

TUGAS BESAR I
IF3270 PEMBELAJARAN MESIN
SEMESTER II TAHUN 2024/2025

Feedforward Neural Network



Kelompok 5:

| | |
|---------------------------|----------|
| Irfan Sidiq Permana | 13522007 |
| Bryan Cornelius Lauwrence | 13522033 |
| Ahmad Hasan Albana | 13522041 |

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

DAFTAR ISI

| | |
|--|----|
| BAB I PENDAHULUAN..... | 3 |
| 1.1. Abstraksi | 3 |
| BAB II IMPLEMENTASI ALGORITMA | 4 |
| 2.1. Deskripsi Kelas | 4 |
| 2.1.1. Kelas WeightInitializer | 4 |
| 2.1.1.1. Kelas ZeroInitializer | 4 |
| 2.1.1.2. Kelas RandomUniformInitializer..... | 4 |
| 2.1.1.3. Kelas RandomNormalInitializer | 4 |
| 2.1.1.4. Kelas GlorotUniformInitializer..... | 4 |
| 2.1.1.5. Kelas HeNormalInitializer | 5 |
| 2.1.1.6. Kelas ActivationFunction..... | 5 |
| 2.1.1.7. Kelas LossFunction..... | 5 |
| 2.1.2. Kelas Tensor..... | 5 |
| 2.1.3. Kelas Optimizer | 7 |
| 2.1.4. Kelas Layer | 7 |
| 2.1.5. Kelas Dense | 8 |
| 2.1.6. Kelas FFNN | 9 |
| 2.1.7. Kelas RegularizedLoss..... | 10 |
| 2.1.8. Kelas RMSNorm..... | 11 |
| 2.2. Forward Propagation..... | 11 |
| 2.3. Backward Propagation | 12 |
| BAB III HASIL PENGUJIAN DAN ANALISIS | 14 |
| 3.1. Pengaruh <i>Depth</i> dan <i>Width</i> | 14 |
| 3.1.1. Hasil Pengujian | 14 |
| 3.1.2. Hasil Analisis | 16 |
| 3.2. Pengaruh Fungsi Aktivasi | 16 |
| 3.2.1. Hasil Pengujian | 16 |
| 3.2.2. Hasil Analisis | 20 |
| 3.3. Pengaruh Learning Rate..... | 21 |
| 3.3.1. Hasil Pengujian | 21 |
| 3.3.2. Hasil Analisis | 23 |
| 3.4. Pengaruh Inisialisasi Bobot..... | 23 |

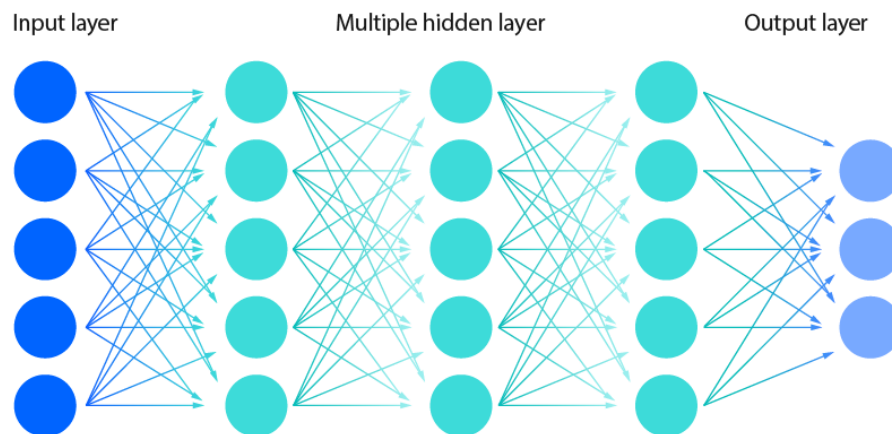
| | | |
|-----------------------------------|------------------------------------|----|
| 3.4.1. | Hasil Pengujian | 23 |
| 3.4.2. | Hasil Analisis | 27 |
| 3.5. | Pengaruh Regularisasi..... | 27 |
| 3.5.1. | Hasil Pengujian | 27 |
| 3.5.2. | Hasil Analisis | 29 |
| 3.6. | Pengaruh Normalisasi RMSNorm | 29 |
| 3.6.1. | Hasil Pengujian | 29 |
| 3.6.2. | Hasil Analisis | 31 |
| 3.7. | Perbandingan dengan SKLearn..... | 31 |
| 3.7.1. | Hasil Pengujian | 31 |
| 3.7.2. | Hasil Analisis | 31 |
| BAB IV KESIMPULAN DAN SARAN | | 32 |
| 4.1. | Kesimpulan | 32 |
| 4.2. | Saran | 32 |
| PEMBAGIAN TUGAS | | 33 |
| REFERENSI | | 34 |

BAB I

PENDAHULUAN

1.1. Abstraksi

Feedforward Neural Network (FFNN) adalah salah satu jenis algoritma dalam *Artificial Neural Network* (ANN) *Architecture*. Koneksi antara simpul bersifat satu arah membentuk graf asiklik dengan setiap simpul dapat dikelompokkan menjadi 3 bagian yaitu *input layer*, *hidden layer*, dan *output layer*. Data mengalir dari *input layer* dan mengalir melalui *hidden layer* menuju *output layer* tanpa adanya *loop* pada graf dengan pada setiap *layer* memiliki fungsi aktivasinya tersendiri dan juga memiliki fungsi *error* untuk menghitung nilai *error* pada saat melakukan *training* data.



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

BAB II

IMPLEMENTASI ALGORITMA

2.1. Deskripsi Kelas

2.1.1. Kelas WeightInitializer

Kelas WeightInitializer merupakan kelas abstrak untuk menginisialisasi bobot.

| Metode | Deskripsi |
|---|---|
| <code>initialize_weight(neuron_size: int, input_size: int) -> numpy.ndarray</code> | Menginisialisasi bobot sebagai matriks berukuran (<code>neuron_size</code> × <code>input_size</code>) |

Kelas ini diturunkan ke kelas *weight initializer* lainnya sesuai dengan metode inisialisasi.

2.1.1.1. Kelas ZeroInitializer

Menginisialisasi bobot yang seluruhnya bernilai 0.

2.1.1.2. Kelas RandomUniformInitializer

Menginisialisasi bobot secara acak, tetapi menghasilkan distribusi bobot *uniform*.

| Metode | Deskripsi |
|--|--|
| <code>__init__(self, lower_bound: int, upper_bound: int, seed: int)</code> | Bobot terdistribusi uniform dengan nilai dari <code>lower_bound</code> sampai <code>upper_bound</code> dan menerima <code>seed</code> untuk random generator |

2.1.1.3. Kelas RandomNormalInitializer

Menginisialisasi bobot secara acak, tetapi menghasilkan distribusi bobot normal.

| Metode | Deskripsi |
|--|---|
| <code>__init__(self, mean: int, variance: int, seed: int)</code> | Bobot terdistribusi normal dengan rata-rata bobot <code>mean</code> dan variansinya <code>variance</code> serta menerima <code>seed</code> untuk random generator |

2.1.1.4. Kelas GlorotUniformInitializer

Menginisialisasi bobot secara acak dan menghasilkan distribusi bobot normal, dengan batas atas dan batas bawah $\pm \sqrt{\frac{6}{\text{jumlah bobot}}}$ (*Xavier initialization*).

| Metode | Deskripsi |
|--------|-----------|
|--------|-----------|

| | |
|--|--------------------------------------|
| <code>__init__(self, seed: int)</code> | Menerima seed untuk random generator |
|--|--------------------------------------|

2.1.1.5. Kelas HeNormalInitializer

Menginisialisasi bobot secara acak dan menghasilkan distribusi bobot normal, dengan standar deviasi $\pm \sqrt{\frac{2}{\text{jumlah node di layer sebelumnya}}}$.

| Metode | Deskripsi |
|--|--------------------------------------|
| <code>__init__(self, seed: int)</code> | Menerima seed untuk random generator |

2.1.1.6. Kelas ActivationFunction

Kelas abstrak untuk melakukan perhitungan *forward* dengan fungsi aktivasi tertentu dan *backward* yang merupakan turunan fungsi aktivasi.

| Metode | Deskripsi |
|---|--|
| <code>forward(x: np.ndarray) -> np.ndarray</code> | Menerima x sebagai vektor satu dimensi untuk diaktivasi |
| <code>backward(x: np.ndarray) -> np.ndarray</code> | Menerima x sebagai vektor satu dimensi untuk dihitung nilai turunannya |

Kelas ini digunakan untuk menerapkan fungsi-fungsi aktivasi, yaitu fungsi linear, fungsi ReLU, fungsi Sigmoid, fungsi tanh, fungsi Softmax, fungsi GELU, dan fungsi SiLU. Khusus fungsi Softmax, fungsi *backward* akan menghasilkan matriks 2x2.

2.1.1.7. Kelas LossFunction

Kelas abstrak untuk menghitung besarnya error dari hasil prediksi dengan fungsi *loss* tertentu.

| Metode | Deskripsi |
|--|--|
| <code>forward(y_true: np.ndarray, y_pred: np.ndarray) -> np.ndarray</code> | Menghitung nilai loss dengan fungsi loss |
| <code>backward(y_true: np.ndarray, y_pred: np.ndarray) -> np.ndarray</code> | Menghitung gradien dari fungsi loss |

Kelas ini digunakan untuk menerapkan beberapa fungsi *loss*, yaitu fungsi MSE, fungsi *Binary Cross-Entropy*, dan fungsi *Categorical Cross-Entropy*.

2.1.2. Kelas Tensor

Kelas Tensor merupakan kelas yang digunakan untuk melakukan *automated differentiation*. Kelas ini berperan sebagai *wrapper* dari nilai data dan gradien, serta memiliki fungsi utama yaitu *backward()* yang digunakan untuk menginisiasi *automated differentiation*.

| Atribut | Deskripsi |
|---|--|
| <code>data: np.ndarray</code> | Berisi nilai utama dari suatu tensor. Misal bila tensor merupakan <code>weight</code> , maka <code>weight.data</code> berisi nilai dari <code>weight</code> tersebut |
| <code>gradient: np.ndarray</code> | Berisi nilai gradien dari suatu tensor. Misal bila tensor merupakan <code>weight</code> , maka <code>weight.gradient</code> menyatakan nilai turunan parsial dari <code>weight</code> tersebut |
| <code>__children: List[Tensor]</code> | Berisi operand dari operasi tensor sebelumnya. Misalkan tensor $X = y + z$, maka <code>X.__children = [y, z]</code> |
| <code>__op: str</code> | Berisi operasi dari tensor sebelumnya. Misalkan tensor $X = y + z$, maka <code>X.__op = '+'</code> |
| <code>__backward: Callable[[], None]</code> | Merupakan fungsi privat dari tensor yang akan dipanggil di dalam proses <i>backpropagation</i> |
| <code>requires_grad: bool</code> | Menyatakan apakah tensor tersebut perlu dihitung gradiennya |
| <code>tensor_type: str</code> | Menyatakan keterangan dari tensor, misalkan tensor merupakan <code>weight</code> maka <code>tensor_type = "weight"</code> |

| Metode | Deskripsi |
|---|---|
| <code>__repr__(self) -> str: -> numpy.ndarray</code> | Mengembalikan string data apa saja yang akan diprint ketika dipanggil <code>print(tensor)</code> |
| <code>add_x0(self) -> Tensor</code> | Menambahkan $x_0 = 1$ ke tensor, digunakan untuk menyiapkan output suatu layer sebagai input dari layer selanjutnya |
| <code>sum(self) -> Tensor</code> | Menghitung jumlah sum dari <code>tensor.data</code> . Digunakan untuk menghitung net |
| <code>compute_activation(self, activation_function: ActivationFunction) -> Tensor</code> | Menghitung nilai aktivasi dari suatu tensor menggunakan fungsi aktivasi yang diberikan |

| | |
|---|--|
| <code>compute_loss(self, y_true: np.ndarray, loss_function: LossFunction) -> Tensor</code> | Menghitung nilai loss dari suatu tensor menggunakan fungsi loss yang diberikan |
| <code>concat(self, tensors: List[Tensor]) -> Tensor</code> | Menggabungkan beberapa tensor menjadi satu tensor. Digunakan untuk menggabungkan output tiap neuron pada suatu layer. |
| <code>backward(self) -> None</code> | Menginisiasi automated differentiation. Digunakan pada tensor loss untuk menghitung gradien dari semua operasi sebelumnya. |

2.1.3. Kelas Optimizer

Kelas Optimizer merupakan kelas yang digunakan untuk mengupdate seluruh parameter dari FFNN. Kelas ini umumnya hanya digunakan pada saat fungsi *fit()* dipanggil, yaitu pada proses pembelajaran model.

| Atribut | Deskripsi |
|---------------------------------------|--|
| <code>parameters: List[Tensor]</code> | Berisi semua tensor parameter yang akan di-update nilainya sesuai learning rate. |
| <code>learning_rate: float</code> | Berisi nilai learning rate. |

| Metode | Deskripsi |
|---|---|
| <code>__init__(self, learning_rate: float = 0.01) -> None</code> | Menginisialisasi optimizer dengan learning rate = 0.01 sebagai default |
| <code>set_parameters(self, parameters: List[Tensor])</code> | Menyimpan parameter yang akan di-update. |
| <code>step(self) -> None</code> | Mengupdate seluruh parameter berdasarkan nilai gradien dan learning rate. |
| <code>zero_grad(self) -> None</code> | Mereset gradien seluruh parameter menjadi 0. |

2.1.4. Kelas Layer

Kelas Layer merupakan *abstract class* yang mendefinisikan atribut dan fungsi yang dimiliki tiap *layer* dari model. Setiap *layer* memiliki fungsi aktivasi, metode inisialisasi bobot, matriks bobot dan matriks gradiennya sendiri.

| Atribut | Deskripsi |
|---------|-----------|
|---------|-----------|

| | |
|--|---|
| activation_function: ActivationFunction | Berisi fungsi aktivasi dari layer |
| weight_initializer: WeightInitializer | Berisi metode inisialisasi bobot dari layer |
| weights: List[Tensor] | Berisi kumpulan bobot pada layer |
| gradients: List[Tensor] | Berisi nilai-nilai gradien untuk tiap neuron pada layer |
| output: Tensor | Berisi hasil output forwardpropagation dari layer |

| Metode | Deskripsi |
|---|---|
| initialize_weights(self, input_size: int) -> None | Menginisialisasi seluruh bobot dari layer dengan mengikuti jumlah input |
| get_parameters(self) | Mengembalikan semua parameter dari layer |
| get_neuron_size(self) -> int | Mengembalikan jumlah neuron dari layer |
| forward(self, input: Tensor) -> Tensor | Melakukan forwardpropagation pada layer |
| plot_dist(self, is_weight: bool, n_layer: int, output_layer: int) | Melakukan plotting distribusi bobot atau gradien dari layer |

2.1.5. Kelas Dense

Kelas Dense merupakan *child class* dari kelas Layer, dimana tiap neuron pada suatu *layer* terhubung dengan semua neuron pada *layer* sebelumnya. Kelas ini berisi implementasi dari fungsi-fungsi yang telah didefinisikan pada kelas Layer.

| Atribut | Deskripsi |
|--|---|
| activation_function: ActivationFunction | Berisi fungsi aktivasi dari layer |
| weight_initializer: WeightInitializer | Berisi metode inisialisasi bobot dari layer |
| weights: List[Tensor] | Berisi kumpulan bobot pada layer |
| gradients: List[Tensor] | Berisi nilai-nilai gradien untuk tiap neuron pada layer |
| output: Tensor | Berisi hasil output forwardpropagation dari layer |

| Metode | Deskripsi |
|--|---|
| <code>initialize_weights(self, input_size: int) -> None</code> | Menginisialisasi seluruh bobot dari layer dengan mengikuti jumlah input |
| <code>get_parameters(self)</code> | Mengembalikan semua parameter dari layer |
| <code>get_neuron_size(self) -> int</code> | Mengembalikan jumlah neuron dari layer |
| <code>forward(self, input: Tensor) -> Tensor</code> | Melakukan forwardpropagation pada layer |
| <code>plot_dist(self, is_weight: bool, n_layer: int, output_layer: int)</code> | Melakukan plotting distribusi bobot atau gradien dari layer |

2.1.6. Kelas FFNN

Kelas FFNN merupakan kelas utama dari program, yaitu keseluruhan model FFNN yang diinstansiasi oleh pengguna. Kelas Model terdiri dari beberapa objek Layer beserta satu Optimizer, LossFunction atau RegularizedLoss, dan RMSNorm.

| Atribut | Deskripsi |
|--|--|
| <code>optimizer: Optimizer</code> | Berisi optimizer yang akan mengupdate bobot tiap layer setelah backpropagation |
| <code>loss_function: LossFunction</code> | Berisi fungsi loss dari output layer |
| <code>layers: List[Layer]</code> | Berisi kumpulan layer yang dimiliki model |
| <code>output: Tensor</code> | Berisi hasil prediksi dari model |
| <code>normalizer: RMSNorm</code> | Berisi kelas yang akan menormalisasi output tiap hidden layer |

| Metode | Deskripsi |
|--|--|
| <code>__init__(self, layers: List[Layer]) -> None</code> | Menginstansiasi model FFNN dengan kumpulan layer yang dimilikinya |
| <code>get_parameters(self) -> List[Tensor]</code> | Mengembalikan seluruh parameter bobot yang dimiliki model |
| <code>print_history(self) -> None</code> | Menampilkan histori training dari model |
| <code>compile(self, optimizer: str Optimizer = "sgd", loss: str = "mean_squared_error", regularization: str RegularizedLoss = None, normalize: bool = False) -> None</code> | Menginstansiasi optimizer, loss function atau regularizer, dan normalizer dari model |

| | |
|--|--|
| <code>forward(self, input: Tensor) -> Tensor</code> | Melakukan prediksi dari input yang diberikan |
| <code>backward(self, y_true: np.ndarray) -> None</code> | Menghitung gradien dari tiap parameter berdasarkan prediksi dan kelas sebenarnya |
| <code>fit(self, X_train: np.ndarray, y_train: np.ndarray, epochs: int = 10, batch_size: int = 32, verbose: bool = 0, validation_data: tuple = None) -> None:</code> | Melakukan training dengan menerima kumpulan data, ukuran batch, epoch, dan sebagainya |
| <code>predict(self, X_test: np.ndarray) -> List[np.ndarray]</code> | Melakukan prediksi dari kumpulan data yang diberikan |
| <code>evaluate(self, X_test: np.ndarray, y_test: np.ndarray) -> tuple[float, float]</code> | Melakukan prediksi dari kumpulan data yang diberikan lalu menghitung metrik evaluasi (loss dan akurasi) berdasarkan nilai kelas sebenarnya |
| <code>__repr__(self) -> str</code> | Menampilkan ilustrasi jaringan keseluruhan model |
| <code>plot_weights(self, layer: List[int]) -> None</code> | Menampilkan distribusi bobot tiap layer yang dipilih |
| <code>plot_gradients(self, layer: List[int]) -> None</code> | Menampilkan distribusi gradien tiap layer yang dipilih |
| <code>save(self, file_path: str) -> None</code> | Menyimpan seluruh state model ke file JSON |
| <code>load(self, file_path: str) -> None</code> | Memuat state model dari file JSON ke model |
| <code>__init__(self, layers: List[Layer]) -> None</code> | Menginstansiasi model FFNN dengan kumpulan layer yang dimilikinya |

2.1.7. Kelas RegularizedLoss

Kelas RegularizedLoss merupakan kelas yang digunakan sebagai *wrapper* dari kelas LossFunction untuk menghitung nilai *loss* dan menghitung *gradient loss* dengan menerapkan metode regularisasi yang dipilih. Pada implementasinya, kelas RegularizedLoss tidak hanya mengembalikan *gradient loss*, tetapi juga secara langsung meng-*update* nilai gradien dari tiap parameter model sesuai metode regularisasi dan konstanta lambda.

| Atribut | Deskripsi |
|---------|-----------|
|---------|-----------|

| | |
|--|--|
| <code>loss_function: LossFunction</code> | Fungsi loss yang digunakan |
| <code>parameters: List[Tensor]</code> | Parameter model yang akan diupdate |
| <code>regularization_type: str</code> | Metode regularisasi yang dipilih |
| <code>lambda_reg: float</code> | Konstanta untuk mengupdate parameter model |

| Metode | Deskripsi |
|---|--|
| <code>def __init__(self, loss_function: LossFunction, regularization_type: str = 'l2', lambda_reg: float = 0.001) -> None</code> | Menginisialisasi objek dengan metode regularisasi dan <code>lambda_reg</code> yang dipilih |
| <code>__name__(self) -> None</code> | Menampilkan fungsi loss dan regularisasi yang dipilih |
| <code>set_parameters(self, parameters: List[Tensor]) -> None</code> | Memuat parameter model yang akan diupdate |
| <code>forward(self, y_true: np.ndarray, y_pred: np.ndarray) -> np.ndarray</code> | Menghitung nilai loss berdasarkan fungsi loss dan metode regularisasi yang dipilih |
| <code>backward(self, y_true: np.ndarray, y_pred: np.ndarray) -> np.ndarray</code> | Menghitung nilai gradien loss dan mengupdate nilai gradien tiap parameter sesuai metode regularisasi |

2.1.8. Kelas RMSNorm

Kelas RMSNorm merupakan kelas yang digunakan untuk menormalisasi *output* dari suatu *layer*. Pada program ini, kelas RMSNorm digunakan untuk menormalisasi *output* dari tiap *hidden layer* sebelum digunakan sebagai *input* untuk *layer* selanjutnya.

| Metode | Deskripsi |
|--|---|
| <code>forward(x: Tensor) -> Tensor</code> | Mengembalikan hasil normalisasi dari Tensor input |

2.2. Forward Propagation

Forward propagation merupakan proses inferensi dari suatu model *neural network*, dimana data input dimasukkan dalam jaringan melalui *input layer* untuk diteruskan ke seluruh neuron pada jaringan dan akhirnya dihasilkan output berupa prediksi di layer terakhir (*output layer*). Selama *forward propagation*, data input dikalikan dengan bobot pada *input layer*, untuk

kemudian dijumlahkan dan dioperasikan menggunakan fungsi aktivasi. Hasil dari *input layer* ini kemudian digunakan sebagai input pada *layer* berikutnya, dan begitu seterusnya hingga diperoleh *output* dari *layer* terakhir yaitu *output layer*.

Data input untuk *forward propagation* pada program kami berupa 1D NumPy array berukuran n , dimana n adalah jumlah fitur dari data input. Implementasi *forward propagation* dari program kami adalah sebagai berikut:

- a. *Input data* dikonversi menjadi objek Tensor terlebih dahulu (untuk nantinya melakukan *automated differentiation*).
- b. Menambahkan $x_0 = 1$ pada input agar dimensinya sesuai dan dapat dikalikan dengan bobot tiap neuron (karena bobot tiap neuron termasuk bias).
- c. *Input* dimasukkan kedalam *input layer* kemudian dikalikan dengan seluruh bobot *layer* yang berupa matriks $m \times n$ dengan m adalah jumlah neuron dan n adalah jumlah fitur tiap data.
- d. Hasil perkalian tiap neuron (sejumlah m baris) masing-masing dijumlahkan, yaitu tiap anggota pada baris yang sama dijumlahkan sehingga diperoleh satu angka pada tiap baris.
- e. Tiap angka ini dimasukkan ke fungsi aktivasi sehingga diperoleh sejumlah m *output*.
- f. Output masing-masing neuron kemudian digabungkan menjadi satu objek Tensor baru, lalu digunakan sebagai input untuk *layer* berikutnya.
- g. Begitu seterusnya hingga diperoleh *output* dari *layer* terakhir (yang juga berupa objek Tensor).

2.3. Backward Propagation

Backward propagation merupakan proses pembelajaran dalam model *neural network* yang terjadi setelah *forward propagation* selesai. Pada tahap ini, model mengevaluasi seberapa jauh hasil prediksi dari *output layer* berbeda dengan nilai target yang sebenarnya menggunakan fungsi *loss*. Selisih atau *error* ini kemudian disebarkan kembali ke belakang melalui setiap *layer* jaringan, dimulai dari *output layer* hingga mencapai *input layer*. Selama proses ini, model menghitung kontribusi setiap bobot terhadap *error* yang terjadi menggunakan turunan dari fungsi aktivasi dan aturan rantai, untuk menentukan arah dan besarnya perubahan yang diperlukan. Gradien yang dihasilkan digunakan untuk memperbarui bobot-bobot jaringan agar di iterasi selanjutnya, prediksi menjadi lebih akurat. Proses ini dilakukan berulang-ulang agar model semakin “belajar” dan menghasilkan *output* yang semakin mendekati nilai sebenarnya.

Implementasi *backward propagation* pada program kami dilakukan dengan menerapkan *automated differentiation*, sebagai berikut:

- a. Hasil *output* dari *output layer* dihitung *loss*-nya berdasarkan nilai target sebenarnya dan fungsi *loss* yang dipilih.
- b. Memulai *automated differentiation* dengan memanggil fungsi *backward()* pada hasil *loss* tersebut (yang berupa objek Tensor).
- c. Fungsi *backward* akan menyusun urutan operasi yang selama ini dilakukan pada fase *forward propagation* (yang berupa pohon operasi) menggunakan *topological sort* sehingga diperoleh urutan operasi mulai dari operasi terakhir yaitu *loss function* hingga operasi pertama yaitu perkalian pada *input layer*.
- d. Menghitung nilai gradien dari tiap operasi terurut dari operasi terakhir (kedalaman 0 pada pohon operasi), lalu hasil gradien tersebut digunakan untuk mengalikan nilai gradien selanjutnya (kedalaman 1 pada pohon operasi).
- e. Begitu seterusnya hingga seluruh elemen pada pohon operasi diperoleh nilai gradiennya.

Pada proses training model, data input dikelompokkan sesuai ukuran *batch* terlebih dahulu lalu untuk tiap *batch* dilakukan *forward propagation* dan *back propagation* untuk tiap anggota *batch*. Hasil kumulatif dari *backward propagation* pada satu *batch* kemudian digunakan untuk meng-*update* seluruh parameter jaringan berdasarkan *learning rate* yang ditentukan.

BAB III

HASIL PENGUJIAN DAN ANALISIS

3.1. Pengaruh *Depth* dan *Width*

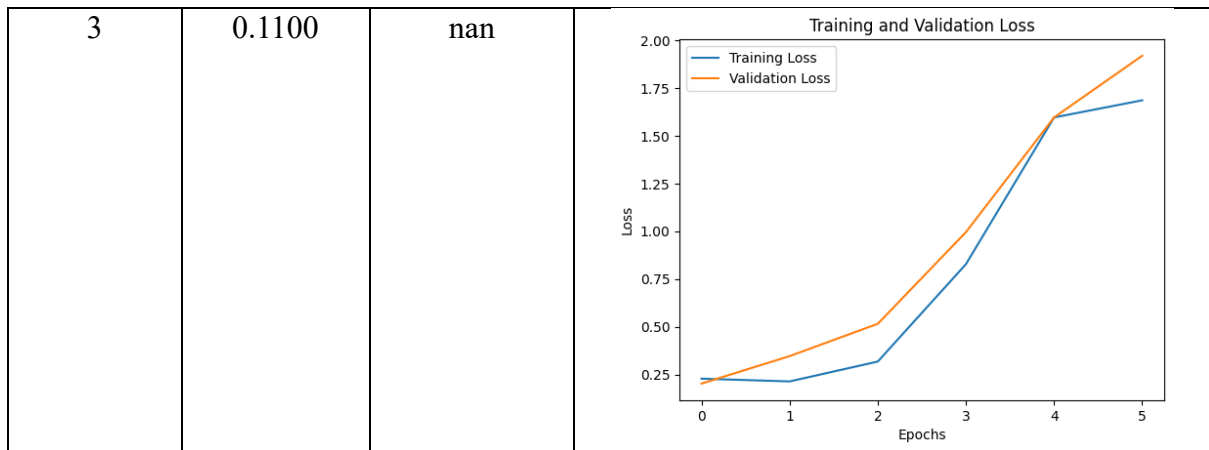
3.1.1. Hasil Pengujian

Pengujian model terhadap pengaruh *depth* dilakukan dengan nilai-nilai berikut (nilai *width* untuk tiap *hidden layer* ditetapkan yaitu 256):

| Percobaan | Depth | Width |
|-----------|-------|--------------------------|
| 1 | 3 | [784, 256, 10] |
| 2 | 4 | [784, 256, 256, 10] |
| 3 | 5 | [784, 256, 256, 256, 10] |

Dari ketiga percobaan, diperoleh *accuracy*, *loss*, serta grafik *training loss* dan *validation loss* tiap *epoch* sebagai berikut:

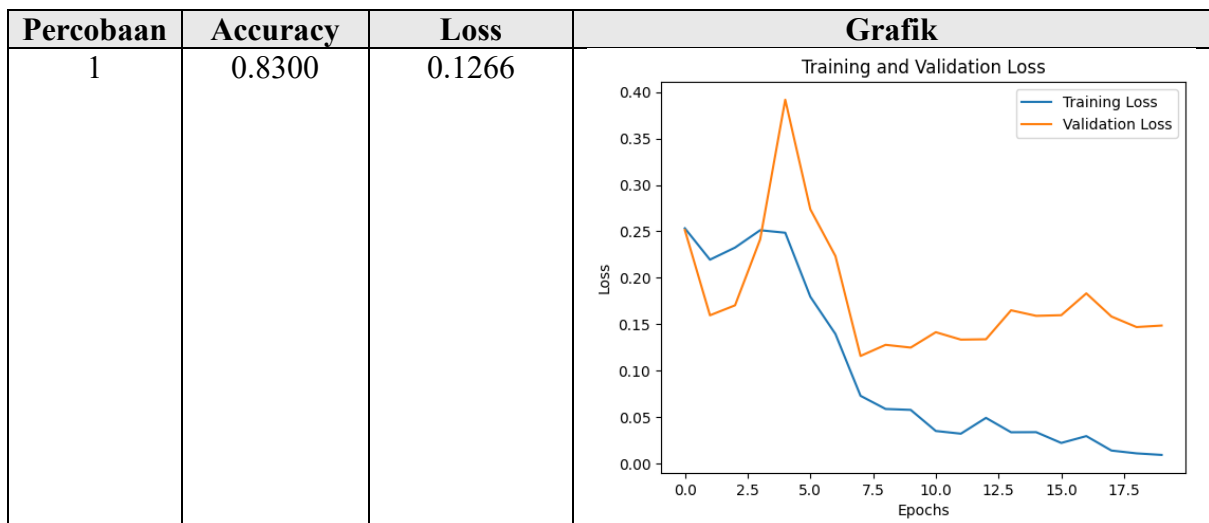
| Percobaan | Accuracy | Loss | Grafik |
|-----------|----------|--------|--------|
| 1 | 0.8350 | 0.0561 | |
| 2 | 0.8300 | 0.1156 | |

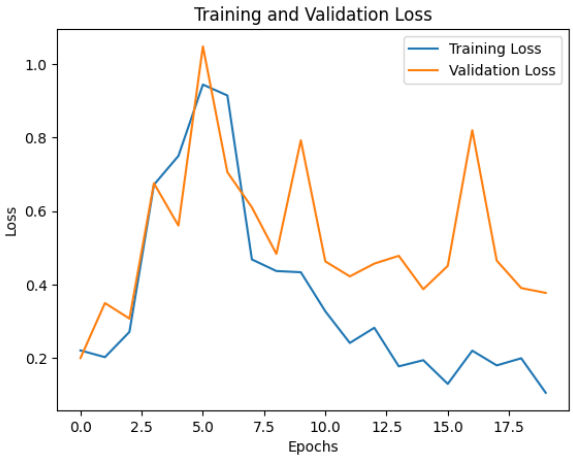
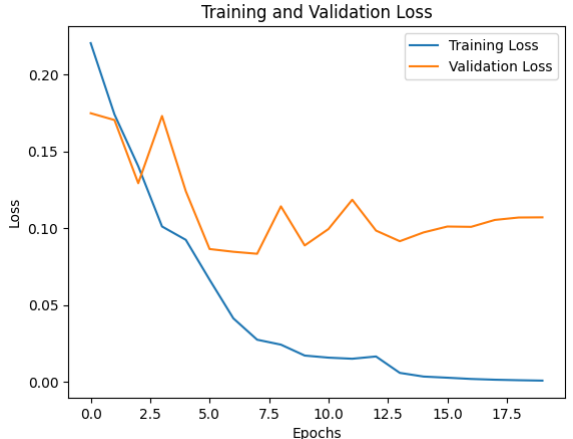


Selanjutnya, dilakukan pengujian model terhadap pengaruh *width* dilakukan dengan nilai-nilai berikut (nilai *depth* ditetapkan yaitu 4, termasuk *output layer*):

| Percobaan | Depth | Width |
|-----------|-------|---------------------|
| 1 | 4 | [784, 256, 64, 10] |
| 2 | 4 | [784, 256, 128, 10] |
| 3 | 4 | [784, 256, 256, 10] |

Dari ketiga percobaan, diperoleh *accuracy*, *loss*, serta grafik *training loss* dan *validation loss* tiap *epoch* sebagai berikut:



| | | | |
|---|--------|--------|---|
| 2 | 0.7700 | 0.3951 |  |
| 3 | 0.8200 | 0.0879 |  |

3.1.2. Hasil Analisis

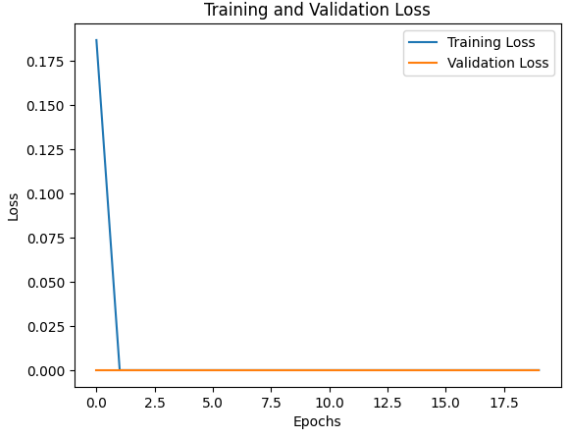
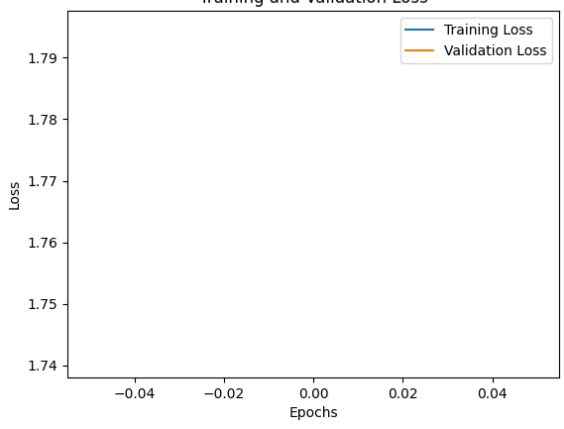
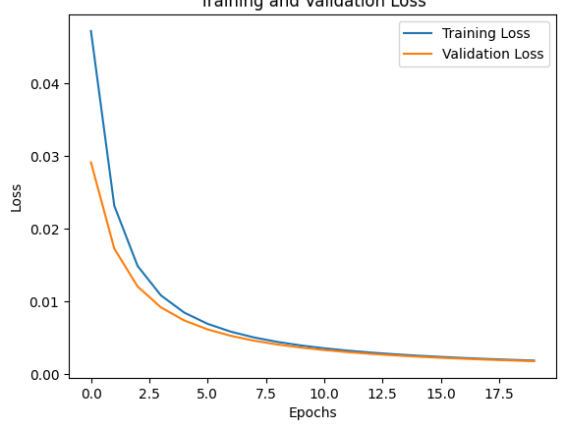
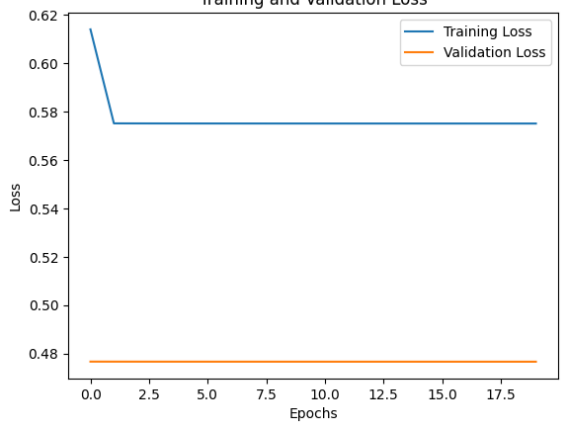
Berdasarkan hasil percobaan di atas, *depth* dan *width* tidak cukup berpengaruh terhadap akurasi model. Model yang memiliki terlalu banyak bobot (*depth* atau *width* yang besar) justru akurasinya lebih rendah dibandingkan dengan model yang lebih sederhana. Bobot yang banyak akan membuat gradien terlalu dekat dengan nol di *layer-layer* awal. Optimasi pun lebih sulit dicapai karena waktu *training* lama dan sulit mencapai konvergensi.

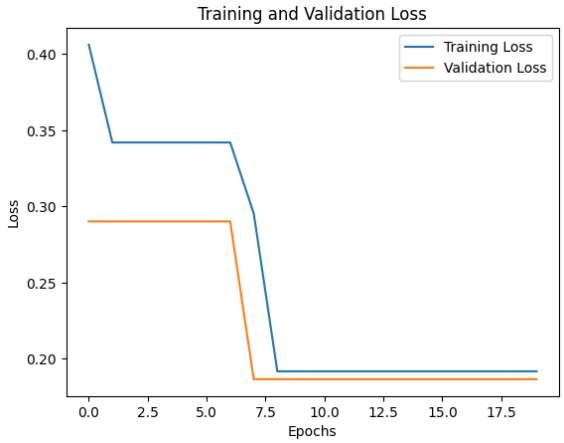
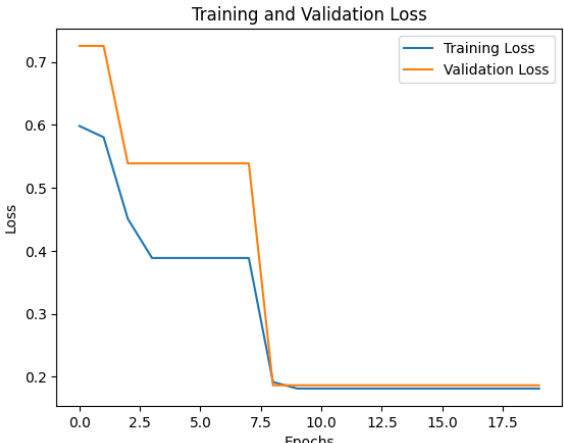
3.2. Pengaruh Fungsi Aktivasi

3.2.1. Hasil Pengujian

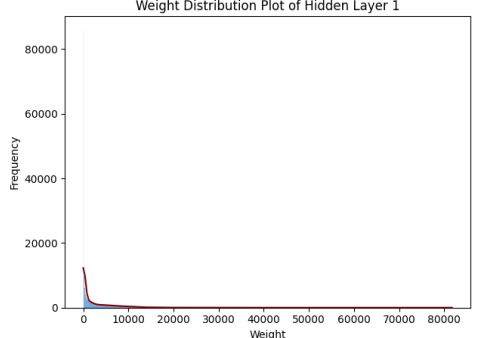
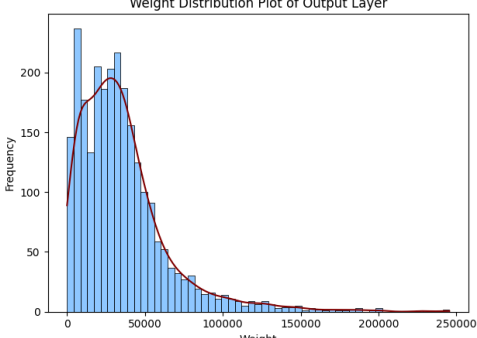
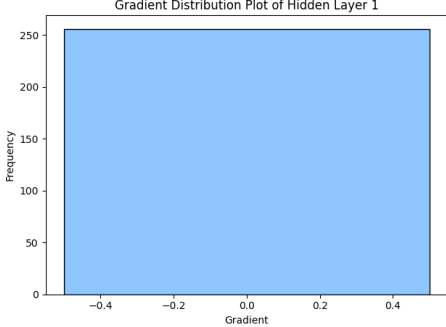
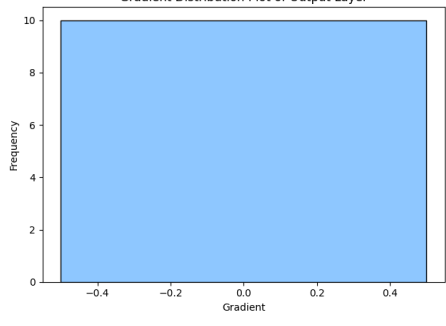
Pengujian model terhadap pengaruh fungsi aktivasi sebagai berikut (nilai *width* untuk *hidden layer* sama yaitu 256):

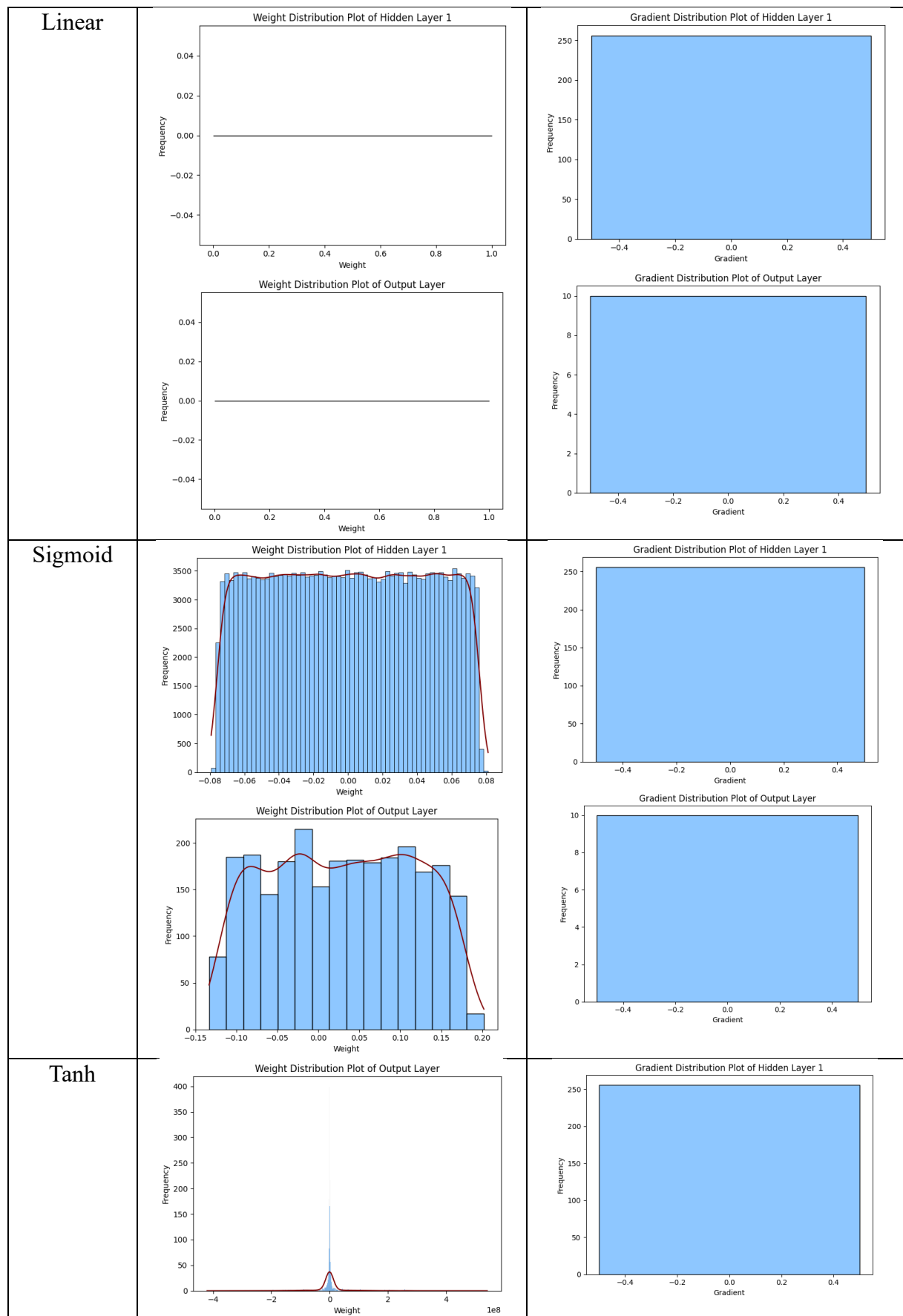
| Fungsi | Accuracy | Loss | Grafik |
|--------|----------|------|--------|
|--------|----------|------|--------|

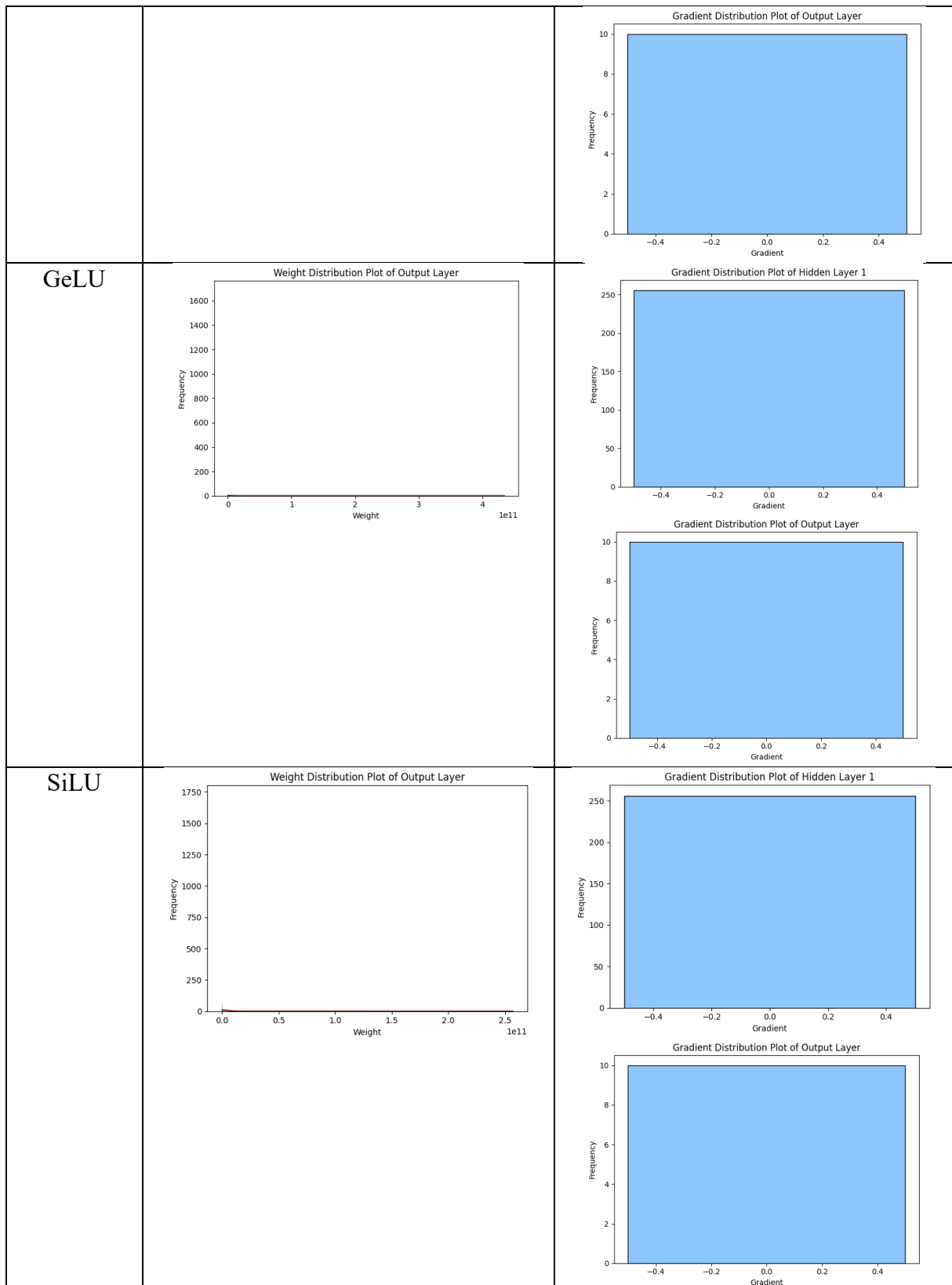
| | | | |
|---------|--------|--------|--|
| ReLU | 0.1100 | 0.0000 |  |
| Linear | 0.1100 | nan |  |
| Sigmoid | 0.1700 | 0.1700 |  |
| Tanh | 0.0900 | 0.5907 |  |

| | | | |
|------|--------|--------|---|
| GeLU | 0.1100 | 0.2383 |  |
| SiLU | 0.1100 | 0.2383 |  |

Serta grafik distribusi bobot dan gradien sebagai berikut:

| Fungsi | Distribusi bobot | Distribusi gradien |
|--------|---|---|
| ReLU | <p>Weight Distribution Plot of Hidden Layer 1</p>  <p>Weight Distribution Plot of Output Layer</p>  | <p>Gradient Distribution Plot of Hidden Layer 1</p>  <p>Gradient Distribution Plot of Output Layer</p>  |





3.2.2. Hasil Analisis

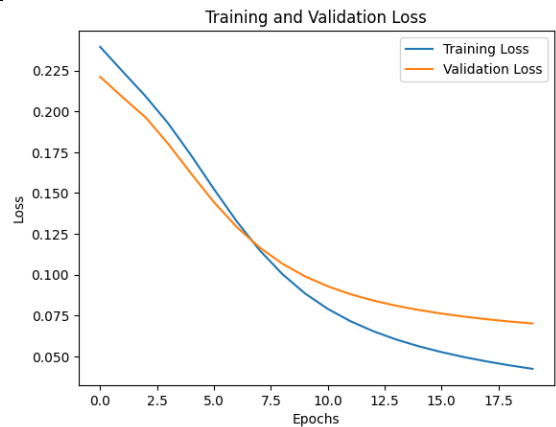
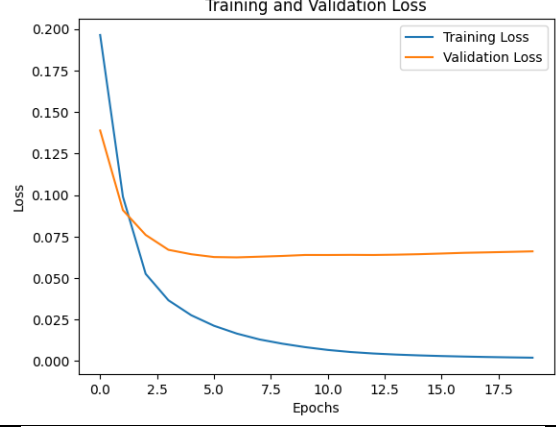
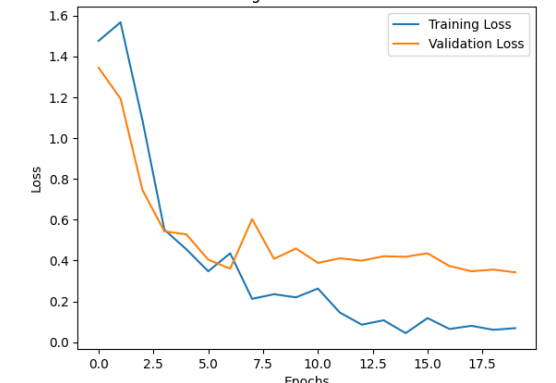
Berdasarkan hasil pengujian, keenam fungsi aktivasi tidak menghasilkan performa yang baik. Hasil tersebut diperoleh karena hanya menggunakan satu *hidden layer*, gradien pun

tidak berubah secara signifikan. Jadi, fungsi aktivasi sebaiknya tetap dikombinasikan dengan jumlah *layer* dan jumlah *node* yang memadai. Bagaimanapun juga, persebaran bobot yang dihasilkan oleh fungsi sigmoid cukup wajar sehingga sigmoid mampu menghasilkan akurasi tertinggi. Fungsi lainnya tidak berhasil menghasilkan model yang baik karena fungsi sigmoid tidak menimbulkan linearitas.

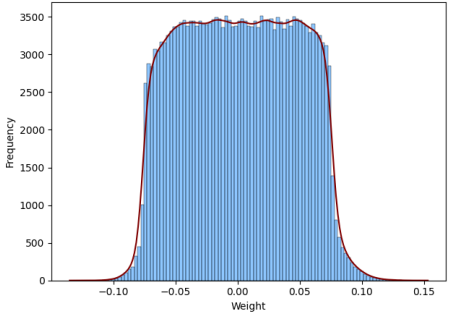
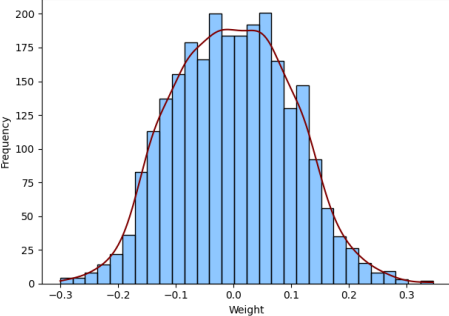
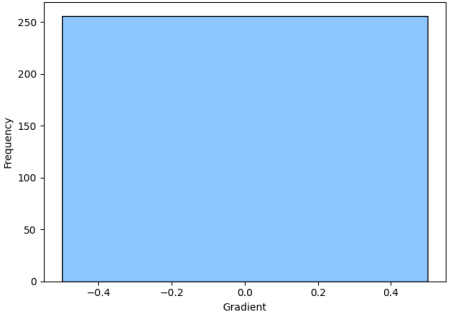
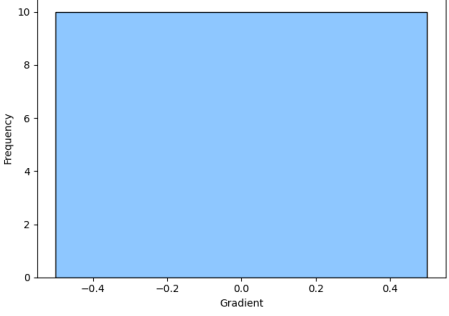
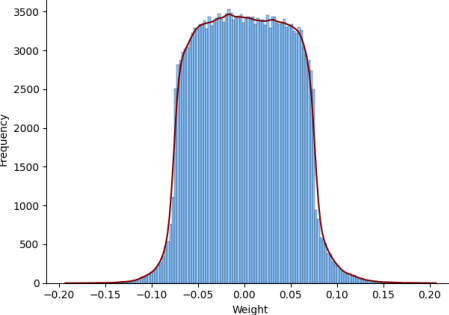
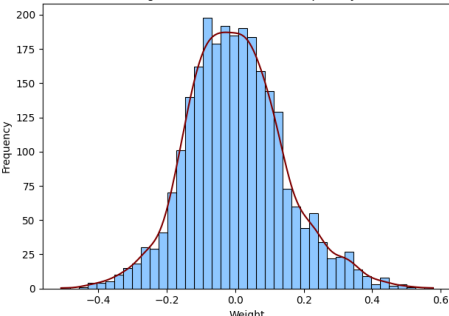
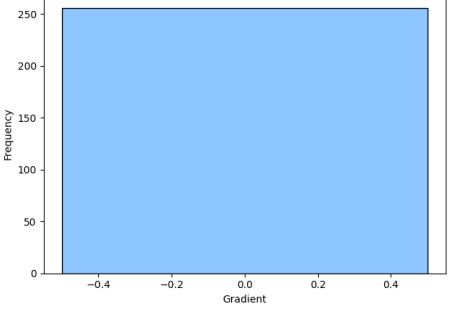
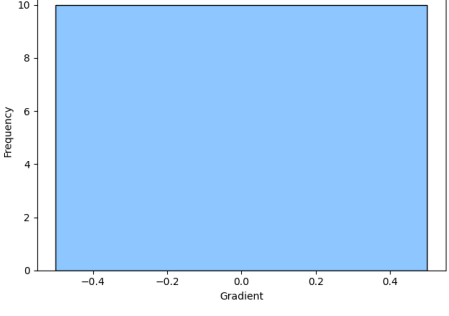
3.3. Pengaruh Learning Rate

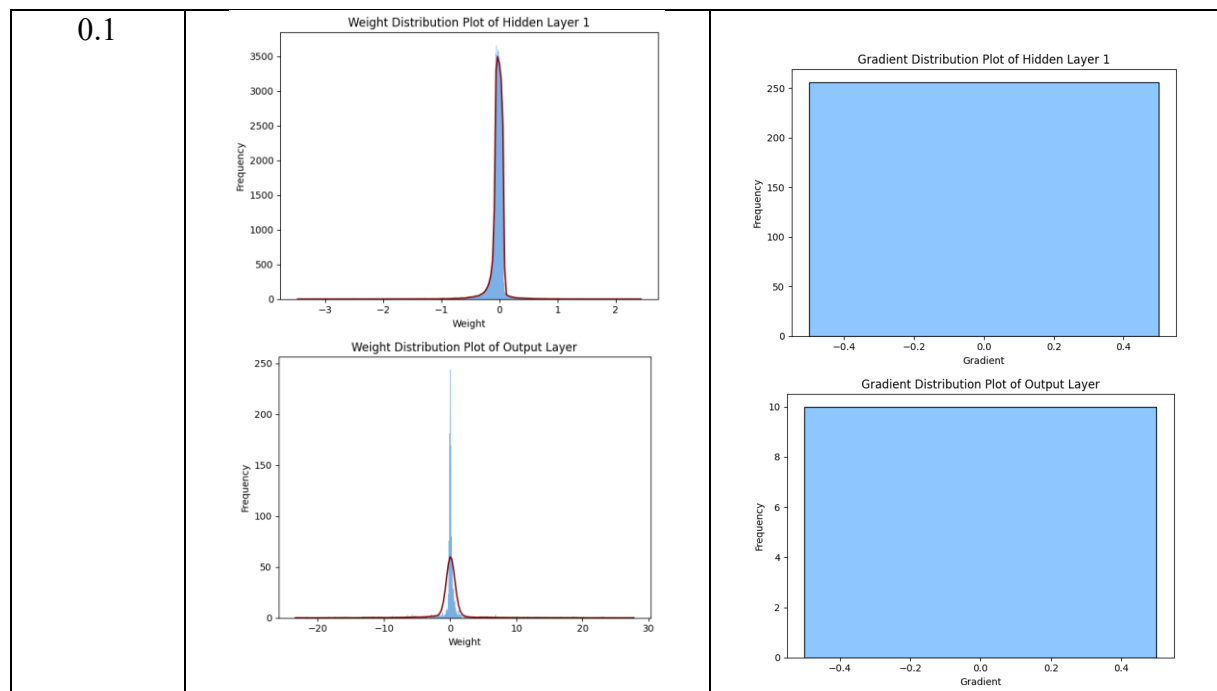
3.3.1. Hasil Pengujian

Pengujian model terhadap pengaruh *learning rate* dilakukan dengan nilai-nilai berikut (nilai *width* untuk *hidden layer* sama, yaitu 256):

| Learning Rate | Accuracy | Loss | Grafik |
|---------------|----------|--------|--|
| 0.001 | 0.8150 | 0.0633 |  |
| 0.01 | 0.8550 | 0.0569 |  |
| 0.1 | 0.7700 | 0.4579 |  |

Serta grafik distribusi bobot dan gradien sebagai berikut:

| Learning Rate | Distribusi bobot | Distribusi gradien |
|---------------|---|---|
| 0.001 | <p data-bbox="523 338 783 360">Weight Distribution Plot of Hidden Layer 1</p>  <p data-bbox="523 674 783 696">Weight Distribution Plot of Output Layer</p>  | <p data-bbox="1034 338 1294 360">Gradient Distribution Plot of Hidden Layer 1</p>  <p data-bbox="1034 674 1294 696">Gradient Distribution Plot of Output Layer</p>  |
| 0.01 | <p data-bbox="523 1099 783 1122">Weight Distribution Plot of Hidden Layer 1</p>  <p data-bbox="523 1442 783 1464">Weight Distribution Plot of Output Layer</p>  | <p data-bbox="1034 1099 1294 1122">Gradient Distribution Plot of Hidden Layer 1</p>  <p data-bbox="1034 1442 1294 1464">Gradient Distribution Plot of Output Layer</p>  |



3.3.2. Hasil Analisis

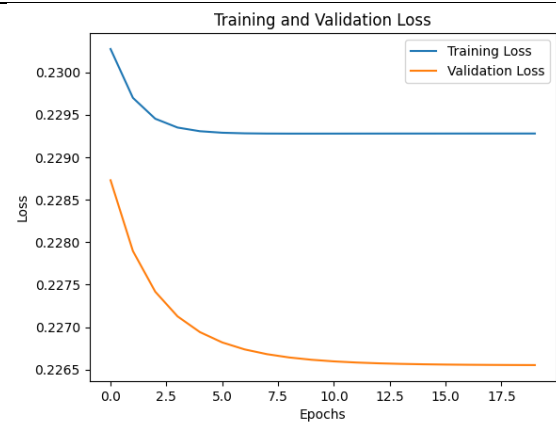
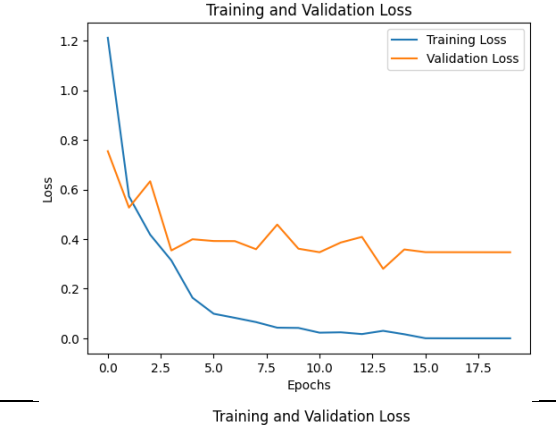
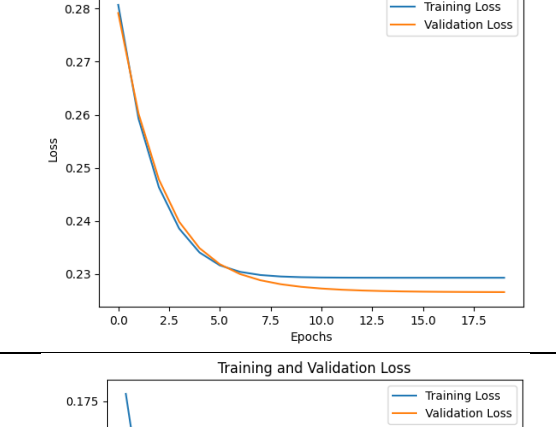
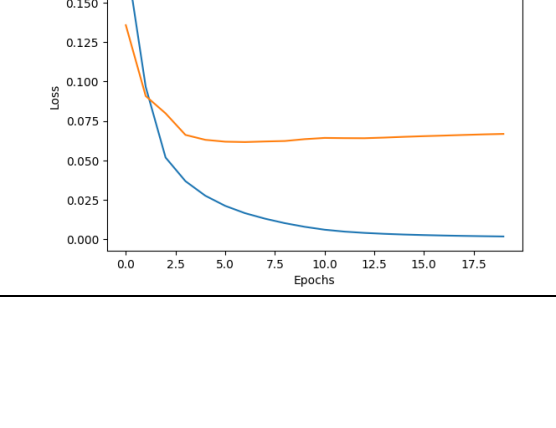
Learning rate berpengaruh pada hasil *training* model. Model terbaik dari pengujian berdasarkan nilai akurasi dan loss dihasilkan oleh nilai *learning rate* 0.01. Berdasarkan hasil percobaan tersebut, *learning rate* yang terlalu besar akan mengurangi akurasi model karena bobot berubah-ubah terlalu banyak seperti pada grafik *training loss and validation loss*, tetapi *learning rate* yang terlalu kecil akan memperlambat perubahan bobot. Pada grafik persebaran bobot, terlihat juga bahwa *learning rate* 0.01 bobotnya tidak terlalu menyebar. Jadi, nilai bobot sebaiknya disesuaikan supaya tidak terlalu besar dan tidak terlalu kecil.

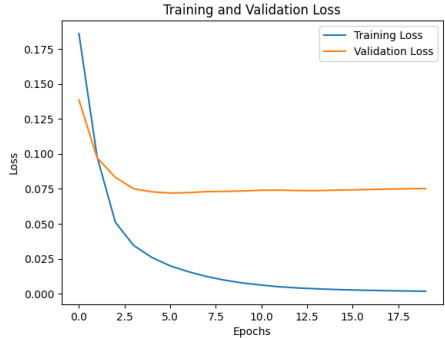
3.4. Pengaruh Inisialisasi Bobot

3.4.1. Hasil Pengujian

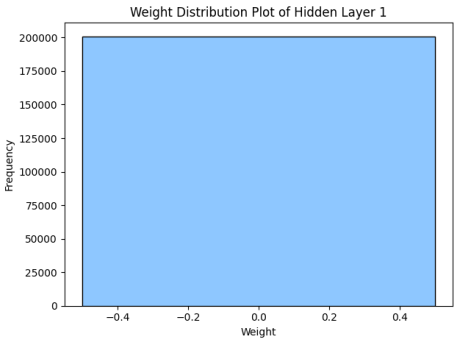
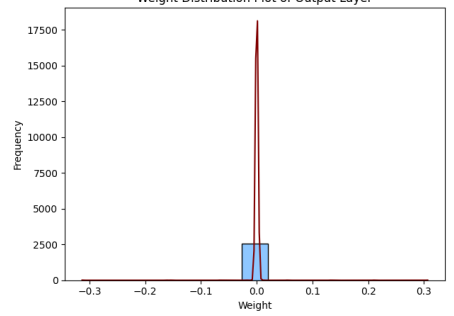
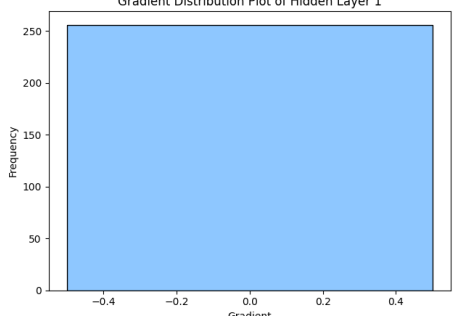
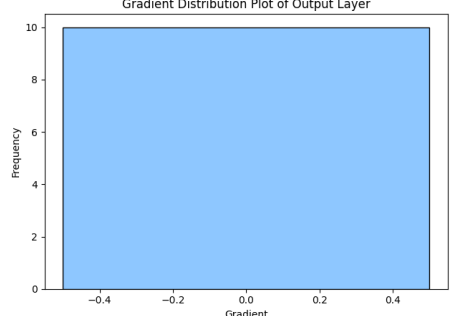
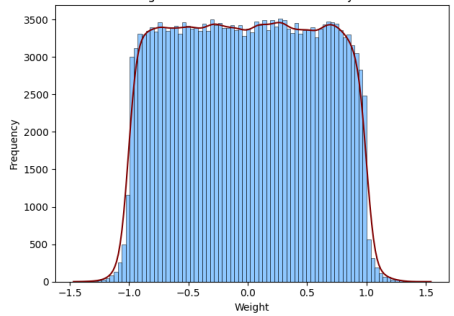
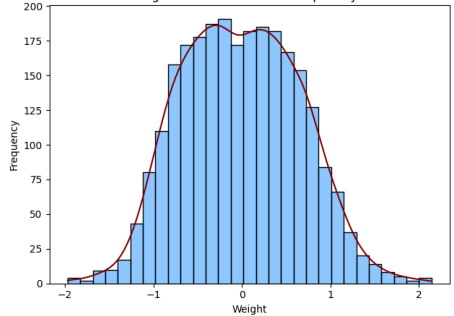
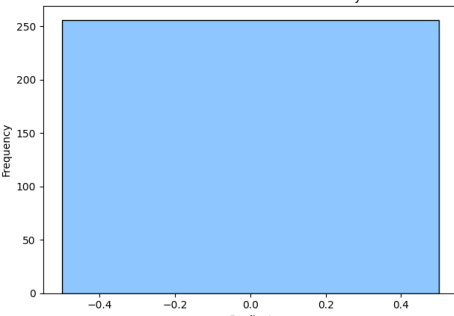
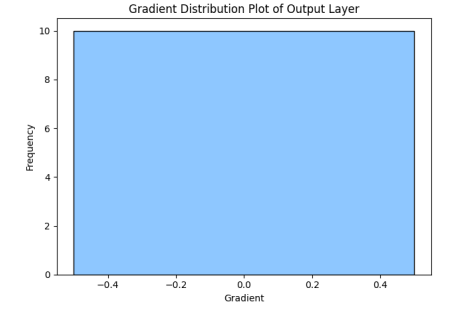
Pengujian model terhadap pengaruh inisiasi bobot dilakukan dengan nilai-nilai berikut (nilai *width* untuk *hidden layer* sama yaitu 256):

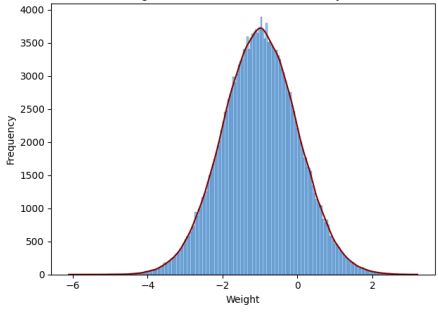
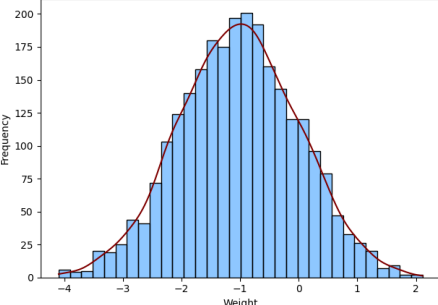
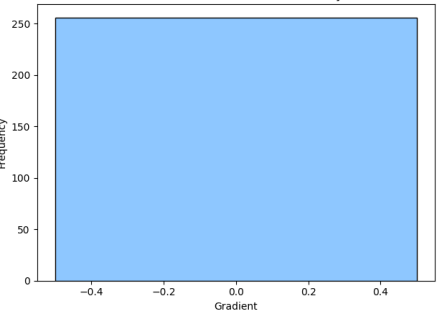
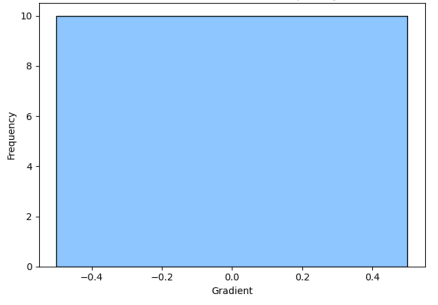
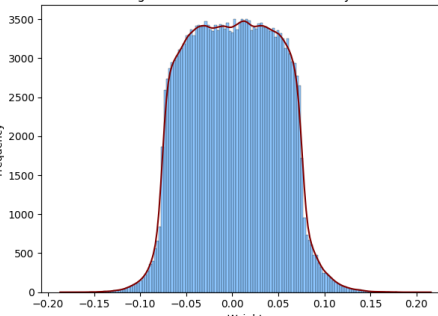
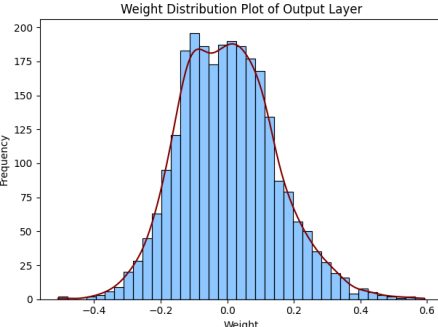
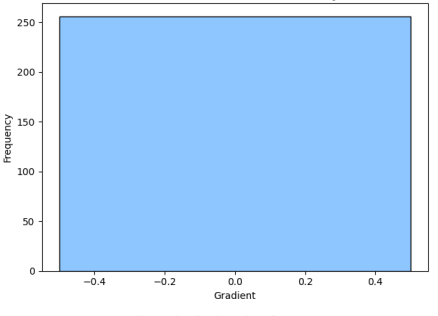
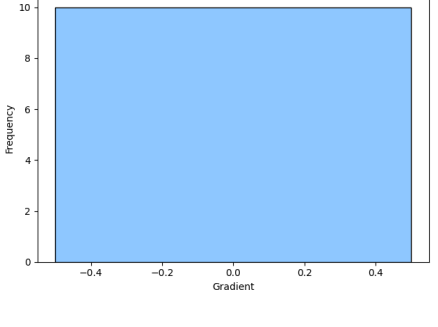
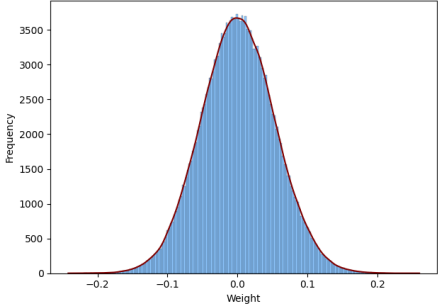
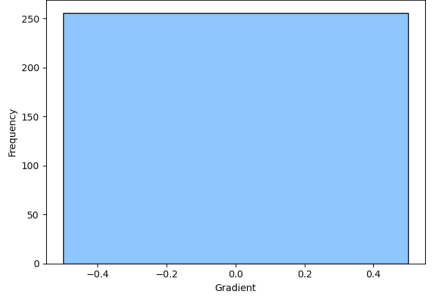
| Inisialisasi bobot | Accuracy | Loss | Grafik |
|--------------------|----------|------|--------|
|--------------------|----------|------|--------|

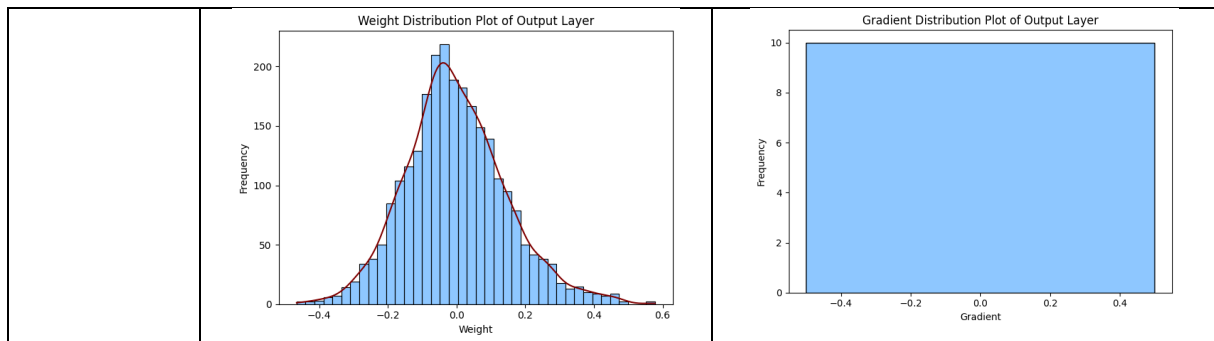
| | | | |
|-----------------------|--------|--------|--|
| Zero Initialization | 0.1100 | 0.2311 |  <p>Training and Validation Loss</p> <p>Loss</p> <p>Epochs</p> <p>Training Loss</p> <p>Validation Loss</p> |
| Random Uniform | 0.7850 | 0.3157 |  <p>Training and Validation Loss</p> <p>Loss</p> <p>Epochs</p> <p>Training Loss</p> <p>Validation Loss</p> |
| Random Normal | 0.1100 | 0.2311 |  <p>Training and Validation Loss</p> <p>Loss</p> <p>Epochs</p> <p>Training Loss</p> <p>Validation Loss</p> |
| Xavier Initialization | 0.8250 | 0.0625 |  <p>Training and Validation Loss</p> <p>Loss</p> <p>Epochs</p> <p>Training Loss</p> <p>Validation Loss</p> |

| | | | |
|-------------------|--------|--------|--|
| He Initialization | 0.8450 | 0.0533 |  |
|-------------------|--------|--------|--|

Serta grafik distribusi bobot dan gradien sebagai berikut:

| Inisialisasi bobot | Distribusi bobot | Distribusi gradien |
|---------------------|---|---|
| Zero Initialization |   |   |
| Random Uniform |   |   |

| | | |
|----------------------------------|--|--|
| <p>Random Normal</p> | <p>Weight Distribution Plot of Hidden Layer 1</p>  <p>Weight Distribution Plot of Output Layer</p>  | <p>Gradient Distribution Plot of Hidden Layer 1</p>  <p>Gradient Distribution Plot of Output Layer</p>  |
| <p>Xavier Initialization</p> | <p>Weight Distribution Plot of Hidden Layer 1</p>  <p>Weight Distribution Plot of Output Layer</p>  | <p>Gradient Distribution Plot of Hidden Layer 1</p>  <p>Gradient Distribution Plot of Output Layer</p>  |
| <p>He Initialization</p> | <p>Weight Distribution Plot of Hidden Layer 1</p>  | <p>Gradient Distribution Plot of Hidden Layer 1</p>  |



3.4.2. Hasil Analisis

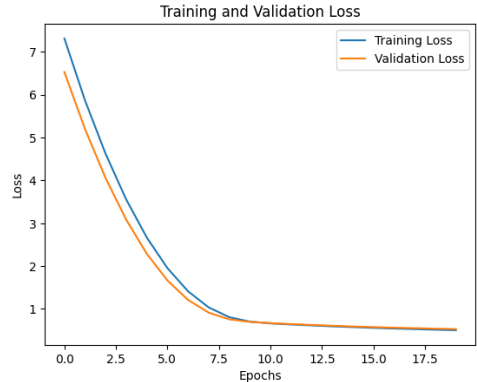
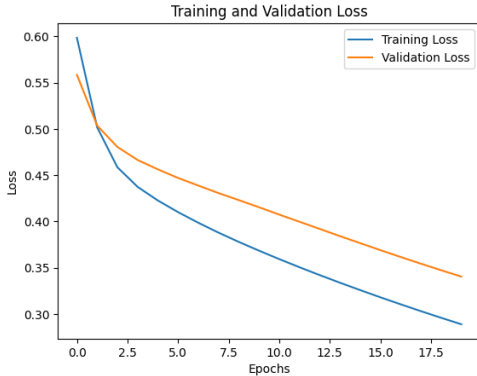
Dari hasil pengujian, pembelajaran dengan *zero initialization*, dan *random normal* menghasilkan model yang buruk, pembelajaran dengan *random uniform* menghasilkan model yang cukup baik, dan pembelajaran dengan *Xavier Initialization* dan *He Initialization* menghasilkan model yang baik. *Zero initialization* menghasilkan model yang buruk karena terlalu *overfit* sehingga dari proses *training*, setiap *layer* memperoleh hasil yang sama, sedangkan *random normal* menghasilkan model yang buruk karena tidak mempertimbangkan jumlah *node* dari *layer* sebelumnya dan sangat tergantung pada nilai *mean* dan *variance*. *Random uniform* menghasilkan model yang cukup baik karena mampu menghasilkan bobot yang tidak terlalu acak, tetapi tidak nol. Namun, model ini masih kurang baik karena tidak mempertimbangkan *layer* sebelumnya. Dua metode inisialisasi terakhir mampu menghasilkan model yang jauh lebih baik karena keduanya mempertimbangkan jumlah input yang mereka terima.

3.5. Pengaruh Regularisasi

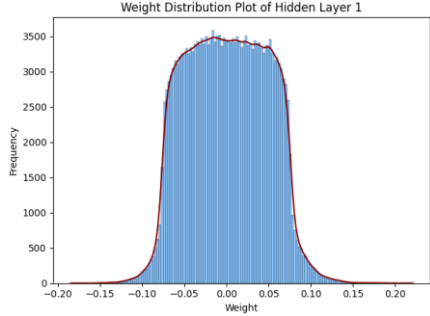
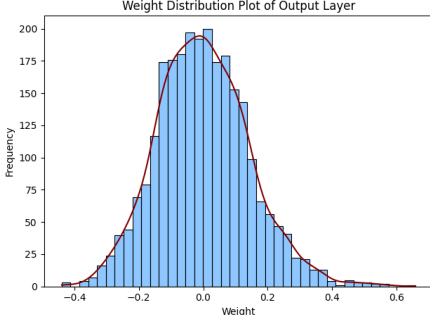
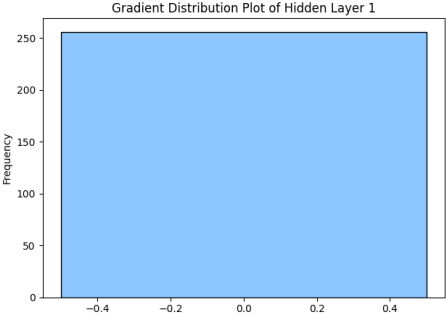
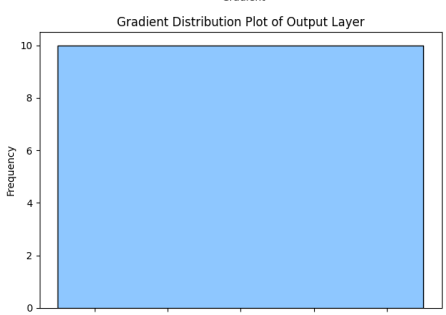
3.5.1. Hasil Pengujian

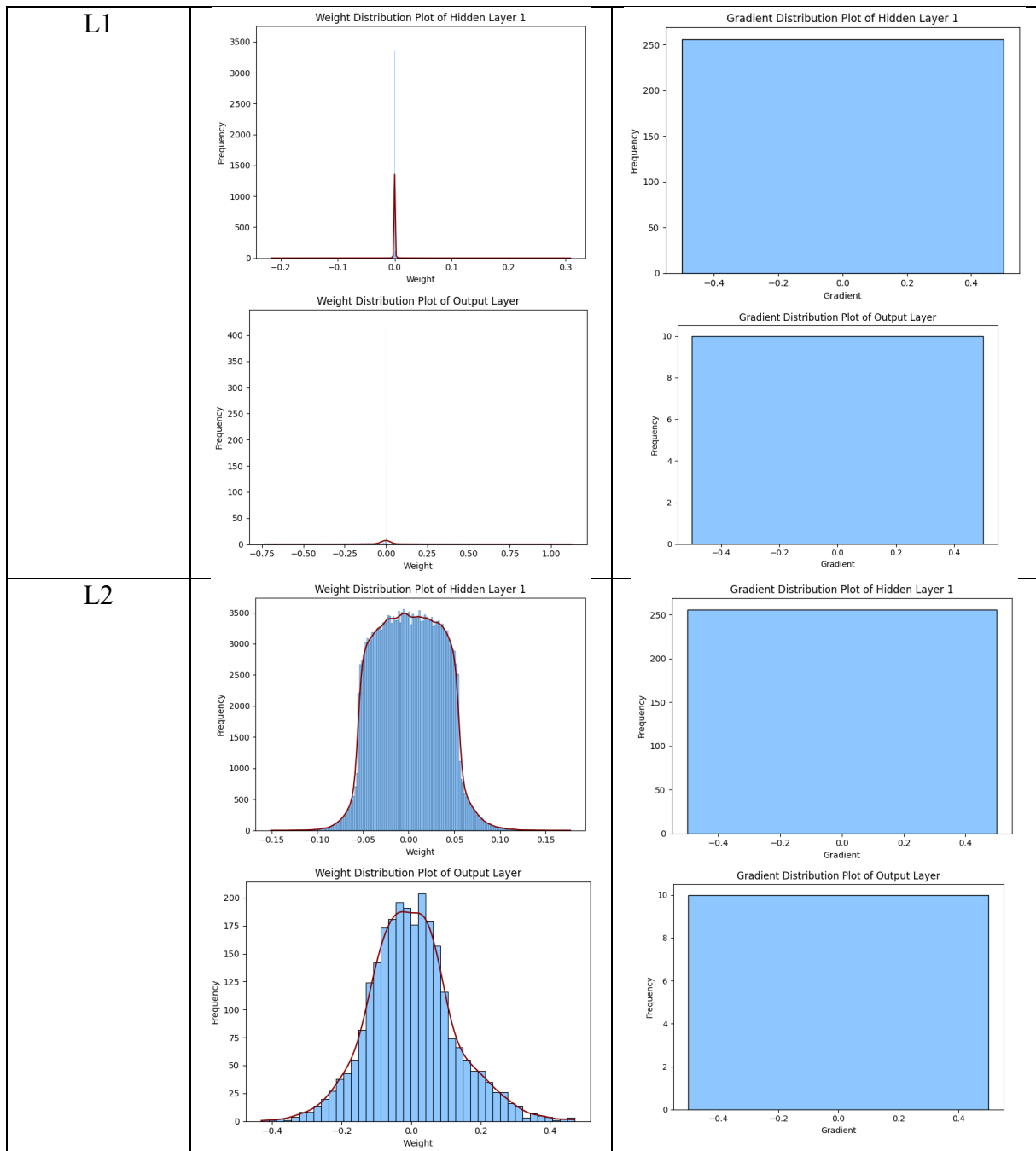
Pengujian model terhadap pengaruh inisiasi bobot dilakukan dengan nilai-nilai berikut (nilai *width* untuk *hidden layer* sama yaitu 256):

| Reguularisasi | Accuracy | Loss | Grafik |
|--------------------|----------|--------|---|
| Tanpa regularisasi | 0.8400 | 0.0557 | <p>The graph shows Training Loss (blue line) starting at approximately 0.19 and decreasing steadily to about 0.005 by epoch 18. Validation Loss (orange line) starts at approximately 0.15, decreases to a minimum of about 0.075 at epoch 5, and then slightly increases to about 0.078 by epoch 18.</p> |

| | | | |
|----|--------|--------|--|
| L1 | 0.7550 | 0.5315 |  |
| L2 | 0.8550 | 0.3333 |  |

Serta grafik distribusi bobot dan gradien sebagai berikut:

| Reguularisasi | Distribusi bobot | Distribusi gradien |
|--------------------|---|---|
| Tanpa regularisasi |   |   |



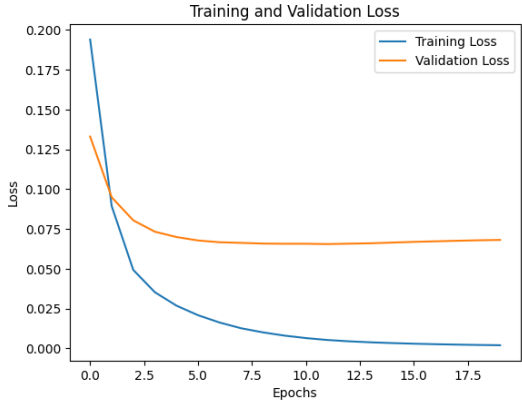
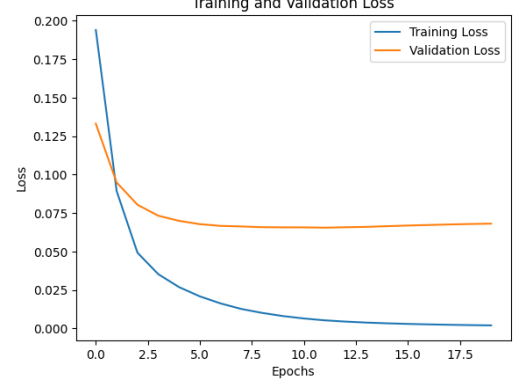
3.5.2. Hasil Analisis

Berdasarkan percobaan, regularisasi dengan L2 menghasilkan model yang baik, tetapi regularisasi dengan L1 menghasilkan model yang lebih buruk daripada model tanpa regularisasi. Regularisasi L1 membuat bobot model terlalu kecil bahkan menyentuh nol sehingga model tidak terlatih dengan baik. Berbeda dengan regularisasi L2 yang bobotnya cukup tersebar. Hal ini diakibatkan penalti yang diberikan L1 terlalu besar sehingga merusak bobot model.

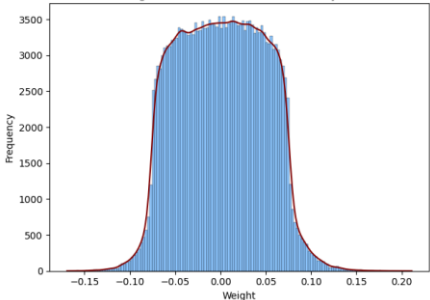
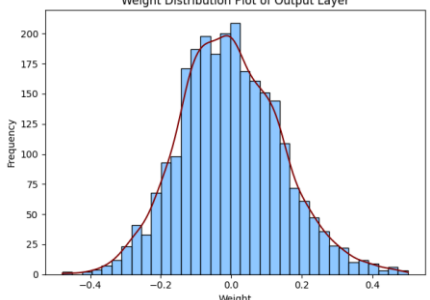
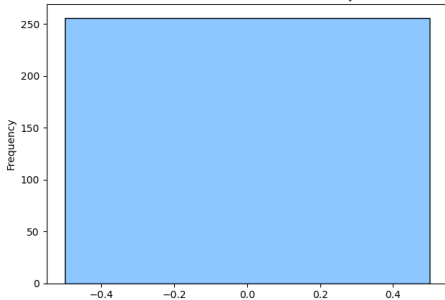
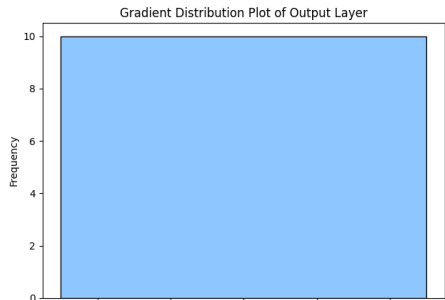
3.6. Pengaruh Normalisasi RMSNorm

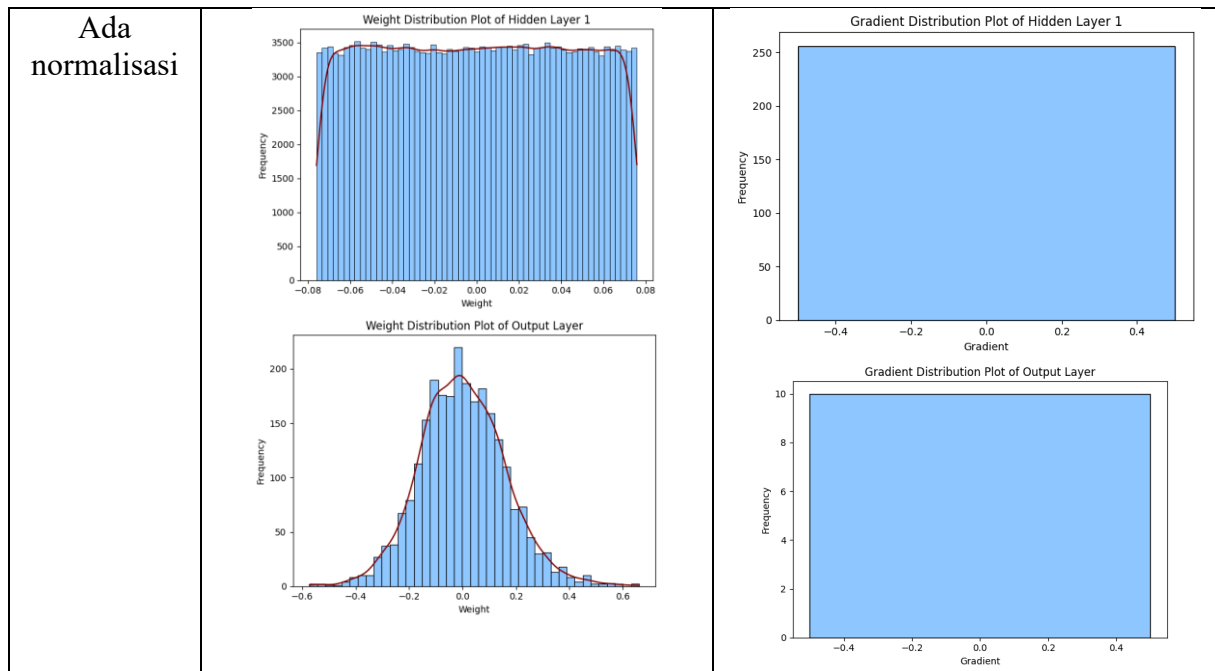
3.6.1. Hasil Pengujian

Pengujian model terhadap pengaruh inisiasi bobot dilakukan dengan nilai-nilai berikut (nilai *width* untuk *hidden layer* sama yaitu 256):

| Normalisasi | Accuracy | Loss | Grafik |
|-------------------|----------|--------|---|
| Tanpa normalisasi | 0.8300 | 0.0593 |  |
| Ada normalisasi | 0.8400 | 0.0582 |  |

Serta grafik distribusi bobot dan gradien sebagai berikut:

| Normalisasi | Distribusi bobot | Distribusi gradien |
|-------------------|--|--|
| Tanpa normalisasi |   |   |



3.6.2. Hasil Analisis

Proses normalisasi berhasil memberikan model yang lebih baik. Hal tersebut dikarenakan data dinormalisasi sehingga persebaran bobot lebih merata. Data yang terdistribusi secara normal sangat mendukung dalam terciptanya model yang baik.

3.7. Perbandingan dengan SKLearn

3.7.1. Hasil Pengujian

Perbandingan model yang kami buat dengan model dari *library* sklearn sebagai berikut (nilai *width* untuk *hidden layer* sama yaitu 256 dengan *depth* 4):

| Model | Accuracy | Loss |
|---------|----------|--------|
| Scratch | 0.8450 | 0.0708 |
| SKLearn | 0.8224 | 0.7397 |

3.7.2. Hasil Analisis

Dari hasil pengujian, *library* sklearn menghasilkan akurasi yang lebih rendah dibandingkan model buatan. Hal tersebut mungkin disebabkan oleh sedikitnya *layer* yang digunakan (2 *hidden layer* dengan 256 *node*). Selain itu, *optimizer* yang digunakan untuk sklearn adalah SGD, padahal sklearn masih memiliki *optimizer* lainnya yang lebih baik dibandingkan SGD, misalnya *optimizer* adam.

BAB IV

KESIMPULAN DAN SARAN

4.1. Kesimpulan

Feedforward Neural Network memiliki bobot yang begitu banyak sehingga waktu pelatihan yang lama belum tentu menghasilkan model yang baik. Banyak faktor yang memengaruhi kualitas model FFNN, misalnya bentuk *training data*, nilai *learning rate*, inisialisasi bobot, bahkan metode yang digunakan dalam FFNN. Hal-hal tersebut menjadi pertimbangan untuk membuat model FFNN yang baik.

4.2. Saran

Untuk membuat FFNN, gunakan fungsi aktivasi yang memang sudah terjamin, misalnya sigmoid dan softmax. Selain itu, *learning rate* pun harus dijaga agar tidak terlalu besar atau terlalu kecil. Jumlah *layer* dan lebar masing-masing *layer* pun tidak boleh terlalu banyak karena mempersulit proses *training*. Regularisasi layak digunakan untuk menghasilkan model yang baik. Data juga sebaiknya dinormalisasi, supaya bobot dapat lebih tersebar. Inisialisasi bobot pun harus diperhatikan. Jadi, pembuatan FFNN memerlukan banyak percobaan.

PEMBAGIAN TUGAS

| Nama | Tugas |
|---------------------------|---|
| Irfan Sidiq Permana | Model FFNN |
| | Insialisasi bobot |
| | Fungsi aktivasi (linear, ReLU) |
| | Fungsi loss (MSE) |
| | Save dan load |
| | Forward dan backward propagation |
| | Regularisasi |
| | Automatic differentiation |
| | RMSNorm |
| Bryan Cornelius Lauwrence | Fungsi aktivasi (sigmoid, tanh, softmax, dan GeLU) |
| | Visualisasi ANN |
| | Visualisasi distribusi bobot dan gradien |
| | Analisis hasil pengujian |
| Ahmad Hasan Albana | Fungsi aktivasi (SiLU) |
| | Fungsi loss (Binary Cross-Entropy, Categorical Cross-Entropy) |
| | Verbose |
| | Histori proses pelatihan |

REFERENSI

- <https://www.geeksforgeeks.org/plotting-histogram-in-python-using-matplotlib/>
- <https://www.geeksforgeeks.org/training-and-validation-loss-in-deep-learning/>
- <https://www.youtube.com/watch?v=VMj-3S1tku0>
- https://keras.io/guides/sequential_model/
- <https://medium.com/data-science/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1>