Laporan Tugas Besar 1
IF3270 - Pembelajaran Mesin

Oleh kelompok 22:

Nama	NIM
Renaldy Arief Susanto	13522022
Nyoman Ganadipa Narayana	13522066
Muhammad Dava Fathurrahman	13522114

Deskripsi Persoalan

Pada tugas ini, mahasiswa diminta untuk mengimplementasikan model Feedforward Neural Network (FFNN) from scratch. Model ini harus mencakup proses forward propagation dan backward propagation, sebagaimana yang telah dipelajari di perkuliahan. Model diimplementasikan dalam bahasa Python dan akan diuji bagaimana pengaruh kedalaman, fungsi aktivasi, learning rate, bobot inisial, dan dibandingkan dengan implementasi yang sudah ada (`sklearn`).

Pembahasan

Penjelasan Implementasi

Pada subbab ini akan dijelaskan tentang implementasi model FFNN.

Deskripsi kelas beserta deskripsi atribut dan methodnya

Implementasi kami terdiri dari 5 *file* utama yang dikemas menjadi modul `lib`, yaitu `activation.py`, `ffnn.py`, `loss.py`, `neural.py`, dan `weight_initializer.py`.

```
lib
├── activation.py
├── ffnn.py
├── loss.py
├── neural.py
├── weight_initializer.py
└── __init__.py
```

Berikut penjelasan masing-masing *file*.

1. `activation.py` – `loss.py` – `weight_initializer.py`

   Ketiga *file* ini berturut-turut berisi kelas fungsi aktivasi, kelas fungsi *loss*, dan kelas fungsi inisialisasi bobot. Ketiga kelas ini masing-masing diabstraksikan menjadi *abstract base class* dan mempunyai fungsi-fungsi turunan yang merupakan varian dari masing-masing fungsi. Contoh, kelas `Activation` pada `activation.py` mempunyai beberapa kelas turunan, salah satunya `Sigmoid`.

   Berikut adalah definisi *abstract base class* untuk masing-masing fungsi dengan masing-masing metodenya yang akan diimplementasi kelas turunannya.

```python
# file: activation.py

class Activation(ABC):
    """Base class for activation functions"""
    @abstractmethod
    def function(self, x: np.ndarray) -> np.ndarray:
        raise NotImplementedError

    @abstractmethod
    def derivative(self, x: np.ndarray) -> np.ndarray:
        raise NotImplementedError
```

```python
# file: loss.py

class Loss(ABC):
    @abstractmethod
    def function(self, y_true: np.ndarray, y_pred: np.ndarray) -> float:
        raise NotImplementedError

    @abstractmethod
    def derivative(self, y_true: np.ndarray, y_pred: np.ndarray) ->
np.ndarray:
        raise NotImplementedError
```

```python
# file: weight_initializer.py

class WeightInitializer(ABC):
    """Abstract base class for neural network weight initialization."""
    @abstractmethod
    def initialize(self, shape):
        raise NotImplementedError
```

2. `neural.py`

   File ini berisi kelas dan struktur data untuk *layer* dan *neural network*. Pada implementasi kami, dibuat sebuah kelas untuk masing-masing objek tersebut. Agar tidak terlalu panjang, kami hanya lampirkan definisi kelas dan informasi yang disimpan saja.

```
class NetworkLayer:
    """A layer in a neural network"""

    nodes: np.ndarray
    activation: act.Activation
    activated_nodes: np.ndarray

    def __init__(self, node_count: int, activation: act.Activation):
        """
        Initialize a network layer

        Args:
            node_count: Number of nodes in this layer
            activation: Activation function for this layer
        """
        self.activation = activation
        self.nodes = np.zeros(node_count)
        self.activated_nodes = np.zeros(node_count)
```

```
class NeuralNetwork:
    loss_function: loss.Loss
    layers: List[NetworkLayer]
    weights: List[np.ndarray] # 3D array
    gradients: List[np.ndarray] # 3D array
    bias_weights: List[np.ndarray] # 2D array
    bias_gradients: List[np.ndarray] # 2D array
```

Penting untuk dicatat bahwa bobot dan gradien disimpan dalam bentuk matriks untuk setiap antardua *layer* sehingga bentuknya array 3 dimensi. Begitupun gradien.

Pada kelas `NeuralNetwork`, terdapat metode-metode berikut.

1. `initialize_weights` untuk menginisialisasi bobot.

2. `show` untuk menampilkan gambar *neural network* (dalam bentuk graf).
3. `plot_weights` untuk menampilkan gambar *plot* bobot.
4. `plot_gradients` untuk menampilkan gambar *plot* gradien bobot.

Perhatikan bahwa tipe *weight initializer* untuk setiap *layer* tidak disimpan, melainkan dipanggil ketika *neural network* pertama kali diinstansiasi sehingga bobot langsung terinisialisasi di awal.

3. `ffnn.py`

   Pada *file* ini didefinisikan model *Feedforward Neural Network* yang sesungguhnya, dengan metode-metode yang umum untuk model *neural network* seperti `fit`, `evaluate`, `save`, dan `load`. Fungsi `forward_prop` dan `back_prop` juga diimplementasi di kelas ini. Pada kelas ini, hanya disimpan *neural network* dan *loss history*. Untuk metode-metodenya, kami tidak jelaskan secara menyeluruh di sini sebab akan dijelaskan di Penjelasan forward propagation dan Penjelasan backward propagation.

   ```
   class FFNN:
       network: NeuralNetwork
       loss_history: Dict[str, List[float]]
   ```

Penjelasan forward propagation

Forward propagation adalah proses mengisi nilai-nilai bobot dan nilai-nilai aktivasi bobot. Proses ini cukup simpel, yaitu untuk setiap *layer* ke-i, kita bisa mendapatkan *layer* ke-i+1 dengan mengaktivasi *layer* ke-i kemudian mengalikannya dengan matriks bobot ke-i. Jika diilustrasikan dengan *pseudocode*, kurang lebih seperti berikut.

```
for i from 1 to N-1 do:
    layer[i + 1] = weight_matrix[i] @ activate(layer[i]) + bias[i + 1]
```

Dengan N adalah banyaknya *layer*, `activate` adalah fungsi aktivasi, dan `@` adalah operator perkalian matriks. Implementasi model kami bisa disamakan dengan *pseudocode* di atas.

Penjelasan backward propagation dan weight update

Sederhananya, *backward propagation* adalah menghitung turunan parsial bobot terhadap fungsi *loss* agar dapat menggunakannya untuk metode *gradient descent*. Dengan kata lain, kita ingin tahu masing-masing bobot itu **seberapa berpengaruh ke hasil *output*** sehingga bobot bisa dimodifikasi agar hasilnya lebih akurat.

Kami tidak akan jelaskan *backward propagation* dari awal, sebab hal tersebut tidak relevan ke laporan ini. Tujuan laporan ini hanya untuk menjelaskan implementasi kami saja.

Pertama, untuk *layer* terakhir, rumus *error term*-nya (disebut dengan **delta**) adalah:

$$\textbf{delta} = A'(x) \times L'(x)$$

Dengan $A'(x)$ adalah turunan dari aktivasi dan $L'(x)$ adalah turunan dari *loss*. Kedua ini sudah didefinisikan berturut-turut di `activation.py` dan `loss.py`. Kemudian, untuk mendapatkan **dW** atau gradien, kita kalikan dengan nilai aktivasi pada *layer* $N - 1$ (N adalah jumlah *layer*).

Untuk memperdetail lagi, implementasi kami menggunakan perkalian matriks dengan `np.dot` agar mempermudah sintaks. Berikut adalah potongan kode untuk proses yang telah dideskripsikan.

```
if (d_activation.ndim == 3): # softmax yea

    output_sz, batch_sz = d_loss.shape
    delta = np.zeros((output_sz, batch_sz))

    for i in range(batch_sz):
        delta[:,i] = np.dot(d_activation[:,:,i], d_loss[:,i])
        # Let there be B data in a batch, then:
        # d_activation[:,:,i] is the i'th Jacobian matrix (N x N)
        # d_loss[:,i] is the i'th loss gradient vector (N x 1)

else:
    delta = d_activation * d_loss
```

Perhatikan bahwa kasus untuk *softmax* dibedakan, tetapi ini bukan karena itu adalah kasus spesial, melainkan karena turunan *activation function* selain *softmax* tidak berbentuk *Jacobian matrix*. Sebenarnya, turunan fungsi aktivasi yang lain juga berbentuk *Jacobian matrix*, tetapi karena hasil matriksnya adalah matriks diagonal, efisiensi bisa ditingkatkan dengan menyimpannya sebagai vektor kolom saja, kemudian perkalian dengan turunan *loss* dilakukan secara *component wise*, bukan perkalian matriks.

Note: untuk meng-*handle* hal ini, pada kode di atas, `d_activation` bersifat fleksibel. Jika *softmax* digunakan, `d_activation` berbentuk matriks, dan jika tidak, `d_activation` berbentuk vektor kolom.

Kemudian untuk gradien bobot pada *layer-layer* berikutnya, proses yang mirip dilakukan. Perbedaannya hanya pada rumus **delta**, yaitu mempertimbangkan nilai **delta** pada *layer* sebelumnya:

$$\textbf{delta} = \text{previous\_delta} \times A'(x)$$

Terakhir, dilakukan proses *update* bobot, yaitu untuk setiap bobot $\mathbf{w}$, akan dihitung perubahan bobot yang akan dilakukan, $\boldsymbol{\Delta w}$ dengan rumus berikut.

$$\boldsymbol{\Delta w} = \mathbf{dW} \times \eta$$

Dengan η adalah *learning rate* dan $\mathbf{dW}$ adalah gradien bobot (sudah dijelaskan).

Hasil Pengujian

Pengaruh depth dan width

Eksperimen dilakukan dengan tiga variasi **kedalaman** jaringan, yaitu 3 lapisan, 4 lapisan, dan 5 lapisan dengan konfigurasi sebagai berikut:

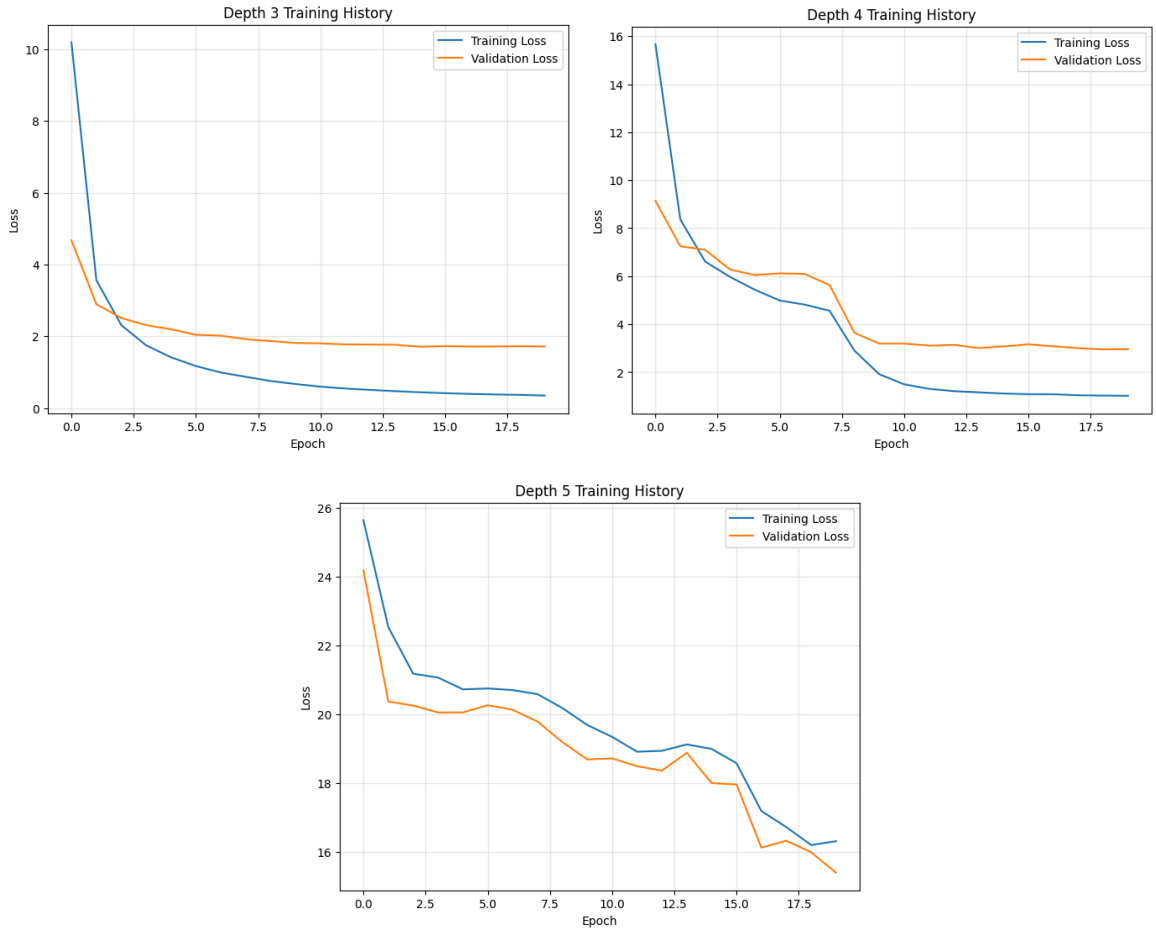
1. Kedalaman 3 lapisan: 784 → 156 → 10
2. Kedalaman 4 lapisan: 784 → 156 → 156 → 10
3. Kedalaman 5 lapisan: 784 → 156 → 156 → 156 → 10

Hasil evaluasi performa model berdasarkan metrik akurasi, presisi, recall, dan F1-score ditunjukkan pada tabel berikut:

Depth	Accuracy	Precision	Recall	F1 Score
3	0.8233000000	0.8244363618	0.8216177327	0.8221891758
4	0.8514000000	0.8519176168	0.8502652613	0.8499188507
5	0.5324000000	0.3465329181	0.5252515830	0.4043920317

Berdasarkan tabel di atas, kedalaman jaringan berpengaruh signifikan terhadap performa model. Jaringan dengan 3 lapisan memiliki performa yang cukup baik. Saat kedalaman meningkat menjadi 4 lapisan, performa model meningkat pada semua metrik, menunjukkan bahwa tambahan satu lapisan dapat membantu model menangkap pola yang lebih kompleks. Namun, ketika jaringan semakin dalam hingga 5 lapisan, performa model justru menurun drastis. Akurasi turun hingga 53,24%, dan F1-score juga anjlok, menandakan bahwa model mengalami kesulitan belajar. Hal ini kemungkinan disebabkan oleh overfitting, vanishing gradient, atau kesulitan dalam konvergensi selama pelatihan.

Grafik loss pelatihan seperti berikut,



Berdasarkan grafik loss di atas, model dengan kedalaman 3 lapisan mencapai konvergensi lebih cepat, tetapi terdapat *gap* yang cukup besar antara training loss dan validation loss, yang menandakan sedikit overfitting. Saat kedalaman ditingkatkan menjadi 4 lapisan, baik training loss maupun validation loss turun lebih signifikan hingga sekitar epoch ke-10. Setelah itu, validation loss mulai stagnan dan sedikit meningkat, yang mengindikasikan tanda awal overfitting. Namun, ketika model diperbesar menjadi 5 lapisan, loss menjadi tidak stabil dan model gagal mencapai konvergensi. Hal ini kemungkinan besar terjadi karena vanishing gradient problem, di mana nilai gradien yang sangat kecil menyebabkan lapisan terdalam sulit diperbarui selama proses backpropagation. Akibatnya, model kesulitan belajar dan tidak dapat bekerja dengan baik.

Kemudian, eksperimen dilakukan dengan tiga variasi **lebar** jaringan (banyak neuron pada hidden layer) dengan konfigurasi sebagai berikut:

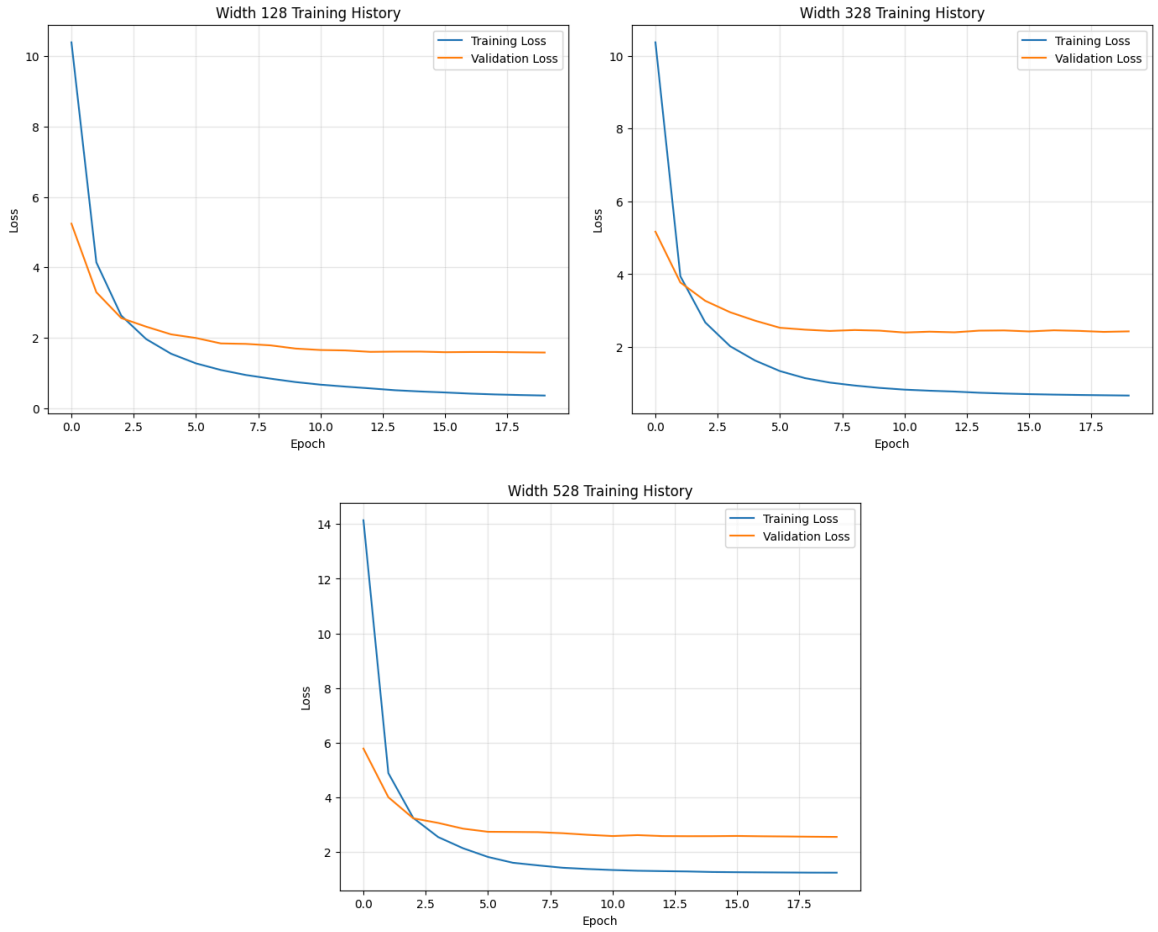
1. Jaringan 1: 784 → 128 → 10
2. Jaringan 2: 784 → 328 → 10
3. Jaringan 3: 784 → 528 → 10

Hasil evaluasi performa model berdasarkan metrik akurasi, presisi, recall, dan F1-score ditunjukkan pada tabel berikut:

Neuron	Accuracy	Precision	Recall	F1 Score
128	0.8307000000	0.8290685679	0.8291396182	0.8288167575
328	0.8306000000	0.8301740964	0.8292553941	0.8294908768
528	0.8462000000	0.8457486204	0.8449861291	0.8447939786

Berdasarkan tabel di atas, peningkatan jumlah neuron dalam hidden layer (lebar jaringan) berpengaruh terhadap performa model. Pada jaringan dengan 128 dan 328 neuron, performa model hampir sama, dengan akurasi, presisi, recall, dan F1-score yang tidak menunjukkan peningkatan signifikan. Namun, ketika jumlah neuron ditingkatkan menjadi 528, terjadi lonjakan performa yang cukup jelas di semua metrik, menunjukkan bahwa model mampu menangkap pola yang lebih kompleks dengan lebih baik. Hal ini menunjukkan bahwa menambah jumlah neuron dapat meningkatkan kemampuan model dalam belajar dan menggeneralisasi data.

Grafik loss pelatihan seperti berikut,



Berdasarkan grafik loss di atas, ketiga model memiliki grafik loss yang cukup mirip, yaitu mencapai konvergensi lebih cepat. Akan tetapi, terdapat gap yang cukup besar antara training loss dan validation loss, yang menandakan sedikit overfitting.

Pengaruh fungsi aktivasi

Eksperimen dilakukan dengan lima variasi fungsi aktivasi, yaitu dengan konfigurasi sebagai berikut:

1. Linear → Linear → Softmax
2. ReLU → ReLU → Softmax
3. Sigmoid → Sigmoid → Softmax
4. Tanh → Tanh → Softmax
5. ELU → ELU → Softmax
6. GELU → GELU → Softmax

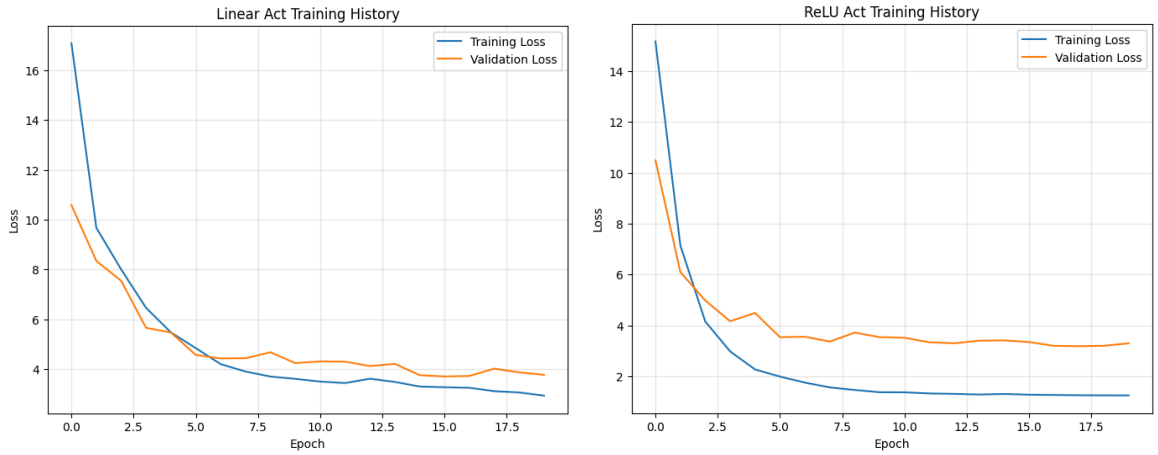
Hasil evaluasi performa model berdasarkan metrik akurasi, presisi, recall, dan F1-score ditunjukkan pada tabel berikut.

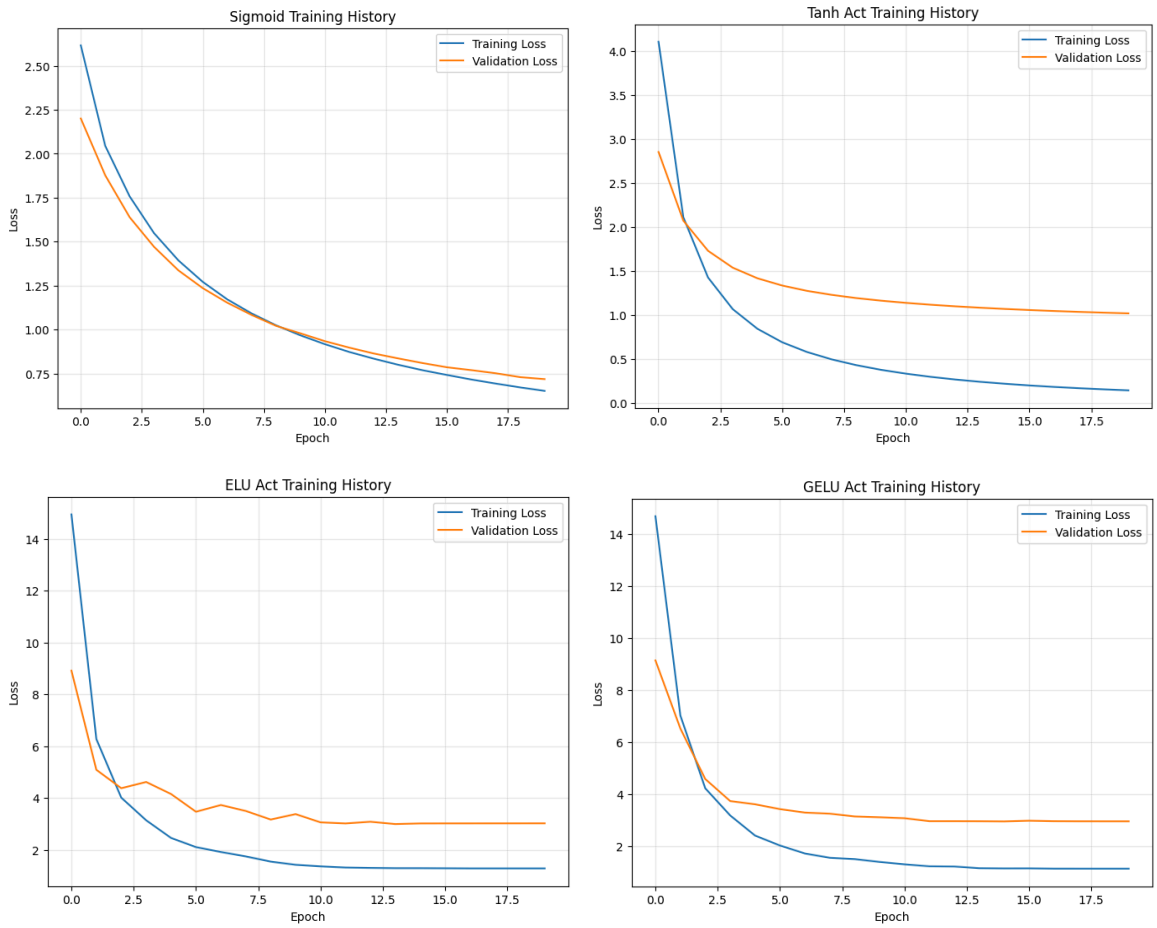
Fungsi Aktivasi	Accuracy	Precision	Recall	F1 Score
Linear	0.8664000000	0.8650109410	0.8643625849	0.8640801026
ReLU	0.8486000000	0.8476814807	0.8461530558	0.8460321352
Sigmoid	0.7793000000	0.7769024963	0.7760609573	0.7753032349
Tanh	0.7252000000	0.7206976546	0.7215156951	0.7206272879
ELU	0.8566000000	0.8548534733	0.8546078939	0.8546379122
GELU	0.8530000000	0.8508432920	0.8512221104	0.8507045616

Berdasarkan informasi di atas, fungsi aktivasi linear yang diikuti softmax menunjukkan performa terbaik dengan mencapai akurasi sebesar 86.64%. Hasil ini cukup mengejutkan mengingat fungsi linear secara teoritis memiliki kemampuan representasi yang lebih terbatas dibandingkan fungsi non-linear. Artinya, dataset yang digunakan (MNIST) cocok dengan linear.

Fungsi aktivasi ReLU dan variannya (ELU dan GELU) menunjukkan performa yang relatif baik meskipun tidak seoptimal fungsi linear. ReLU mencapai akurasi 84.86%, sementara ELU dan GELU masing-masing mencapai 85.66% dan 85.30%. Di sisi lain, sigmoid hanya mencapai akurasi 77.93%, sedangkan tanh lebih rendah lagi di 72.52%.

Grafik loss pelatihan seperti berikut,





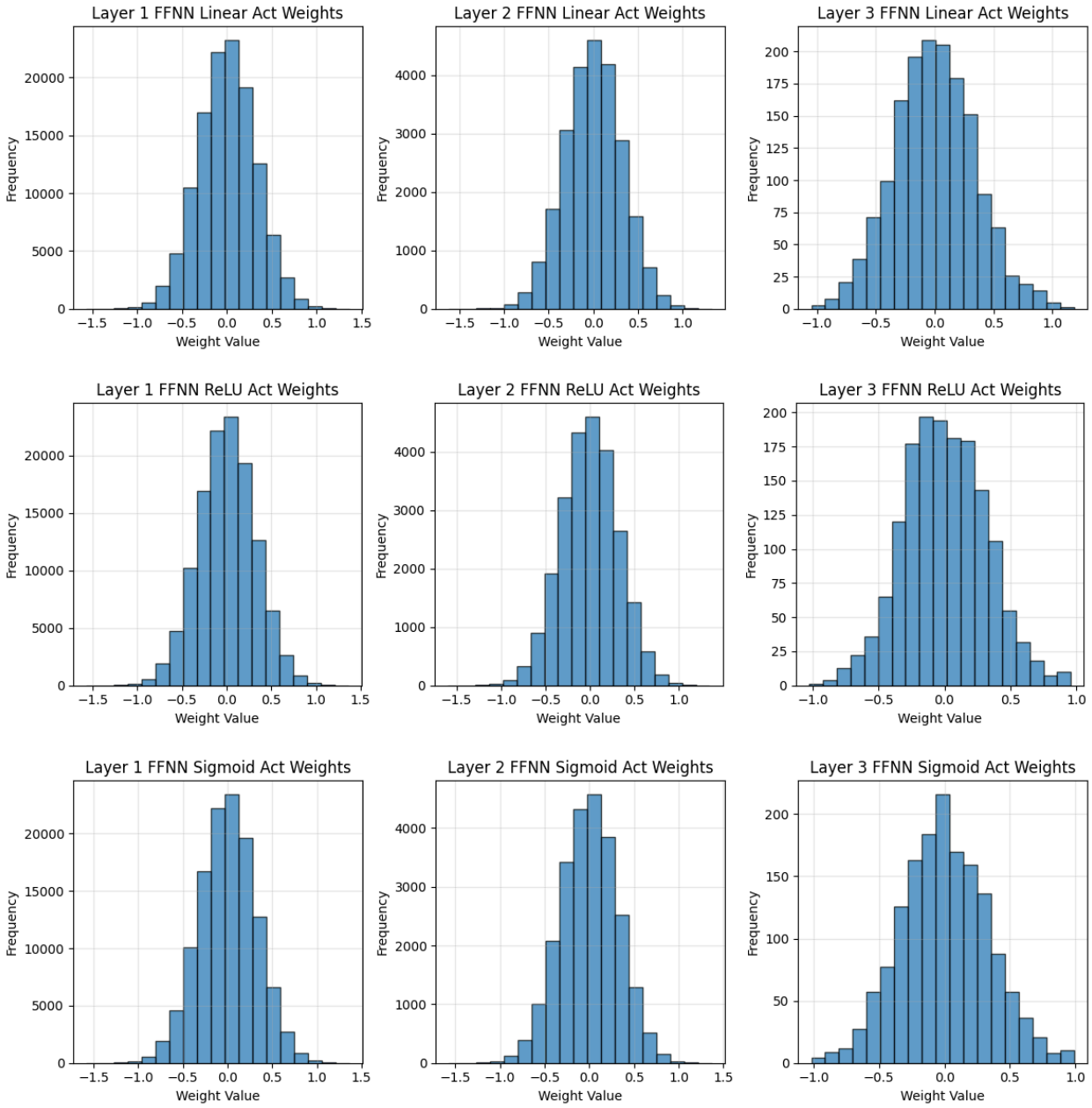
Berdasarkan grafik loss di atas, pada model dengan fungsi aktivasi linear, setelah sekitar epoch ke-5, validation loss mulai stagnan dan bahkan menunjukkan sedikit peningkatan serta fluktuasi. Fluktuasi kecil pada validation loss menunjukkan bahwa model cukup stabil, tetapi mungkin ada sedikit ketidakcocokan antara data pelatihan dan validasi. Tidak ada tanda-tanda overfitting atau underfitting yang signifikan.

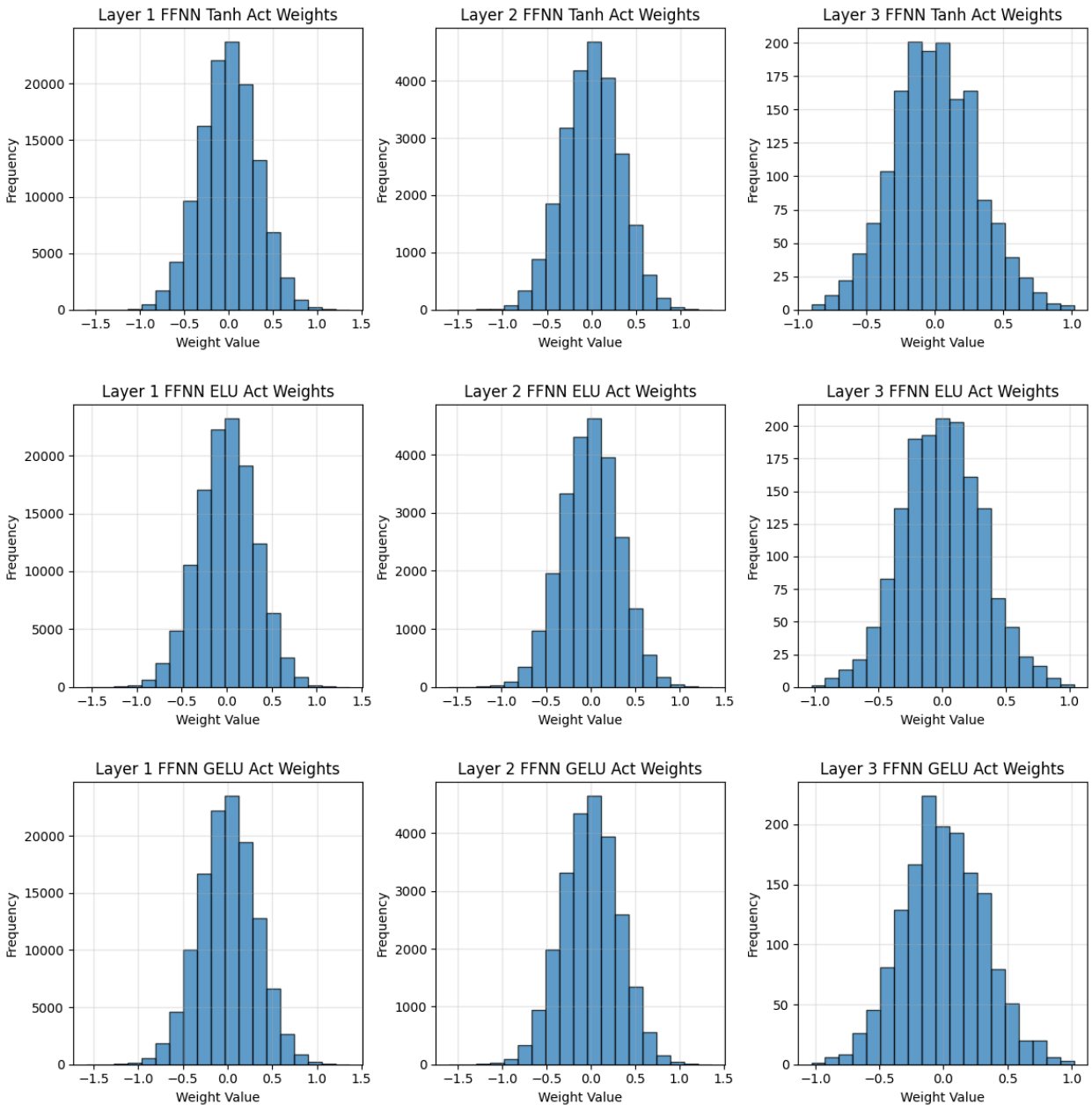
Sementara itu, model dengan fungsi aktivasi ReLU, ELU, dan GELU mencapai konvergensi lebih cepat, yakni sekitar epoch ke-9. Namun, sedikit lonjakan pada validation loss mengindikasikan adanya kecenderungan overfitting.

Pada model dengan fungsi aktivasi Sigmoid, loss terlihat lebih stabil, dengan training loss dan validation loss yang seimbang serta cenderung rendah. Meskipun demikian, penurunan loss yang berlanjut hingga akhir epoch mengindikasikan bahwa model membutuhkan waktu lebih lama untuk mencapai konvergensi, yang bisa menjadi tanda sedikit underfitting.

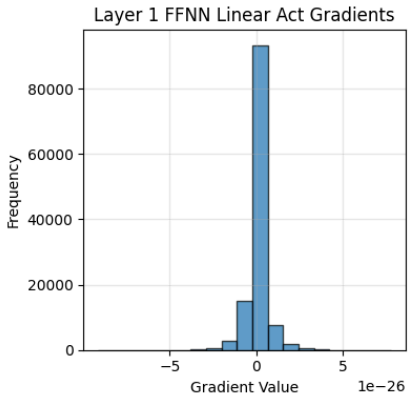
Pada model dengan fungsi aktivasi Tanh loss terus menurun di akhir plot, mungkin lama dalam mencapai konvergensi. Akan tetapi, terdapat gap yang cukup besar antara training loss dan validation loss, yang menunjukkan indikasi overfitting.

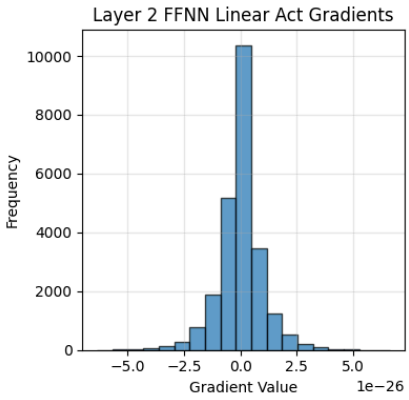
Grafik distribusi bobot seperti berikut,

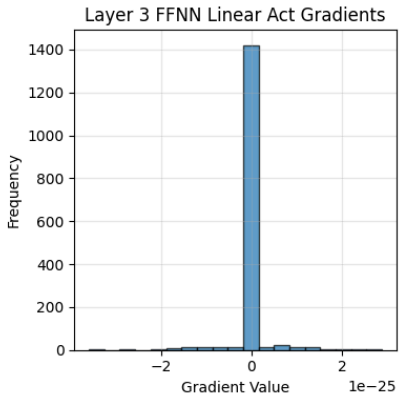


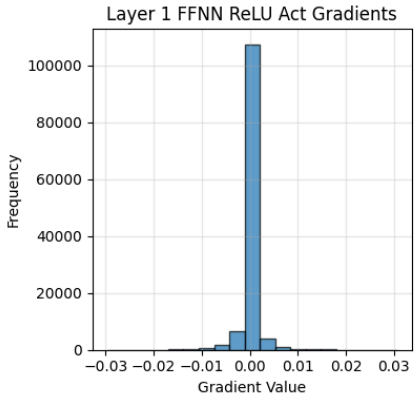


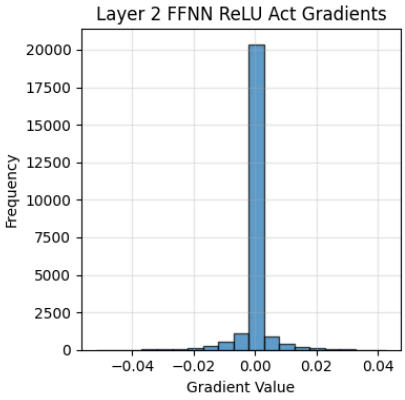
Grafik distribusi gradien seperti berikut,

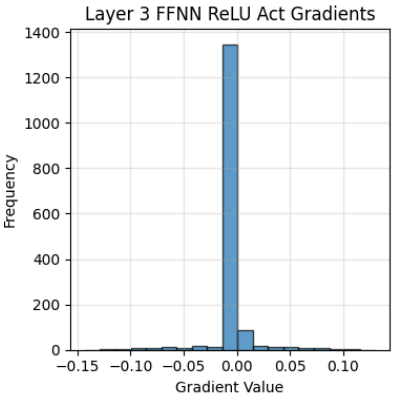

Layer 1 FFNN Linear Act Gradients
Frequency
Gradient Value
1e−26

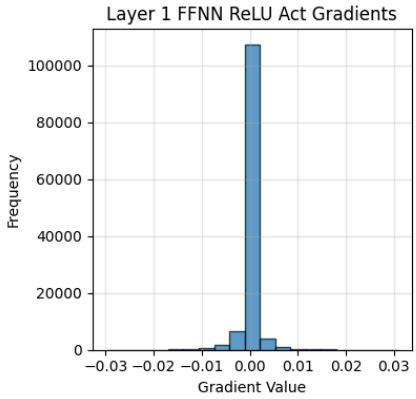

Layer 2 FFNN Linear Act Gradients
Frequency
Gradient Value
1e−26

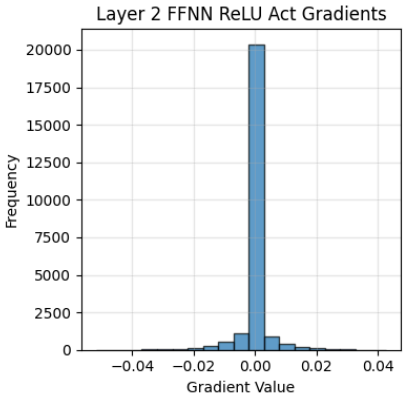

Layer 3 FFNN Linear Act Gradients
Frequency
Gradient Value
1e−25

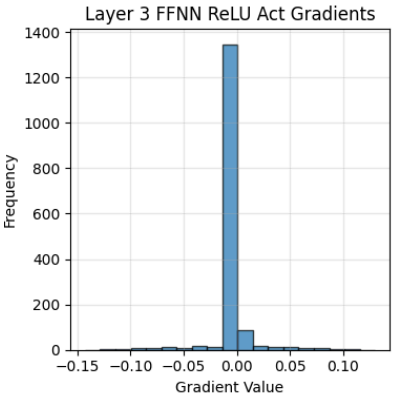

Layer 1 FFNN ReLU Act Gradients
Frequency
Gradient Value

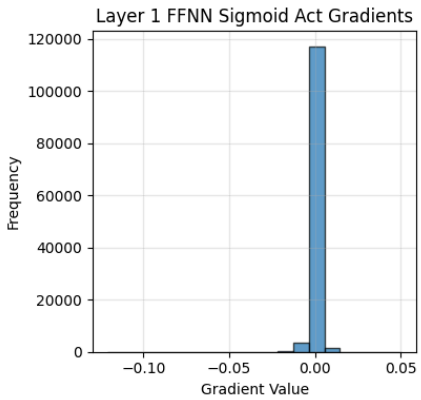

Layer 2 FFNN ReLU Act Gradients
Frequency
Gradient Value

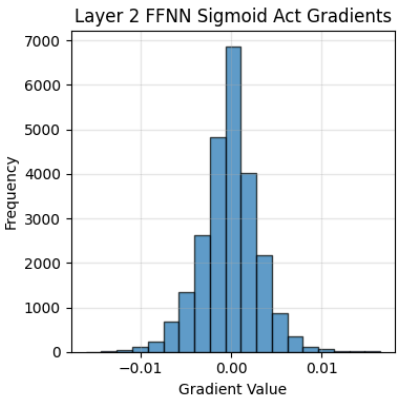

Layer 3 FFNN ReLU Act Gradients
Frequency
Gradient Value

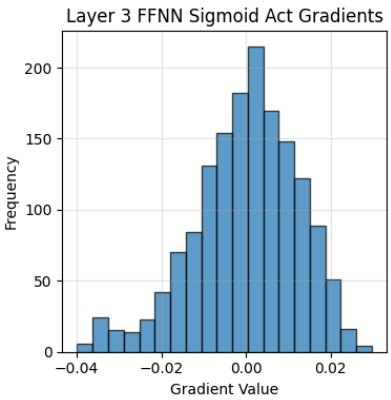

Layer 1 FFNN ReLU Act Gradients
Frequency
Gradient Value

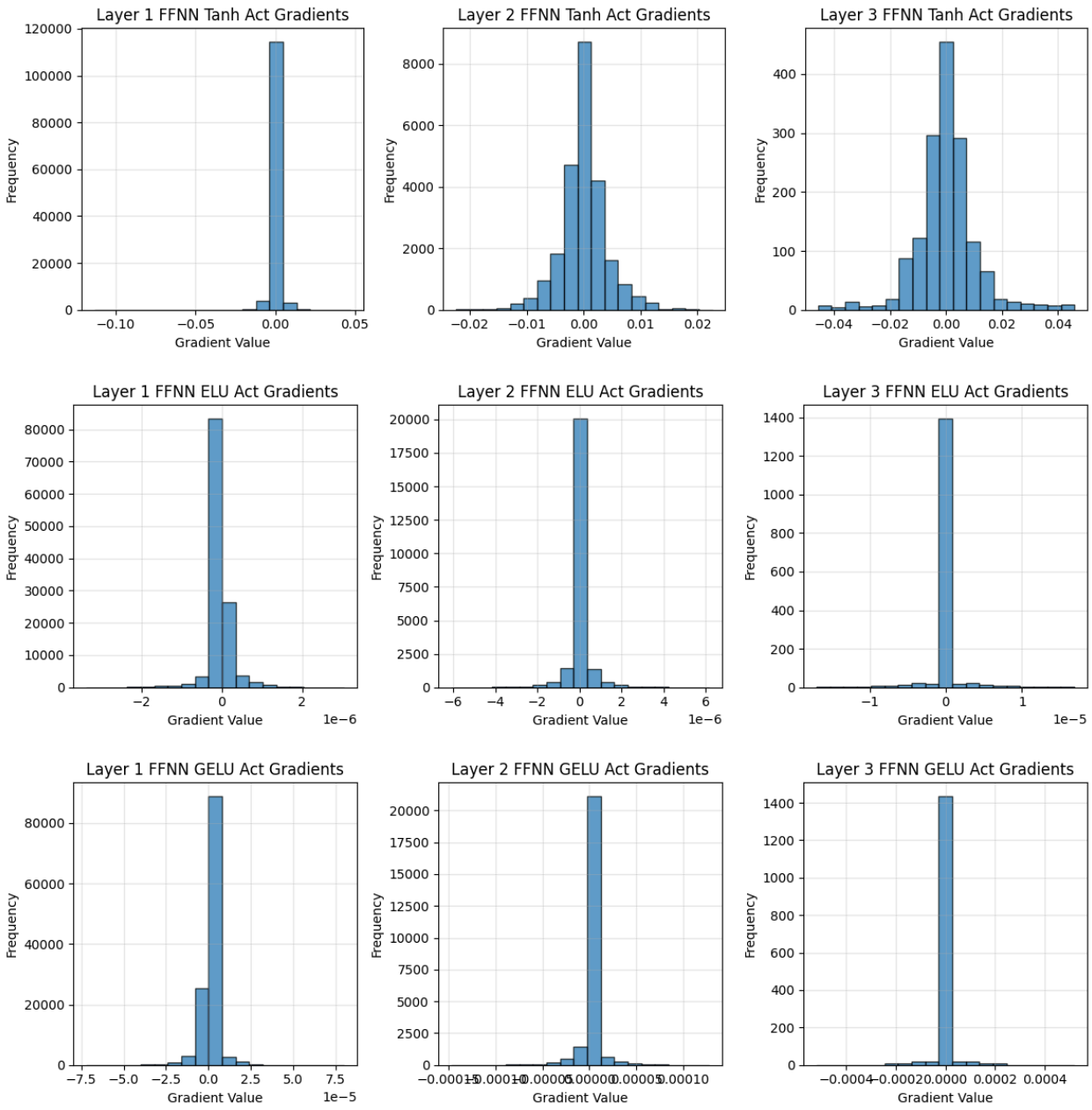

Layer 2 FFNN ReLU Act Gradients
Frequency
Gradient Value


Layer 3 FFNN ReLU Act Gradients
Frequency
Gradient Value


Layer 1 FFNN Sigmoid Act Gradients
Frequency
Gradient Value


Layer 2 FFNN Sigmoid Act Gradients
Frequency
Gradient Value


Layer 3 FFNN Sigmoid Act Gradients
Frequency
Gradient Value



Distribusi gradien dan bobot cukup normal dan tidak banyak perbedaan, dicerminkan juga dari tabel akurasinya. Adapun hal yang *worth mentioning* adalah distribusi gradien fungsi Sigmoid di *layer* ketiga yang sedikit lebih merata dari yang lain.

Pengaruh learning rate

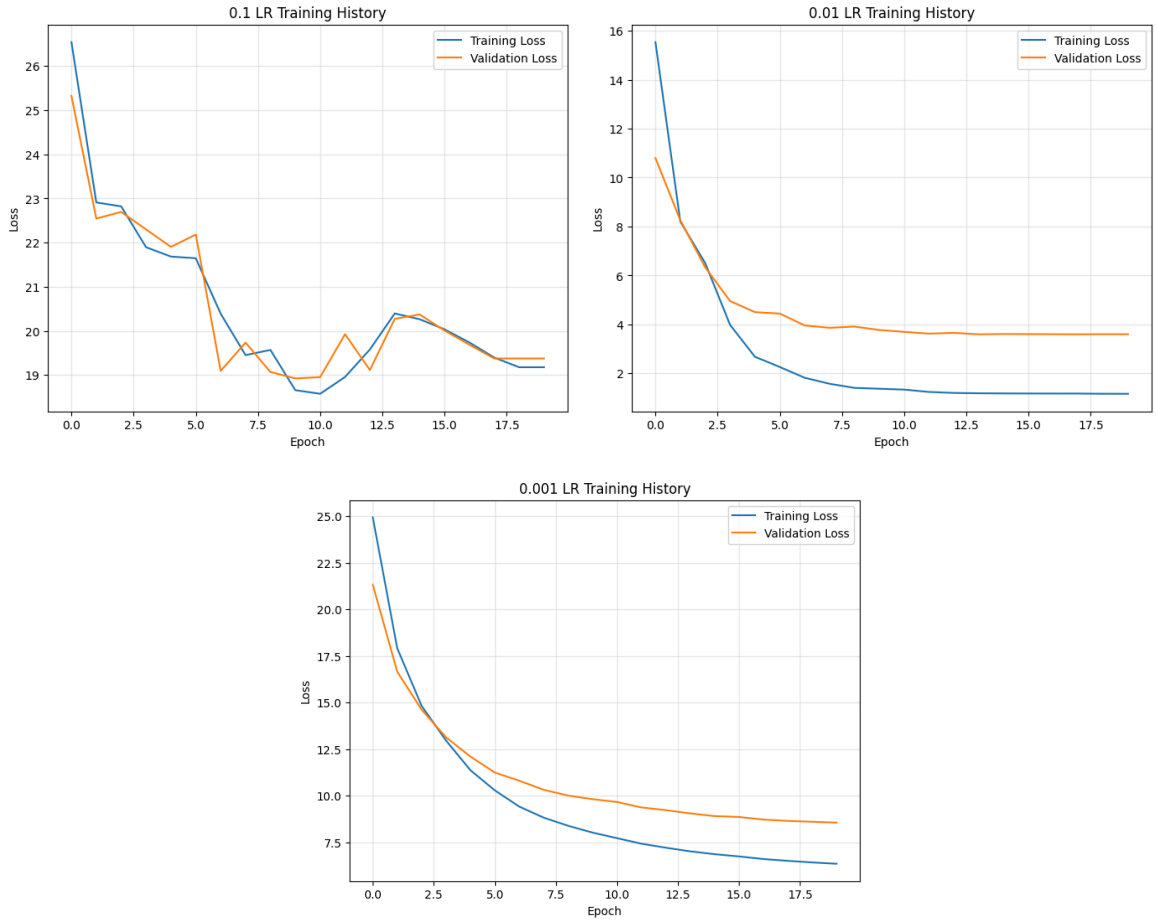
Eksperimen dilakukan dengan tiga variasi learning rate, yaitu 0.1, 0.01, dan 0.001.

Hasil evaluasi performa model berdasarkan metrik akurasi, presisi, recall, dan F1-score ditunjukkan pada tabel berikut:

Learning Rate	Accuracy	Precision	Recall	F1 Score
0.1	0.4439000000	0.2520676902	0.4377460533	0.3111700642
0.01	0.8429000000	0.8420035582	0.8414948678	0.8416514036
0.001	0.6862000000	0.6278698312	0.6841697178	0.6505221708

Hasil eksperimen menunjukkan bahwa pemilihan learning rate sangat berpengaruh terhadap performa model. Dengan learning rate yang terlalu besar (0.1), model mengalami kesulitan dalam konvergensi, sehingga performanya sangat buruk dengan akurasi hanya 44.39%. Hal ini kemungkinan karena langkah pembaruan bobot terlalu besar, menyebabkan model melewati solusi optimal tanpa bisa mencapai titik minimum yang baik. Sebaliknya, ketika learning rate diturunkan menjadi 0.01, performa model meningkat drastis dengan akurasi 84.29%, menunjukkan bahwa model mampu belajar secara efektif. Namun, saat learning rate semakin kecil menjadi 0.001, performa model justru menurun dengan akurasi 68.62%. Ini mungkin disebabkan oleh proses pembelajaran yang terlalu lambat, sehingga model belum sepenuhnya mencapai performa optimal dalam jumlah epoch yang diberikan.

Grafik loss pelatihan seperti berikut,



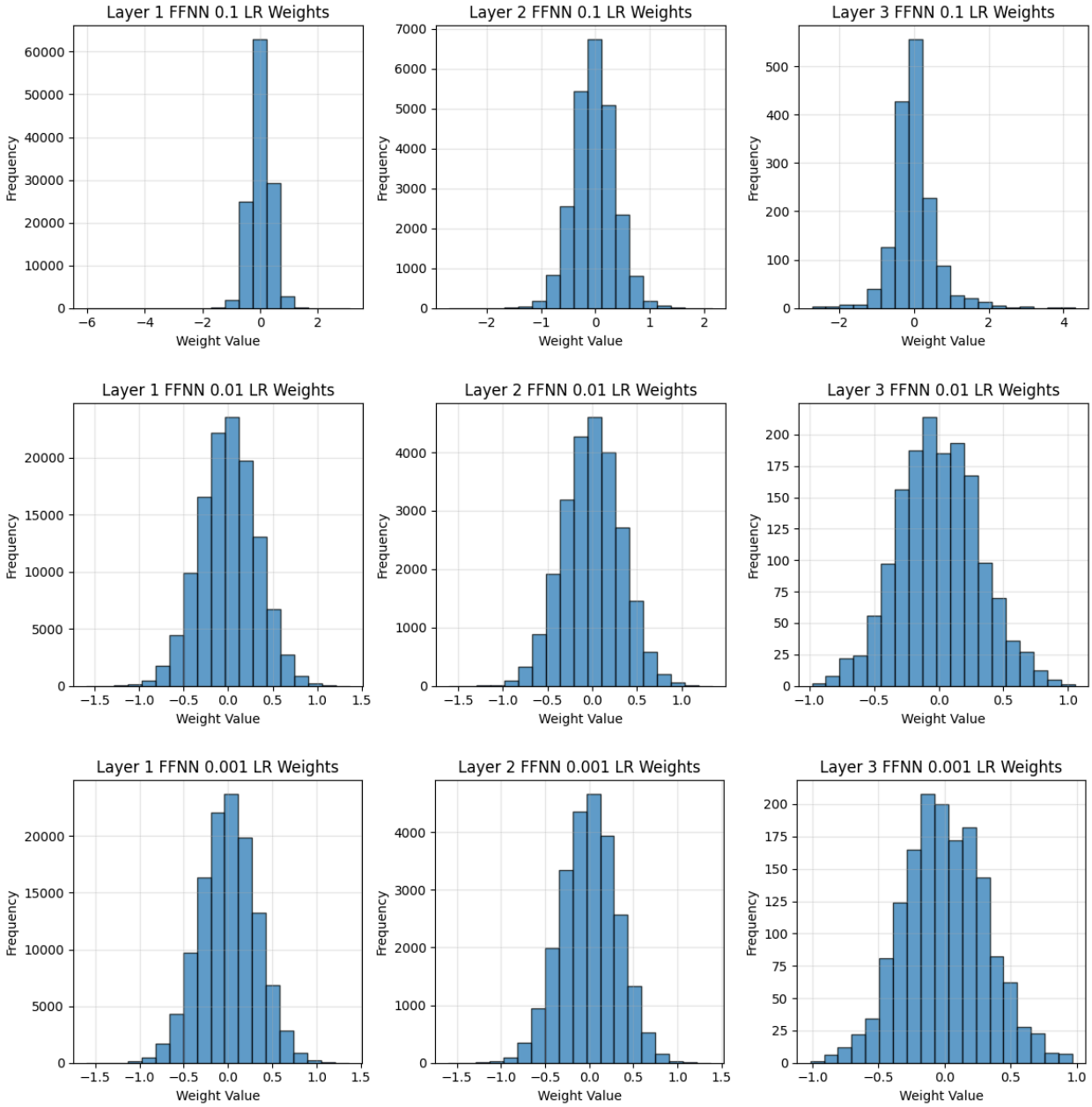
Berdasarkan grafik di atas, model dengan learning rate 0.1 mengalami fluktuasi yang sangat drastis sepanjang pelatihan. Konvergensi model tergolong buruk karena baik training loss maupun validation loss tetap tinggi dan tidak stabil. Hal ini menunjukkan bahwa model mengalami underfitting, di mana model gagal mengenali pola dalam data pelatihan maupun validasi.

Sementara itu, model dengan learning rate 0.01 mencapai konvergensi lebih cepat, sekitar epoch ke-9. Training loss menurun secara stabil, sementara validation loss juga menunjukkan tren penurunan sebelum akhirnya mulai stagnan. Karena perbedaan antara keduanya tidak terlalu besar, model tidak mengalami overfitting yang signifikan. Model masih mampu melakukan generalisasi dengan baik.

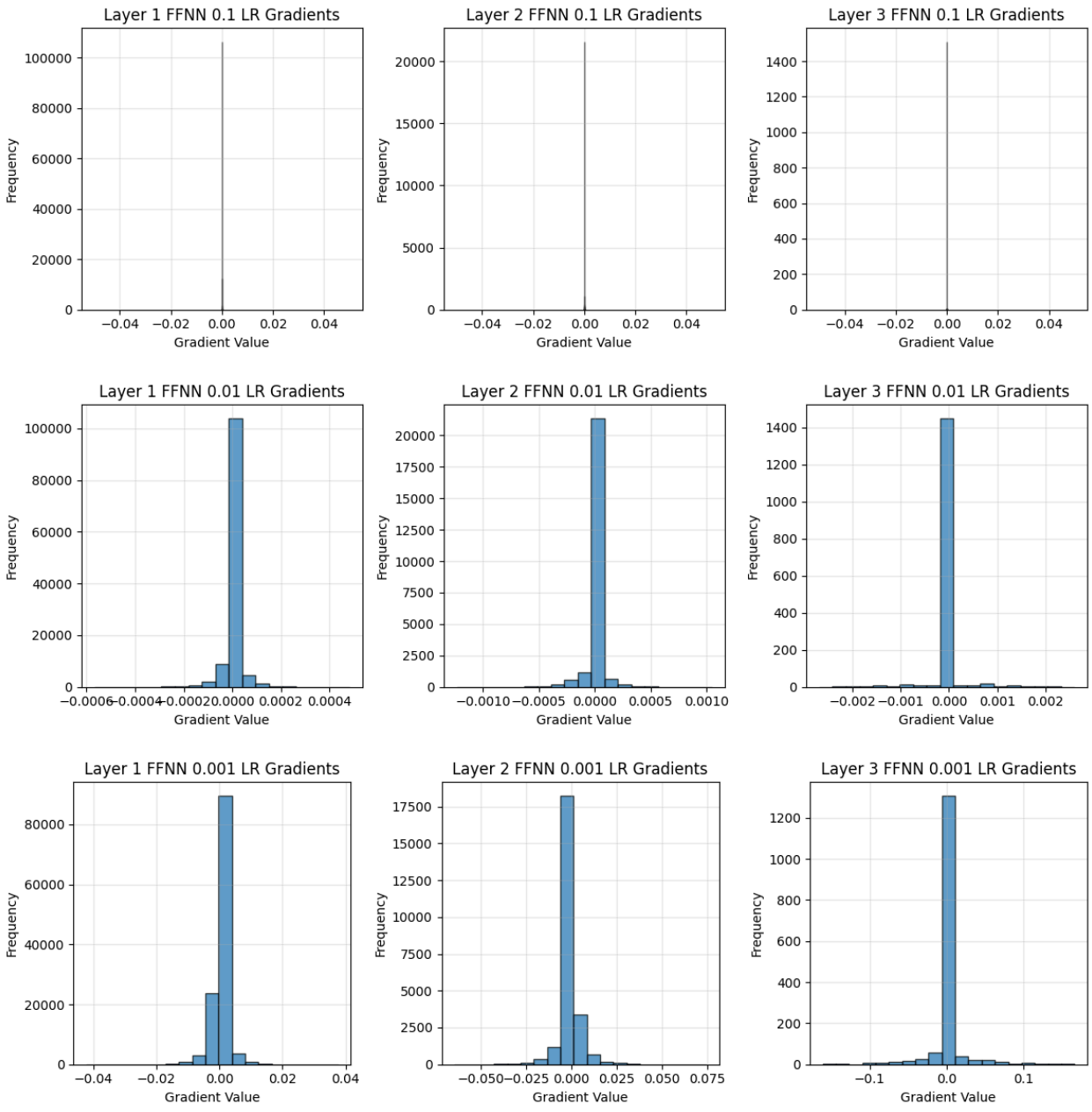
Pada model dengan learning rate 0.001, proses konvergensi berlangsung lebih lambat. Baik training loss maupun validation loss terus menurun, tetapi penurunannya terjadi secara bertahap dan belum mencapai titik stabil dalam jumlah epoch yang diberikan. Selain itu, gap antara training loss dan validation loss masih cukup besar, yang menandakan bahwa

model masih dalam tahap belajar dan belum sepenuhnya mampu melakukan generalisasi secara optimal.

Grafik distribusi bobot seperti berikut,



Grafik distribusi gradien seperti berikut,



Perhatikan bahwa *learning rate* terkecil menghasilkan gradien yang paling mendekati nol. Hal ini jelas karena *learning rate* terbesar menghasilkan *descent* yang lebih besar sehingga gradiennya akan menurun dengan lebih cepat.

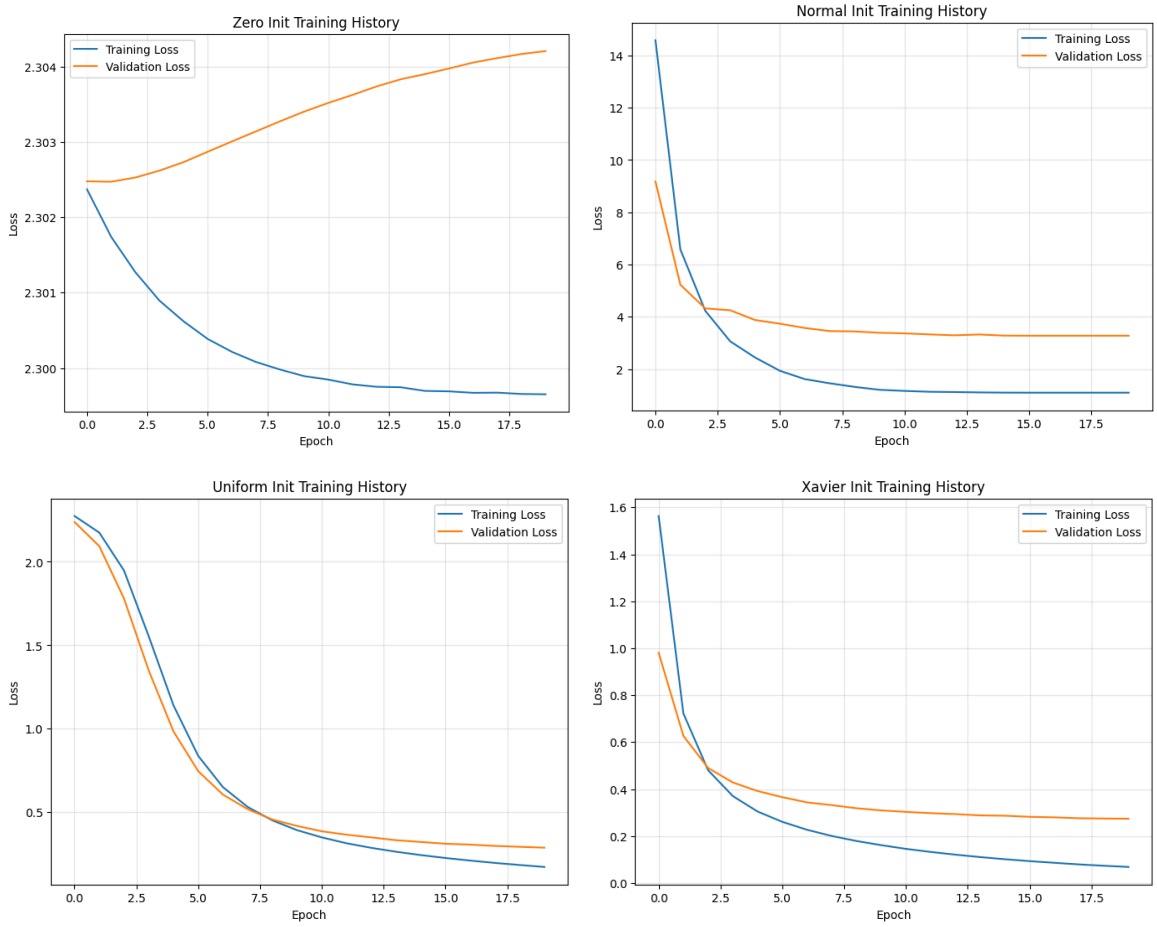
Pengaruh inisialisasi bobot

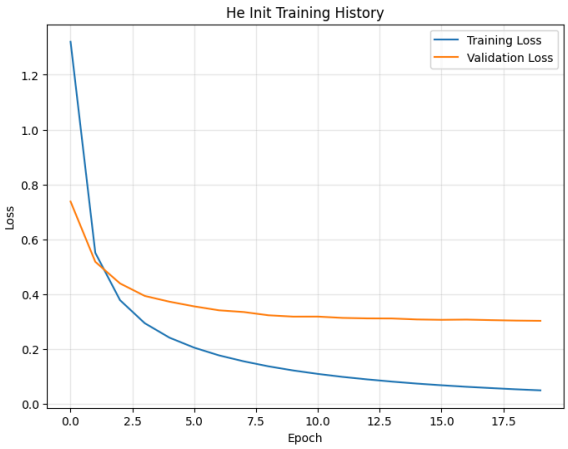
Eksperimen dilakukan dengan lima variasi metode inisialisasi bobot, yaitu `ZeroInitializer`, `NormalInitializer`, `UniformInitializer`, `XavierInitializer`, dan `HeInitializer`

Hasil evaluasi performa model berdasarkan metrik akurasi, presisi, recall, dan F1-score ditunjukkan pada tabel berikut:

Metode Inisialisasi	Accuracy	Precision	Recall	F1 Score
`ZeroInitializer`	0.1091	0.0109100000	0.1000000000	0.0196736092
`NormalInitializer`	0.8396	0.8391032168	0.8388527512	0.8386456003
`UniformInitializer`	0.8991	0.8986023715	0.8985286441	0.8984839249
`XavierInitializer`	0.9035	0.9029991755	0.9029709215	0.9028810250
`HeInitializer`	0.9025	0.9020384737	0.9019696908	0.9019155043

Grafik loss pelatihan seperti berikut,





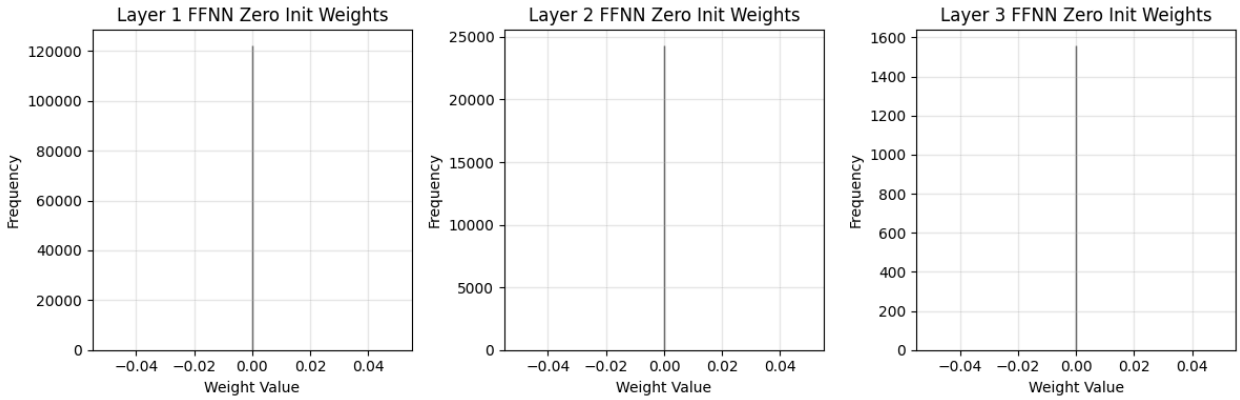
Berdasarkan grafik di atas, model dengan inisialisasi bobot `ZeroInitializer` mengalami kesulitan dalam konvergensi. Meskipun training loss perlahan menurun, validation loss justru menunjukkan tren peningkatan. Hal ini menandakan bahwa model gagal belajar secara efektif dari data yang diberikan.

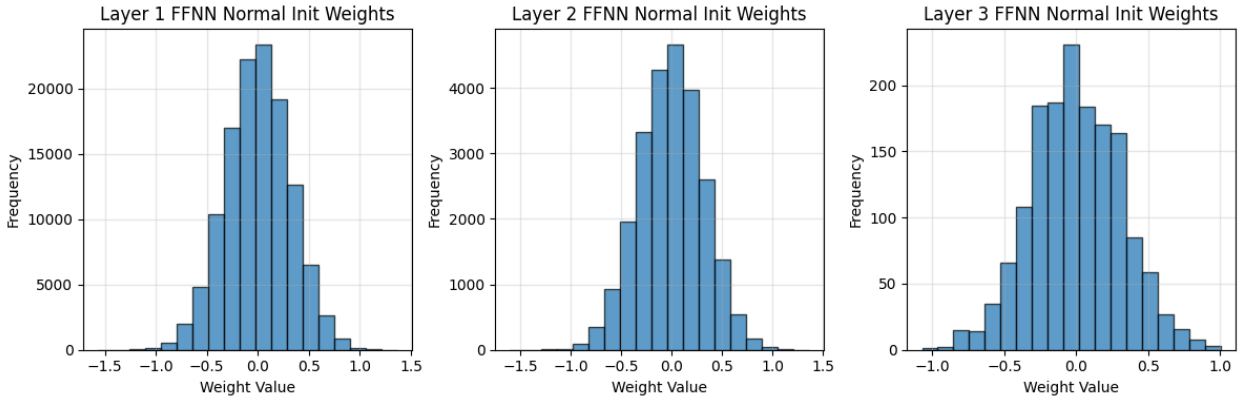
Pada model dengan inisialisasi `NormalInitializer`, konvergensi tercapai lebih cepat, sekitar epoch ke-9. Namun, terdapat sedikit perbedaan antara training loss dan validation loss, yang mengindikasikan adanya gejala overfitting.

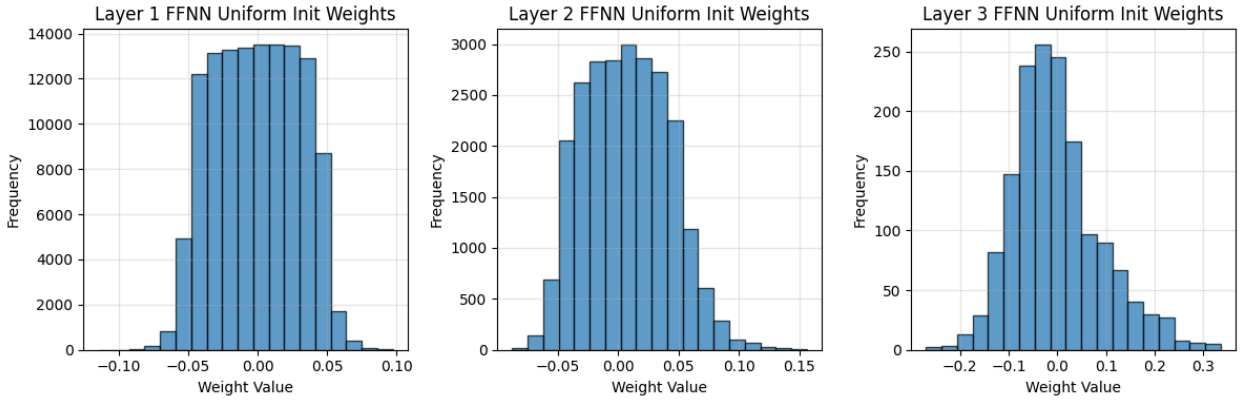
Sementara itu, model dengan inisialisasi `UniformInitializer` menunjukkan indikasi overfitting ringan. Training loss sedikit lebih rendah dibandingkan validation loss, tetapi selisihnya kecil. Model masih mampu melakukan generalisasi dengan baik karena validation loss tetap stabil dan berada pada nilai yang rendah.

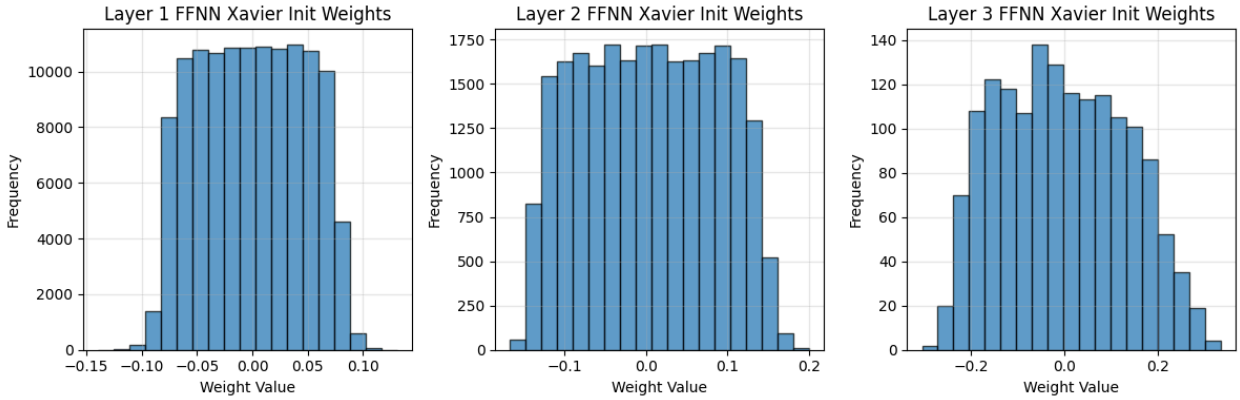
Model dengan inisialisasi `XavierInitializer` dan `HeInitializer` mencapai konvergensi lebih cepat, sekitar epoch ke-9. Training loss menurun dengan cepat, sedangkan validation loss juga mengalami penurunan, meskipun terdapat sedikit jarak antara keduanya. Hal ini menunjukkan potensi overfitting ringan, tetapi model masih dapat melakukan generalisasi dengan cukup baik karena validation loss tetap rendah dan tidak mengalami lonjakan yang signifikan.

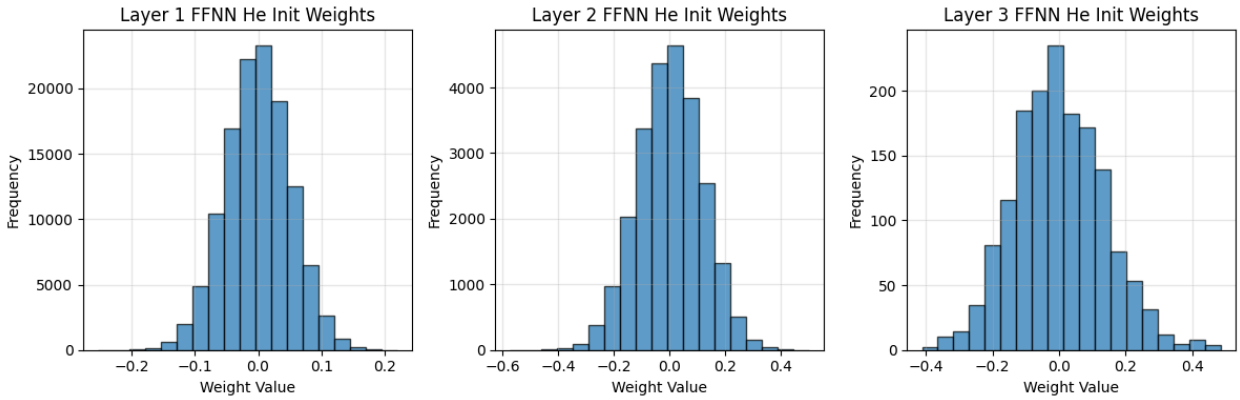
Grafik distribusi bobot seperti berikut,


Layer 1 FFNN Zero Init Weights
Frequency
Weight Value
Layer 2 FFNN Zero Init Weights
Frequency
Weight Value
Layer 3 FFNN Zero Init Weights
Frequency
Weight Value

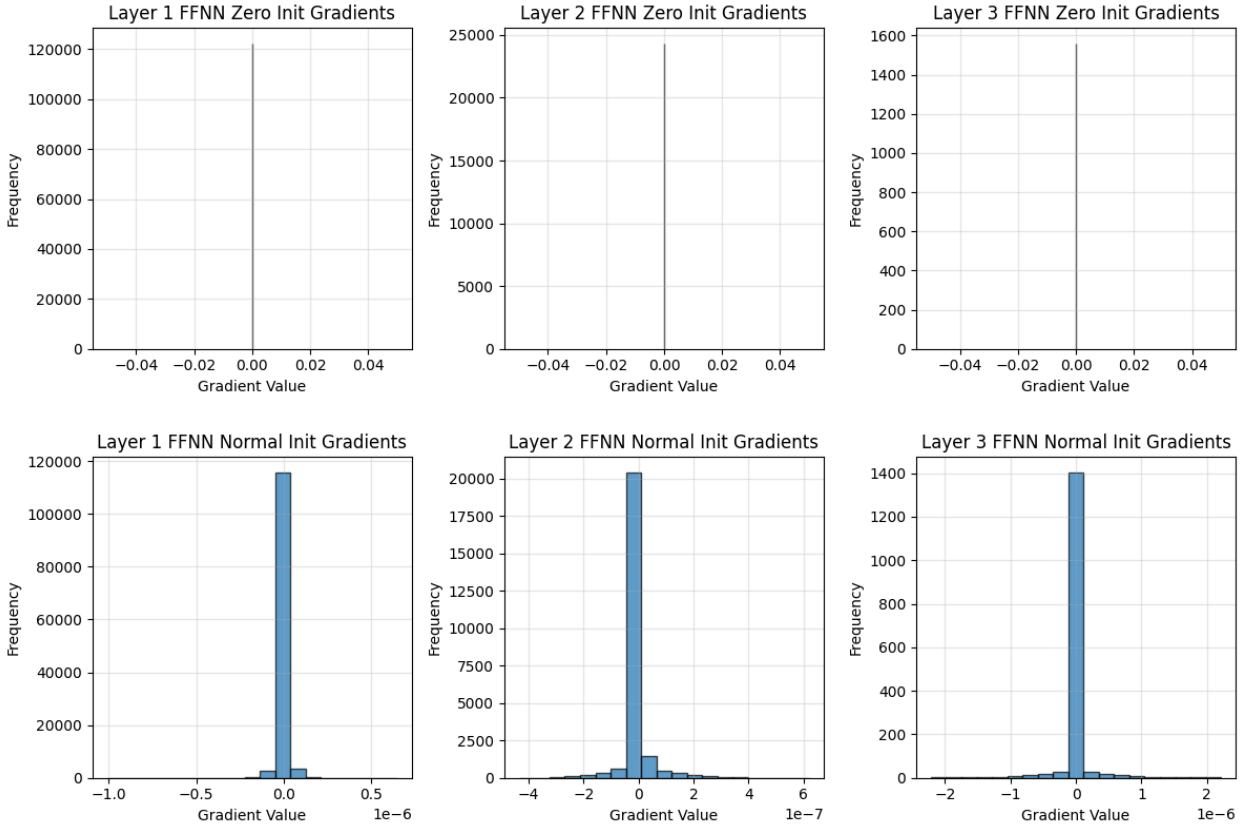

Layer 1 FFNN Normal Init Weights
Frequency
Weight Value
Layer 2 FFNN Normal Init Weights
Frequency
Weight Value
Layer 3 FFNN Normal Init Weights
Frequency
Weight Value

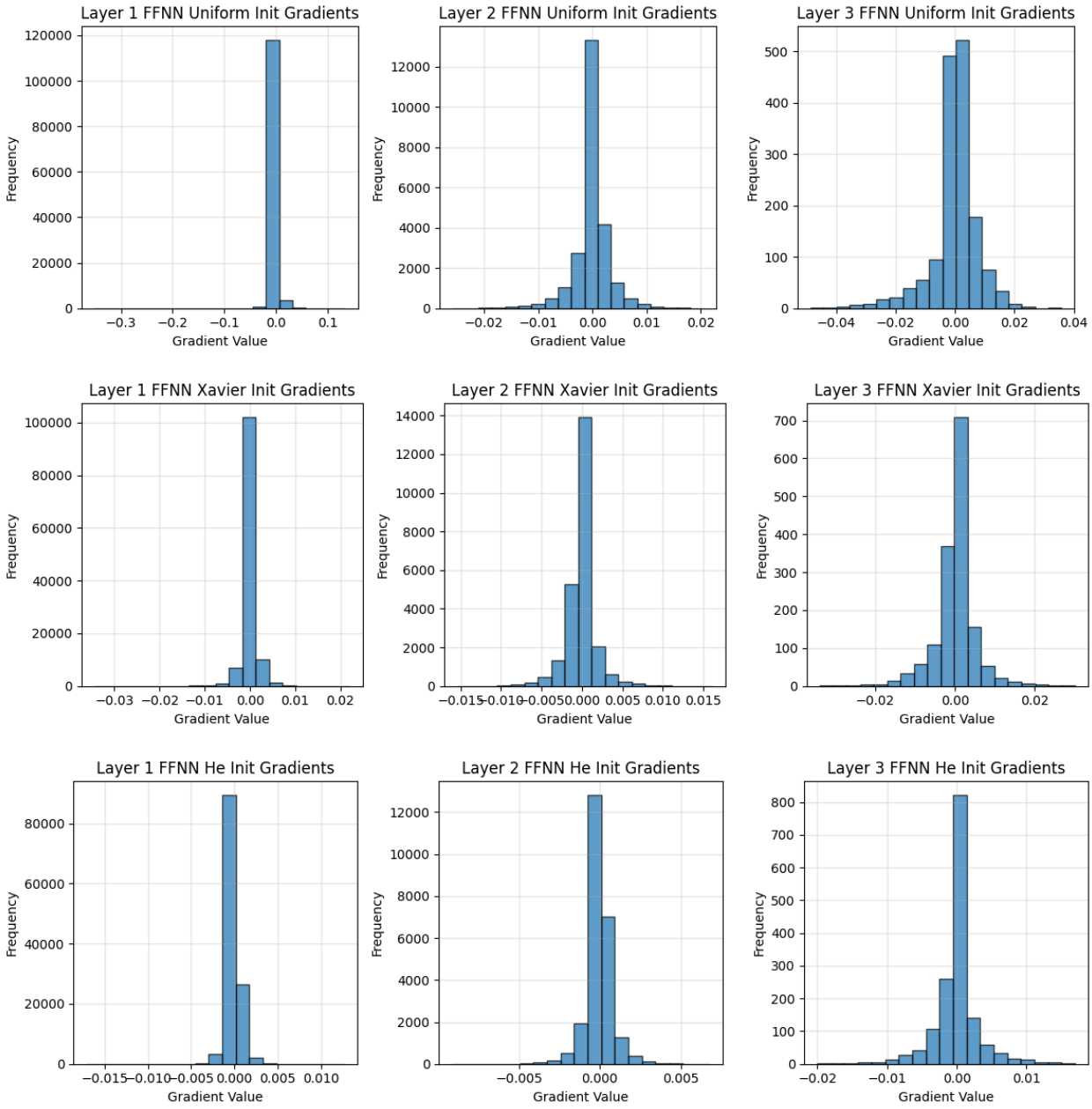

Layer 1 FFNN Uniform Init Weights
Frequency
Weight Value
Layer 2 FFNN Uniform Init Weights
Frequency
Weight Value
Layer 3 FFNN Uniform Init Weights
Frequency
Weight Value


Layer 1 FFNN Xavier Init Weights
Frequency
Weight Value
Layer 2 FFNN Xavier Init Weights
Frequency
Weight Value
Layer 3 FFNN Xavier Init Weights
Frequency
Weight Value



Grafik distribusi gradien seperti berikut,





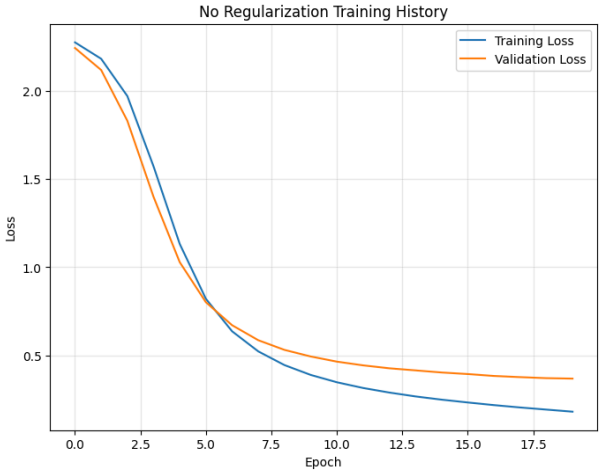
Terlihat jelas dari grafik-grafik di atas bahwa *zero initializer* bersifat **beda sendiri**. Semua bobot dan gradiennya nol sebab pada awalnya nol dan algoritma *backward propagation* tidak mampu menaikkan bobotnya.

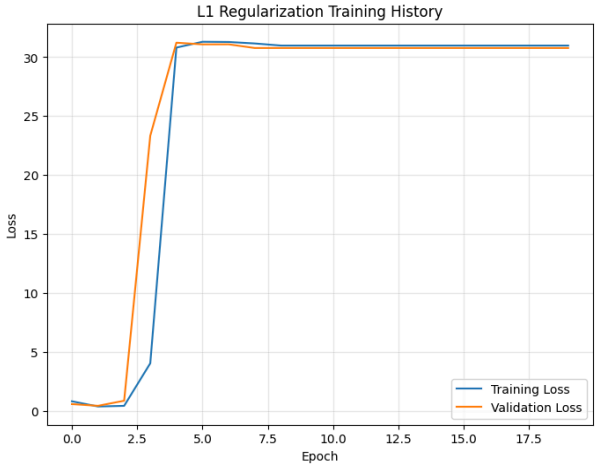
Pengaruh regularisasi

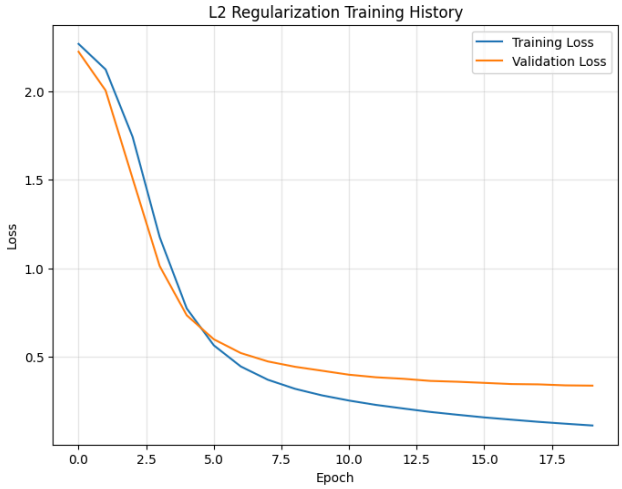
Eksperimen dilakukan dengan tiga variasi regularisasi: metode L1, metode L2, dan tanpa regularisasi. Berikut adalah evaluasi ketiga model ini.

Teknik Regularisasi	Accuracy	Precision	Recall	F1 Score
Tidak ada	0.9036000000	0.9031474808	0.9018701325	0.9018769747
L1	0.0978000000	0.0097800000	0.1000000000	0.0178174531
L2	0.9148000000	0.9138195126	0.9133908438	0.9133626332

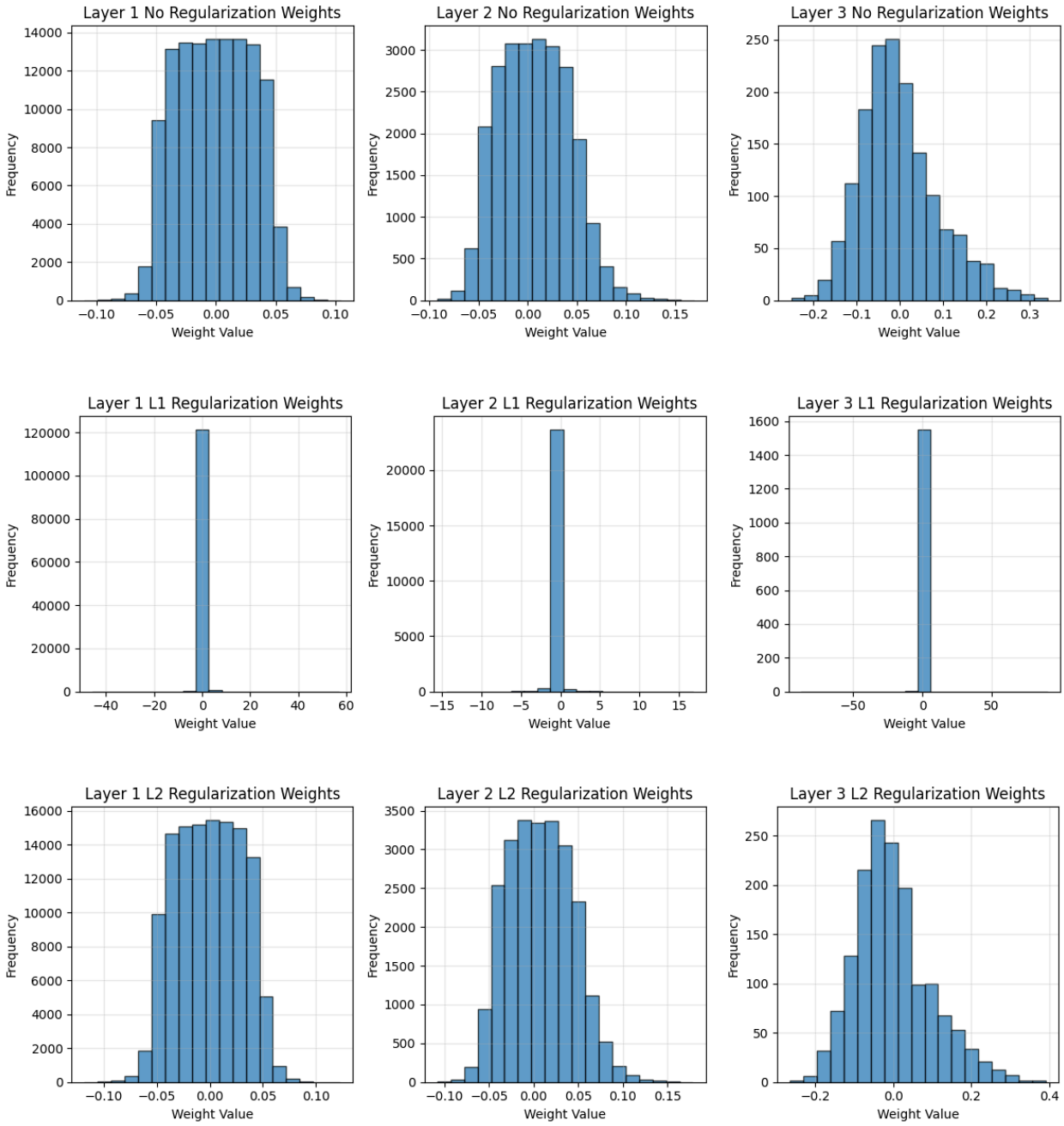
Perhatikan performa model ketika tanpa menggunakan regularisasi dengan menggunakan regularisasi. Terdapat **peningkatan** performa ketika menggunakan metode regularisasi L2, tetapi metode regularisasi L1 justru membuat performa menjadi sangat buruk. Hal ini bisa dijelaskan dengan melihat grafik loss pelatihan seperti berikut,







Kemudian distribusi bobotnya adalah sebagai berikut.

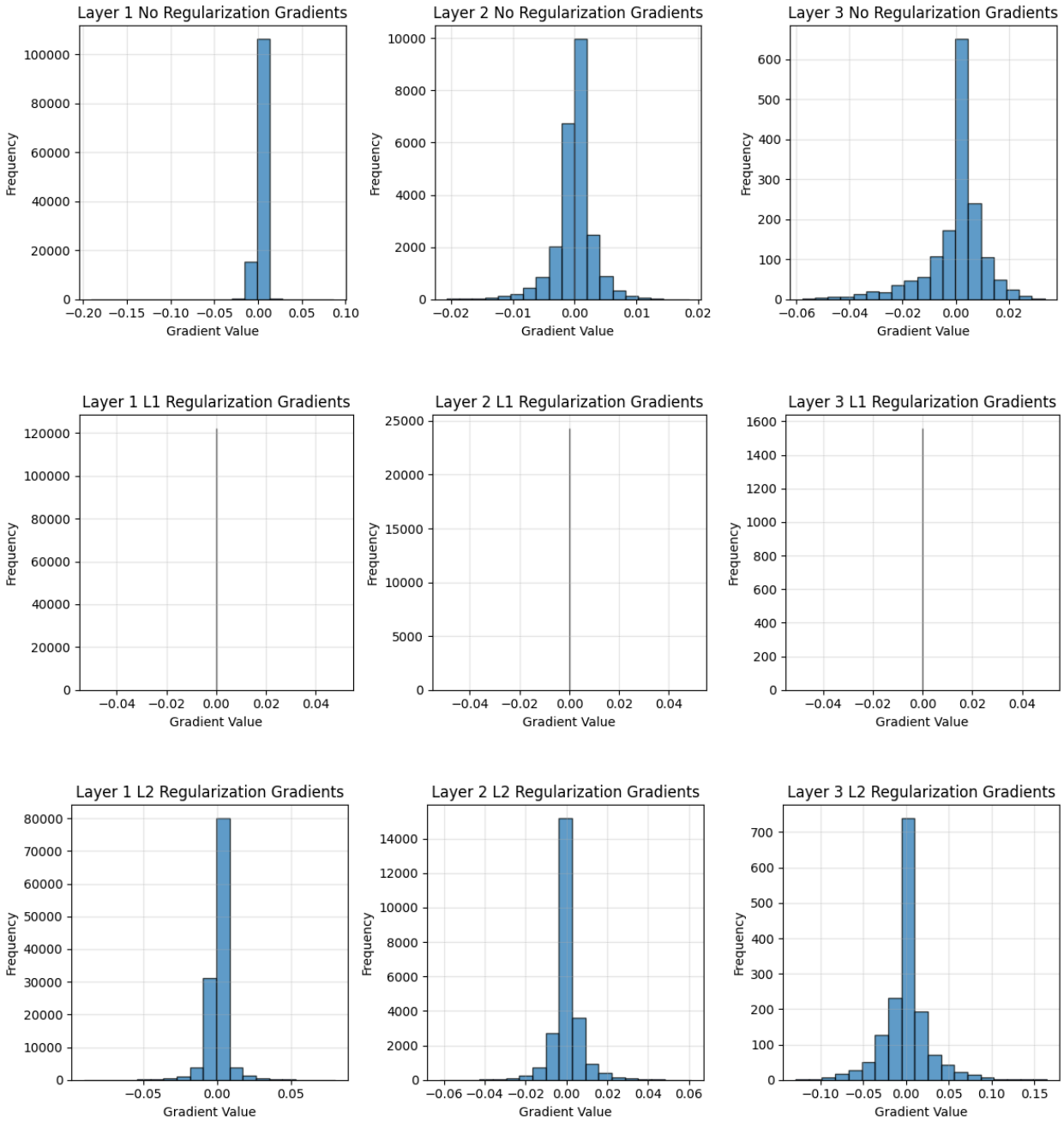


Bisa dilihat bahwa L1 mempunyai *sparsity* yang besar, seperti yang diindikasikan skala *x-axis* pada distribusi bobotnya. Banyak juga bobot yang nilainya sangat mendekati nol Kemudian, jika kita mengobservasi grafik *loss history*, metode L1 mempunyai tanjakan loss yang tajam di tengah-tengah.

Semua hasil tersebut kemungkinan besar diakibatkan karena sifat metode L1 yang menambahkan *sparsity* pada model. Artinya, nilai bobot mungkin lebih tersebar, kemudian metode ini seolah-olah melakukan “seleksi” sejumlah bobot yang akan berpengaruh besar,

sedangkan bobot yang lain diberikan pengaruh yang kecil. Ini menjelaskan kenapa tiba-tiba ada tanjakan di tengah, artinya nilai dari beberapa bobot telah meningkat signifikan sedangkan yang lainnya menjadi insignifikan.

Distribusi gradien adalah sebagai berikut. Bisa dilihat juga bahwa metode L1 sendiri yang sudah mempunyai gradien nol pada semua bobot, artinya mencapai kekonvergenan lebih cepat.

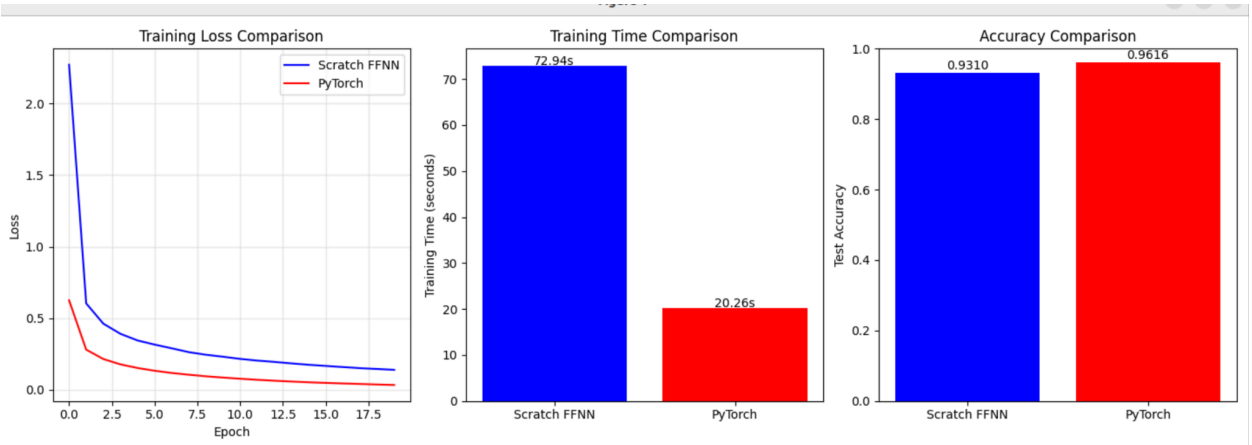


Perbandingan dengan library sklearn

Terakhir, bagaimana hasilnya apabila kami membandingkan model FFNN yang kami buat dari *scratch* dengan model FFNN pada sklearn dengan konfiugrasi yang hampir sama, yaitu tidak menggunakan regularisasi, dan aturan *default* lainnya.

Model	Accuracy	Precision	Recall	F1 Score
Our FFNN	0.9100000000	0.9093596145	0.9087720831	0.9086435034
SKLearn FFNN	0.9280952381	0.9276115993	0.9271996335	0.9273010257

Data tersebut menunjukkan bahwa tidak ada perbedaan signifikan antara implementasi kami dengan implementasi SKLearn, perbedaan sebanyak ~2% pada setiap bagian dapat disebabkan karena perbedaan inisialisasi bobot awal maupun konfigurasi-konfigurasi default SKlearn lainnya yang mungkin kami belum ketahui. Perbandingan yang mungkin lebih *fair* adalah membandingkan implementasi kami dengan milik pytorch. Hasil perbandingannya adalah sebagai berikut.



Tetap tidak terlalu sama, namun lebih mendekati, hal ini dapat terjadi karena internal dari pytorch dapat menggunakan simplifikasi beberapa pendekatan, seperti simplifikasi delta apabila menggunakan CCE sebagai loss function dan Softmax sebagai activation function.

Kesimpulan dan Saran

1. Kedalaman (depth) jaringan sangat memengaruhi performa model. Jika terlalu dalam, model berisiko mengalami overfitting atau vanishing gradient problem.
2. Lebar (width) jaringan berperan dalam mengenali pola yang lebih kompleks, tetapi jika terlalu lebar, dapat meningkatkan risiko overfitting.
3. Fungsi aktivasi memengaruhi akurasi, kecepatan konvergensi, serta potensi overfitting atau underfitting.
4. Learning rate yang terlalu besar dapat menyebabkan model tidak stabil dan sulit belajar, sedangkan learning rate yang terlalu kecil membuat proses pembelajaran menjadi lambat.
5. Metode inisialisasi bobot berpengaruh terhadap stabilitas dan konvergensi model. Inisialisasi yang kurang tepat, seperti Zero initialization pada kasus ini, dapat menghambat proses pembelajaran.
6. Implementasi FFNN *from scratch* yang kami buat sudah mendekati performa model SKLearn, dengan perbedaan yang kemungkinan berasal dari optimasi dalam library tersebut.

Pembagian Tugas

Nama	NIM	Pembagian tugas
Renaldy Arief Susanto	13522022	- Implementasi Backward Propagation - Implementasi Struktur data Neural Network dan Layer - Laporan Bagian Penjelasan Implementasi - Implementasi, Pengujian, dan Laporan Bagian Regularisasi
Nyoman Ganadipa Narayana	13522066	- Implementasi Forward Propagation - Implementasi Fungsi Aktivasi - Implementasi Weight Initializer - Pengujian dan Laporan Bagian Pengujian
Muhammad Dava Fathurrahman	13522114	- Implementasi Fungsi Loss - Implementasi Save, Load, serta metode-metode Plot dan Graf - Pengujian dan Laporan Bagian Pengujian

Referensi

https://www.jasonosajima.com/backprop

https://medium.com/@hunter-j-phillips/a-simple-introduction-to-softmax-287712d69bac

https://edunexcontentprodhot.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://wandb.ai/mostafaibrahim17/ml-articles/reports/A-Deep-Dive-Into-Learning-Curves-in-Machine-Learning--Vmlldzo0NjA1ODY0

https://rstudio-conf-2020.github.io/dl-keras-tf/notebooks/learning-curve-diagnostics.nb.html