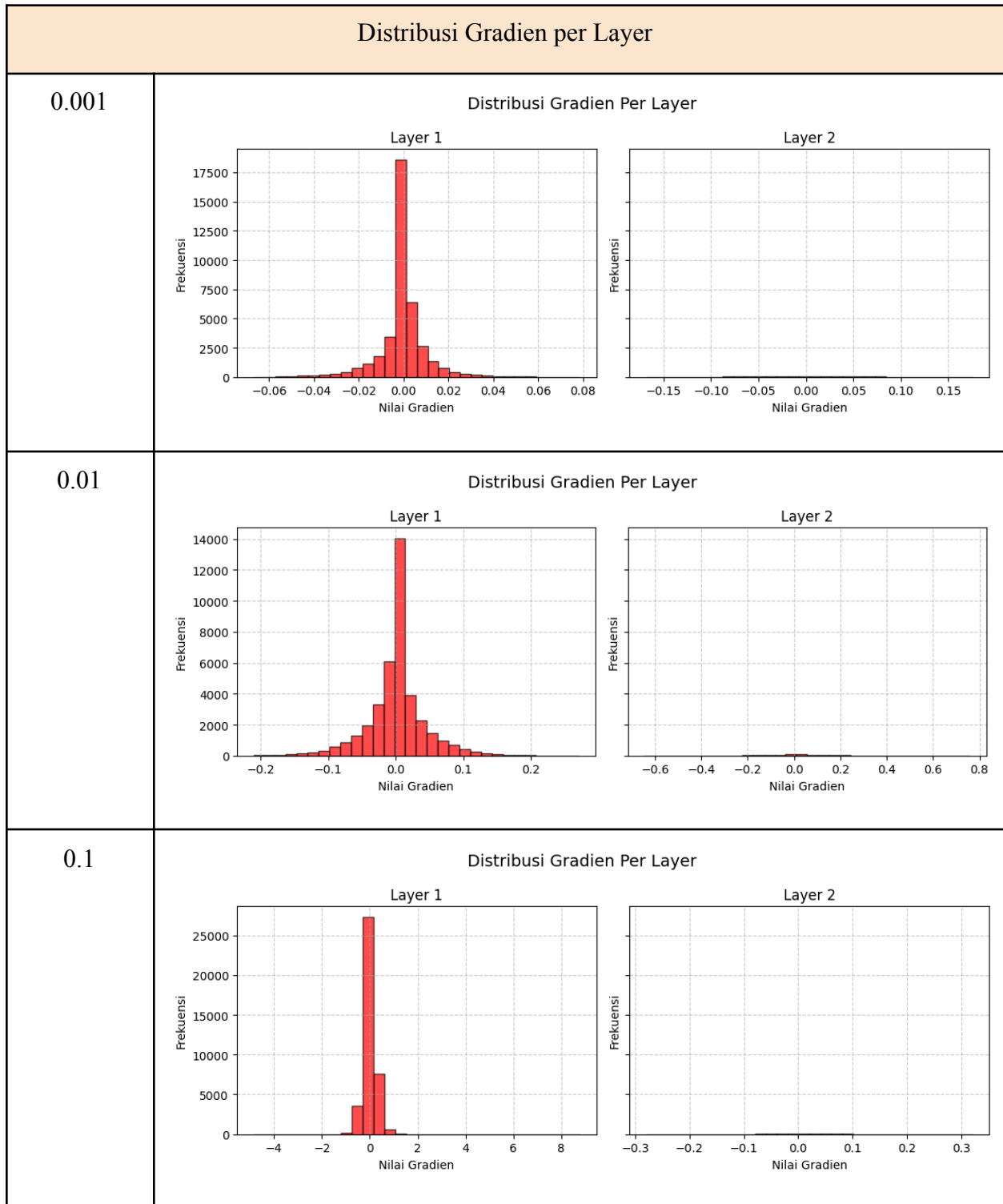




Learning Rate	0.001	0.01	0.1
Akurasi	41.23%	42.56%	29.65%

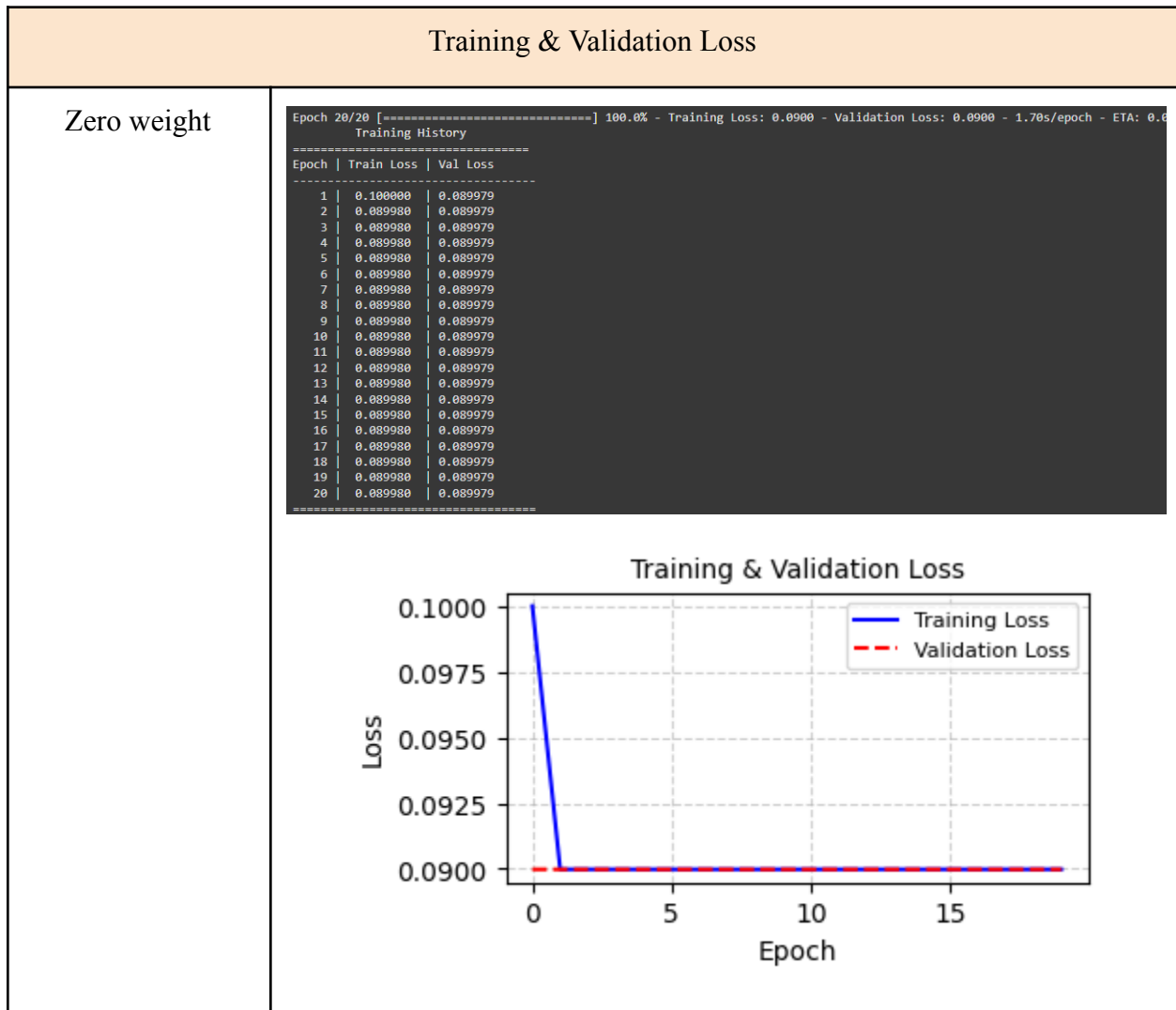
Pada learning rate 0.001, distribusi bobot masih relatif normal dan terpusat di sekitar nol tapi tersebar secara merata, menunjukkan pembelajaran berlangsung dengan stabil. Ketika learning rate meningkat menjadi 0.01, nilai bobot mulai membesar dengan beberapa nilai ekstrim, menandakan adanya peningkatan variabilitas dalam pembaruan bobot. Pada learning rate 0.1, distribusi bobot semakin menyebar luas dengan nilai yang jauh lebih besar, yang mengindikasikan kemungkinan *exploding gradient* atau pembaruan bobot menjadi terlalu besar akibat akumulasi gradien yang terus meningkat, menyebabkan nilai bobot membesar secara tidak terkendali. Hal ini dapat menghambat konvergensi model dan bahkan membuat training menjadi tidak stabil. Oleh karena itu, bisa disimpulkan bahwa semakin tinggi nilai learning rate

menyebabkan bobot model membesar secara tidak stabil, yang dapat mengarah ke divergensi selama training.



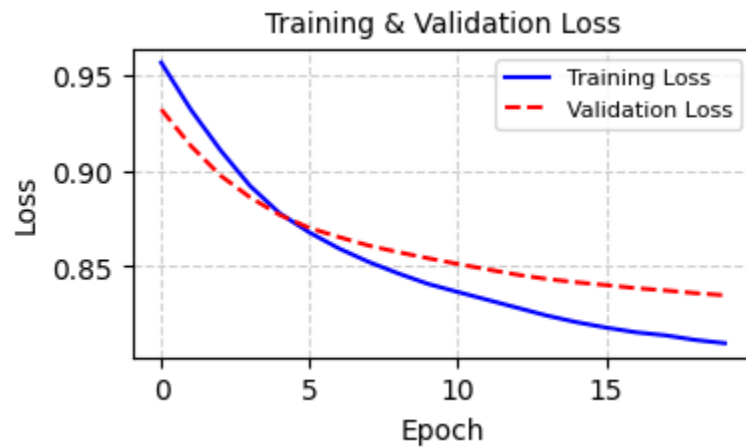
Distribusi gradien per layer menunjukkan bahwa semakin besar nilai *learning rate*, semakin tinggi risiko terjadinya *exploding gradient*. Pada *learning rate* 0.001, distribusi gradien masih terkendali dan mayoritas nilai berada di sekitar nol, terutama pada layer pertama, sementara layer kedua hampir tidak aktif. Saat *learning rate* dinaikkan ke 0.01, rentang gradien mulai melebar, mengindikasikan awal ketidakstabilan pembaruan bobot meskipun pusat distribusi tetap di nol. Pada *learning rate* 0.1, gradien mengalami lonjakan drastis dengan nilai yang sangat besar pada layer pertama, menandakan *exploding gradient* yang serius, sedangkan layer kedua tetap pasif. Pola ini memperlihatkan bahwa peningkatan *learning rate* secara signifikan dapat mengganggu stabilitas pelatihan model.

3.4 Pengaruh Inisialisasi Bobot



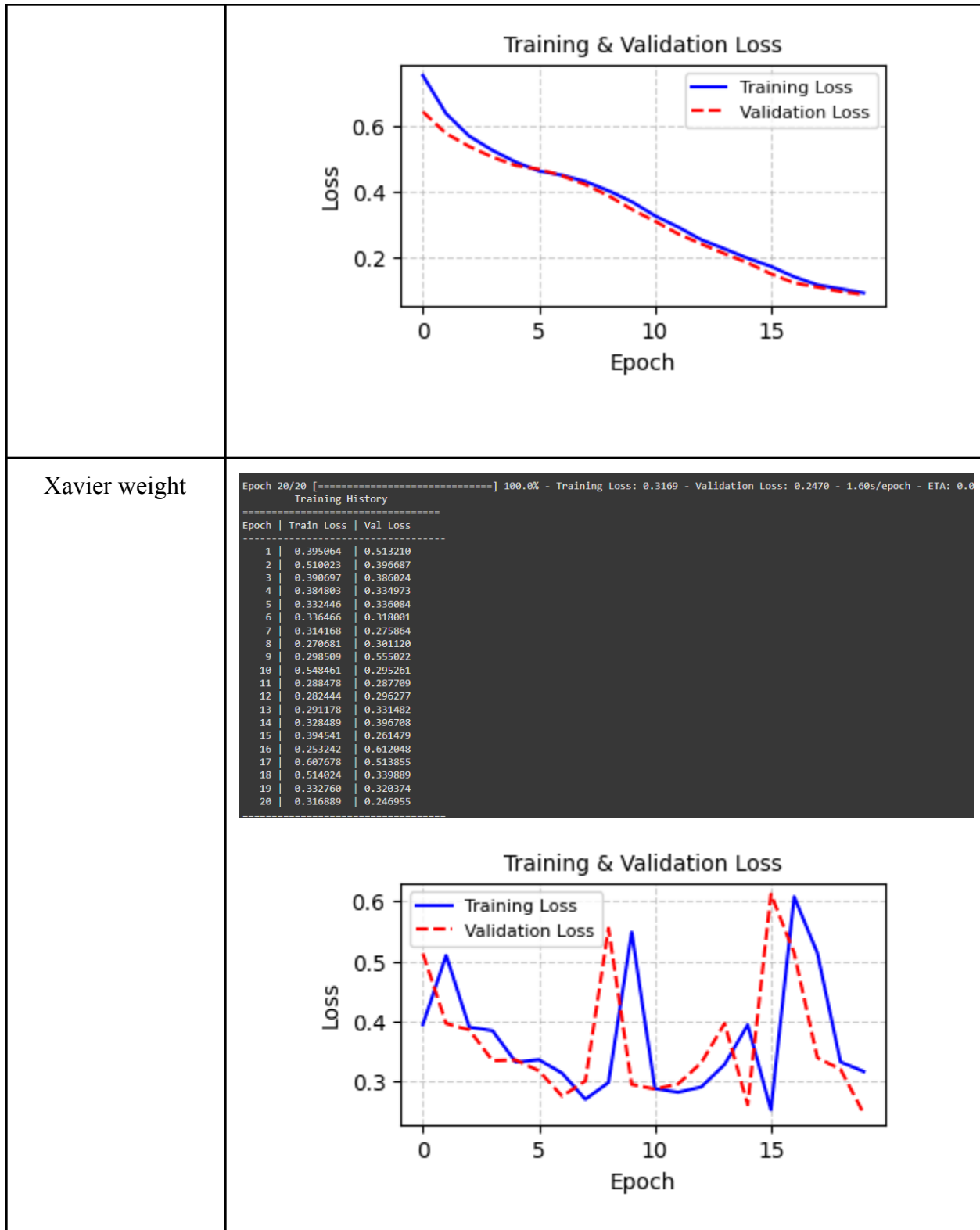
Normal weight

```
normal weight initializations Result
Epoch 20/20 [=====] 100.0% - Training Loss: 0.8101 - Validation Loss: 0.8350 - 1.70s/epoch - ETA: 0.0
Training History
=====
Epoch | Train Loss | Val Loss
-----|-----|-----
1 | 0.956252 | 0.931899
2 | 0.931711 | 0.912844
3 | 0.910565 | 0.897374
4 | 0.891930 | 0.885984
5 | 0.877648 | 0.876865
6 | 0.867699 | 0.870079
7 | 0.859345 | 0.865309
8 | 0.852353 | 0.860885
9 | 0.846342 | 0.857446
10 | 0.840907 | 0.854300
11 | 0.836715 | 0.851333
12 | 0.832624 | 0.848467
13 | 0.828523 | 0.845739
14 | 0.824432 | 0.843500
15 | 0.820994 | 0.841771
16 | 0.818162 | 0.840310
17 | 0.815733 | 0.838734
18 | 0.814127 | 0.837521
19 | 0.811734 | 0.836141
20 | 0.810056 | 0.834996
=====
```



Uniform weight

```
Epoch 20/20 [=====] 100.0% - Training Loss: 0.0923 - Validation Loss: 0.0873 - 1.70s/epoch - ETA: 0.0
Training History
=====
Epoch | Train Loss | Val Loss
-----|-----|-----
1 | 0.750921 | 0.640806
2 | 0.635111 | 0.574884
3 | 0.567030 | 0.535634
4 | 0.523974 | 0.503481
5 | 0.488618 | 0.478186
6 | 0.461566 | 0.466951
7 | 0.448635 | 0.446773
8 | 0.430132 | 0.420592
9 | 0.401707 | 0.386617
10 | 0.368704 | 0.345344
11 | 0.326004 | 0.309495
12 | 0.291741 | 0.271141
13 | 0.253313 | 0.240568
14 | 0.225790 | 0.211225
15 | 0.197232 | 0.183159
16 | 0.172527 | 0.150069
17 | 0.141063 | 0.122311
18 | 0.116537 | 0.110888
19 | 0.104767 | 0.096099
20 | 0.092327 | 0.087267
=====
```

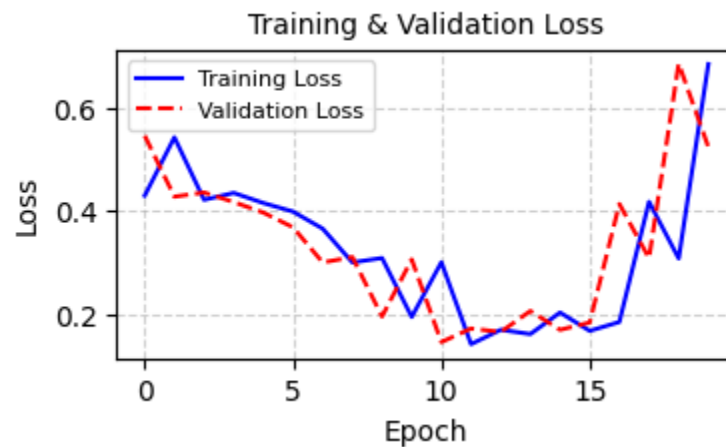


He weight

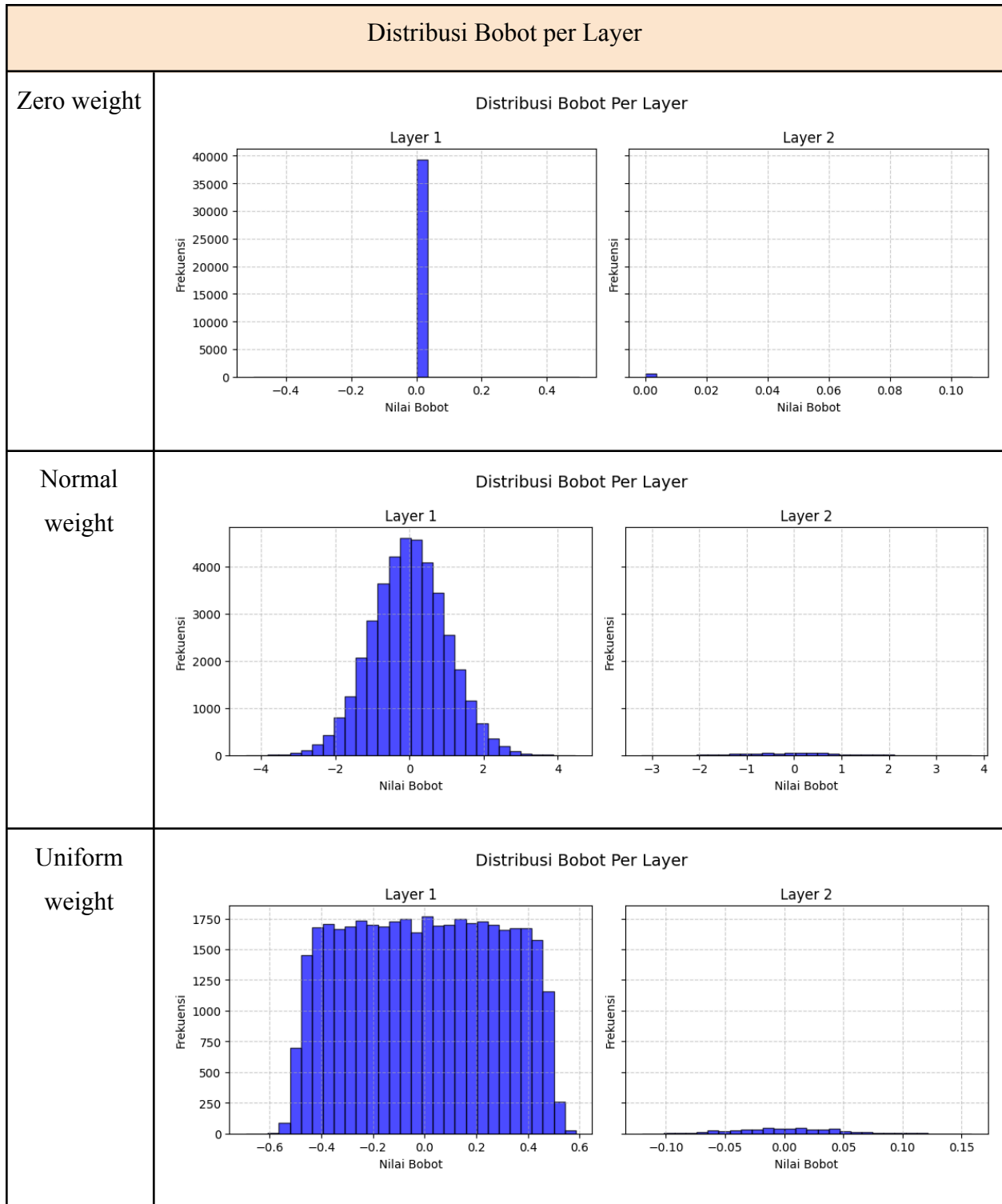
poch 20/20 [=====] 100.0% - Training Loss: 0.6852 - Validation Loss: 0.5243 - 1.70s/epoch - ETA: 0.00

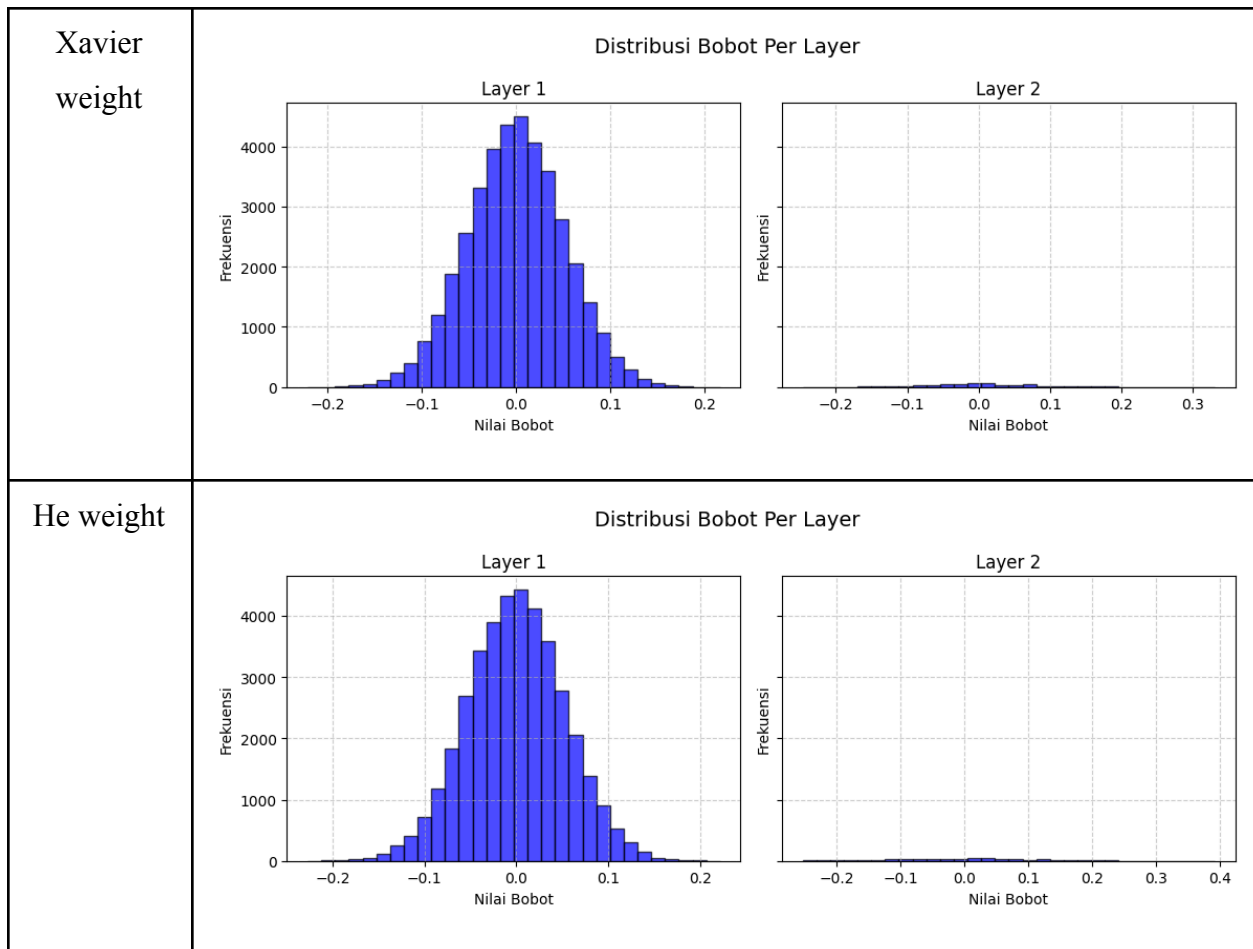
Training History

poch	Train Loss	Val Loss
1	0.430087	0.546343
2	0.543166	0.428887
3	0.421928	0.436881
4	0.435326	0.417561
5	0.415683	0.397107
6	0.399061	0.368510
7	0.365893	0.300374
8	0.300414	0.310841
9	0.308976	0.195518
10	0.194674	0.306215
11	0.300852	0.145818
12	0.141828	0.172217
13	0.169734	0.165469
14	0.161457	0.206227
15	0.203209	0.169657
16	0.167656	0.183806
17	0.184878	0.413893
18	0.418095	0.308720
19	0.308077	0.686225
20	0.685205	0.524347



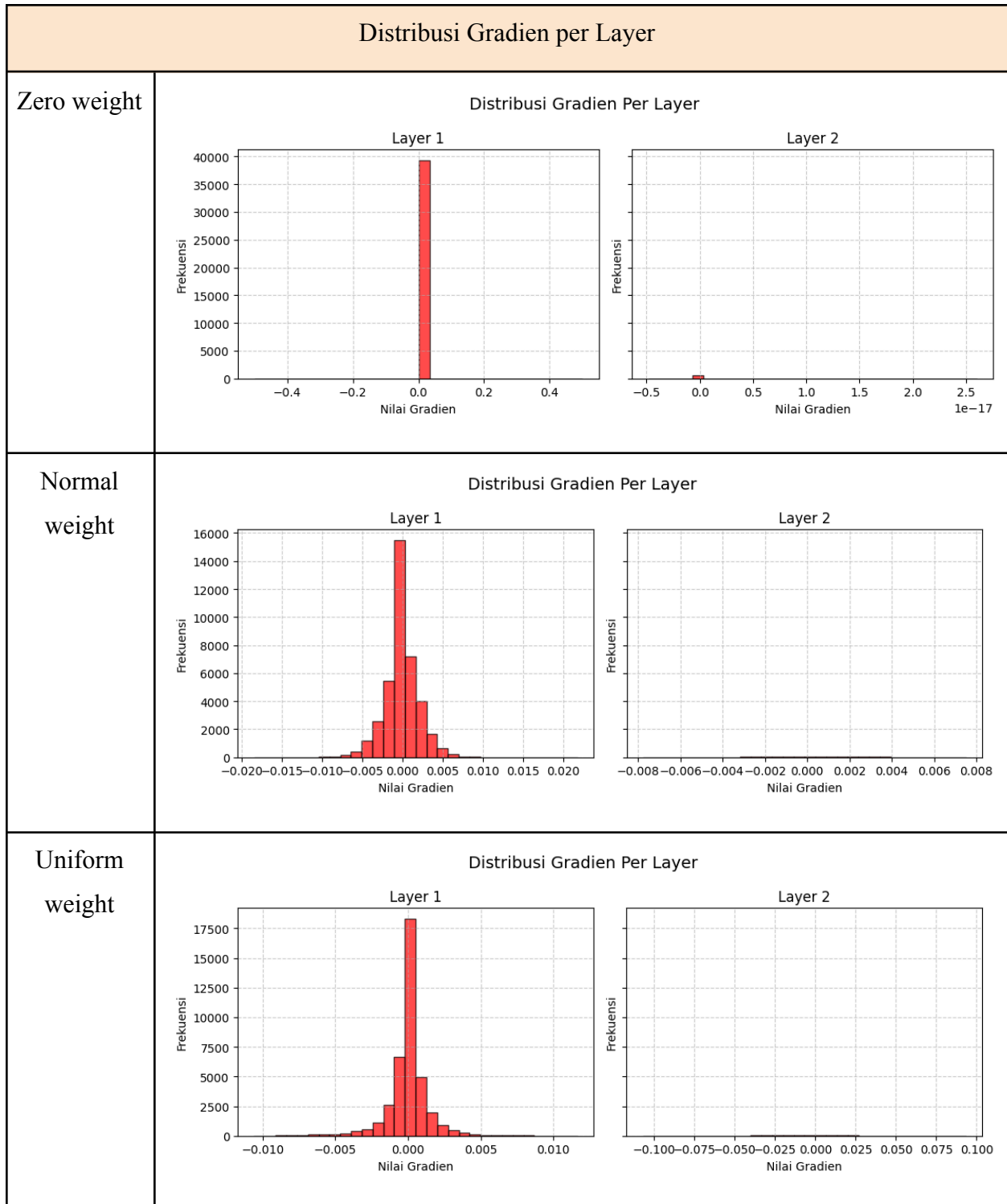
Dari hasil uji coba, penggunaan inisialisasi Uniform weight memberikan model yang cukup optimal. Hal ini bisa dilihat dari grafik validation error yang menurun dengan nilai loss validation akhir 0.087 namun hal ini memiliki potensi overfit pada train data. Selanjutnya Xavier memiliki nilai validation yang cukup rendah di akhir yakni, 0.246 meskipun grafik naik turun pada validation loss dan training loss xavier memberikan nilai yang rendah di akhir epoch dan menunjukkan model tidak *overfitting*. Pada zero weight proses penurunan loss terjadi sangat lambat sehingga perubahan loss yang terjadi pada tiap epoch tidak signifikan. Sedangkan inisialisasi He menghasilkan model yang overfit karena nilai validation yang meningkat pada akhir epoch. Untuk normal, terlihat grafik yang terus menurun namun penurunan loss yang terjadi juga sangat lambat untuk setiap epochnya.

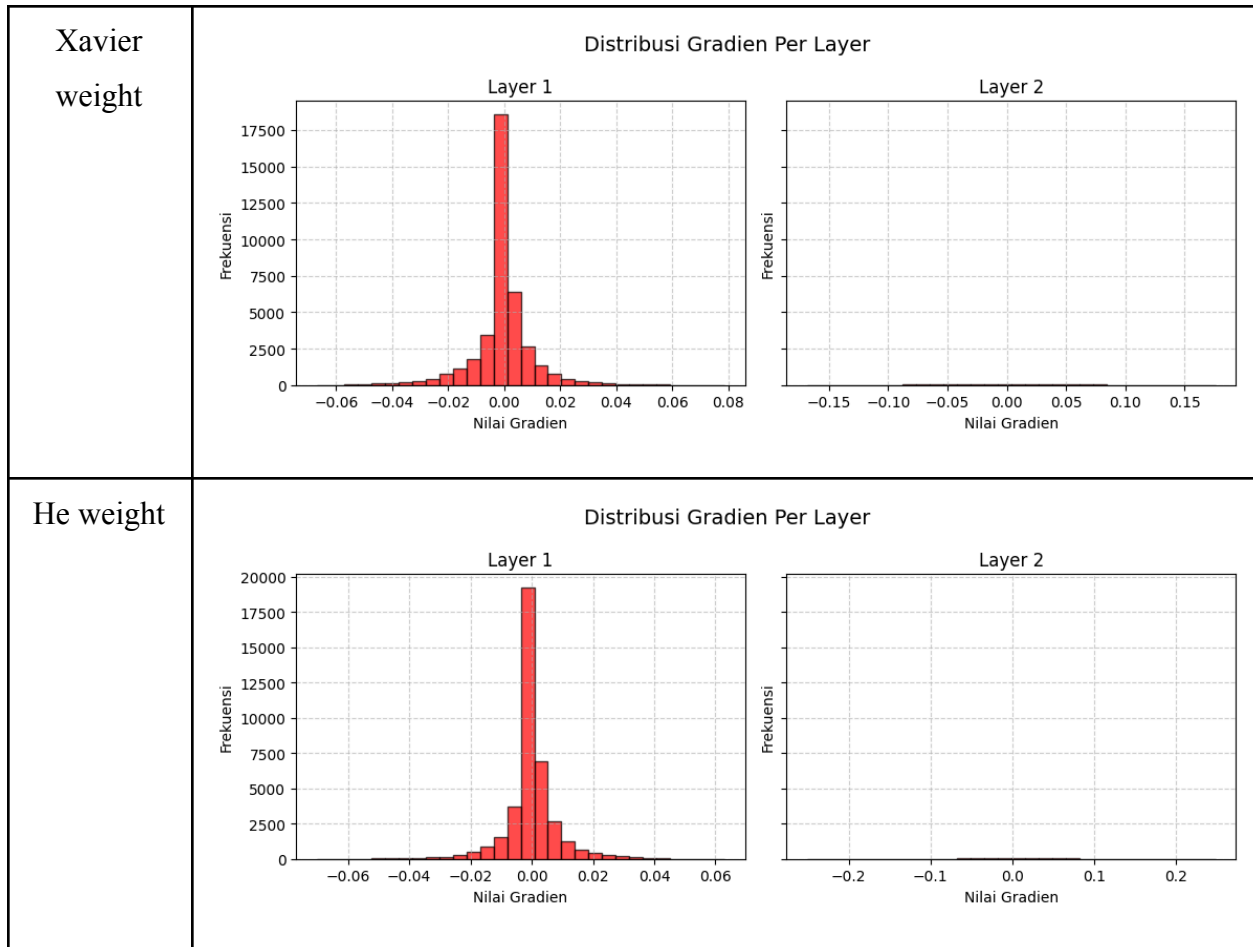




Bobot	zero	normal	uniform	xavier	he
Akurasi	9.81%	35.18%	60.32%	41.23%	26.65%

Dari berbagai metode inisialisasi bobot, Xavier dan He Weight menunjukkan distribusi bobot yang lebih stabil dibandingkan metode lainnya, menjadikannya pilihan terbaik. Zero Weight menyebabkan semua neuron memiliki output yang sama. Normal weight dan Uniform Weight cenderung mengalami distribusi bobot pada layer yang lebih tinggi, yang dapat menyebabkan menurunnya nilai gradien saat backpropagation. Meski begitu, pada hasil akurasi pada test set uniform memiliki nilai akurasi yang paling tinggi dibanding yang lainnya. Dengan demikian, pemilihan metode inisialisasi yang tepat dapat berpengaruh terhadap stabilitas dan konvergensi model selama proses training.

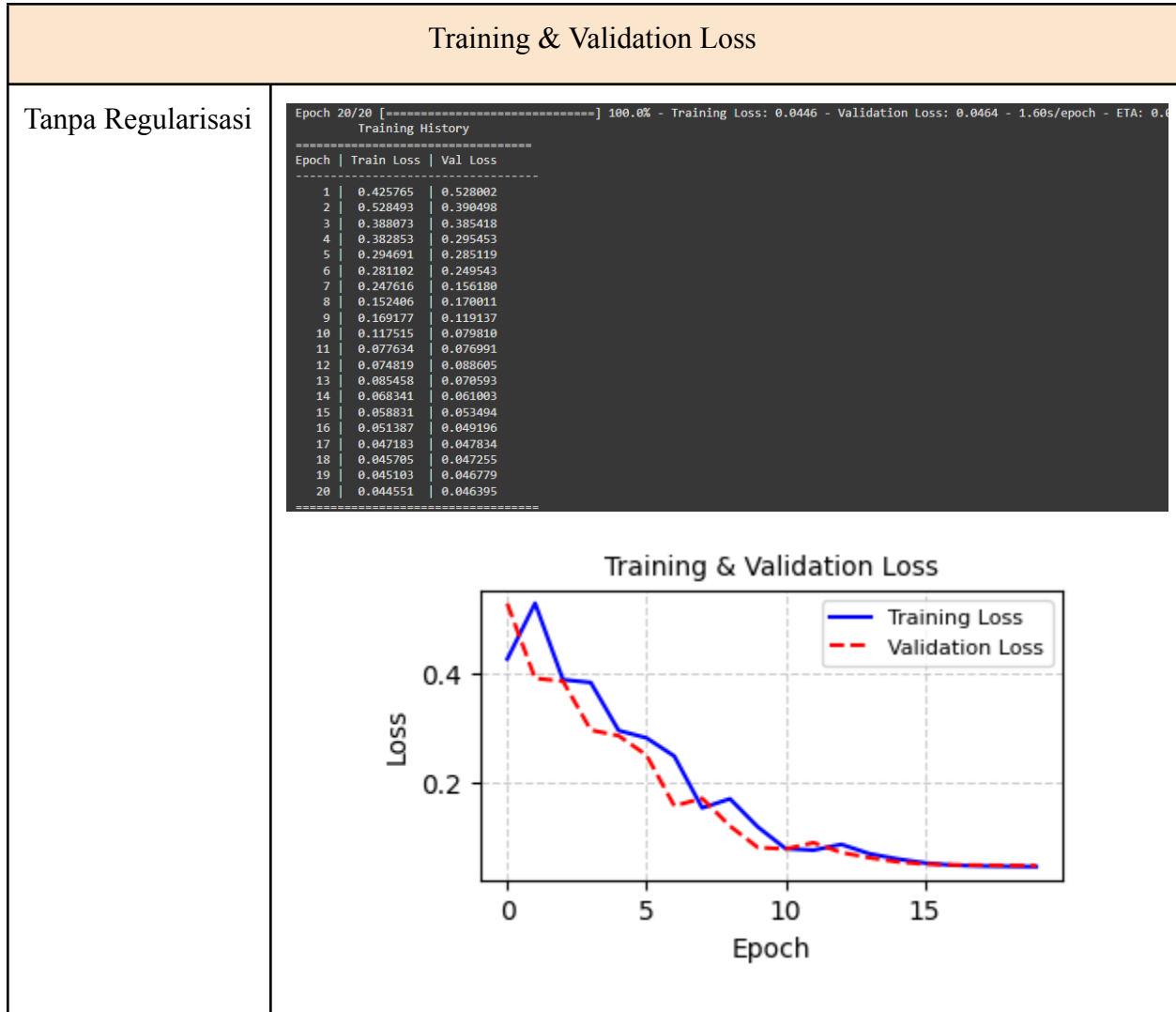




Dari analisis distribusi gradien pada berbagai metode inisialisasi bobot, terlihat bahwa Zero Weight menyebabkan gradien seluruh layer bernilai nol, yang menyebabkan model tidak melakukan pembelajaran karena backpropagation tidak berjalan. He dan Uniform Weight menghasilkan gradien yang terlalu besar di beberapa layer, berpotensi menyebabkan *exploding gradient*. Xavier dan normal mendistribusikan gradien dengan lebih stabil di seluruh layer, cocok untuk aktivasi sigmoid dan tanh.

3.5 Perbandingan Regularisasi

Berikut hasil pengujian model tanpa regularisasi, dengan regularisasi l1 dan regularisasi l2.



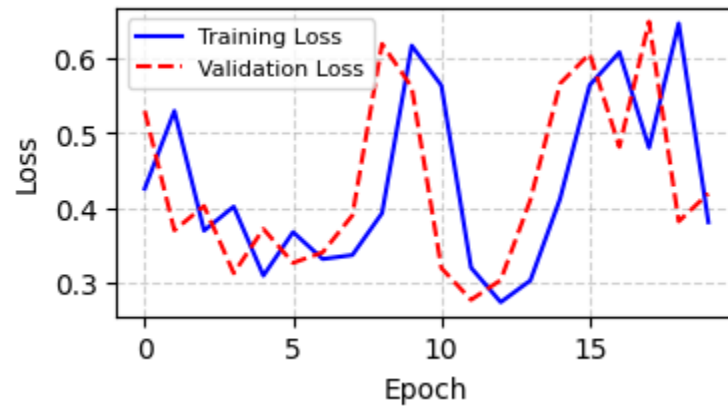
Regularisasi l1

Epoch 20/20 [=====] 100.0% - Training Loss: 0.3810 - Validation Loss: 0.4194 - 1.50s/epoch - ETA: 0.00s

Training History

Epoch	Train Loss	Val Loss
1	0.425765	0.529849
2	0.529853	0.370120
3	0.369690	0.402736
4	0.401821	0.312756
5	0.309464	0.371958
6	0.367675	0.326495
7	0.331689	0.340631
8	0.336993	0.390124
9	0.393106	0.619489
10	0.617231	0.561857
11	0.563902	0.319586
12	0.320134	0.277066
13	0.273926	0.303498
14	0.302941	0.410728
15	0.412141	0.566483
16	0.564330	0.606342
17	0.608336	0.481696
18	0.480579	0.649369
19	0.646981	0.382059
20	0.381010	0.419377

Training & Validation Loss

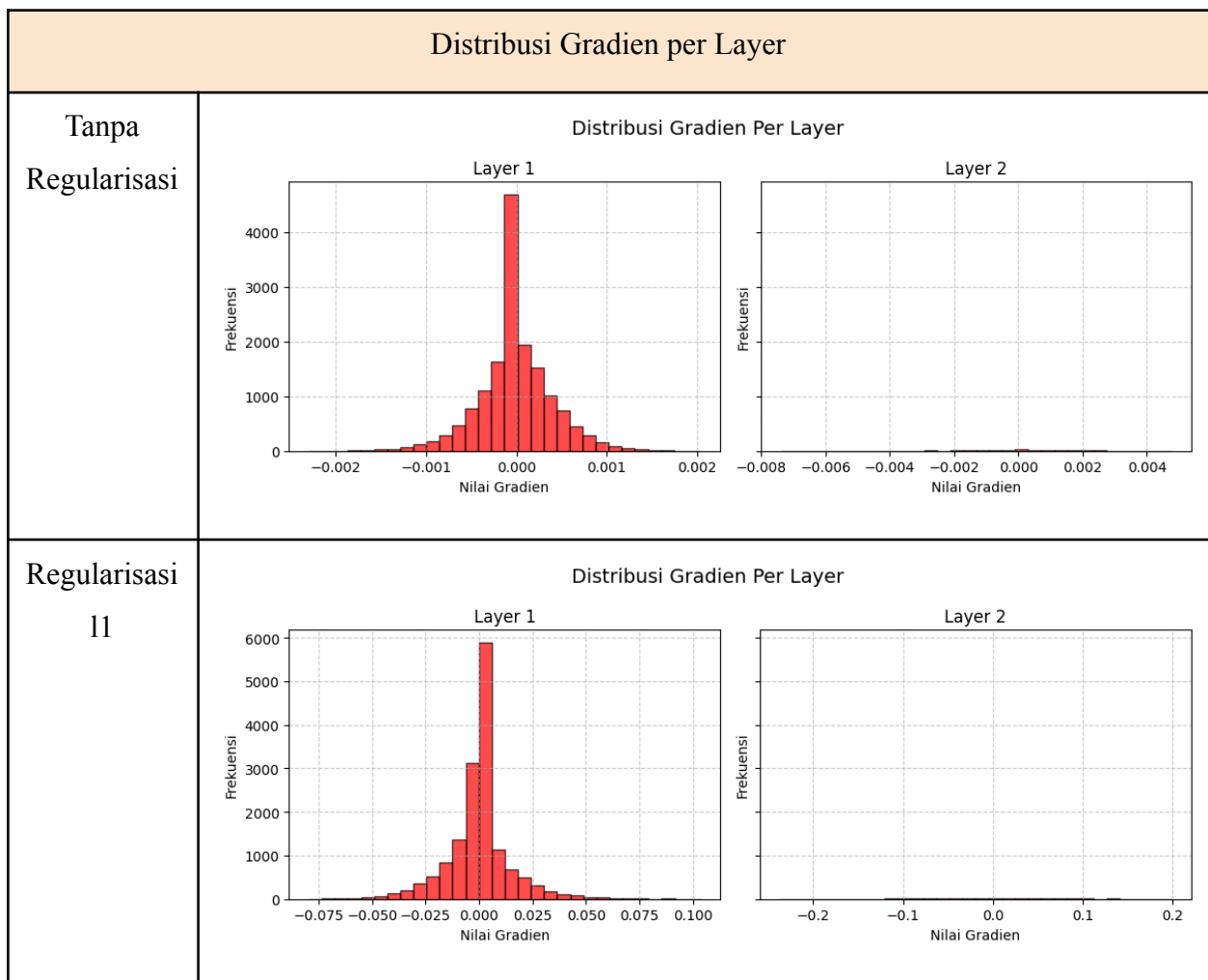
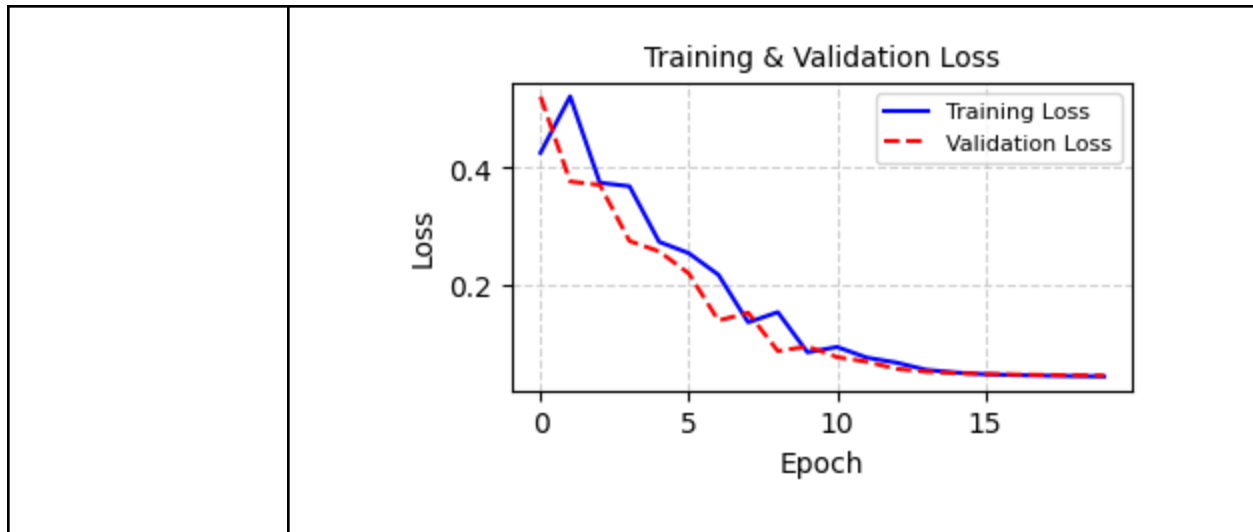


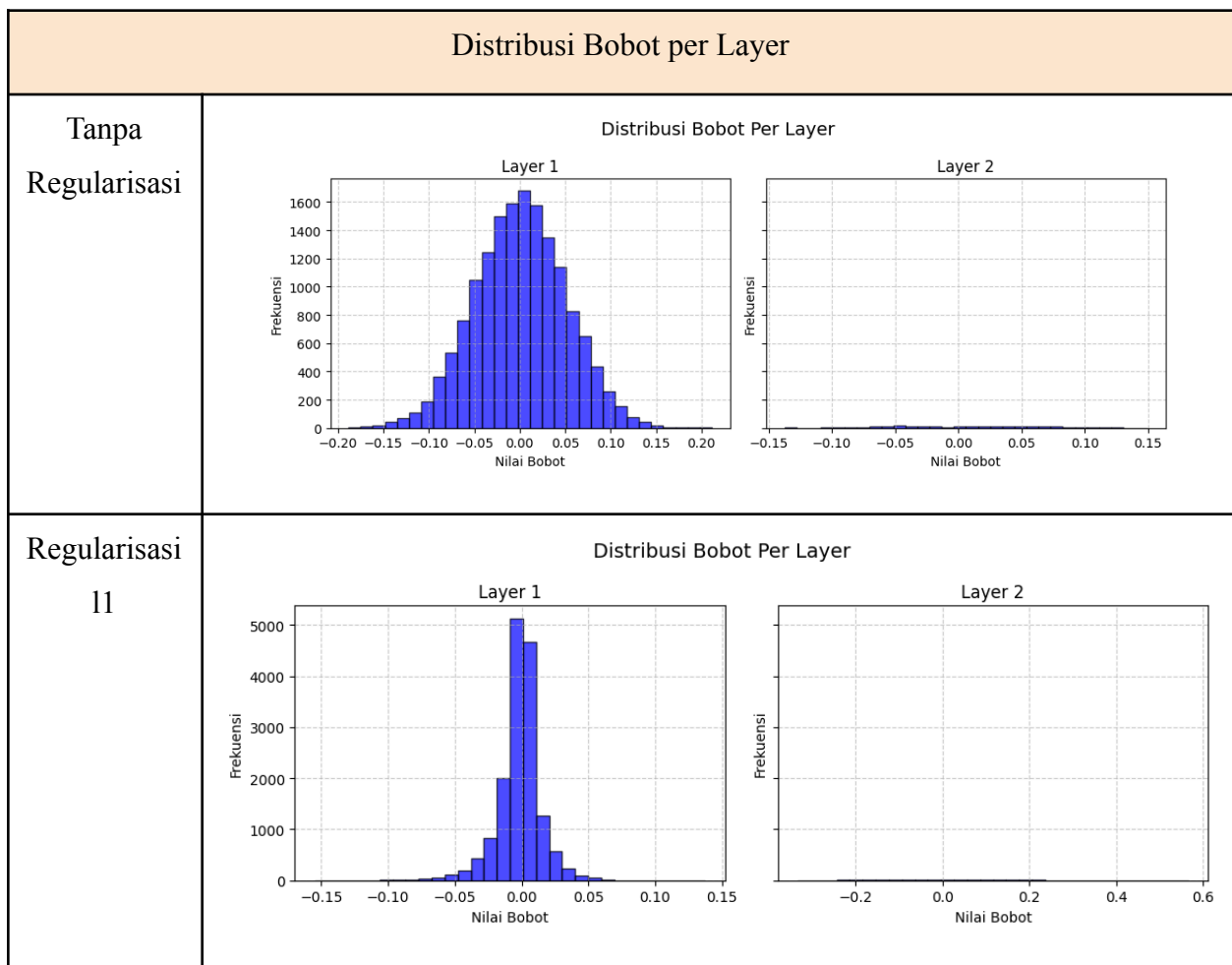
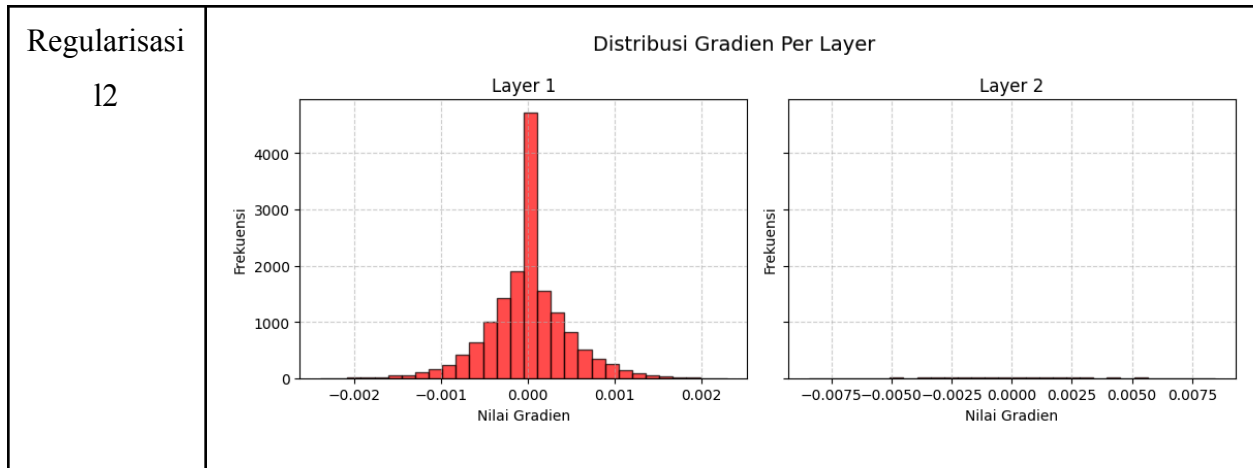
Regularisasi l2

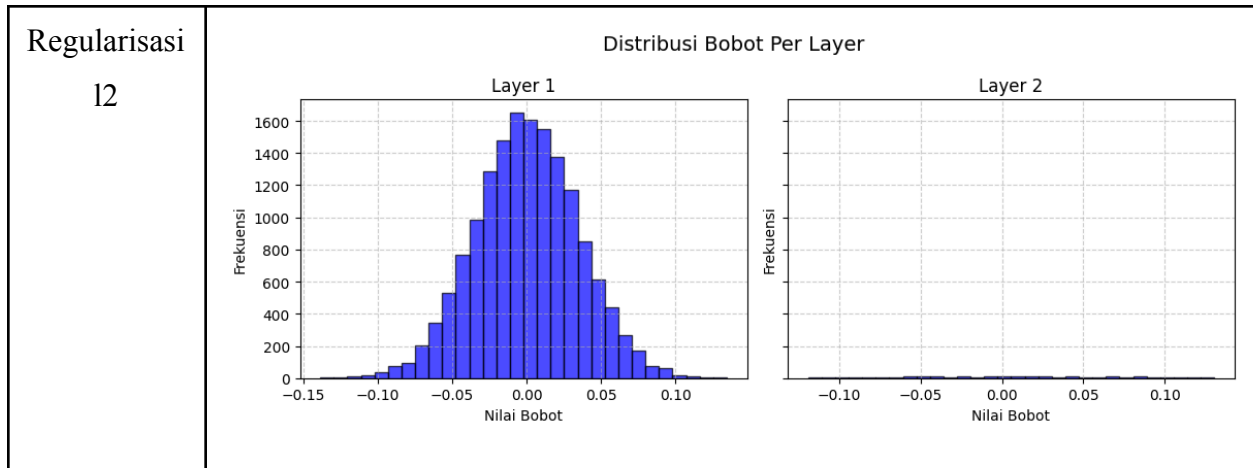
Epoch 20/20 [=====] 100.0% - Training Loss: 0.0428 - Validation Loss: 0.0444 - 1.50s/epoch - ETA: 0.00s

Training History

Epoch	Train Loss	Val Loss
1	0.425765	0.521850
2	0.522210	0.376527
3	0.374381	0.370709
4	0.368439	0.274617
5	0.273215	0.256868
6	0.253767	0.218945
7	0.216720	0.138471
8	0.135401	0.151742
9	0.152773	0.086298
10	0.084115	0.093983
11	0.093187	0.075839
12	0.074969	0.067967
13	0.066423	0.055744
14	0.054037	0.050947
15	0.049408	0.048305
16	0.046491	0.046065
17	0.045147	0.046011
18	0.044105	0.045417
19	0.043430	0.044814
20	0.042769	0.044405







Model	Tanpa Regularisasi	Regularisasi l1	Regularisasi l2
Akurasi	82.55%	21.49%	83.63%

Model yang dijalankan dengan Regularisasi l2 memberikan nilai akurasi yang paling baik yaitu 83.63 %. Hal ini bisa ditinjau dari nilai validation loss terkecil yang diberikan diantara ketiganya yaitu di angka 0.044.

Jika ditinjau dari persebaran bobot per layernya persebaran untuk model tanpa regularisasi dan regularisasi l2 hampir mirip dan persebaran di sekitar 0 nya lebih sedikit <1700 sedangkan untuk regularisasi l1 >5000. Artinya, model ini akan mengupdate bobotnya tidak seperti regularisasi yang memberikan lebih banyak nilai bobot bernilai nol, yang menyebabkan model tidak melakukan pembelajaran karena backpropagation tidak berjalan.

Jika ditinjau dari persebaran gradien per layernya menunjukkan bahwa penggunaan regularisasi berpengaruh terhadap stabilitas pembelajaran dalam model. Pada model tanpa regularisasi, gradien di layer pertama terdistribusi merata dengan pusat di sekitar nol <5000, sementara layer kedua hampir tidak aktif. Saat regularisasi L1 diterapkan, distribusi gradien melebar dengan nilai-nilai ekstrem yang mulai muncul, dan persebaran berada di sekitaran nol yang mengartikan backpropagation tidak berjalan dengan baik mengindikasikan bahwa L1 mendorong sparsity namun juga dapat menyebabkan ketidakstabilan pada pembaruan bobot. Sementara itu, regularisasi L2 menghasilkan distribusi gradien yang merata dan nilai gradien nol nya lebih sedikit <5000, menandakan bahwa L2 lebih efektif dalam menjaga stabilitas pembelajaran dengan menekan nilai-nilai gradien ekstrem. Secara keseluruhan, L1 dan L2 memiliki efek berbeda dalam mengontrol skala gradien, di mana L2 cenderung memberikan hasil yang lebih stabil dibandingkan tanpa regularisasi maupun L1.

3.6 Perbandingan Normalisasi RMSNorm

Training & Validation Loss

Tanpa Normalisasi

Epoch 20/20 [=====] 100.0% - Training Loss: 0.0485 - Validation Loss: 0.0489 - 1.50s/epoch - ETA: 0.00s

Training History

Epoch	Train Loss	Val Loss
1	0.358051	0.432641
2	0.432547	0.313025
3	0.312197	0.333579
4	0.331318	0.237307
5	0.231833	0.271147
6	0.262449	0.209541
7	0.205193	0.220975
8	0.213605	0.132430
9	0.131030	0.184045
10	0.183884	0.142037
11	0.140517	0.169235
12	0.167602	0.144507
13	0.146449	0.114805
14	0.114591	0.101174
15	0.098608	0.071080
16	0.068529	0.055855
17	0.054313	0.052423
18	0.050402	0.051540
19	0.049726	0.050568
20	0.048510	0.048941

Training & Validation Loss

Epoch	Train Loss	Val Loss
1	0.358051	0.432641
2	0.432547	0.313025
3	0.312197	0.333579
4	0.331318	0.237307
5	0.231833	0.271147
6	0.262449	0.209541
7	0.205193	0.220975
8	0.213605	0.132430
9	0.131030	0.184045
10	0.183884	0.142037
11	0.140517	0.169235
12	0.167602	0.144507
13	0.146449	0.114805
14	0.114591	0.101174
15	0.098608	0.071080
16	0.068529	0.055855
17	0.054313	0.052423
18	0.050402	0.051540
19	0.049726	0.050568
20	0.048510	0.048941

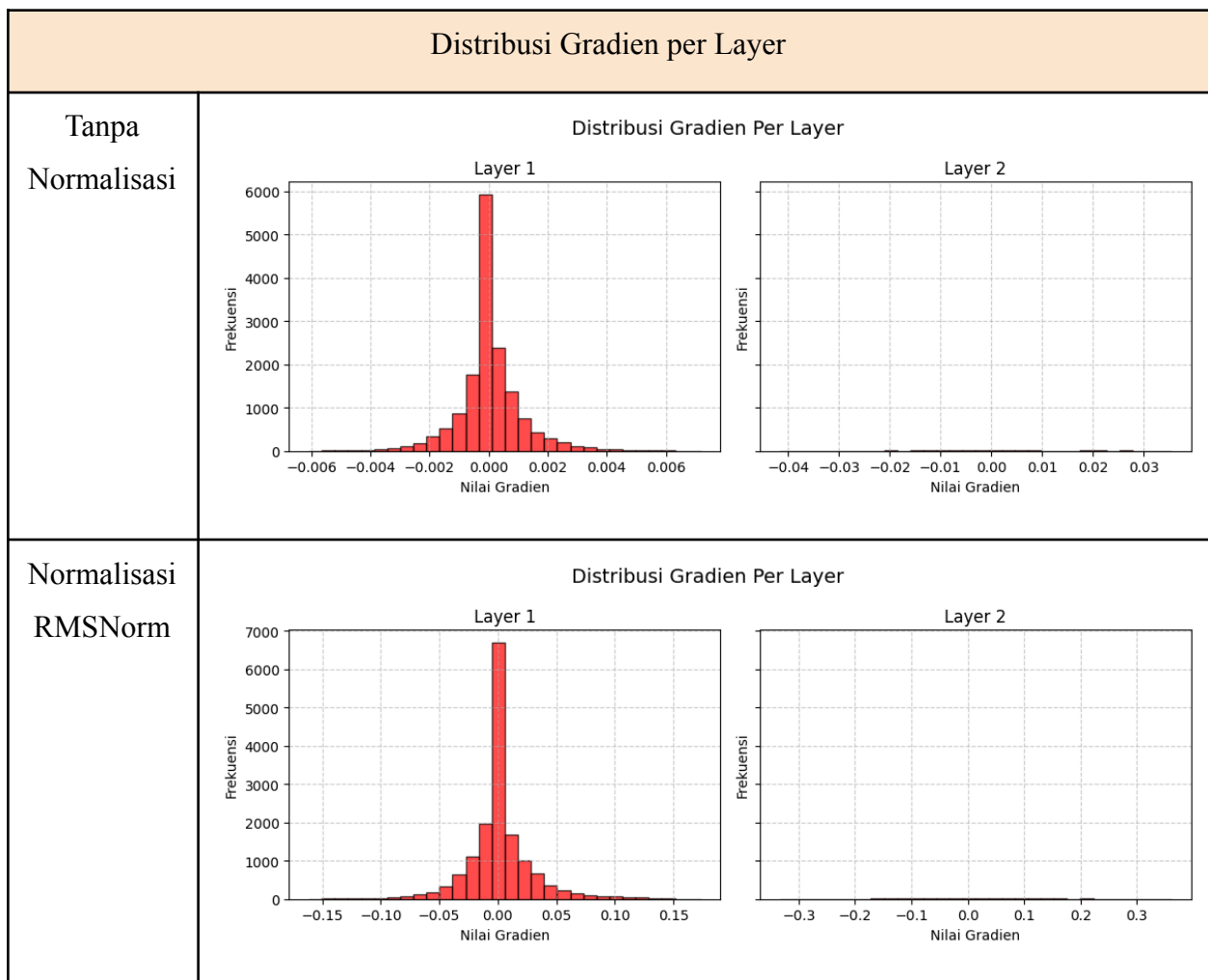
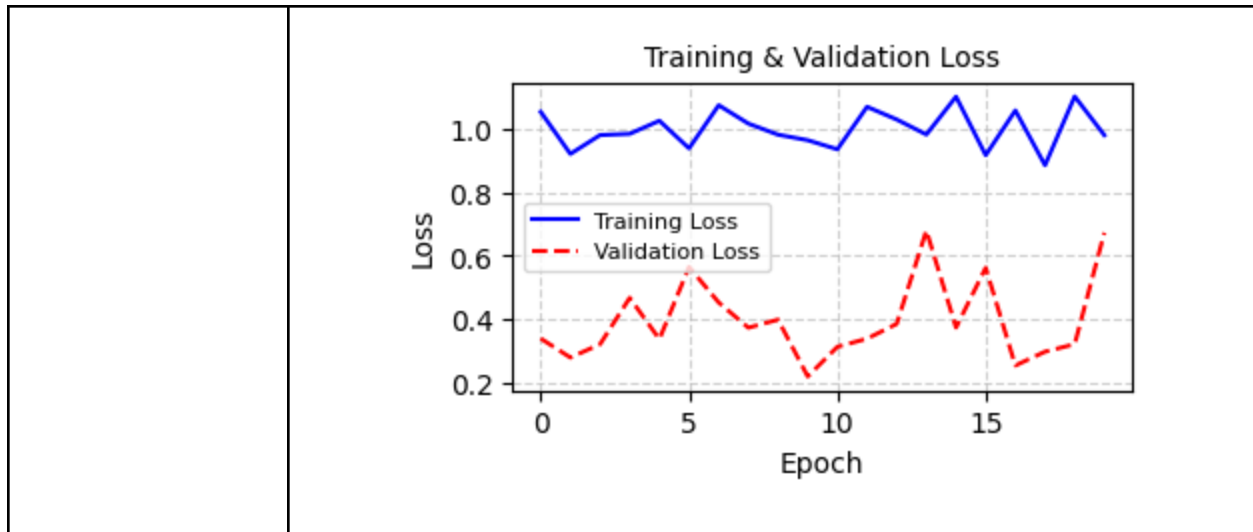
Normalisasi

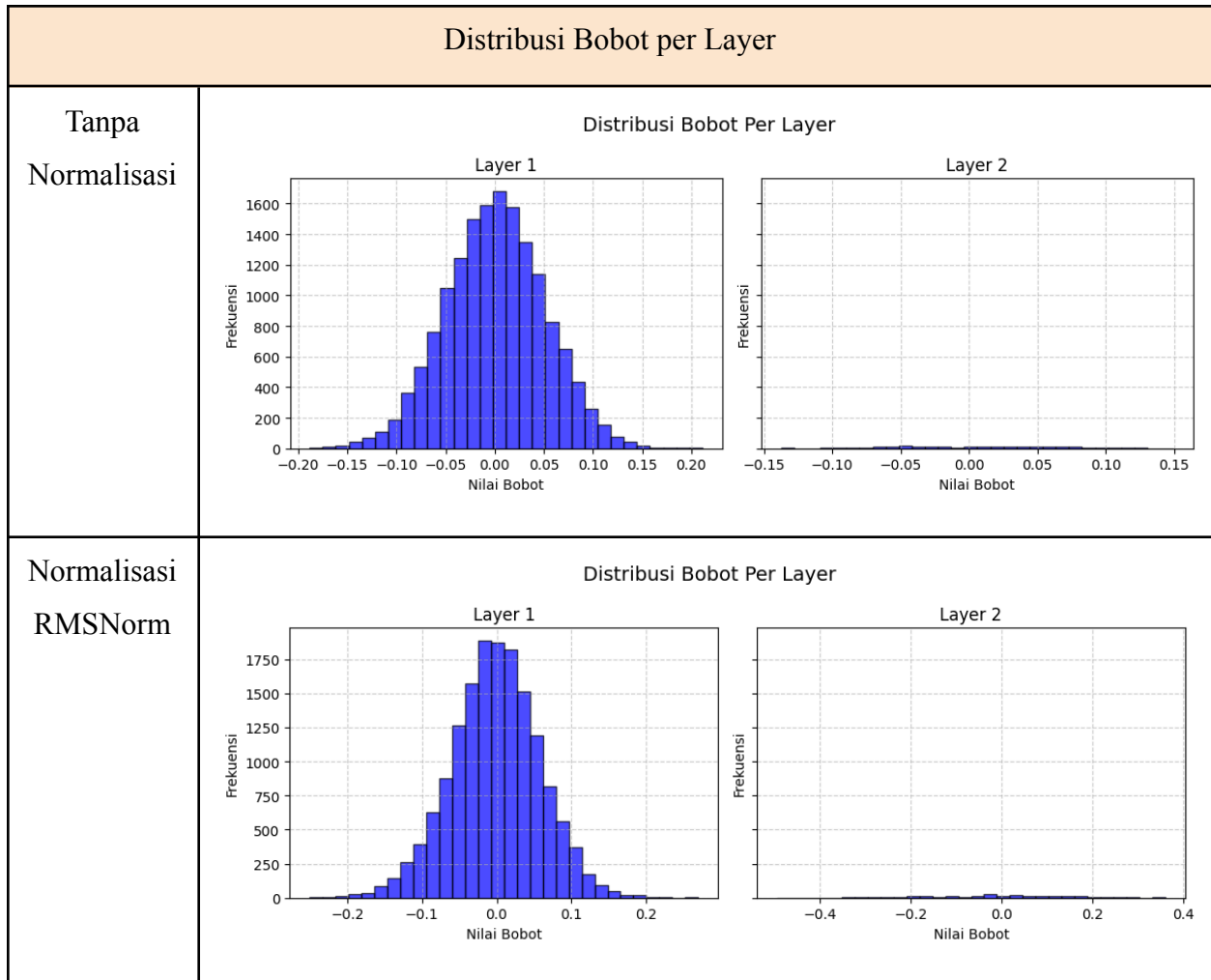
RMSNorm

Epoch 20/20 [=====] 100.0% - Training Loss: 0.9812 - Validation Loss: 0.6730 - 1.39s/epoch - ETA: 0.00s

Training History

Epoch	Train Loss	Val Loss
1	1.054928	0.338654
2	0.921973	0.278527
3	0.981581	0.320554
4	0.985534	0.467233
5	1.026886	0.336744
6	0.939629	0.564736
7	1.075973	0.452480
8	1.017794	0.372775
9	0.982815	0.398285
10	0.965377	0.218848
11	0.936629	0.312424
12	1.070937	0.338237
13	1.030759	0.384850
14	0.983311	0.679902
15	1.102903	0.373410
16	0.918122	0.562078
17	1.059338	0.253345
18	0.886232	0.297060
19	1.103671	0.320818
20	0.981228	0.672971

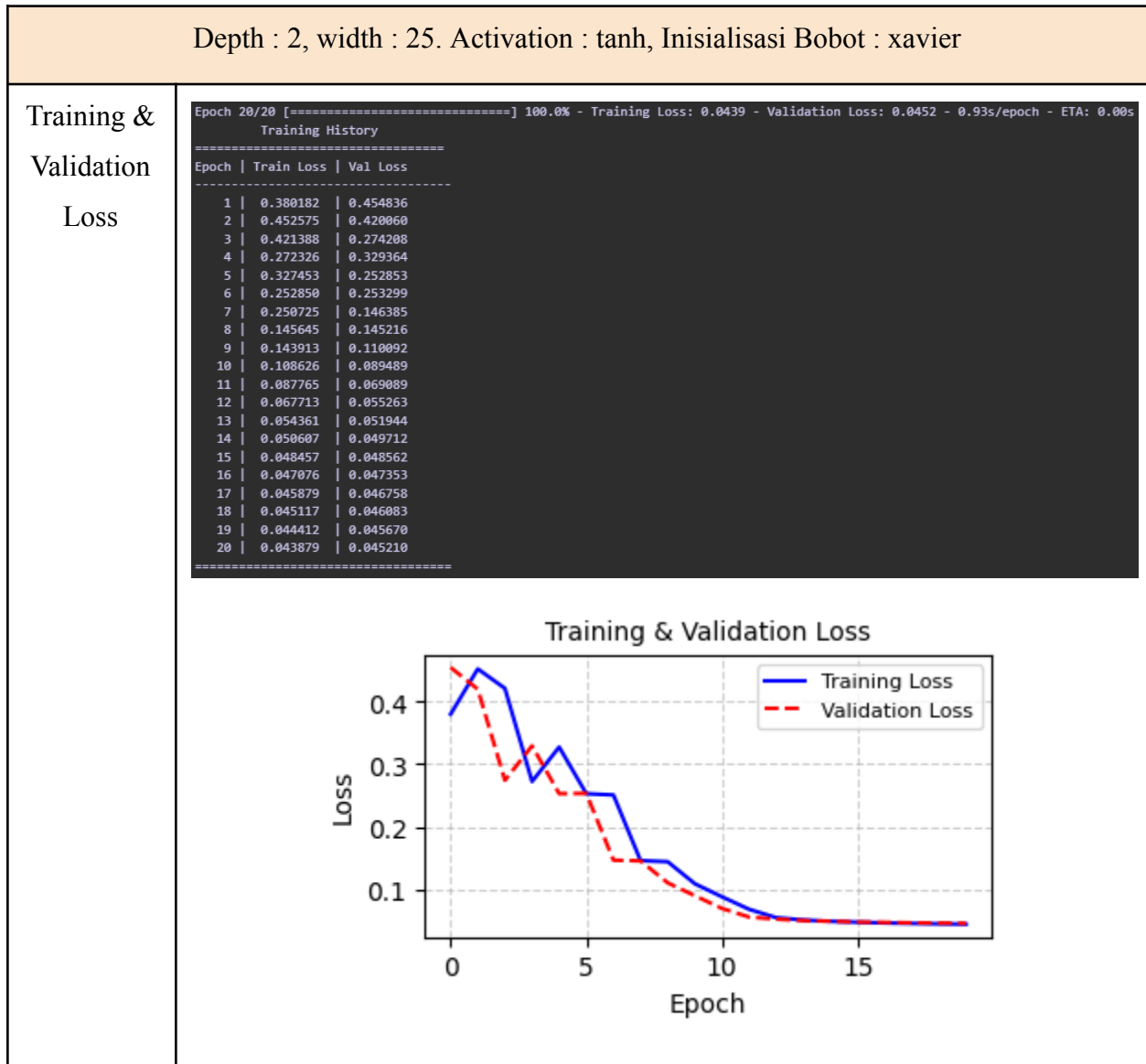


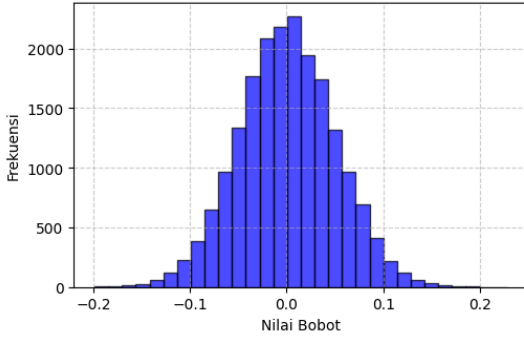
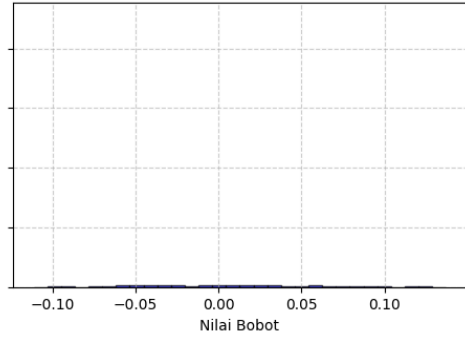
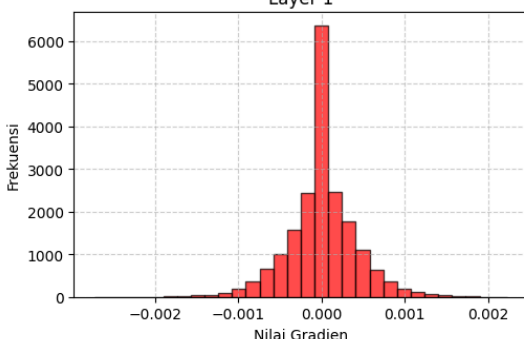
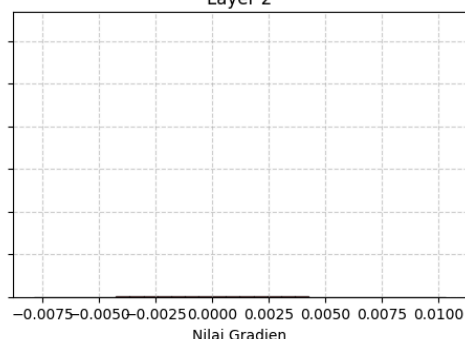


Model	Tanpa Normalisasi	Dengan Normalisasi
Akurasi	82.40%	9.97%

Berdasarkan hasil pengujian dengan parameter yang sama, model FFNN tanpa normalisasi mencapai akurasi sebesar 82.40%, yang jauh lebih tinggi dibandingkan dengan penggunaan RMSNorm yang hanya mencatat 9.97%. Perbedaan ini kemungkinan besar disebabkan oleh ketidakcocokan normalisasi dengan fungsi aktivasi tanh dan inisialisasi bobot xavier. Hal ini terlihat dari grafik loss yang menunjukkan fluktuasi yang signifikan pada penggunaan normalisasi, yang menandakan adanya overfitting. Selain itu, distribusi bobot dan gradien pada lapisan yang lebih dalam menunjukkan persebaran nilai yang lebih luas, yang dapat mengakibatkan masalah exploding gradient dan berdampak negatif pada akurasi data.

3.7 Perbandingan dengan library sklearn



Distribusi Bobot Per Layer	<p style="text-align: center;">Distribusi Bobot Per Layer</p> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p>Layer 1</p>  </div> <div style="text-align: center;"> <p>Layer 2</p>  </div> </div>
Distribusi Gradien Per Layer	<p style="text-align: center;">Distribusi Gradien Per Layer</p> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p>Layer 1</p>  </div> <div style="text-align: center;"> <p>Layer 2</p>  </div> </div>
Akurasi	<div style="background-color: #333; color: white; padding: 10px;"> <p>Akurasi Model FFNN : 83.26%</p> <p>Akurasi MLP Sklearn : 0.90%</p> </div>

Berdasarkan hasil pengujian dengan parameter yang sama, model FFNN menunjukkan akurasi 83.26%, jauh lebih tinggi dibandingkan MLP dari Sklearn yang hanya mencapai 0.90%. Perbedaan ini kemungkinan besar disebabkan oleh MLP Sklearn yang digunakan dengan parameter default tanpa tuning dan tidak memakai regularisasi, sehingga tidak optimal dalam menangani dataset ini. Distribusi bobot dan gradien per layer menunjukkan bahwa model mengalami *vanishing gradient* pada layer yang lebih dalam, yang dapat menghambat proses learning. Training loss yang menurun dan meningkat juga mengindikasikan adanya overfitting atau learning rate yang tidak stabil, yang perlu diperbaiki dengan penyesuaian lebih lanjut pada parameter seperti inisialisasi bobot, fungsi aktivasi, dan lainnya untuk meningkatkan nilai akurasi.

BAB IV

KESIMPULAN DAN SARAN

4.1 Kesimpulan

Berikut adalah beberapa kesimpulan yang diperoleh dari penelitian ini terkait implementasi Feedforward Neural Network pada tugas kali ini:

1. Pemilihan model FFNN yang optimal dipengaruhi oleh jumlah neuron, kedalaman layer, fungsi aktivasi, dan strategi regularisasi yang digunakan.
2. Kombinasi jumlah neuron yang tepat, kedalaman yang tepat, dan fungsi aktivasi yang sesuai dapat meningkatkan stabilitas serta kinerja model.
3. Pengaturan *hyperparameter* yang optimal, seperti *learning rate* dan metode inisialisasi bobot, berperan penting dalam mempercepat konvergensi serta mencegah *exploding* atau *vanishing gradients*.
4. Regularisasi yang efektif dapat mengurangi risiko overfitting, sehingga model memiliki kinerja yang lebih baik pada data baru.

4.2 Saran

Berdasarkan hasil penelitian ini, beberapa saran yang dapat dipertimbangkan untuk pengembangan lebih lanjut adalah sebagai berikut:

1. Mengeksplorasi model FFNN yang lebih kompleks dan menguji berbagai kombinasi *hyperparameter*.
2. Menerapkan strategi regularisasi tambahan untuk meningkatkan generalisasi model.
3. Melakukan evaluasi performa model pada berbagai dataset untuk memastikan kemampuan model terhadap variasi data.

PEMBAGIAN TUGAS

Nama	Pembagian Tugas
Muhammad Yusuf Rafi	Membuat fungsi save & load, Membuat fungsi distribusi bobot dan gradien, Implementasi Fungsi Aktivasi linear, dan implementasi Fungsi Loss Categorical Cross-Entropy
Debrina Veisha Rashika W	Membuat model FFNN, Inisialisasi bobot, Implementasi Fungsi Aktivasi Tanh dan Softmax, dan Implementasi Fungsi Loss MSE
Melati Anggraini	Membuat visualisasi graf, Implementasi Fungsi Aktivasi ReLU dan Sigmoid, dan Implementasi Fungsi Loss Binary Cross-Entropy

LINK REPOSITORY

<https://github.com/debrinashika/FFNN-Machine-Learning.git>

DAFTAR PUSTAKA

<https://www.jasonosajima.com/forwardprop>

<https://www.jasonosajima.com/backprop>

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

<https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>

<https://medium.com/@akshayush007/journey-llm-8-activation-functions-498accbe78c3>