Laporan Tugas Besar 1 IF3270 Pembelajaran Mesin

Feedforward Neural Network

Semester 2 Tahun 2024/2025

Disusun oleh:

Hugo Sabam Augusto	13522129
Muhamad Zaki	13522136
Ahmad Rafi Maliki	13522137

TEKNIK INFORMATIKA

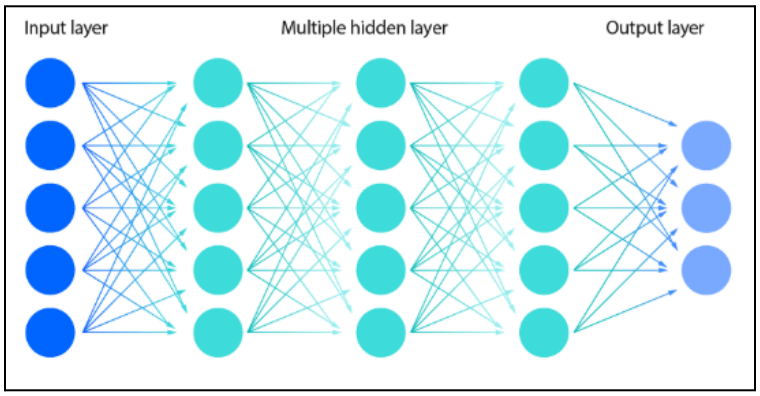
INSTITUT TEKNOLOGI BANDUNG

2025

Daftar Isi

Bagian 1.
Deskripsi Persoalan

Pada Tugas Besar 1 mata kuliah IF3270 Pembelajaran Mesin, kami akan mengimplementasikan **Feed Forward Neural Network (FFNN)** dalam bahasa pemrograman Python dari nol tanpa menggunakan pustaka pembelajaran mesin seperti Scikit-Learn, Tensorflow, atau PyTorch. Tugas ini bertujuan untuk memberikan pemahaman mendalam tentang cara kerja *Artificial Neural Network*, termasuk *forward propagation*, *backward propagation*, *activation function,* dan *loss function*.



Gambar 1. *Artificial Neural Network*

Kami akan membangun FFNN menggunakan berbagai kombinasi antara *activation function* dan juga *loss function,* mengujinya menggunakan dataset serta melakukan analisis terhadap hasil eksperimen. Implementasi ini akan memberikan wawasan tentang *artificial neural network* khususnya yang bertipe *feed forward* serta pengaruh *hyper-parameter*-nya.

Model yang dibangun ini nantinya akan digunakan untuk memprediksi *dataset* mnist_784 yaitu berupa gambar digit yang ditulis tangan sehingga menjadi tantangan bagi mesin untuk memprediksi digit apa sebenarnya tertulis pada gambar tersebut.

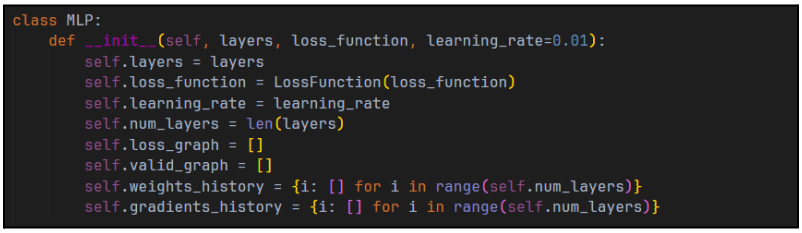
Bagian 2.
Pembahasan

2.1. Penjelasan Implementasi

2.1.1. Deskripsi kelas beserta atribut dan methodnya

1. Kelas MLP (Multi Layer Perceptron)

Kelas MLP adalah inti dari model yang kami buat, kelas ini merupakan representasi dari sebuah *artificial neural network* yang kami modelkan dalam bahasa pemrograman Python. Pada kelas ini mekanisme *forward propagation* dan *backward propagation* didefinisikan.



```python
class MLP:
    def __init__(self, layers, loss_function, learning_rate=0.01):
        self.layers = layers
        self.loss_function = LossFunction(loss_function)
        self.learning_rate = learning_rate
        self.num_layers = len(layers)
        self.loss_graph = []
        self.valid_graph = []
        self.weights_history = {i: [] for i in range(self.num_layers)}
        self.gradients_history = {i: [] for i in range(self.num_layers)}
```

Gambar 2.1.1.1 Atribut Kelas MLP

Tabel 2.1.1.1 Atribut Kelas MLP

Atribut	Tipe	Deskripsi
layers	list[class.Layer]	List yang berisikan *instance* dari kelas *layer*
loss_function	class.LossFunc	Instance dari kelas LossFunc
learning_rate	float	Menyimpan data *learning rate*
num_layers	int	Banyaknya layer (hidden + output) yang dimiliki oleh MLP
loss_graph	list	Menyimpan data *loss* untuk plot

valid_graph	list	Menyimpan data *val* untuk plot
weights_history	dictionary	Menyimpan data *weights* untuk plot
gradients_history	dictionary	Menyimpan data *gradient* untuk plot

Gambar 2.1.1.2 Method Kelas MLP

Tabel 2.1.1.2 Method Kelas MLP

1.	**prosedur forward(X)**
X: nilai input yang akan di propagasikan melalui sebuah layer, berupa matriks	
Prosedur ini mendefinisikan mekanisme ***forward propagation*** pada MLP. Cara kerjanya adalah melakukan perkalian matriks nilai input X dari seluruh *instance* data pada batch dengan nilai *weight* yang berkorespondensi. Fungsi ini memanfaatkan perkalian array pada pustaka numpy untuk melakukan perhitungan nilai secara sekaligus per *batch* per *layer*. Setelah prosedur ini selesai dijalankan, berarti proses propagasi maju pada FFNN sudah selesai	

untuk sebuah *batch.*	
2.	**prosedur backward(X, y_true)**
X: nilai input yang digunakan saat *forward propagation,* berupa matriks y_true: nilai label dari X, berupa matriks	
Prosedur ini mendefinisikan mekanisme ***backward propagation*** pada MLP. Cara kerja nya adalah dengan melakukan *looping* mundur dari nilai *loss* sampai ke *hidden layer* pertama. Di tiap layer, dihitung nilai delta *weight* yang dan disimpan. Baru setelah *batch* selesai, nilai *weight* akan di update secara sekaligus.	
3.	**prosedur train(X, y, X_val, Y_val, epochs, batch_size, verbose)**
X: nilai input yang akan digunakan untuk pembelajaran, berupa matriks y: nilai label dari X, berupa matriks X_val: nilai input yang akan digunakan untuk menghitung nilai validasi, berupa matriks y_val: nilai label dari X_val, berupa matriks epochs: banyaknya *epoch* saat proses pembelajaran, berupa integer batch_size: banyaknya *instance* input saat melakukan propagasi maju verbose: nilai boolean, jika True maka *progress* dari proses train akan dicetak	
Prosedur yang meng-abstraksi pemanggilan prosedur forward dan juga prosedur backward. Pada prosedur ini proses belajar dari sebuah MLP akan dilaksanakan secara utuh. Mulai dari *forward propagation*, *backward propagation*, hingga *weight update* untuk tiap *batch* sebanyak *epoch* yang telah ditentukan.	
4.	**fungsi accuracy(X, y_true) → accuracy_score**
X: nilai input yang akan digunakan untuk uji akurasi	

Y_true: nilai label dari X, berupa matriks output: nilai skor akurasi model, berupa float	
Menghitung skor akurasi model berdasarkan data uji, mengembalikan skor akurasi.	
5.	**fungsi predict(X) → predictions**
X: nilai input yang akan diprediksi kelasnya output: *list* berupa kelas hasil prediksi tiap *instance*	
Melakukan prediksi terhadap data uji menggunakan model yang sudah dilatih, mengembalikan list berupa kelas data uji.	
6.	**fungsi plot_loss() → loss_graph**
tidak memiliki parameter	
Memanfaatkan pustaka python ‘matplotlib’ untuk menghasilkan grafik *loss function.*	
7.	**fungsi plot_weight_distribution() → wight_dist_graph**
tidak memiliki parameter	
Memanfaatkan pustaka python ‘matplotlib’ untuk menghasilkan grafik distribusi *weight.*	
8.	**fungsi plot_gradient_distribution() → gradient_dist_graph**
tidak memiliki parameter	
Memanfaatkan pustaka python ‘matplotlib’ untuk menghasilkan grafik	

distribusi *gradient.*	
9.	**prosedur save(filepath)**
filepath: alamat file pada komputer untuk menyimpan data model	
Memanfaatkan pustaka python ‘pickle’ untuk menyimpan *object instance* dari model MLP ke *binary file*.	
10.	**prosedur load(filepath)**
filepath: alamat file pada komputer tempat data model disimpan	
Memanfaatkan pustaka python ‘pickle’ untuk memuat *object instance* dari model MLP dari *binary file.*	

2. Kelas Layer

Kelas Layer merupakan pemodelan dari tiap lapisan pada *artificial neural network.* Lapisan yang dimaksud bisa berupa *hidden layer* dan *output layer*.

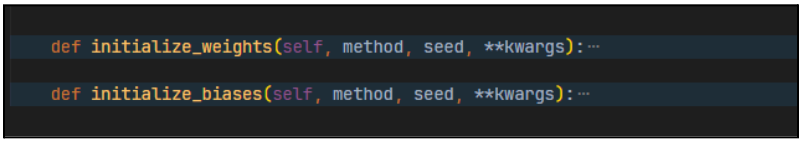
```
class Layer:
    def __init__(self, input_size, n_neurons, activation,
                 weight_init='random_normal', bias_init='zeros',
                 seed=42, **kwargs):

        self.input_size = input_size
        self.n_neurons = n_neurons
        self.activation = Activation(activation)
        self.seed = seed
        self.weights = self.initialize_weights(weight_init, seed, **kwargs)
        self.biases = self.initialize_biases(bias_init, seed, **kwargs)
```

Gambar 2.1.1.3 Atribut Kelas Layer

Tabel 2.1.1.3 Atribut Kelas Layer

Atribut	Tipe	Deskripsi
input_size	int	Banyaknya *input* yang dapat

		diterima oleh tiap neuron pada *layer*
n_neurons	int	Banyaknya neuron pada *layer*
activations	class.Activations	*Instance* dari kelas Activations, menyimpan data fungsi aktivasi beserta turunannya.
seed	int	Seed untuk menghasilkan hasil *random* yang serupa
weights	list.list.float	Menyimpan *weight* yang dimiliki oleh tiap neuron pada *layer* berupa list float dua dimensi (matriks)
biases	list.list.float	Menyimpan *weight* yang dimiliki oleh bias neuron pada *layer* berupa list float dua dimensi (matriks)



Gambar 2.1.1.4 Method Kelas Layer

Tabel 2.1.1.4 Method Kelas Layer

1.	**Fungsi intialize_weights(method, seed, **kwargs) → weight_list**
method: metode inisialisasi weight, berupa string seed: random seed untuk *reproducibility*, berupa int	

**kwargs: argumen lainnya untuk tiap metode inisialisasi bobot	
Melakukan inisialisasi bobot sesuai metodenya masing-masing.	
2.	**Fungsi intialize_biases(method, seed, **kwargs) → weight_list**
method: metode inisialisasi weight, berupa string seed: random seed untuk *reproducibility*, berupa int **kwargs: argumen lainnya untuk tiap metode inisialisasi bobot	
Melakukan inisialisasi bobot sesuai metodenya masing-masing.	

3. Kelas Activation

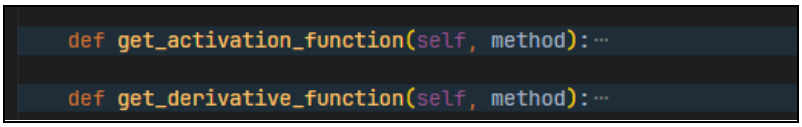
Kelas Activation merupakan kelas yang mengabstraksi fungsi aktivasi dan juga turunannya. Kelas ini dapat diinstansiasi sesuai metode perhitungan aktivasi yang diinginkan.

```
class Activation:
    def __init__(self, method):
        self.method = method
        self.activate = self.get_activation_function(method)
        self.derive = self.get_derivative_function(method)
```

Gambar 2.1.1.5 Atribut Kelas Activation

Tabel 2.1.1.5 Atribut Kelas Activation

Atribut	Tipe	Deskripsi
method	string	Nilai string dari sebuah jenis fungsi aktivasi
activate	function	Fungsi aktivasi itu sendiri
derive	function	Fungsi turunan dari fungsi aktivasi



Gambar 2.1.1.6 Atribut Kelas Activation

Tabel 2.1.1.6 Method Kelas Activation

1.	**Fungsi get_activation_function(method) → function**
method: metode aktivasi output: fungsi aktivasi	
Mengembalikan fungsi aktivasi sesuai metode yang ditentukan saat instansiasi objek.	
2.	**Fungsi get_derivative_function(method) → function**
method: metode aktivasi output: fungsi turunan dari fungsi aktivasi	
Mengembalikan fungsi turunan dari fungsi aktivasi sesuai metode yang ditentukan saat instansiasi objek.	

```
    def linear(self, x):...

    def linear_derivative(self, x):...

    def relu(self, x):...

    def relu_derivative(self, x):...

    def sigmoid(self, x):...

    def sigmoid_derivative(self, x):...

    def tanh(self, x):...

    def tanh_derivative(self, x):...

    def softmax(self, x):...

    def softmax_derivative(self, x):...
```

Gambar 2.1.1.7 Fungsi Aktivasi dan Fungsi Turunannya

4. Kelas WeightInit

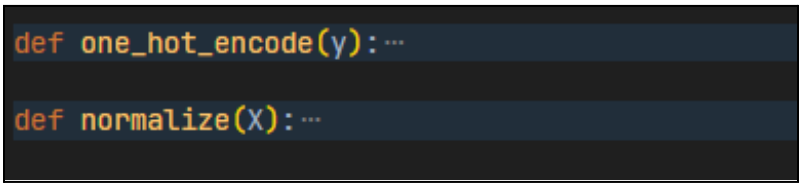
Kelas ini berisikan *static functions* untuk tiap metode inisialisasi bobot yang diimplementasikan. Kelas ini akan dipanggil oleh kelas Layer saat melakukan instansiasi. Implementasi dari tiap fungsi mengikuti keterangan formula yang disediakan pada dokumen spesifikasi.

```
class WeightInit:
    def zeros(size):...

    def random_uniform(size, low=-1, high=1, seed=42):...

    def random_normal(size, mean=0, std=1, seed=42):...

    def xavier_uniform(size, seed=42):...

    def xavier_normal(size, seed=42):...

    def he_uniform(size, seed=42):...

    def he_normal(size, seed=42):...
```

Gambar 2.1.1.8 Fungsi Inisialisasi Bobot

5. Fungsi Utilitas

Berikut merupakan kumpulan dari fungsi utilitas yang kami implementasikan untuk melakukan abstraksi terhadap beberapa fungsi.



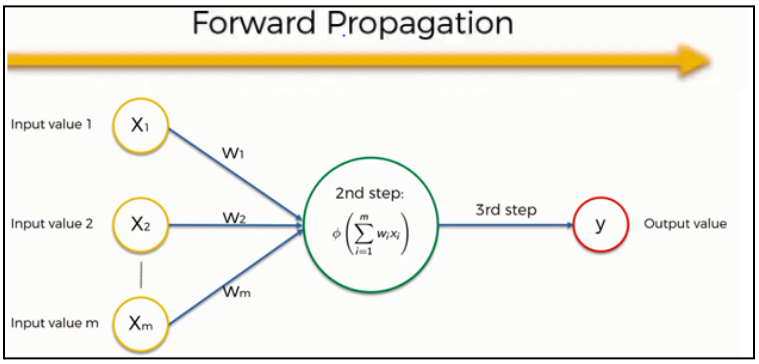
Gambar 2.1.1.9 Fungsi Utilitas

Tabel 2.1.1.7 Method Kelas Activation

3.	**Fungsi one_hot_encode(y) → y**
y: nilai label dari dataset, berupa matriks output: nilai label yang sudah ter-*encode*, berupa matriks	
Melakukan *one hot encoding* terhadap data label, mengembalikan label yang sudah ter-*encode*	
4.	**Fungsi normalize(X) → X**
X: nilai fitur pada dataset, berupa matriks output: nilai X yang sudah dinormalisasi	
Melakukan normalisasi terhadap nilai fitur menjadi rentang 0-1	

2.1.2. Penjelasan *forward propagation*

Forward propagation (tahapan inferensi) adalah proses perubahan *input* atau masukan menjadi *output* atau keluaran dalam sebuah *feedforward neural network (***FFNN***)*. Namun pada implementasinya nanti, tahapan ini diakhiri oleh perhitungan nilai *loss* yaitu nilai kesalahan yang dihasilkan oleh **FFNN** relatif terhadap nilai sebenarnya.



Gambar 2.1.2. *Forward Propagation*

Tahapan ini melibatkan seluruh neuron pada **FFNN** yang terdiri dari *input layer, hidden layer (neurons),* dan *output layer*. Input layer adalah nilai fitur yang dimiliki oleh sebuah *instance* data, sedangkan hidden layer yang terdiri dari banyak neuron berperan sebagai perantara untuk melakukan berbagai operasi matematis yang akan mengkombinasikan nilai-nilai yang diperoleh dari *layer* sebelumnya. Terakhir di *output layer* akan dihasilkan keluaran dari **FFNN** tersebut.

Tiap *layer* akan terdiri dari satu atau banyak neuron (direpresentasikan sebagai sebuah simpul pada graf) yang independen satu sama lain, namun terhubung secara lengkap dengan semua simpul yang terletak di *layer* sebelum atau sesudahnya yang dihubungkan oleh *edge* yang memiliki *weight* masing-masing.

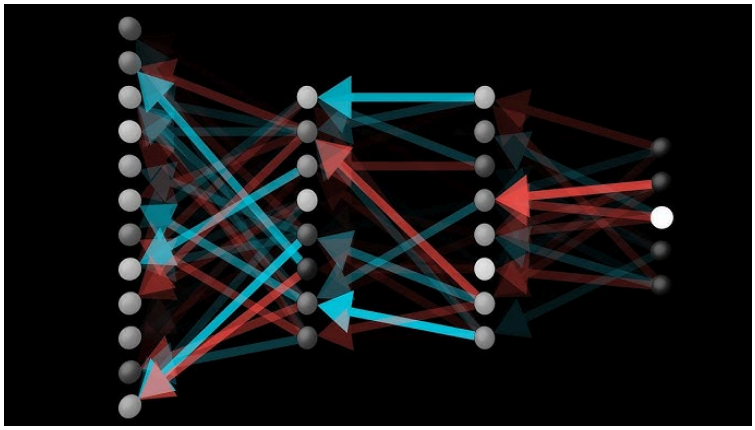
Pada *forward propagation*, dimulai dari *layer* pertama yaitu *input layer* (pada gambar simpul berwarna kuning), seluruh nilai fitur dari sebuah *instance* data dan juga nilai bias akan dikombinasikan secara linear dengan nilai bobot *(weight*) nya masing-masing, hasil kombinasi ini kemudian disebut sebagai *'net'* dan akan dimasukkan kedalam fungsi aktivasi sebuah neuron pada *layer* selanjutnya (bergerak maju) dan akan menjadi nilai yang dimiliki oleh neuron pada *layer* tersebut (pada gambar simpul berwarna hijau). Nilai neuron pada *layer* ini kemudian akan berperan sebagai *input* untuk neuron yang berada pada *layer* setelahnya, hal ini terus berulang sampai tibanya di *layer* terakhir yaitu *output layer*.

Penggunaan pustaka Python NumPy beserta implementasi mekanisme *batch gradient descent*, mengizinkan implementasi kode kami untuk melakukan perkalian matriks antara nilai fitur tiap *instance* pada batch dengan bobotnya secara sekaligus tanpa perlu memperbaharui bobotnya. Hal ini mempercepat waktu eksekusi secara signifikan.

Nilai yang dihasilkan pada *output layer* merupakan keluaran atau hasil prediksi (inferensi) dari **FFNN** terhadap *instance* data tersebut. Pada proses pembelajaran, akan dihitung pula nilai *loss* nya relatif dengan nilai sebenarnya untuk mengevaluasi seberapa baik model yang telah dibangun. Nilai ini nantinya akan bermanfaat untuk proses evaluasi dan pembelajaran.

2.1.3. Penjelasan *backward propagation* dan *weight update*

Backward propagation (tahapan belajar) merupakan tahapan setelah *forward propagation*. Pada tahapan ini, model akan belajar untuk menjadi model yang lebih baik. Berkebalikan dengan *forward propagation*, aliran data pada proses ini bergerak mundur dari nilai *loss*, ke *output layer*, hingga ke *input layer*.



Gambar 2.1.3 *Backward Propagation*

Inti dari tahapan ini adalah untuk menghitung pengaruh tiap *edge* (*weight*) yang menghubungkan tiap simpul (neuron) terhadap nilai *loss*. Secara matematis, yang dicari adalah gradien atau turunan dari tiap simpul terhadap nilai *loss.* Nilai turunan pada simpul yang terletak pada *layer* yang tidak berhubungan secara langsung oleh nilai *loss* dapat diperoleh dengan aturan rantai pada kalkulus.

Tujuan dari mengetahui nilai gradien tiap simpul adalah untuk melakukan *weight update.* Mengetahui nilai gradien berarti mengetahui pengaruh nilai dari tiap *edge* terhadap nilai *loss*. Model yang baik adalah model dengan nilai *loss* yang minimal. Dengan mengetahui pengaruh ini, kita dapat meng-*update* bobot tiap *edge* ke arah yang akan meminimalkan nilai loss.

Pada implementasinya saat melakukan *weight update*, bobot tiap *edge* akan ditambah oleh negatif dari hasil perkalian antara gradien *edge* tersebut dengan *learning rate*. Nilai negatif ini yang akan berkontribusi terhadap menurunnya nilai *loss*. Nilai *learning rate* adalah nilai yang menjadi faktor penentu seberapa cepat nilai *weight* sebuah edge akan berubah terhadap faktor gradien.

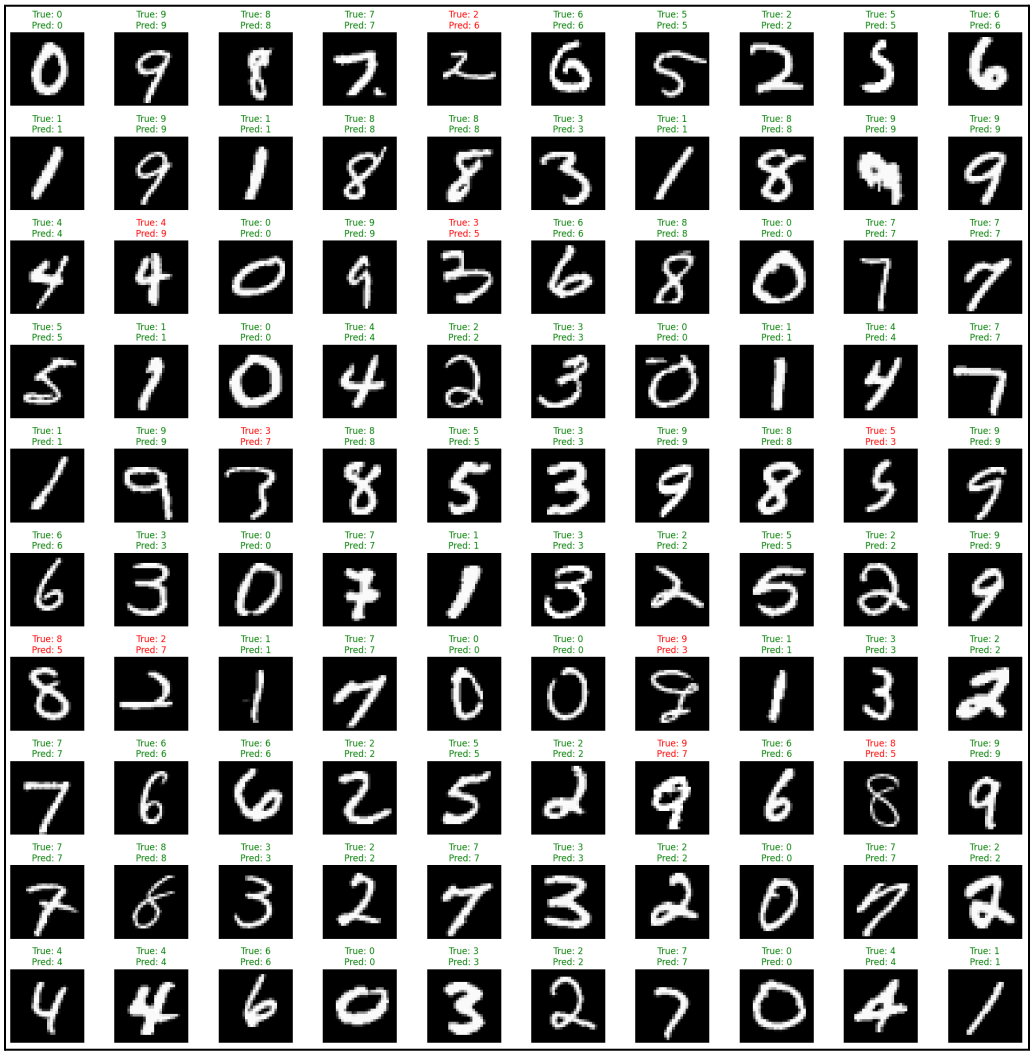
Karena mengimplementasikan mekanisme *batch gradient descent*, nilai *weight* tiap *edge* baru diperbaharui setelah propagasi selesai untuk *batch* tersebut.

Selain menggunakan mekanisme manual, kami juga mencoba untuk mengimplementasikan bonus *autodiff*. Menggunakan mekanisme ini, proses *backward propagation* dan juga *weight update* sudah diabstraksi dalam kelas Value, sehingga secara pemrograman, mekanisme *backward propagation* tidak perlu secara eksplisit diimplementasikan. Sayangnya mekanisme ini membuat proses *forward propagation* sangat lambat, sehingga tidak diimplementasikan pada hasil akhir.

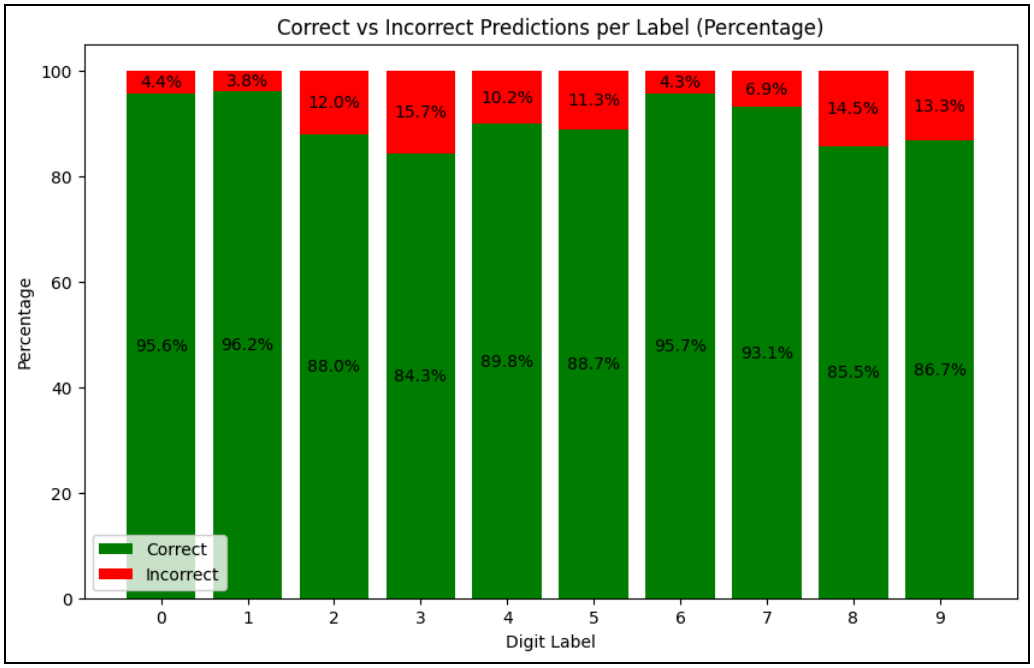
Setelah siklus *forward propagation* dan *backward propagation* selesai, yang diakhiri oleh *weight update* pada model, satu *epoch* telah dilalui oleh model **FFNN** ini. Nilai *weight* tiap *edge* sudah berubah dan diharapkan model menjadi lebih baik dalam melakukan prediksi data. Siklus ini dapat dilakukan berulang kali sampai model menghasilkan nilai *loss* yang kecil atau sudah memuaskan.

2.2. Hasil Pengujian

2.2.1. Tangkapan Layar Hasil Pengujian



Gambar 2.2.1. Hasil Pengujian Dengan 100 Sampel Acak



Gambar 2.2.2. Statistik Pengujian Dengan 1000 Sampel Uji

Bisa diamati pada gambar 2.2.2. Bahwa digit 3 dan 8 paling banyak mengalami kesalahan prediksi. Hal tersebut mungkin saja terjadi karena bentuk 3 dan 8 yang cukup mirip sehingga membuat model salah memprediksi.

2.2.2. Pengaruh *depth* dan *width*

Pada **FFNN**, *depth* (kedalaman) adalah seberapa dalam atau berapa banyak *layer* yang dimiliki oleh model tersebut. Sedangkan, *width* (lebar) adalah seberapa banyak neuron di tiap *layer*-nya.

Sejatinya **FFNN**, hanyalah sekumpulan persamaan matematis kompleks yang direpresentasikan sebagai sebuah graph. Secara teoritis, semakin dalam sebuah **FFNN** maka model tersebut dapat melakukan komputasi matematis yang lebih kompleks namun pengaruh (gradien) di tiap *layer* nya akan semakin kecil dan semakin sulit untuk mencapai konvergensi. Sehingga proses pembelajaran akan lebih lama. Disisi lain, semakin lebar sebuah **FFNN**, akan lebih mudah belajar karena pengaruh tiap *edge* terhadap simpul akhir akan semakin kuat, sehingga gradien akan mencapai konvergensi lebih cepat.

Kombinasi tepat dari kedua komponen ini menjadi penentu seberapa baik semua model dalam melakukan pembelajaran.

Pada eksperimen yang kami lakukan dengan model yang kami bangun terhadap *data set* mnist_784, kami mendapati bahwa dengan menggunakan width (jumlah neuron per layer) yang lebih besar didapatkan pula *accuracy* yang lebih baik, akan tetapi depth (jumlah hidden layer) tidak serta merta meningkatkan *accuracy*. Untuk detail pengujian kami berikan data dibawah ini.

Tabel 2.2.2.1 *Hyperparameter* pengujian *depth* dan *width*

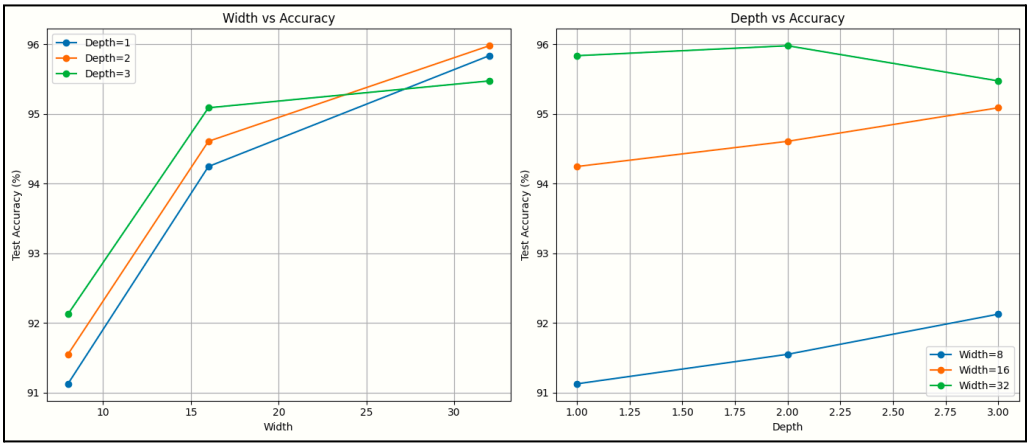
Hyperparameter			
Input Layer	784 Neuron	Wh He Uniform	Wb Zero
Hidden layer ReLu	Diujikan	Wh He Uniform	Wb Zero
Output Layer SoftMax	10 neuron	Wh He Uniform	Wb Zero
Loss Function	CCE		
Learning Rate	0.1		
Epoch	10		
Mini batch size	64		
Train data (MNIST)	60.000		

Didapatkan hasil,

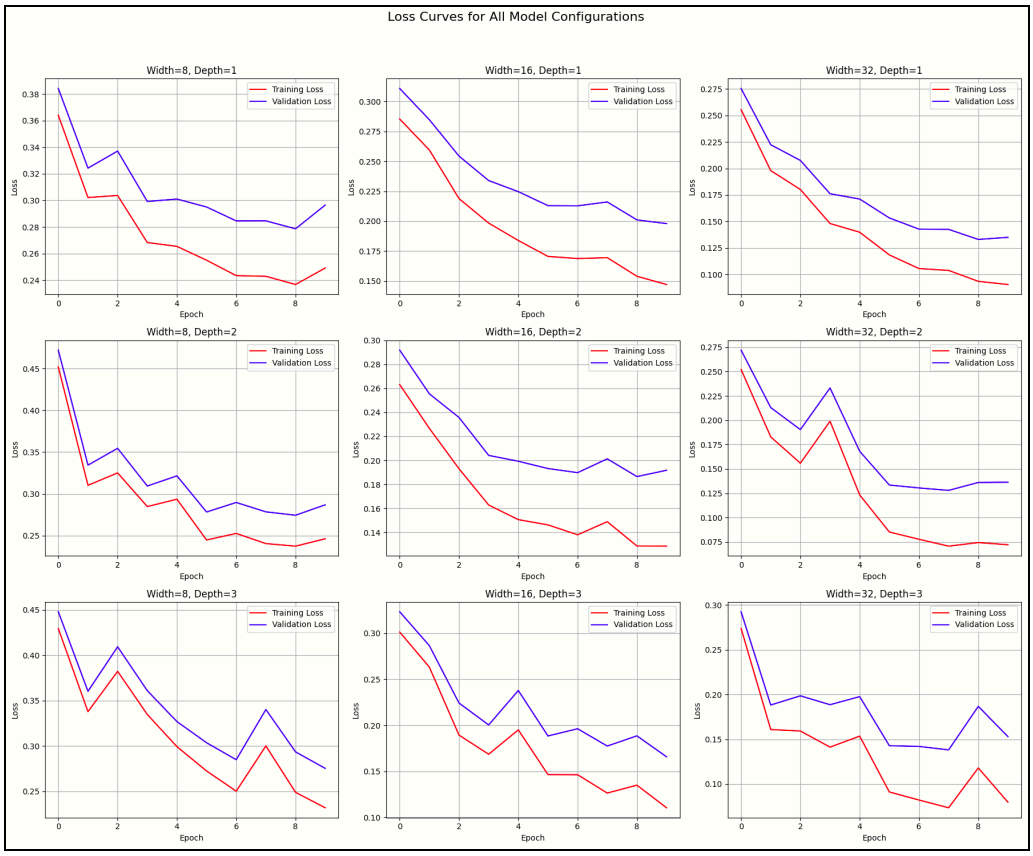
Tabel 2.2.2.2 Hasil pengujian *depth* dan *width*

	Width	**Depth**	**Test Accuracy**
1	8	1	91.12

2	8	2	91.55
3	8	3	92.12
4	16	1	94.24
5	16	2	94.6
6	16	3	95
7	32	1	95.8
8	32	2	95.9
9	32	3	95.4



Gambar 2.2.2.1 Grafik *width* dan *depth* terhadap akurasi



Gambar 2.2.2.2 Grafik *plot loss* semua model

Dari pengujian diatas kami mengetahui bahwa model terbaik adalah model yang menggunakan width 32 dengan akurasi 95.9 % pada depth 2. Hal ini kami simpulkan karena dengan semakin banyaknya neuron dalam suatu layer, memungkinkan model untuk belajar lebih kompleks askan tetapi menambahkan lebih banyak hidden layer tidak selalu meningkatkan akurasi karena bisa menyebabkan adanya *overfitting* sehingga performa model malah menurun. Pengaruh depth terhadap overfitting terlihat dari meningkatnya training loss pada model dengan width 32 depth 3, yang menunjukkan bahwa model menjadi lebih sulit untuk dioptimalkan. Hal ini semakin diperjelas dengan tren validation loss yang tidak stabil atau meningkat, yang menunjukkan bahwa model tidak dapat mempertahankan kinerja optimalnya pada data validasi.

2.2.3. Pengaruh fungsi aktivasi hidden layer

Fungsi aktivasi adalah fungsi matematis yang dimiliki oleh tiap simpul untuk melakukan manipulasi lebih lanjut terhadap nilai *net* yang merupakan hasil kombinasi linear dari seluruh nilai input. Fungsi aktivasi sangat banyak jenisnya, yang kami implementasikan pada tugas ini adalah Linear, ReLU, Sigmoid, Hyperbolic Tangent (tanh), dan Softmax.

Tiap fungsi aktivasi yang digunakan akan mempengaruhi bagaimana sebuah model belajar karena nilai tiap simpul akan berbeda. Untuk mengetahui fungsi aktivasi mana yang paling cocok untuk digunakan pada sebuah data perlu dilakukan analisis dan juga eksperimen.

Pada eksperimen yang kami lakukan dengan model yang kami bangun terhadap *data set* mnist_784, kami mendapatkan hasil sebagai berikut.

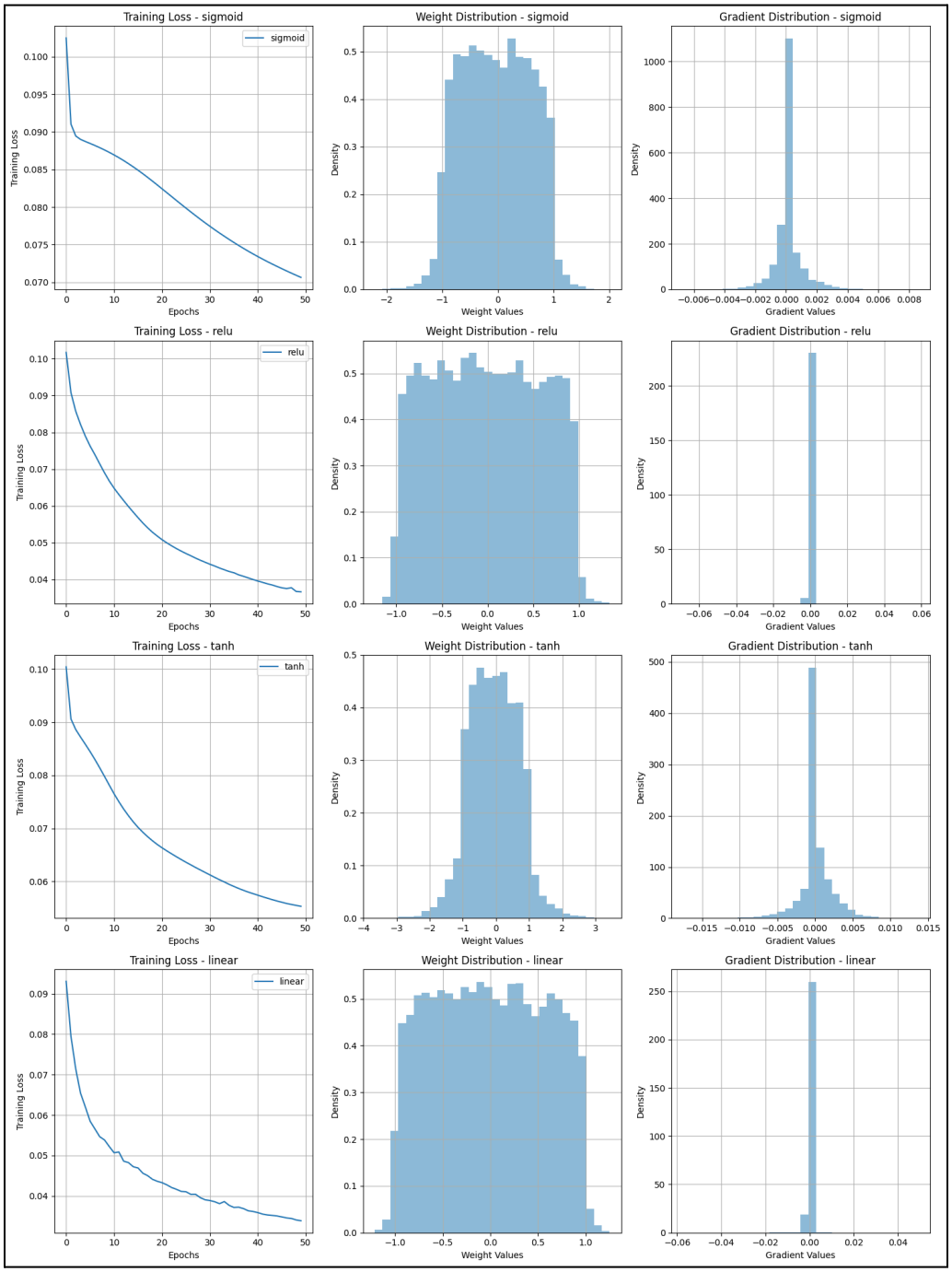
Tabel 2.2.3.1 *Hyperparameter* pengujian aktivasi hidden layer

Hyperparameter			
Input Layer	784 Neuron		
Hidden layer <aktivasi diujikan>	16 Neuron	Wh Uniform (-1,1)	Wb Zero
Hidden Layer <aktivasi diujikan>	8 Neuron	Wh Uniform (-1,1)	Wb Zero
Output Layer Sigmoid	10 neuron	Wh Uniform (-1,1)	Wb Zero
Loss Function	MSE		
Epoch	50		
LR	0.1		
Mini batch size	1000		

Train data (MNIST)	49.000
Test data (MNIST)	21.000

Tabel 2.2.3.2 *Hyperparameter* pengujian *aktivasi*

	Activation Function			
	ReLu	**Sigmoid**	**Linear**	**Tanh**
Test Accuracy	76.12%	53.61%	82.50%	59.69%
Train Loss	0.0366	0.0706	0.0339	0.0553
Val Loss	0.0368	0.0707	0.0341	0.0556



Gambar 2.2.3.2 Grafik *plot loss* semua model

Seperti yang diketahui, dataset MNIST merupakan klasifikasi gambar dengan 784 fitur / gambar dengan ukuran 28x28 bit dengan isi warna 0-255 dari warna putih ke hitam. Dilihat dari hasil plotting diatas serta keakuratan tiap

aktivasi, fungsi aktivasi sigmoid memiliki tingkat keakuratan paling kecil yaitu 53.61% dan tingkat keakuratan terbesar adalah fungsi aktivasi linear dengan nilai 82.50%. Sigmoid & tanh memiliki gradien yang lebih tersebar dibandingkan linear dan relu yang lebih terkonsentrasi di 0.00 atau -0.01.

Pengaruh aktivasi terhadap gradien itu sendiri terutama dalam sigmoid & tanh bisa menyebabkan gradient vanishing, kondisi dimana turunan menjadi sangat kecil dan akan dikalikan ulang berulang kali di setiap lapisan sebelumnya, menyebabkan bobot di lapisan awal sulit diperbaharui, sehingga pelatihan tidak baik dan tidak optimal. Pengaruh loss function juga berperan cukup penting dalam klasifikasi, karena MSE mengasumsikan keluaran bersifat kontinu padahal klasifikasi bersifat diskrit (kelas 0-9), gradien MSE juga memperparah aktivasi sigmoid di output layer.

Selanjutnya mengapa vanishing gradient tidak terjadi pada linear dan relu karena bila dilihat keakuratan, mereka cenderung cukup baik. Kita tahu bahwa rumus aktivasi linear itu adalah f(x) = x dan turunannya adalah 1. Disini gradien selalu 1 sehingga tidak mengalami vanishing gradient. Untuk relu juga lebih tidak mengalami vanishing gradient karena rumus relu ReLU(x) = max(0,x) dan turunannya yaitu ambil 1 bila x > 0 dan 0 bila x <= 0. x >0 membuat gradien tetap 1 dan x < 0 gradien menjadi 0, namun lebih stabil daripada sigmoid / tanh.

2.2.4. Pengaruh *learning rate*

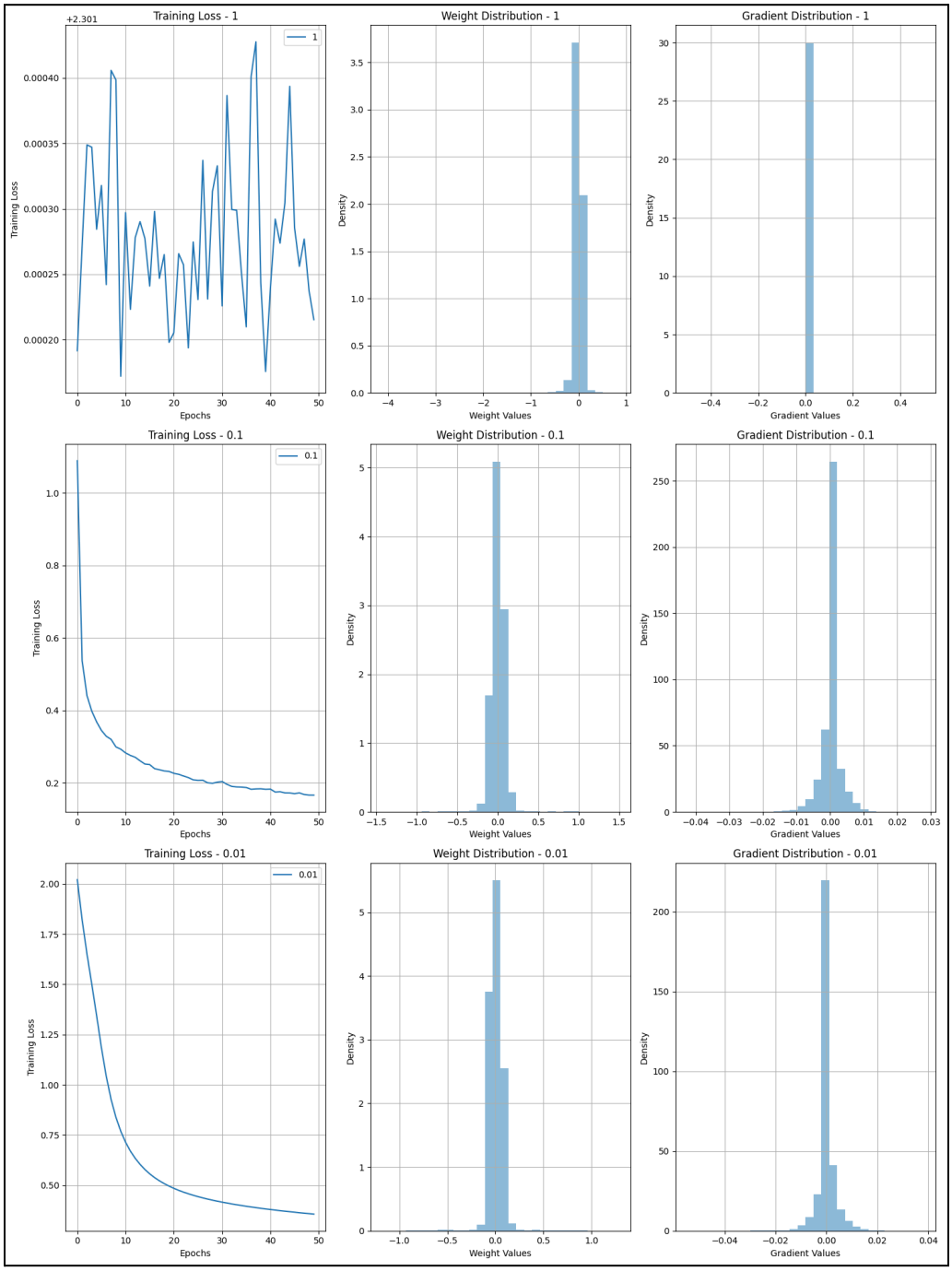
Tabel 2.2.4.1 *Hyperparameter* pengujian *learning rate*

Hyperparameter			
Input Layer	784 Neuron		
Hidden layer 1 ReLu	16 Neuron	Wh He Uniform	Wb Zero
Hidden Layer 2 ReLu	8 Neuron	Wh He Uniform	Wb Zero
Output Layer SoftMax	10 neuron	Wh He Uniform	Wb Zero
Loss Function	CCE		
Epoch	50		
Mini batch size	1000		
Train data (MNIST)	49.000		
Test data (MNIST)	21.000		

Cek hasil akhir,

Tabel 2.2.4.2 Hasil pengujian *learning rate*

1	0.1	0.01
Test Accuracy=11.25% Train Loss=2.3012 Val Loss=2.3012	Test Accuracy=94.15% Train Loss=0.1662 Val Loss=0.2055	Test Accuracy=89.45% Train Loss=0.3565 Val Loss=0.3762



Gambar 2.2.4.1 *Plotting* pengujian learning rate

Berdasarkan data diatas, terlihat jelas bahwa perbedaan laju learning rate sangat mempengaruhi bobot yang tersebar di seluruh layer. Hal ini bisa terjadi karena sesuai dengan rumus learning rate:

$$W_{new} = W_{old} - \alpha \left(\frac{\partial \mathcal{L}}{\partial W_{old}} \right)$$

α = Learning rate

Gambar 2.2.4.2 Fungsi *weight update*

Learning rate mengatur seberapa cepat bobot di tiap layer untuk fit atau menuju arah yang diharapkan dengan bantuan gradien, turunan dari loss terhadap bobotnya. Dibuktikan dengan eksperimen learning 0.1 dan 0.01. Hasil yang didapatkan dari learning rate 0.1 lebih baik dibandingkan 0.01 dengan tingkat keakuratan lebih tinggi sekitar 4-5%. Tak hanya itu kami juga testing bila learning rate dinaikkan lebih tinggi yaitu 1. Hasilnya cukup terduga karena learning rate yang terlalu tinggi dapat menyebabkan gradient exploding, yaitu kondisi dimana loss tidak turun atau bahkan meningkat karena perubahan bobot yang terlalu ekstrim sehingga akhirnya menjadi 0 (kondisi **overconfidence** , gambar distribusi plot LR = 1).

2.2.5. Pengaruh inisialisasi bobot

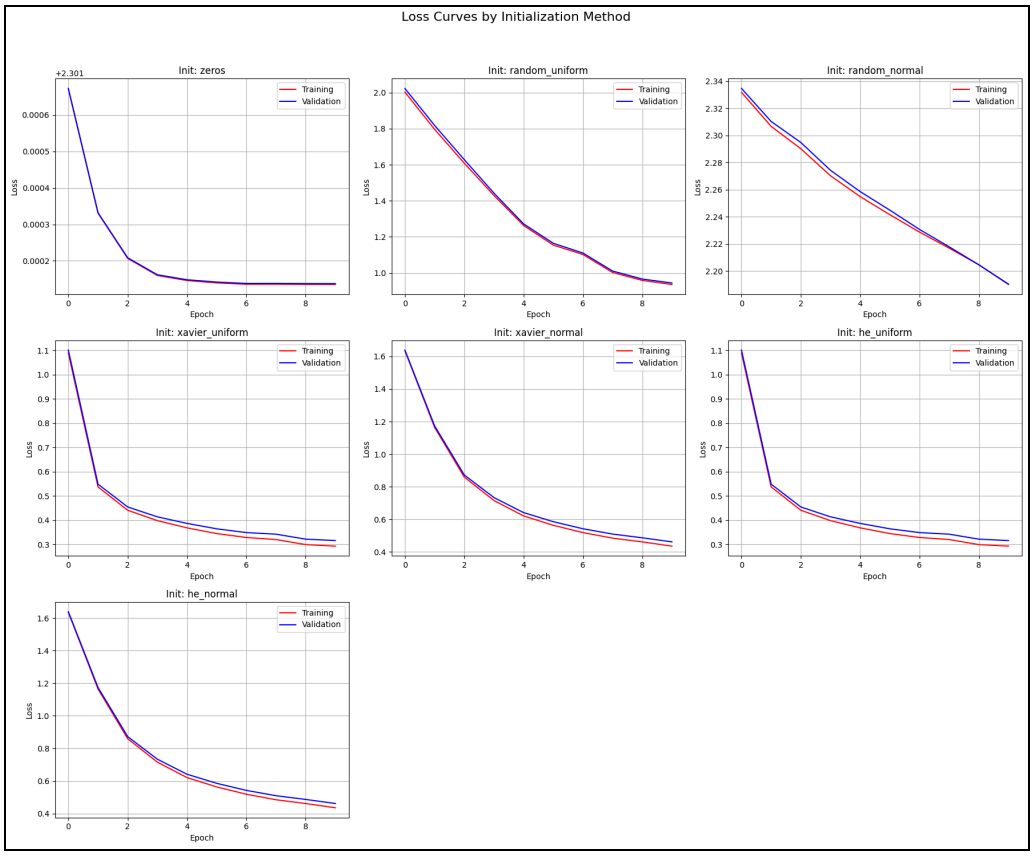
Tabel 2.2.5.1 *Hyperparameter* pengujian *aktivasi*

Hyperparameter			
Input Layer	784 Neuron		
Hidden layer ReLu	16	Wh He Uniform	Wb Zero
Hidden layer ReLu	8	Wh He Uniform	Wb Zero
Output Layer SoftMax	10 neuron	Wh He Uniform	Wb Zero

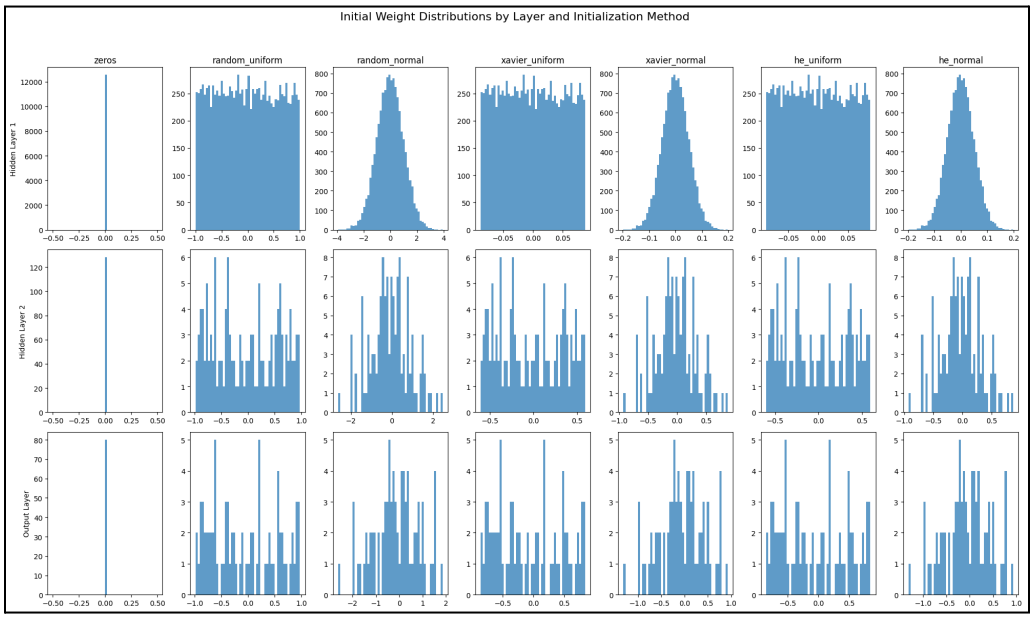
Loss Function	CCE
Learning Rate	0.1
Epoch	10
Mini batch size	1000
Train data (MNIST)	60.000

Tabel 2.2.5.2 Hasil Pengujian

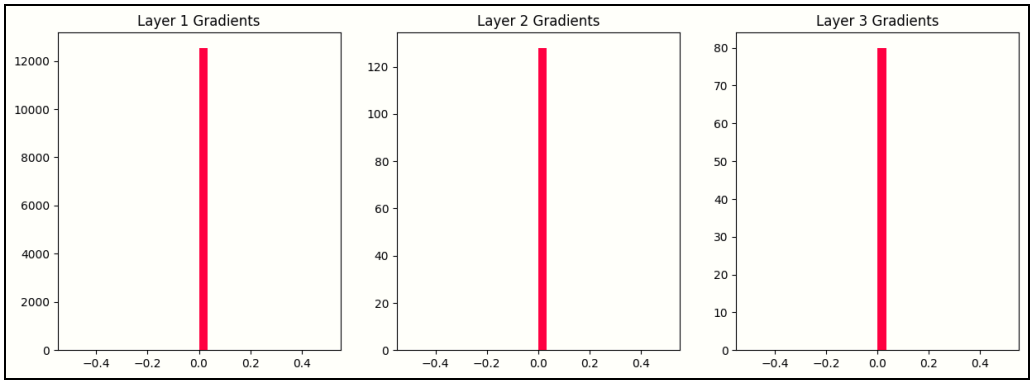
	Aktivasi	**Test Accuracy**
1	zeros	11.25
2	random_uniform	69.11
3	random_normal	17.50
4	xavier_uniform	90.97
5	xavier_normal	87.90
6	he_uniform	90.97
7	he_normal	87.90



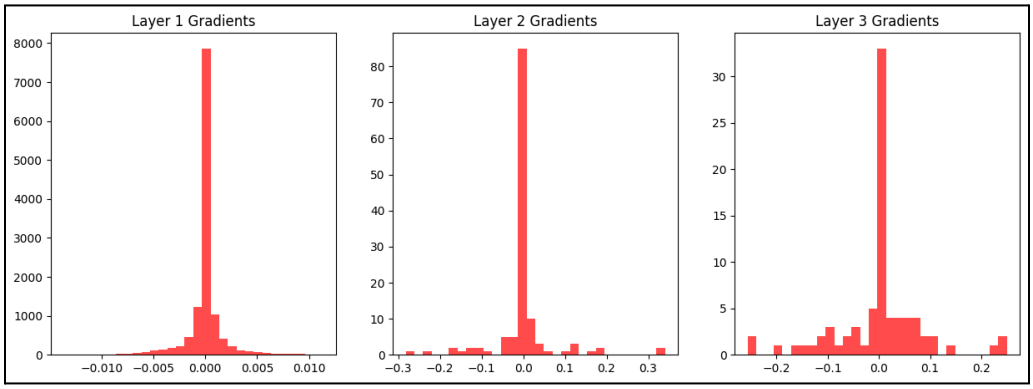
Gambar 2.2.5.1 Grafik kurva *loss function*



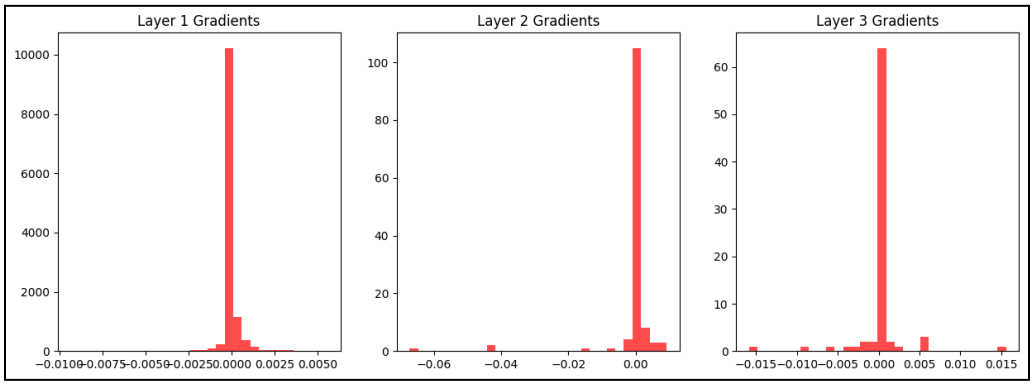
Gambar 2.2.5.2 Distribusi Bobot Awal



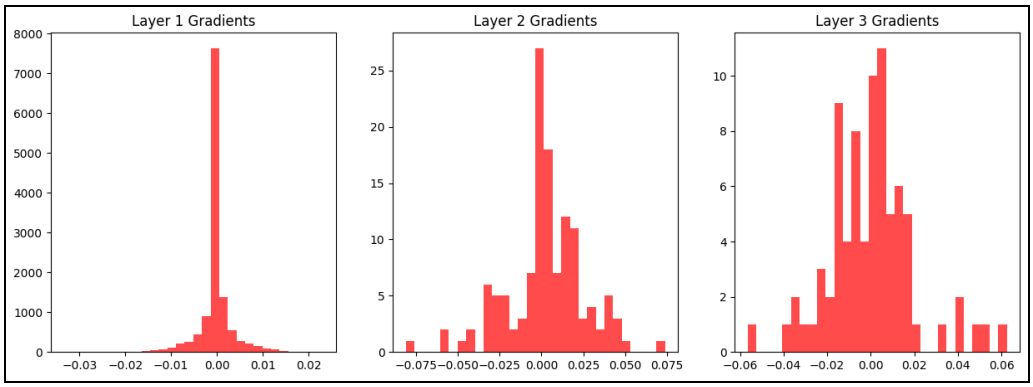
Gambar 2.2.5.3 Grafik distribusi gradient bobot Zero



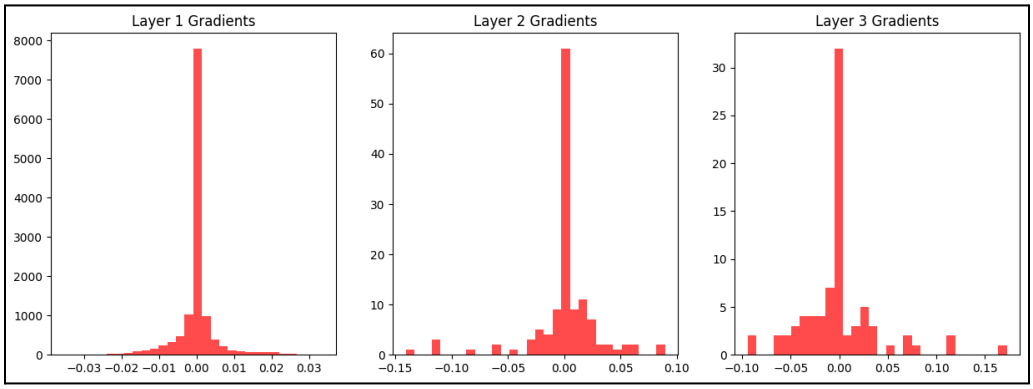
Gambar 2.2.5.4 Grafik distribusi gradient bobot random_uniform



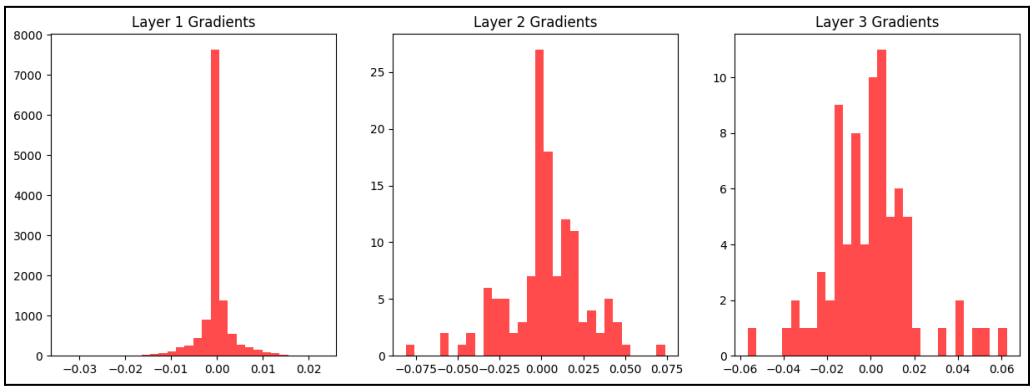
Gambar 2.2.4.5 Grafik distribusi gradient bobot random_normal



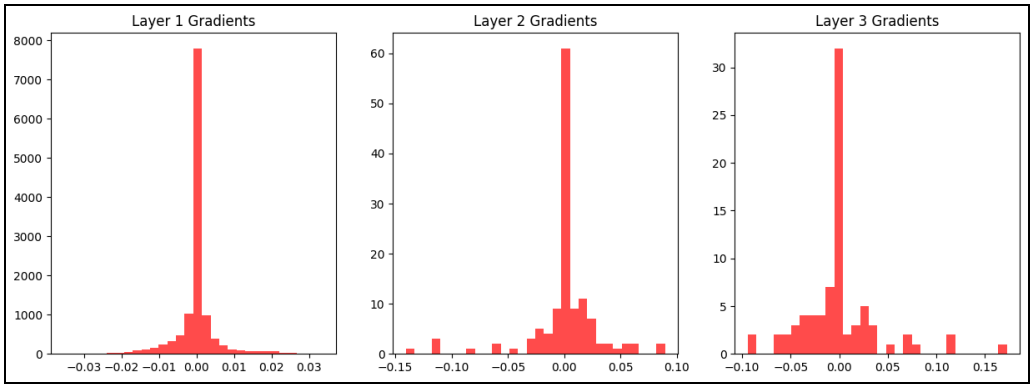
Gambar 2.2.5.6 Grafik distribusi gradient bobot xavier_uniform



Gambar 2.2.5.7 Grafik distribusi gradient bobot xavier_normal



Gambar 2.2.5.8 Grafik distribusi gradient bobot he_uniform



Gambar 2.2.5.9 Grafik distribusi gradient bobot he_normal

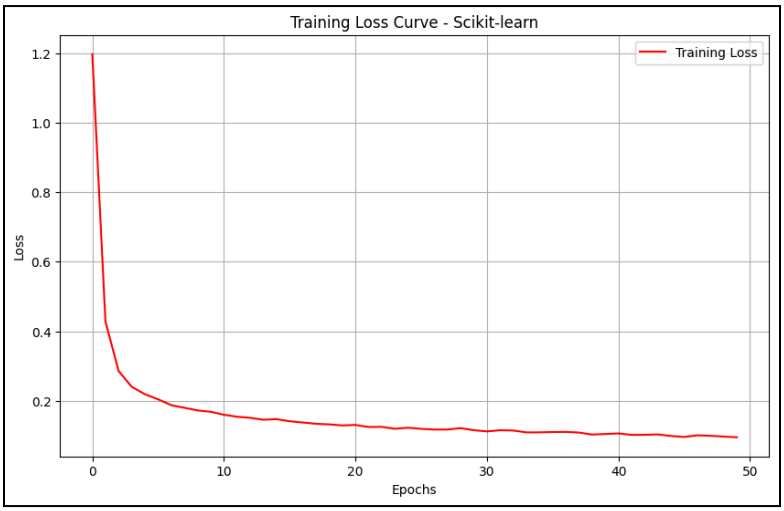
Inisialisasi bobot memiliki pengaruh yang signifikan terhadap kinerja model, karena tidak semua metode inisialisasi cocok dengan fungsi aktivasi yang digunakan. Pada pengujian di atas, terlihat bahwa metode Xavier Uniform dan He Uniform menghasilkan akurasi yang lebih tinggi dibandingkan metode lainnya, seperti zeros initialization.

Dari distribusi gradien yang ditampilkan, dapat diamati bahwa beberapa layer memiliki gradien yang terkonsentrasi di sekitar 0, yang dapat mengindikasikan masalah vanishing gradient. Hal ini terutama terjadi jika inisialisasi bobot tidak sesuai dengan fungsi aktivasi yang digunakan. Jadi pemilihan inisialisasi harus diimbangi dengan pengetahuan tentang fungsi aktivasi yang dipilih.

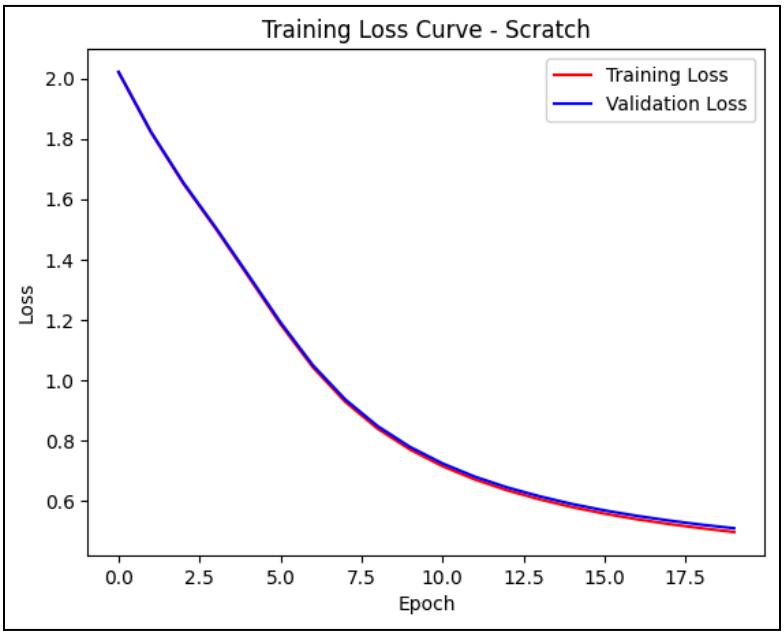
2.2.6. Perbandingan dengan pustaka scikit-learn

Tabel 2.2.6.1 *Hyperparameter* pengujian dengan pustaka Scikit-Learn

Hyperparameter			
Input Layer	784 Neuron		
Hidden layer ReLu	16	Wh He Uniform	Wb Zero
Hidden layer ReLu	8	Wh He Uniform	Wb Zero
Output Layer SoftMax	10 neuron	Wh He Uniform	Wb Zero
Loss Function	CCE		
Learning Rate	0.01		
Epoch	20		
Mini batch size	1000		
Train data (MNIST)	60.000		



Gambar 2.2.6.1 Grafik kurva *loss function* pustaka Scikit-Learn



Gambar 2.2.6.2 Grafik kurva *loss function* implementasi *scratch*

Tabel 2.2.6.2 Tabel akurasi hasil pengujian perbandingan pustaka scikit-learn dan implementasi scratch

Model	Accuracy
Scratch	85.74%
Scikit MLP	94.07%

Dilihat dari hasil pengujian diatas, terbuktikan bahwa scikit MLP menghasilkan tingkat keakuratan yang lebih tinggi dibandingkan scratch. Hal ini bisa terjadi karena Scikit learn menggunakan solver adam, yang mana berfungsi secara adaptif mengatur learning rate dan momentum. Terbuktikan dari bentuk kurva loss train yang dihasilkan pada gambar plot diatas. Kurva plot train loss dari scikit lebih curam di awal epoch dibandingkan train loss dari scratch. Tak hnya itu Scikit juga sudah builtin regularisasi L2, alpha sebesar 0.0001.

Bagian 3.
Kesimpulan dan Saran

Kesimpulan

- Pengaruh depth and width
    - Model terbaik dengan width 32 dan depth 2 mencapai akurasi 95.9%
    - Lebih banyak neuron meningkatkan kapasitas model untuk belajar pola kompleks
    - Menambah hidden layer tidak selalu meningkatkan akurasi dan bisa menyebabkan overfitting
- Pengaruh aktivasi pada hidden layer
    - Sigmoid memiliki akurasi terendah sebesar 53.61% sedangkan linear memiliki akurasi tertinggi sebesar 82.5%
    - Sigmoid & tanh mengalami vanishing gradient, ditunjukkan oleh gradien yang lebih tersebar dalam plotting
    - Linear dan ReLU tidak mengalami vanishing grad karena linear memiliki turunan tetap, sama halnya juga dengan relu yang lebih stabil dibanding sigmoid/tanh
- Pengaruh learning rate
    - Learning rate 0.1 lebih baik dibanding 0.01 dengan akurasi lebih tinggi sekitar 4-5%
    - Learning rate yang terlalu tinggi bisa (di eksperimen LR = 1) bisa menyebabkan grad exploding, kondisi dimana loss tidak turun atau naik
- Pengaruh inisialisasi bobot
    - Inisialisasi bobot harus disertai dengan pemahaman terkait fungsi aktivasinya
    - Tidak bisa men-generalisir penggunaan bobot pada semua fungsi aktivasi
    - Inisialisasi dengan xavier dan he memiliki performa yang baik pada aktivasi ReLu dan softmax
- Perbandingan dengan scikit-learn MLP
    - Model dengan implementasi sci-kit menghasilkan performa yang lebih baik

Saran

1. Temukan cara untuk mengintegrasikan mekanisme auto-diff seperti pustaka micrograd dengan pustaka NumPy agar dapat melakukan proses propagasi maju dengan lebih cepat dan efisien.
2. Jika sudah, lakukan pull request ke repository micrograd agar berkontribusi terhadap repository milik co-founder OpenAI.
3. Lakukan pengujian menggunakan pustaka yang sudah ada seperti scikit-learn dari awal agar bisa membandingkan kinerjanya dengan model yang di-implementasi dari scratch (agar tidak cepat puas dengan hasil sendiri).
4. Lakukan eksplorasi di internet mengenai *hyperparameter* apa yang cocok untuk dataset mnist_784 agar tidak kebingungan kenapa model saya jelek sekali, ternyata *hyperparameter*-nya memang tidak cocok.
5. Implementasikan bonus yang belum diimplementasikan.
6. Lanjutkan pengujian untuk data set huruf.
7. Lanjutkan proyek sampai bisa membaca sederet angka pada foto.

Pembagian Tugas

Nama	NIM	Pekerjaan
Hugo Sabam Augusto	13522129	Implementasi kelas MLP, Implementasi fungsi loss MSE, ReLU, tanh , linear, dan fitur tambahan seperti plotting dan graf, menulis laporan.
Muhammad Zaki	13522136	Debug MLP, Implementasi save and load, inisialisasi bobot, sigmoid, menulis laporan.
Ahmad Rafi Maliki	13522137	Debug MLP, Implementasi fungsi *loss* BCE dan CCE, kelas Activation, kelas Layer, menulis laporan.

Lampiran

Pranala *repository* GitHub

https://github.com/miannetopokki/IF3270_Tubes1_MachineLearning

Referensi

A. Karpathy, "The spelled-out intro to neural networks and backpropagation: Building micrograd". YouTube, Agustus. 17, 2020. [Video Daring]. Tersedia: https://www.youtube.com/watch?v=VMj-3S1tku0.

Gambar 1. https://www.nature.com/articles/s41598-025-88845-0

Gambar 2.1.2. https://rpubs.com/la`xmikant/NeuralNetworks

Gambar 2.1.3. https://www.youtube.com/watch?v=Ilg3gGewQ5U

Sumber: https://medium.com/geekculture/the-story-behind-random-seed-42-in-machine-learning-b838c4ac290a