

Feedforward Neural Network

Laporan Tugas Besar 1
IF3270 Pembelajaran Mesin



Disusun Oleh Kelompok 41:

12821046 Fardhan Indrayesa

13522064 Devinzen

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

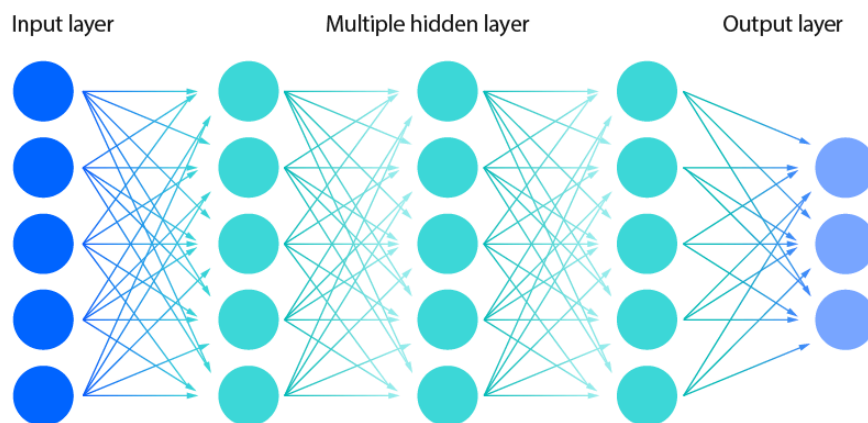
2025

Bab 1

Deskripsi Persoalan

Feedforward Neural Network (FFNN) adalah salah satu tipe *Artificial Neural Network* (ANN) yang menggunakan informasi mengalir dalam satu arah, yaitu dari *layer input* melalui *hidden layer* ke *layer output*, tanpa *looping* ataupun *feedback*. Tools ini biasanya digunakan untuk *pattern recognition* seperti klasifikasi gambar dan suara. Pada FFNN terdapat struktur *layer* yang menggambarkan aliran informasi diproses secara sekuensial melalui setiap *layer*, seperti yang ditunjukkan pada gambar di bawah. Struktur tersebut di antaranya.

1. *Input layer*: Layer ini terdiri atas sejumlah neuron yang menerima data input. Setiap neuron pada layer ini merepresentasikan *feature* pada data input.
2. *Hidden layer*: Satu atau lebih *hidden layer* ditempatkan di antara *layer input* dan *output*. *Layer-layer* ini digunakan dalam pembelajaran untuk mengenali pola kompleks dari data. Setiap neuron dalam *hidden layer* menerapkan jumlah input yang sudah dikali dengan bobot dan diikuti oleh fungsi aktivasi non-linear.
3. *Output layer*: Layer *output* menghasilkan *output* final dari suatu *network*. Jumlah neuron pada layer ini bergantung pada jumlah kelas dalam masalah klasifikasi atau jumlah *output* pada masalah regresi.



Setiap hubungan antara neuron-neuron pada *layer* ini memiliki bobot yang sudah disesuaikan pada saat proses pelatihan dengan tujuan untuk meminimalkan eror pada saat prediksi. Pada FFNN terdapat sebuah fungsi yang dinamakan fungsi aktivasi. Fungsi ini dapat mengenali

non-linearitas ke dalam network yang memungkinkannya mempelajari dan memodelkan pola data yang kompleks. Contoh fungsi aktivasi yang biasa digunakan, yaitu

1. Sigmoid : $\sigma(x) = 1/(1 + e^{-x})$
2. Tanh : $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$
3. ReLU : $\text{ReLU}(x) = \max(0, x)$

Pelatihan FFNN melibatkan penyesuaian bobot neuron untuk meminimalkan error antara *output* prediksi dan *output* sebenarnya. Proses ini biasanya dilakukan dengan menggunakan *backpropagation* dan *gradient descent*.

1. *Forward propagation*: Ketika dilakukan *forward propagation*, data input melewati *network* dan *output* dihitung.
2. Perhitungan *loss*: *Loss* (error) dihitung menggunakan fungsi *loss* seperti *Mean Squared Error* (MSE) untuk regresi atau *Cross-Entropy Loss* untuk klasifikasi.
3. *Backpropagation*: Pada *backpropagation*, nilai error dipropagasikan kembali melalui *network* untuk *update* bobot. Nilai gradien dari fungsi *loss* terhadap setiap bobot dihitung dan bobot-bobot tersebut disesuaikan menggunakan *gradient descent*.

Gradient descent adalah algoritma optimisasi yang digunakan untuk meminimalkan fungsi *loss* dengan secara iteratif meng-update bobot-bobot dengan arah gradien negatif. Beberapa variasi *gradient descent* di antaranya.

1. *Batch gradient descent*: *Update* bobot setelah menghitung gradien pada seluruh dataset.
2. *Stochastic Gradient Descent* (SGD): *Update* bobot untuk setiap sampel secara individu.
3. *Mini-batch gradient descent*: *Update* bobot setelah menghitung gradien pada sebagian kecil sampel pelatihan.

Bab 2

Pembahasan

2.1. Implementasi

Untuk implementasi FFNN *from scratch*, penulis menggunakan berbagai *library* sebagai perhitungan dasar matematika, organisasi data, dan grafis, seperti Numpy, Scikit-Learn (untuk unduh data), dan Matplotlib.

2.1.1. Value

Berikut merupakan implementasi class Value yang dapat menyimpan data berupa angka baik integer ataupun float.

Class	
Class Value	Kelas untuk menyimpan data berupa angka. Kelas ini juga memanfaatkan computational graph untuk memudahkan perhitungan saat mencari nilai gradien (autodiff).
Attributes	
data	Atribut untuk menyimpan nilai data Value.
grad	Menyimpan nilai gradien.
_backward	Menyimpan fungsi backward setiap operasi.
_prev	Menyimpan elemen parent Value sebelumnya yang membentuk nilai Value saat ini.
_op	Keterangan operasi yang menghasilkan nilai Value saat ini.
label	Label untuk data Value (x, o, w, atau h).
Fungsi dan Prosedur	
__repr__	Method untuk menampilkan data Value dalam bentuk kelas Value.
__mul__	Fungsi ini merupakan magic method untuk operasi perkalian antara Value dengan Value.
abs	Fungsi untuk mengkonversi angka menjadi nilai mutlaknya.
__rmul__	Fungsi ini merupakan magic method untuk operasi perkalian

	antara object selain Value dan dengan object Value
<code>__pow__</code>	Fungsi ini merupakan magic method untuk operasi perpangkatan antara object Value dengan Value.
<code>__rpow__</code>	Fungsi ini merupakan magic method untuk operasi perpangkatan dengan basis object selain Value dan pangkat object Value.
<code>__add__</code>	Fungsi ini merupakan magic method untuk operasi penjumlahan antara object Value dengan object selain Value (atau Value)
<code>__radd__</code>	Fungsi ini merupakan magic method untuk operasi penjumlahan antara object selain Value dengan object Value.
<code>__neg__</code>	Fungsi ini merupakan magic method untuk mengubah nilai Value menjadi negatif.
<code>__sub__</code>	Fungsi ini merupakan magic method untuk operasi pengurangan antara object Value dengan object Value atau selain Value.
<code>__rsub__</code>	Fungsi ini merupakan magic method untuk operasi pengurangan antara object selain Value atau object Value.
<code>__truediv__</code>	Fungsi ini merupakan magic method untuk operasi pembagian antara object Value dengan object selain Value atau object Value.
<code>__rtruediv</code>	Fungsi ini merupakan magic method untuk operasi pembagian antara object selain Value dengan object Value.
<code>exp</code>	Fungsi ini merupakan method untuk operasi pangkat basis eksponensial dengan pangkat dalam object Value.
<code>log</code>	Fungsi ini merupakan method untuk operasi logaritma natural.
<code>clip</code>	Fungsi ini merupakan method untuk membatasi nilai Value dengan limit tertentu.
<code>backward</code>	Fungsi ini merupakan prosedur backpropagation untuk mendapatkan nilai gradien.
<code>linear</code>	Fungsi ini merupakan fungsi aktivasi linear beserta turunannya.

relu	Fungsi aktivasi ReLU beserta turunannya untuk backpropagation.
sigmoid	Fungsi aktivasi sigmoid beserta turunannya untuk backpropagation.
tanh	Fungsi aktivasi tanh beserta turunannya untuk backpropagation.

Berikut merupakan potongan *source code* untuk class Value. Potongan *source code* ini hanya menampilkan sebagian kecil *method* pada class Value.

```
class Value:
    def __init__(self, data, _children=(), _op="", label=""):
        self.data = data          # Data value
        self.grad = 0             # Grad initialization = 0
        self._backward = lambda: None # Local backward function
        self._prev = set(_children) # Previous Values
        self._op = _op            # Operator
        self.label = label        # Label (variabel name, e.g., x, net, o,
h)

    # Data display when printed
    def __repr__(self):
        return f"Value({self.data})"

    # Multiply operator
    def __mul__(self, other):
        # self * other
        if isinstance(other, ValueTensor):
            return other + self
        elif isinstance(other, Value):
            other = other
        else:
            other = Value(other)

        out = Value(self.data * other.data, (self, other), "*")

        # Local backpropagation (derivative of out w.r.t self and other)
        def _backward():
            other.grad += self.data * out.grad
            self.grad += other.data * out.grad

        out._backward = _backward # add _backward function to Value out

        return out

    def abs(self):
        out = Value(np.abs(self.data), (self,), "abs")

        def _backward():
            self.grad += np.sign(self.data) * out.grad
        out._backward = _backward

        return out
```

```

# Reverse multiply operator
def __rmul__(self, other):
    # other * self
    return self * other

# Power operator
def __pow__(self, other):
    # self**other
    if isinstance(other, (int, float)):
        other = other
    elif isinstance(other, Value):
        other = float(other.data)
    else:
        other = float(other)

    out = Value(self.data**other, (self,), f"**{other}")

    def _backward():
        self.grad += other * (self.data**(other - 1)) * out.grad

    out._backward = _backward

    return out

def __rpow__(self, other):
    # other**self
    if isinstance(other, (int, float)):
        other = other
    elif isinstance(other, Value):
        other = float(other.data)
    else:
        other = float(other)

    out = Value(other**self.data, (self,), f"{other}**")

    def _backward():
        self.grad += out.data * np.log(other) * out.grad

    out._backward = _backward

    return out

# Add operator
def __add__(self, other):
    # self + other
    if isinstance(other, ValueTensor):
        return other + self
    elif isinstance(other, Value):
        other = other
    else:
        other = Value(other)

    out = Value(self.data + other.data, (self, other), "+")

    def _backward():
        other.grad += out.grad
        self.grad += out.grad

    out._backward = _backward

```

```

    return out

# reverse add operator
def __radd__(self, other):
    # other + self
    return self + other

# negative operator
def __neg__(self):
    # -self
    return self * -1

# subtract operator
def __sub__(self, other):
    # self - other
    return self + (-other)

# reverse subtract operator
def __rsub__(self, other):
    # other - self
    return other + (-self)

# Division operator
def __truediv__(self, other):
    # self / other
    return self * other**(-1)

# reverse division operator
def __rtruediv__(self, other):
    # other / self
    return other * self**(-1)

def exp(self):
    out = Value(np.exp(self.data), (self,), "e**")

    def _backward():
        self.grad += out.data * out.grad
        out._backward = _backward

    return out

def log(self):
    out = Value(np.log(self.data), (self,), "log")

    def _backward():
        self.grad += 1/self.data * out.grad
        out._backward = _backward

    return out

def clip(self, min_val, max_val):
    out_data = np.clip(self.data, min_val, max_val)
    out = Value(out_data, (self,), "clip")

    def _backward():
        if min_val < self.data < max_val:
            self.grad += out.grad
        else:
            self.grad += 0

```



```

    out._backward = _backward
    return out

# Global backward
def backward(self):
    # Use topological order
    topo = []
    visited = set()
    def build_topo(val):
        if val not in visited:
            visited.add(val)
            for child in val._prev:
                build_topo(child)
            topo.append(val)

    build_topo(self)

    for val in reversed(topo):
        val.grad = 0

    # Set grad to 1 and apply the chain rule
    self.grad = 1
    for val in reversed(topo):
        val._backward()

# Activation function
# Linear
def lin(self):
    out = Value(self.data, (self,), "Linear")

    def _backward():
        self.grad += 1 * out.grad
        out._backward = _backward

    return out

# ReLU
def relu(self):
    out = Value(max(0, self.data), (self,), "ReLU")

    def _backward():
        self.grad += (0 if self.data <= 0 else 1) * out.grad
        out._backward = _backward

    return out

# Sigmoid
def sigmoid(self):
    out = Value(1/(1 + np.exp(-self.data)), (self,), "Sigmoid")

    def _backward():
        self.grad += out.data * (1 - out.data) * out.grad
        out._backward = _backward

    return out

# Hyperbolic tangent
def tanh(self):
    out = Value((np.exp(self.data) - np.exp(-self.data)) \

```

```

        / (np.exp(self.data) + np.exp(-self.data)),
        (self,), "tanh")

    def _backward():
        self.grad += (2/(np.exp(self.data) - np.exp(-self.data)))**2 *
out.grad
        out._backward = _backward

    return out

```

2.1.2. ValueTensor

Berikut merupakan implementasi class ValueTensor yang dapat menyimpan data berupa angka baik integer, float, ataupun list.

Class	
Class ValueTensor	Kelas yang menghasilkan array dalam bentuk Value versi multidimensi. Kelas ini merupakan list of Value yang merupakan termasuk class turunan dari Value.
Attributes	
shape	Atribut untuk menyimpan shape data
dim	Atribut untuk menyimpan banyak dimensi dari data
data	Atribut untuk menyimpan data ValueTensor
label	Atribut untuk menyimpan label data ValueTensor
Fungsi dan Prosedur	
grad	Fungsi untuk menampilkan nilai gradien setiap elemen ValueTensor.
T	Fungsi untuk men-transpose ValueTensor.
__repr__	Fungsi untuk menampilkan data ValueTensor dalam bentuk object ValueTensor.
__getitem__	Fungsi untuk mengambil elemen ValueTensor dengan index tertentu.
__setitem__	Fungsi untuk menetapkan elemen ValueTensor di index tertentu.
append	Fungsi untuk menambahkan nilai pada object ValueTensor

sum	Penjumlahan seluruh elemen ValueTensor.
mean	Rata-rata ValueTensor berdasarkan axis tertentu.
abs	Fungsi untuk operasi mutlak setiap elemen.
clip	Menetapkan nilai setiap elemen ValueTensor berdasarkan rentang tertentu.
__add__	Magic method untuk penjumlahan setiap elemen ValueTensor dengan object yang memiliki shape yang sama.
__radd__	Magic method untuk penjumlahan setiap elemen suatu matriks (selain ValueTensor) dengan object ValueTensor.
__mul__	Magic method untuk operasi perkalian antar elemen (element wise) ValueTensor dengan matriks lain.
__rmul__	Magic method untuk operasi perkalian antar elemen (element wise) suatu matriks dengan object ValueTensor.
__pow__	Magic method untuk operasi pangkat antar elemen (element wise) dengan basis setiap elemen ValueTensor.
__rpow__	Magic method untuk operasi pangkat antar elemen dengan basis suatu angka skalar.
exp	Fungsi perpangkatan dengan basis eksponensial antar elemen ValueTensor.
log	Fungsi logaritma natural yang diterapkan pada seluruh elemen ValueTensor.
__matmul__	Magic method untuk operasi perkalian antar matriks ValueTensor dengan matriks lain.
__rmatmul__	Magic method untuk operasi perkalian antar matriks lain dengan matriks ValueTensor.
__neg__	Magic method untuk mengubah tanda seluruh elemen ValueTensor.
__sub__	Magic method untuk operasi pengurangan antar elemen matriks ValueTensor dengan matriks lain.
__rsub__	Magic method untuk operasi pengurangan antar elemen matriks dengan matriks ValueTensor.

<code>__truediv__</code>	Magic method untuk operasi pembagian seluruh elemen matriks ValueTensor.
<code>__rtruediv__</code>	Magic method untuk operasi pembagian suatu matriks lain terhadap matriks ValueTensor.
<code>linear</code>	Fungsi aktivasi linear yang diterapkan pada seluruh elemen matriks ValueTensor.
<code>relu</code>	Fungsi aktivasi ReLU yang diterapkan pada seluruh elemen matriks ValueTensor.
<code>sigmoid</code>	Fungsi aktivasi sigmoid yang diterapkan pada seluruh elemen matriks ValueTensor.
<code>tanh</code>	Fungsi aktivasi tanh yang diterapkan pada seluruh elemen matriks ValueTensor.
<code>softmax</code>	Fungsi aktivasi softmax yang diterapkan pada setiap baris (instance/example) matriks ValueTensor.
<code>backward</code>	Fungsi untuk menerapkan backpropagation pada seluruh elemen matriks ValueTensor sehingga didapatkan nilai gradien seluruh elemennya.

Berikut merupakan potongan *source code* untuk class ValueTensor. Potongan *source code* ini hanya menampilkan sebagian kecil *method* pada class ValueTensor.

```
class ValueTensor:
    def __init__(self, data, label="(h)"):
        if isinstance(data, ValueTensor):
            self.shape = data.shape
            self.dim = data.dim
            self.data = data.data
            self.label = data.label
            return

        if isinstance(data, (list, int, float, Value)):
            data = np.array(data, dtype=object)

        self.shape = data.shape
        self.dim = len(self.shape)
        if self.dim > 0:
            ilabel = np.full(self.shape, np.arange(1, self.shape[-1]+1))
        else:
            ilabel = np.full(self.shape, np.arange(1, 1+1))
        ufunc = np.frompyfunc(lambda val, i: Value(val, label=f"{label}{i}")
        if not isinstance(val, Value) else val, 2, 1)
        self.data = ufunc(data, ilabel)
        self.label = np.array(np.vectorize(lambda x: x.label)(self.data),
        dtype=object)
```

```

@property
def grad(self):
    return np.vectorize(lambda x: x.grad)(self.data)

@property
def T(self):
    return ValueTensor(self.data.T)

def __repr__(self):
    if self.dim > 1:
        return f"ValueTensor(\n{np.vectorize(lambda x: x.data)(self.data)})"
    else:
        return f"ValueTensor({np.vectorize(lambda x: x.data)(self.data)})"

def __getitem__(self, idx):
    item = self.data[idx]
    if isinstance(item, np.ndarray):
        return ValueTensor(item)
    return item

def __setitem__(self, idx, val):
    if isinstance(val, (int, float)):
        self.data[idx] = Value(val)

    elif isinstance(val, Value):
        self.data[idx] = val

    elif isinstance(val, (list, np.ndarray)):
        val = np.array(val, dtype=object)
        item = np.vectorize(lambda x: Value(x) if not isinstance(x, Value)
else x)(val)
        self.data[idx] = item

    elif isinstance(val, ValueTensor):
        self.data[idx] = val.data

def append(self, val, axis=0, label="b"):
    if isinstance(val, (int, float)):
        val = Value(val)

    elif isinstance(val, (list, np.ndarray)):
        val = np.array(val, dtype=object)
        val = np.vectorize(lambda x: Value(x) if not isinstance(x,
Value) else x)(val)

    elif isinstance(val, ValueTensor):
        val = val.data

    else:
        raise TypeError("Unsupported type for append")

    new_data = np.append(self.data, val, axis=axis)

    return ValueTensor(new_data, label=label)

def sum(self, axis=0, **kwargs):
    result = np.sum(self.data, axis=axis, **kwargs)
    return ValueTensor(result)

```

```

def mean(self, axis=0, **kwargs):
    result = np.mean(self.data, axis=axis, **kwargs)
    return ValueTensor(result)

def abs(self):
    # abs(self)
    result = np.vectorize(lambda x: x.abs())(self.data)
    return result

def clip(self, min_val, max_val):
    result = np.vectorize(lambda x: x.clip(min_val, max_val))(self.data)
    return ValueTensor(result)

# Element wise addition
def __add__(self, other):
    # self + other
    if isinstance(other, (int, float, Value)):
        if isinstance(other, Value):
            other = other.data
        else:
            other = other
        result = np.vectorize(lambda x: x + other)(self.data)

    elif isinstance(other, ValueTensor):
        if (other.dim == 0):
            result = np.vectorize(lambda x: x + other)(self.data)
        elif self.shape != other.shape:
            raise ValueError("Shapes do not match")
        result = np.vectorize(lambda x, y: x + y)(self.data, other.data)

    elif isinstance(other, (list, np.ndarray)):
        other_tensor = ValueTensor(other)
        return self + other_tensor

    return ValueTensor(result)

# Element wise reverse addition
def __radd__(self, other):
    # other + self
    return self + other

# Element wise multiplication
def __mul__(self, other):
    # self * other
    if isinstance(other, (int, float, Value)):
        if isinstance(other, Value):
            other = other.data
        else:
            other = other
        result = np.vectorize(lambda x: x * other)(self.data)

    elif isinstance(other, ValueTensor):
        if (other.dim == 0):
            result = np.vectorize(lambda x: x * other)(self.data)
        elif self.shape != other.shape:
            raise ValueError("Shapes do not match")
        result = np.vectorize(lambda x, y: x * y)(self.data, other.data)

    elif isinstance(other, (list, np.ndarray)):
        other_tensor = ValueTensor(other)

```

```

        return self * other_tensor

    return ValueTensor(result)

# Element wise reverse multiplication
def __rmul__(self, other):
    # other * self
    return self * other

# Element wise power
def __pow__(self, other):
    # self**other
    if isinstance(other, (int, float, Value)):
        if not isinstance(other, Value):
            other = Value(other)
        else:
            other = other
        result = np.vectorize(lambda x: x ** other.data)(self.data)

    elif isinstance(other, ValueTensor):
        if self.shape != other.shape:
            raise ValueError("Shapes do not match")
        result = np.vectorize(lambda x, y: x ** y)(self.data, other.data)

    elif isinstance(other, (list, np.ndarray)):
        other_tensor = ValueTensor(other)
        return self ** other_tensor

    return ValueTensor(result)

# Element wise reverse power
def __rpow__(self, other):
    # other**self
    if isinstance(other, (int, float, Value)):
        if not isinstance(other, Value):
            other = Value(other)
        else:
            other = other
        result = np.vectorize(lambda x: other.data ** x)(self.data)

    elif isinstance(other, ValueTensor):
        if self.shape != other.shape:
            raise ValueError("Shapes do not match")
        result = np.vectorize(lambda x, y: x ** y)(other.data, self.data)

    elif isinstance(other, (list, np.ndarray)):
        other_tensor = ValueTensor(other)
        return other_tensor ** self

    return ValueTensor(result)

def exp(self):
    # e**self
    result = np.vectorize(lambda x: x.exp())(self.data)

    return ValueTensor(result)

def log(self):
    # log(self)
    result = np.vectorize(lambda x: x.log())(self.data)

```

```

    return ValueTensor(result)

def __matmul__(self, other):
    if not isinstance(other, ValueTensor):
        raise TypeError(f"Cannot multiply ValueTensor with {type(other)}")

    if self.shape[-1] != other.shape[0]:
        raise ValueError("Shapes do not match for matrix multiplication")

    result_data = np.empty((self.shape[0], other.shape[1]), dtype=object)

    for i in range(self.shape[0]):
        for j in range(other.shape[1]):
            result_data[i, j] = sum(self.data[i, k] * other.data[k, j] for
k in range(self.shape[1]))

    return ValueTensor(result_data)

def __rmatmul__(self, other):
    if not isinstance(other, ValueTensor):
        raise TypeError(f"Cannot right-multiply ValueTensor with
{type(other)}")

    if other.shape[-1] != self.shape[0]:
        raise ValueError("Shapes do not match for matrix multiplication")

    result_data = np.empty((other.shape[0], self.shape[1]), dtype=object)

    for i in range(other.shape[0]):
        for j in range(self.shape[1]):
            result_data[i, j] = sum(other.data[i, k] * self.data[k, j] for
k in range(other.shape[1]))

    return ValueTensor(result_data)

# negative operator
def __neg__(self):
    # -self
    return self * -1

# subtract operator
def __sub__(self, other):
    # self - other
    return self + (-other)

# reverse subtract operator
def __rsub__(self, other):
    # other - self
    return other + (-self)

# Division operator
def __truediv__(self, other):
    # self / other
    return self * other**(-1)

# reverse division operator
def __rtruediv__(self, other):
    # other / self
    return other * self**(-1)

```



```

def linear(self):
    result = np.vectorize(lambda x: x.lin())(self.data)
    return ValueTensor(result)

def relu(self):
    result = np.vectorize(lambda x: x.relu())(self.data)
    return ValueTensor(result)

def sigmoid(self):
    result = np.vectorize(lambda x: x.sigmoid())(self.data)
    return ValueTensor(result)

def tanh(self):
    result = np.vectorize(lambda x: x.tanh())(self.data)
    return ValueTensor(result)

def softmax(self, axis=-1):
    exp_data = self.exp()
    sum_exp = np.sum(np.vectorize(lambda x: x.data)(exp_data.data),
axis=axis, keepdims=True)

    result = np.vectorize(lambda x, s: x / Value(s))(exp_data.data,
sum_exp)
    out = ValueTensor(result)

    def _backward():
        soft_vals = np.vectorize(lambda x: x.data)(out.data)
        grad_output = np.vectorize(lambda x: x.grad)(out.data)

        for i in range(soft_vals.shape[0]):
            s = soft_vals[i].reshape(-1, 1)
            jacobian = np.diagflat(s) - (s @ s.T)

            grad_input = jacobian @ grad_output[i].reshape(-1, 1)

            for j in range(soft_vals.shape[1]):
                out.data[i, j].grad += grad_input[j, 0]

        out._backward = _backward

    return out

def backward(self):
    visited = set()

    def traverse(val):
        if val not in visited:
            visited.add(val)
            for child in val._prev:
                traverse(child)

    for val in np.ravel(self.data):
        traverse(val)

    for val in visited: # set gradien 0
        val.grad = 0

    for val in np.ravel(self.data):
        val.grad = 1

```

```

topo = []
visited_topo = set()

def build_topo(val):
    if val not in visited_topo:
        visited_topo.add(val)
        for child in val._prev:
            build_topo(child)
        topo.append(val)

for val in np.ravel(self.data):
    build_topo(val)

for val in reversed(topo):
    val._backward()

```

Selain itu, terdapat kelas lain yang menyimpan berbagai fungsi loss, seperti mse, binary cross entropy, dan categorical cross entropy beserta turunan-turunannya, yaitu class criterion.

Class	
class criterion	Kelas ini bertujuan untuk menghitung loss hasil prediksi model FFNN.
Fungsi dan Prosedur	
mse	Fungsi untuk menghitung mean squared error antara data target dengan data hasil prediksi FFNN.
binary_cross_entropy	Fungsi untuk menghitung loss binary cross entropy antara data target dengan data hasil prediksi FFNN.
categorical_cross_entropy	Fungsi untuk menghitung loss categorical cross entropy antara data target dengan data hasil prediksi FFNN.
mse_errors	Fungsi untuk menghitung turunan loss MSE terhadap hasil output yang belum dikali dengan fungsi aktivasi.
bce_errors	Fungsi untuk menghitung turunan loss binary cross entropy terhadap hasil output yang belum dikali dengan fungsi aktivasi.
cce_errors	Fungsi untuk menghitung turunan loss categorical cross entropy terhadap hasil output yang belum dikali dengan fungsi aktivasi.

Berikut merupakan potongan *source code* untuk class criterion. Potongan *source code* ini hanya menampilkan beberapa *method* yang penulis anggap penting.

```
class criterion:
    # Loss
    # MSE
    def mse(y_true, y_pred):
        y_true = ValueTensor(y_true)
        y_pred = ValueTensor(y_pred)

        mean_ = ((y_true-y_pred)**2).mean(axis=-1)
        mean_ = ValueTensor(np.expand_dims(mean_.data, axis=0))
        return mean_.mean(axis=-1)

    # BCE
    def binary_cross_entropy(y_true, y_pred):
        y_true = ValueTensor(y_true)
        y_pred = ValueTensor(y_pred.clip(1e-10, 1 - 1e-10))

        t1 = y_pred.log()
        t2 = y_true * t1
        t3 = (1 - y_true)
        t4 = (1 - y_pred).log()
        t5 = t3 * t4
        t6 = t2 + t5
        t7 = t6.mean(axis=-1)
        t7 = ValueTensor(np.expand_dims(t7.data, axis=0))
        t8 = t7.mean(axis=-1)
        return -t8

    # CCE
    def categorical_cross_entropy(y_true, y_pred):
        y_true = ValueTensor(y_true)
        y_pred = ValueTensor(y_pred.clip(1e-10, 1 - 1e-10))

        t1 = y_pred.log()
        t2 = y_true * t1
        t3 = t2.sum(axis=-1)
        t3 = ValueTensor(np.expand_dims(t3.data, axis=0))
        t4 = t3.mean(axis=-1)
        return -t4

    # derivatives
    # output hasil yang belum dikali turunan fungsi aktivasi
    def mse_errors(y_true, y_pred):
        return -2 * (y_true - y_pred) / y_pred.shape[0]

    def bce_errors(y_true, y_pred):
        return -1 * (y_pred - y_true) / (y_pred * (1 - y_pred) *
y_pred.shape[0])

    def cce_errors(y_true, y_pred):
        return -1 * y_true / (y_pred * y_pred.shape[0])
```

2.1.3. Inisialisasi Bobot

Berikut merupakan implementasi class initialization yang dapat membuat inisialisasi bobot dengan berbagai jenis, yaitu zero, random

Class	
class initialization	Kelas ini bertujuan untuk membuat matriks bobot untuk setiap layer. Metode inisialisasi bobot yang tersedia, yaitu zero, uniform, normal, Xavier, dan He.
Fungsi dan Prosedur	
zero	Fungsi untuk inisialisasi bobot dengan nilai nol.
uniform	Fungsi inisialisasi bobot dengan sifat data bobot terdistribusi secara uniform. Terdapat dua metode lain pada fungsi ini, yaitu Xavier dan He
normal	Fungsi inisialisasi bobot dengan sifat data bobot terdistribusi secara normal. Terdapat dua metode lain pada fungsi ini, yaitu Xavier dan He

Berikut merupakan potongan *source code* untuk class initialization. Potongan *source code* ini hanya menampilkan beberapa *method* yang penulis anggap penting.

```
class initialization:
    # kelas untuk inisialisasi bobot tiap neuron layer
    # beberapa cara inisialisasi: zero, uniform/normal distribution,
    # xavier/he(bonus)
    # size: tuple (jumlah neuron input, jumlah neuron output)
    def zero(size):
        return np.zeros(size)

    def uniform(size, lower_bound=-1, upper_bound=1, seed=None,
method="random"):
        if seed is not None:
            np.random.seed(seed)

        low, high = lower_bound, upper_bound
        if (method == "xavier"):
            x = np.sqrt(6 / (size[0] + size[1]))
            low, high = -x, x
        elif (method == "he"):
            x = np.sqrt(6 / size[0])
            low, high = -x, x

        return np.random.uniform(low=low, high=high, size=size)

    def normal(size, mean=0, std=1, seed=None, method="random"):
        if seed is not None:
            np.random.seed(seed)
```

```

loc, scale = mean, std
if (method == "xavier"):
    loc, scale = 0, np.sqrt(2 / (size[0] + size[1]))
elif (method == "he"):
    loc, scale = 0, np.sqrt(2 / size[0])

return np.random.normal(loc=loc, scale=scale, size=size)

```

2.1.4. Layer

Berikut merupakan implementasi class Layer yang berfungsi sebagai perhitungan neuron dalam satu layer.

Class	
class Layer	Kelas ini bertujuan untuk menghitung output setiap layer. Kelas ini mulai menyimpan nilai bobot setiap neuron dalam satu layer. Selain itu, kelas ini juga terdapat proses perhitungan network serta perhitungan update bobot.
Attributes	
input_size	Atribut untuk menyimpan jumlah neuron dalam satu layer.
output_size	Menyimpan jumlah neuron untuk layer selanjutnya
activation_function	Menyimpan string fungsi aktivasi untuk digunakan saat forward propagation
weights	Menyimpan bobot (termasuk bobot bias) antara layer ini dengan selanjutnya.
neuron_values	Menyimpan nilai neuron dan bias yang sudah diberikan fungsi aktivasi dalam satu layer.
next_raw	Menyimpan data nilai layer selanjutnya yang belum diberi fungsi aktivasi.
next_activated	Untuk menyimpan nilai layer selanjutnya yang sudah diberi fungsi aktivasi.
next_error	Menyimpan gradien untuk backpropagation dan update weight.
Fungsi dan Prosedur	
forward	Fungsi untuk forward propagation, yaitu menghitung nilai

	neuron dalam satu layer hingga perhitungan dengan fungsi aktivasi. Fungsi ini menerima argumen inputs, yaitu nilai neuron (neuron_values) yang sudah dihitung oleh layer sebelumnya.
backward_and_update_weights	Fungsi untuk backpropagation sekaligus update bobot dengan gradient descent. Fungsi ini menerima argumen next_gradien yang merupakan nilai eror di layer selanjutnya yang sudah dikalikan dengan bobot pada layer selanjutnya, learning_rate yang berperan sebagai step/langkah pembelajaran dalam pencarian bobot optimal, dan is_last yang merupakan kondisi layer terakhir.

Berikut merupakan potongan *source code* untuk class Layer. Potongan *source code* ini hanya menampilkan beberapa *method* yang penulis anggap penting.

```
class Layer:
    def __init__(self, input_size, output_size, activation_function="linear",
weight_init="normal", weight_low_or_mean=None, weight_high_or_std=1,
weight_seed=None, weight_type="random"):
        self.input_size = input_size # jumlah neuron di dalam layer ini
        self.output_size = output_size # jumlah neuron di layer selanjutnya.
        untuk weight
        self.activation_function = activation_function # string??? idk.
        activation function yg digunakan

        # weight_init (string): zero, uniform, atau normal
        # weight_low_or_mean: untuk lower bound kalau pakai uniform atau mean
        kalau pakai normal
        # weight_high_or_std: untuk upper bound kalau pakai uniform atau std
        kalau pakai normal
        # weight_seed: seed untuk inisialisasi weight. untuk reproducibility
        # weight_type (string): random, xavier (bonus), atau he (bonus)
        # inisialisasi weight semua neuron dan bias. size = (input_size + 1,
        output_size)
        weights_array = None
        if (weight_init == "uniform"):
            if (weight_low_or_mean == None): weights_array =
initialization.uniform((input_size + 1, output_size), -1,
weight_high_or_std, weight_seed, weight_type)
            else: weights_array = initialization.uniform((input_size + 1,
output_size), weight_low_or_mean, weight_high_or_std, weight_seed,
weight_type)
        elif (weight_init == "normal"):
            if (weight_low_or_mean == None): weights_array =
initialization.normal((input_size + 1, output_size), 0,
weight_high_or_std, weight_seed, weight_type)
            else: weights_array = initialization.normal((input_size + 1,
output_size), weight_low_or_mean, weight_high_or_std, weight_seed,
weight_type)
        else: # weight_init == "zero"
            weights_array = initialization.zero((input_size + 1, output_size))
        self.weights = ValueTensor(weights_array)
```

```

        self.neuron_values = None # berisikan semua nilai neuron dan bias
        dalam satu layer, dalam satu batch??? yang neuronnya sudah dikasih fungsi
        aktivasi
        self.next_raw = None # untuk simpan data nilai layer selanjutnya yang
        belum diberi fungsi aktivasi
        self.next_activated = None # untuk simpan data nilai layer selanjutnya
        yang sudah diberi fungsi aktivasi
        self.next_error = None # untuk simpan gradien untuk backpropagation
        dan update weight
        self.weight_gradients = None # untuk simpan gradien bobot untuk update
        weight dan plotting???

    def forward(self, inputs): # untuk forward propagation
        if not isinstance(inputs, ValueTensor): inputs = ValueTensor(inputs)

        self.neuron_values = ValueTensor(np.hstack((inputs.data,
        np.ones((inputs.shape[0], 1)))) # sekalian isiin bias
        self.next_raw = self.neuron_values @ self.weights

        # activation function
        if (self.activation_function == "relu"): self.next_activated =
self.next_raw.relu()
        elif (self.activation_function == "sigmoid"): self.next_activated =
self.next_raw.sigmoid()
        elif (self.activation_function == "tanh"): self.next_activated =
self.next_raw.tanh()
        elif (self.activation_function == "softmax"): self.next_activated =
self.next_raw.softmax()
        else: self.next_activated = self.next_raw.linear() # activation
        function == "linear"

        return self.next_activated

    def backward_and_update_weights(self, next_gradients, learning_rate,
    is_last): # untuk back propagation dan sekaligus update weights
        # next_gradients itu error layer selanjutnya lagi yg sudah dikaliin
        dengan weights layer selanjutnya
        # is_last: true kalau bukan yang terakhir
        if not isinstance(next_gradients, ValueTensor): next_gradients =
ValueTensor(next_gradients)

        self.next_activated.backward() # untuk self.next_raw.grad

        self.next_error = ValueTensor(self.next_raw.grad) * next_gradients

        if not is_last:
            weight_T_no_bias = ValueTensor(np.array([row[:-1] for row in
self.weights.data.T], dtype=object))

            self.weight_gradients = ValueTensor(self.neuron_values.data.T) @
self.next_error

            # update weights
            self.weights -= learning_rate * self.weight_gradients

        if not is_last: return (self.next_error @ weight_T_no_bias)
        else: return

```

2.1.5. Output Layer

Berikut merupakan implementasi class OutputLayer yang berfungsi sebagai perhitungan untuk output layer.

Class	
class OutputLayer	Kelas ini merupakan perhitungan khusus untuk output layer. Kelas ini berfungsi untuk memprediksi nilai output, menghitung loss, dan menghitung turunan loss. Kelas ini menerima argumen output_size dan loss_function.
Attributes	
predicted	Menyimpan ValueTensor matriks hasil prediksi
target	Menyimpan data target output.
loss_function	Menyimpan jenis fungsi loss dalam tipe string.
loss	Atribut untuk menyimpan nilai loss
loss_derivatives	Atribut untuk menyimpan hasil turunan loss yang belum dikali turunan nilai output layer.
Fungsi dan Prosedur	
setPredictions	Prosedur untuk menerima data hasil prediksi dan data target. Fungsi ini menerima argumen predicted (data hasil prediksi) dan target (data target).
calculateLoss	Prosedur untuk menyimpan nilai loss berdasarkan data prediksi dan target yang sudah diterima pada fungsi setPredictions.
lossDerivatives	Prosedur untuk menyimpan nilai turunan loss berdasarkan data hasil prediksi dan target yang sudah diterima pada fungsi setPredictions.

Berikut merupakan potongan *source code* untuk class OutputLayer. Potongan *source code* ini hanya menampilkan beberapa *method* yang penulis anggap penting.

```
class OutputLayer:
    def __init__(self, output_size, loss_function="mse"):
        self.predicted = None # ValueTensor matriks (batch_size, output_size)
        prediksi
        self.target = None # target output
        self.loss_function = loss_function # mse, bce, cce
```



```

self.loss = None # nilai loss
self.loss_derivatives = None # nilai hasil turunan loss yang belum
dikali turunan nilai output layer dan sudah dibagi batch size. matriks

def setPredictions(self, predicted, target):
    if not isinstance(predicted, ValueTensor): self.predicted =
ValueTensor(predicted)
    else: self.predicted = predicted
    if not isinstance(target, ValueTensor): self.target =
ValueTensor(target)
    else: self.target = target

def calculateLoss(self, weights=None, regularization=None,
lambda_=None):
    # asumsi sudah ada self.predicted dan self.target
    if regularization == None:
        if (self.loss_function == "bce"): self.loss =
criterion.binary_cross_entropy(self.target, self.predicted)
        elif (self.loss_function == "cce"): self.loss =
criterion.categorical_cross_entropy(self.target, self.predicted)
        else: self.loss = criterion.mse(self.target, self.predicted) #
self.loss_function == "mse"
        elif regularization == "L1":
            reg = ValueTensor([weights[i].abs().sum().data for i in
range(len(weights))]).sum()
            if (self.loss_function == "bce"): self.loss =
criterion.binary_cross_entropy(self.target, self.predicted) + (lambda_ *
reg)
            elif (self.loss_function == "cce"): self.loss =
criterion.categorical_cross_entropy(self.target, self.predicted) +
(lambda_ * reg)
            else: self.loss = criterion.mse(self.target, self.predicted) +
(lambda_ * reg) # self.loss_function == "mse"
        elif regularization == "L2":
            reg = ValueTensor([(weights[i]**2).sum().sum().data for i in
range(len(weights))]).sum()
            if (self.loss_function == "bce"): self.loss =
criterion.binary_cross_entropy(self.target, self.predicted) + (lambda_ *
reg)
            elif (self.loss_function == "cce"): self.loss =
criterion.categorical_cross_entropy(self.target, self.predicted) +
(lambda_ * reg)
            else: self.loss = criterion.mse(self.target, self.predicted) +
(lambda_ * reg) # self.loss_function == "mse"

def lossDerivatives(self):
    # asumsi sudah ada self.predicted dan self.targets
    if (self.loss_function == "bce"): self.loss_derivatives =
criterion.bce_errors(self.target, self.predicted)
    elif (self.loss_function == "cce"): self.loss_derivatives =
criterion.cce_errors(self.target, self.predicted)
    else: self.loss_derivatives = criterion.mse_errors(self.target,
self.predicted) # self.loss_function == "mse"

```

2.1.6. FFNN

Berikut merupakan implementasi class FFNN yang berfungsi sebagai untuk implementasi *Feedforward Neural Network*.

Class	
class FFNN	Kelas untuk implementasi model FFNN. menerima input input_size, hidden_size_array, output_size, activation_function, loss_function, dan weight_init
Attributes	
input_size	Jumlah features data pada data input sebagai layer input.
hidden_size_array	Kumpulan jumlah neuron tiap hidden layer dalam tipe array
output_size	Ukuran output pada output layer.
num_neurons	Kumpulan jumlah neuron seluruh layer (input, hidden, dan output) dalam tipe array.
activation_function	Kumpulan fungsi aktivasi seluruh hidden layer dan output layer dalam tipe array.
loss_function	Nama fungsi loss hanya untuk output layer dalam bentuk string.
weight_init	Konfigurasi dalam bentuk tuple untuk inisialisasi bobot.
input_and_hidden_layers	Hidden layer pertama sebagai asumsi model mempunyai minimal satu hidden layer.
output_layer	Object kelas output layer model FFNN.
Fungsi dan Prosedur	
forward_propagation	Fungsi untuk forward propagation dari input menuju output yang dilakukan setiap satu batch. Fungsi ini mengembalikan nilai loss hasil forward propagation.
back_propagation	Prosedur untuk melakukan back propagation dari loss menuju parameter setiap layer yang dilakukan setiap satu batch. Fungsi ini melakukan backward dan update bobot di setiap layer.
train_model	Prosedur ini bertujuan untuk melatih model FFNN dengan epoch tertentu dan jumlah batch data tertentu. Fungsi ini

	mengembalikan nilai loss hasil training dan validasi dalam bentuk array.
weight_distribution	Prosedur untuk menampilkan distribusi data bobot pada setiap layer.
gradient_distribution	Prosedur untuk menampilkan distribusi data gradien bobot pada setiap layer.
save_model	Prosedur untuk menyimpan model FFNN.
load_model	Prosedur untuk load model FFNN.

Berikut merupakan potongan *source code* untuk class FFNN. Potongan *source code* ini hanya menampilkan beberapa *method* yang penulis anggap penting.

```
class FFNN:
    def __init__(self, input_size, hidden_size_array, output_size,
activation_function, loss_function, weight_init, regularization=None,
lambda_=0.01):
        self.input_size = input_size # 784
        self.hidden_size_array = np.array(hidden_size_array).astype(int) #
array jumlah neuron tiap hidden layer
        self.output_size = output_size # 10
        self.num_neurons = np.insert(hidden_size_array, 0, input_size)
        self.num_neurons = np.append(self.num_neurons,
output_size).astype(int) # array jumlah neuron termasuk input dan output
layer
        self.activation_function = activation_function # array fungsi aktivasi
setiap layer (termasuk output)
        self.loss_function = loss_function # hanya untuk output layer. MSE,
Binary cross entropy, atau Categorical cross entropy
        self.weight_init = weight_init # array tuple (weight_init,
weight_low_or_mean, weight_high_or_std, weight_seed, weight_type)
inisialisasi bobot tiap layer (termasuk input)
        self.regularization = regularization # L1, L2, atau None
        self.lambda_ = lambda_ # lambda untuk regularization

        # asumsi model punya minimal satu hidden layer
        self.input_and_hidden_layers = [Layer(self.num_neurons[i],
self.num_neurons[i+1], activation_function[i], weight_init[i][0],
weight_init[i][1], weight_init[i][2], weight_init[i][3],
weight_init[i][4]) for i in range(len(hidden_size_array) + 1)]
        self.output_layer = OutputLayer(output_size, loss_function)

    def forward_propagation(self, data, target):
        # forward propagation satu kali dalam satu batch
        values = data
        weights = [self.input_and_hidden_layers[i].weights for i in
range(len(self.num_neurons) - 1)]
        for i in range(len(self.input_and_hidden_layers)):
            values = self.input_and_hidden_layers[i].forward(values)
        self.output_layer.setPredictions(values, target)
        if (self.regularization is not None):
```

```

        self.output_layer.calculateLoss(weights, self.regularization,
self.lambda_)
    else:
        self.output_layer.calculateLoss()
    return self.output_layer.loss

def back_propagation(self, learning_rate):
    # backward propagation satu kali dalam satu batch
    # asumsi sudah melakukan forward_propagation sebelum ini
    self.output_layer.lossDerivatives()
    values = self.output_layer.loss_derivatives
    for i in range (len(self.input_and_hidden_layers) - 1, 0, -1):
        values =
self.input_and_hidden_layers[i].backward_and_update_weights(values,
learning_rate, False)
        self.input_and_hidden_layers[0].backward_and_update_weights(values,
learning_rate, True)
    return

def train_model(self, batch_size, learning_rate, num_epochs, x_train,
y_train, x_val, y_val, verbose=0):
    # verbose == 0: tidak menampilkan apa-apa
    # verbose == 1: menampilkan progress bar, kondisi training_loss dan
validation_loss
    X_batches_train = np.array_split(x_train, np.ceil(len(x_train) /
batch_size))
    Y_batches_train = np.array_split(y_train, np.ceil(len(y_train) /
batch_size))
    X_batches_val = np.array_split(x_val, np.ceil(len(x_val) /
batch_size))
    Y_batches_val = np.array_split(y_val, np.ceil(len(y_val) /
batch_size))
    num_of_batches_train = len(X_batches_train)
    num_of_batches_val = len(X_batches_val)
    training_loss_array = []
    val_loss_array = []
    batches_loss_array = np.array([])
    for i in range (num_epochs):
        progress = range(num_of_batches_train + num_of_batches_val)
        if (verbose == 1): # show progress bar
            progress = tqdm(progress, desc=f"Epoch {i+1}/{num_epochs}",
unit="batch")

        batches_loss_array = np.array([])
        for j in range (num_of_batches_train):
            batches_loss_array = np.append(batches_loss_array,
self.forward_propagation(X_batches_train[j], Y_batches_train[j])[0].data)
            if (verbose == 1):
                progress.set_postfix({"Batch Loss": batches_loss_array[j]})
                progress.update(1)
            self.back_propagation(learning_rate)
            training_loss_array.append(batches_loss_array.mean())
            batches_loss_array = np.array([])
            for j in range (num_of_batches_val):
                batches_loss_array = np.append(batches_loss_array,
self.forward_propagation(X_batches_val[j], Y_batches_val[j])[0].data)
                if (verbose == 1): progress.update(1)
                val_loss_array.append(batches_loss_array.mean())

        if (verbose == 1):

```

```

        (print(f"Epoch {i+1}: Train Loss = {training_loss_array[i]}, Val
Loss = {val_loss_array[i]}"))
    return training_loss_array, val_loss_array

def predict(self, x_val):
    predictions = np.empty((0, self.output_size))
    for i in range(len(x_val)):
        values = [x_val[i]]
        for j in range(len(self.input_and_hidden_layers)):
            values = self.input_and_hidden_layers[j].forward(values)
        predictions = np.vstack((predictions, np.vectorize(lambda x:
x.data)(values.data)))
    predictions = np.argmax(predictions, axis=1)
    return predictions

def weight_distribution(self, layers_list):
    # layers_list itu list of integer layer mana saja yang weightnya di
    plot (mulai dari 0 itu input layer)
    for i in range(len(layers_list)):
        weight_flat = np.vectorize(lambda x:
x.data)(self.input_and_hidden_layers[layers_list[i]].weights.data).flatte
n()
        plt.figure(figsize=(8, 5))
        sns.histplot(weight_flat, bins=50, kde=True, color="blue")
        plt.title("Weight Distribution for layer " + str(layers_list[i]))
        plt.xlabel("Weight Value")
        plt.ylabel("Frequency")
        plt.grid(True)
        plt.show()

def gradient_distribution(self, layers_list):
    for i in range(len(layers_list)):
        weight_gradient_flat = np.vectorize(lambda x:
x.data)(self.input_and_hidden_layers[layers_list[i]].weight_gradients.dat
a).flatten()
        plt.figure(figsize=(8, 5))
        sns.histplot(weight_gradient_flat, bins=50, kde=True, color="blue")
        plt.title("Weight Gradient Distribution for layer " +
str(layers_list[i]))
        plt.xlabel("Weight Gradient Value")
        plt.ylabel("Frequency")
        plt.grid(True)
        plt.show()

def save_model(self, filename):
    with open(filename, "wb") as f:
        dill.dump(self, f)
    print(f"Model saved to {filename}")

@staticmethod
def load_model(filename):
    with open(filename, "rb") as f:
        model = dill.load(f)
    print(f"Model loaded from {filename}")
    return model

def draw_graph(self):
    visualizer = GraphNN(self)
    visualizer.draw_graph()

```

```

class GraphNN:
    def __init__(self, model):
        self.layer = model.num_neurons
        self.weights = [model.input_and_hidden_layers[i].weights for i in
range(len(self.layer)-1)]
        self.graph = nx.DiGraph()
        self.node_pos = {}
        self.node_lab = {}

        self.build_graph()

    def build_graph(self):
        x_offset = 0
        max_neurons = max(self.layer)

        id_node = 0
        prev_layer = []

        for layer_idx, num_neuron in enumerate(self.layer):
            y_offset = (max_neurons - num_neuron) / 2
            current_layer = []

            for i in range(num_neuron):
                if layer_idx > 0 and layer_idx < len(self.layer) - 1:
                    label = "h"
                elif layer_idx == len(self.layer) - 1:
                    label = "o"
                else:
                    label = "i"
                neuron_label = f"{label}{i+1}"
                self.graph.add_node(id_node, label=neuron_label)
                self.node_pos[id_node] = (x_offset, -i -y_offset)
                self.node_lab[id_node] = neuron_label
                current_layer.append(id_node)
                id_node += 1

            if prev_layer:
                weight = self.weights[layer_idx - 1].data
                for j, prev_node in enumerate(prev_layer):
                    for k, curr_node in enumerate(current_layer):
                        weight_val = weight[j, k].data
                        grad_val = weight[j, k].grad
                        self.graph.add_edge(prev_node, curr_node,
weight=round(weight_val, 2), grad=round(grad_val, 2))

                prev_layer = current_layer
                x_offset += 2

    def draw_graph(self):
        plt.figure(figsize=(10,6))
        edges = self.graph.edges(data=True)

        nx.draw(self.graph, pos=self.node_pos, with_labels=True,
labels=self.node_lab, node_size=800, node_color="lightgreen",
font_size=10, edge_color="gray")

        edge_labels = {(u, v): f"w={d['weight']}, g={d['grad']}" for u,v,d in
edges}
        nx.draw_networkx_edge_labels(self.graph, pos=self.node_pos,
edge_labels=edge_labels, font_size=8)

```

```
plt.title("Struktur Jaringan dan Nilai Bobot")  
plt.show()
```

2.2. Forward Propagation

Forward propagation pada implementasi FFNN dilakukan satu kali dalam satu batch. Setiap method forward dipanggil dari class Layer, maka akan dilakukan perhitungan network dan beserta hasilnya setelah menggunakan fungsi aktivasi dalam satu layer. Selain itu, dalam class FFNN, forward propagation dihitung untuk seluruh layer sesuai dengan jumlah layer yang didefinisikan oleh user. Pada class dan method yang sama, nilai loss juga akan dihitung dalam proses forward propagation sehingga method ini menghasilkan nilai loss dari forward propagation yang sudah dilakukan pada seluruh layer.

2.3. Backward Propagation dan Weight Update

Backward propagation pada implementasi FFNN menggunakan autodiff yang dapat menghasilkan nilai gradien pada setiap operasi. Setiap pemanggilan backward pada implementasi ini akan langsung otomatis menghasilkan gradien untuk setiap parameter. Hasil gradien ini kemudian akan digunakan untuk memperbarui bobot pada setiap layer dengan menggunakan Gradient Descent. Jenis Gradient Descent disesuaikan dengan jumlah batch yang sudah ditetapkan.

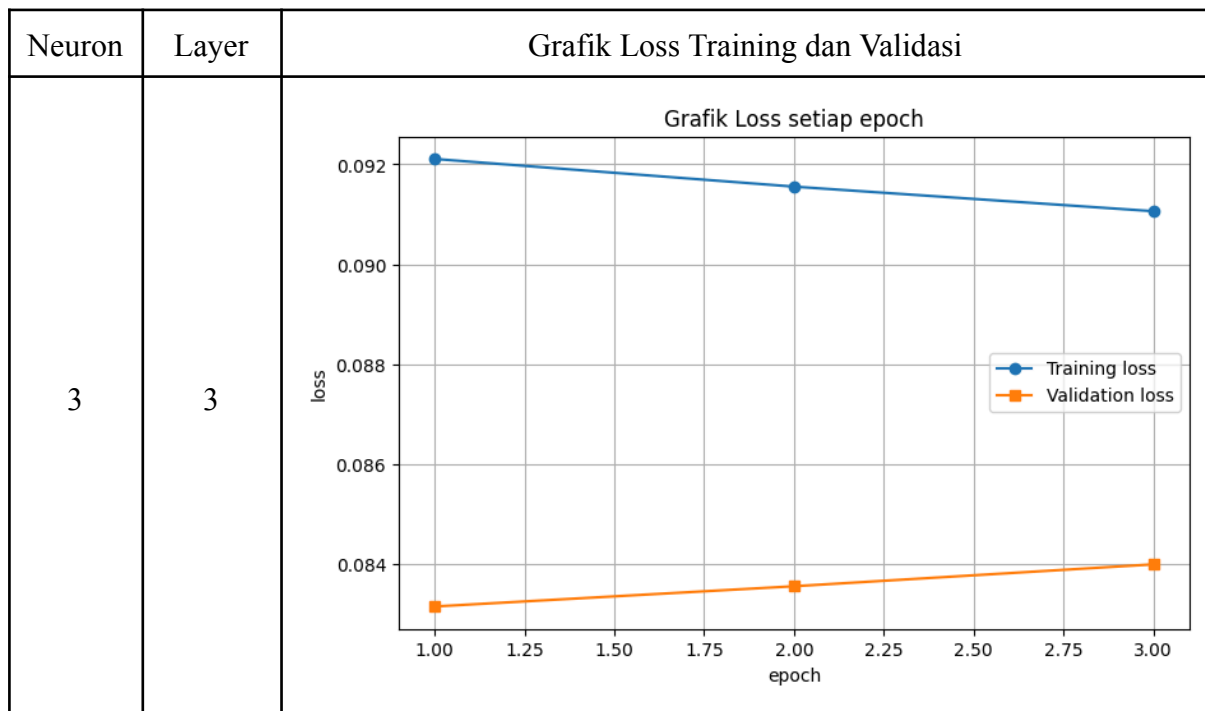
Bab 3

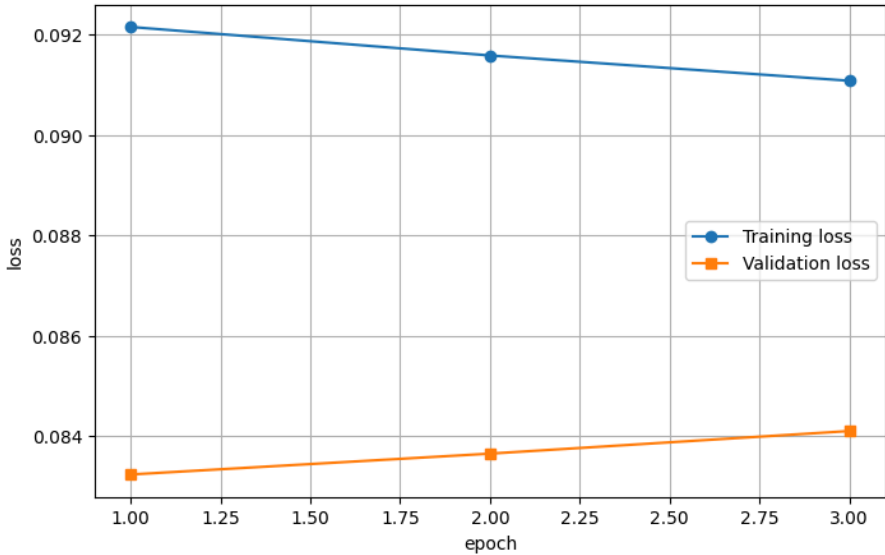
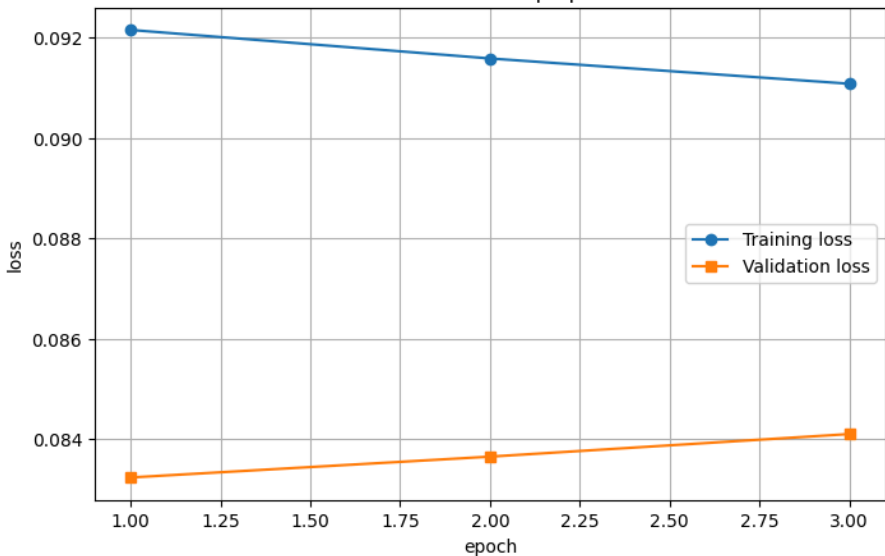
Hasil dan Analisis

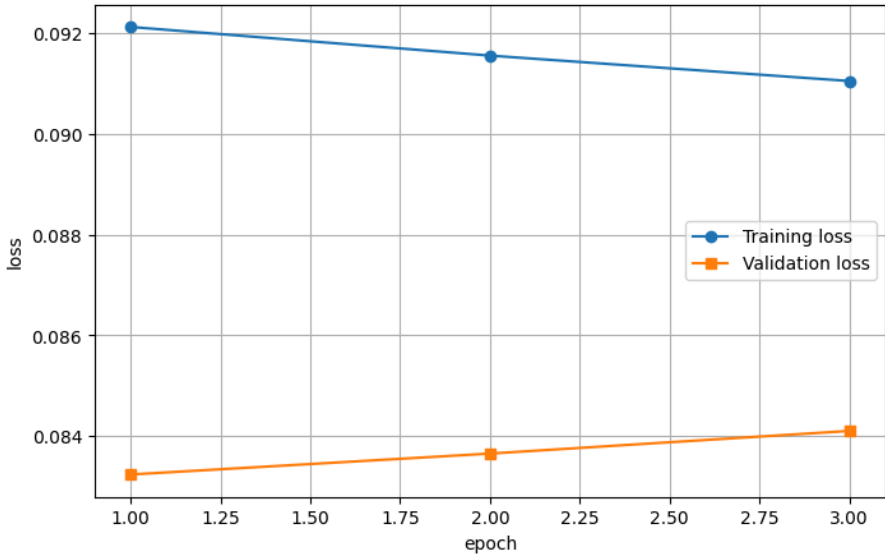
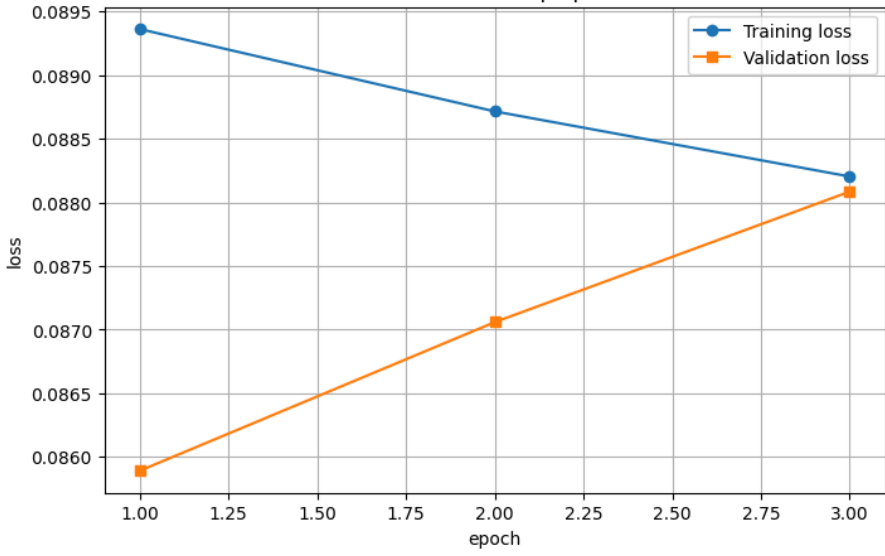
Pengujian dilakukan pada sebagian kecil dataset karena model yang diimplementasikan cukup memakan waktu pada saat *training* dan validasi.

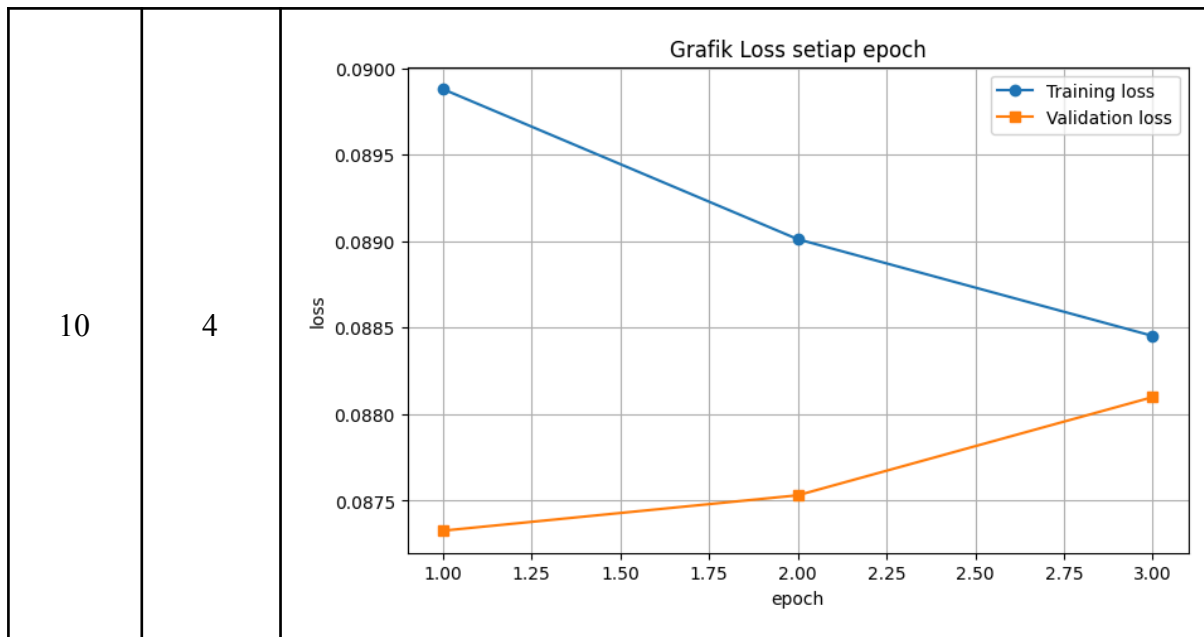
3.1. Pengaruh Jumlah Neuron dan Banyak *Layer*

Jumlah neuron dan layer yang digunakan untuk pengujian beserta hasilnya adalah sebagai berikut dengan fungsi aktivasi setiap hidden layer adalah sigmoid dan output layer adalah softmax, fungsi loss MSE, ukuran input 784, ukuran output 10, dan inisialisasi bobot uniform random dari -1 sampai 1 (dengan random state 42). Selain itu, ukuran batch yang digunakan adalah 2, learning rate 0.1, dan jumlah epoch 3.



3	5	<p>Grafik Loss setiap epoch</p>  <table border="1"> <thead> <tr> <th>epoch</th> <th>Training loss</th> <th>Validation loss</th> </tr> </thead> <tbody> <tr> <td>1.00</td> <td>0.0922</td> <td>0.0832</td> </tr> <tr> <td>2.00</td> <td>0.0916</td> <td>0.0836</td> </tr> <tr> <td>3.00</td> <td>0.0911</td> <td>0.0841</td> </tr> </tbody> </table>	epoch	Training loss	Validation loss	1.00	0.0922	0.0832	2.00	0.0916	0.0836	3.00	0.0911	0.0841
epoch	Training loss	Validation loss												
1.00	0.0922	0.0832												
2.00	0.0916	0.0836												
3.00	0.0911	0.0841												
3	7	<p>Grafik Loss setiap epoch</p>  <table border="1"> <thead> <tr> <th>epoch</th> <th>Training loss</th> <th>Validation loss</th> </tr> </thead> <tbody> <tr> <td>1.00</td> <td>0.0922</td> <td>0.0832</td> </tr> <tr> <td>2.00</td> <td>0.0916</td> <td>0.0836</td> </tr> <tr> <td>3.00</td> <td>0.0911</td> <td>0.0841</td> </tr> </tbody> </table>	epoch	Training loss	Validation loss	1.00	0.0922	0.0832	2.00	0.0916	0.0836	3.00	0.0911	0.0841
epoch	Training loss	Validation loss												
1.00	0.0922	0.0832												
2.00	0.0916	0.0836												
3.00	0.0911	0.0841												

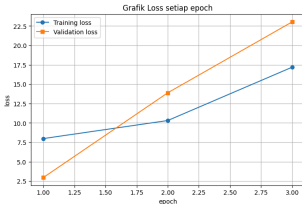
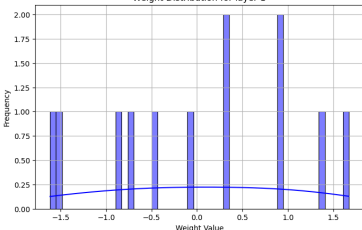
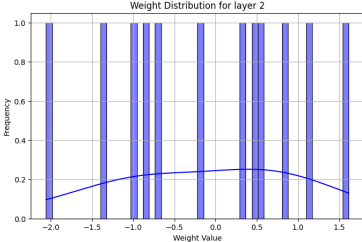
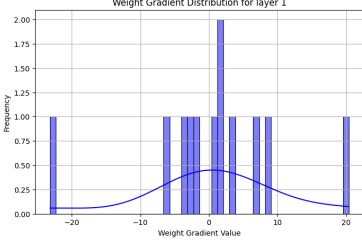
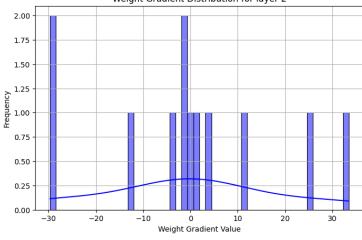
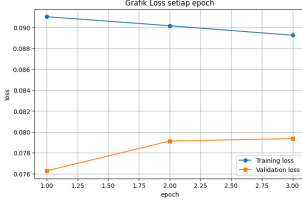
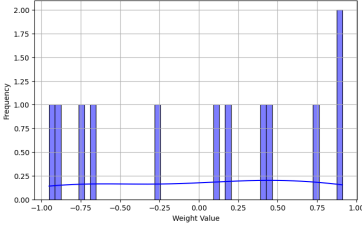
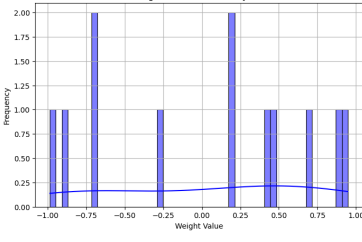
3	4	<p style="text-align: center;">Grafik Loss setiap epoch</p>  <table border="1"> <thead> <tr> <th>epoch</th> <th>Training loss</th> <th>Validation loss</th> </tr> </thead> <tbody> <tr> <td>1.00</td> <td>0.092</td> <td>0.083</td> </tr> <tr> <td>2.00</td> <td>0.0915</td> <td>0.0835</td> </tr> <tr> <td>3.00</td> <td>0.091</td> <td>0.084</td> </tr> </tbody> </table>	epoch	Training loss	Validation loss	1.00	0.092	0.083	2.00	0.0915	0.0835	3.00	0.091	0.084
epoch	Training loss	Validation loss												
1.00	0.092	0.083												
2.00	0.0915	0.0835												
3.00	0.091	0.084												
5	4	<p style="text-align: center;">Grafik Loss setiap epoch</p>  <table border="1"> <thead> <tr> <th>epoch</th> <th>Training loss</th> <th>Validation loss</th> </tr> </thead> <tbody> <tr> <td>1.00</td> <td>0.0894</td> <td>0.0859</td> </tr> <tr> <td>2.00</td> <td>0.0887</td> <td>0.0871</td> </tr> <tr> <td>3.00</td> <td>0.0882</td> <td>0.0881</td> </tr> </tbody> </table>	epoch	Training loss	Validation loss	1.00	0.0894	0.0859	2.00	0.0887	0.0871	3.00	0.0882	0.0881
epoch	Training loss	Validation loss												
1.00	0.0894	0.0859												
2.00	0.0887	0.0871												
3.00	0.0882	0.0881												

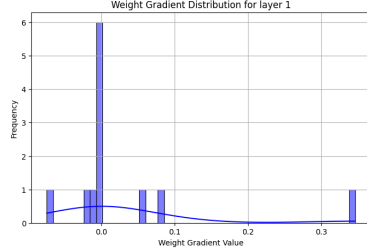
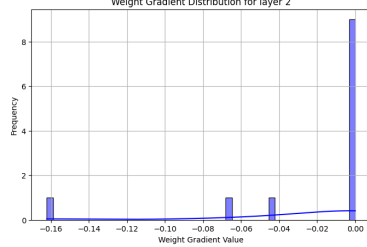
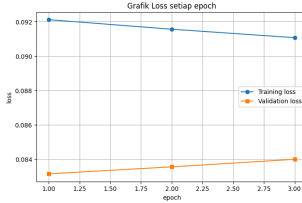
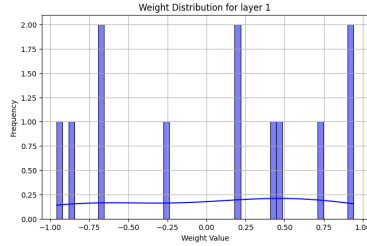
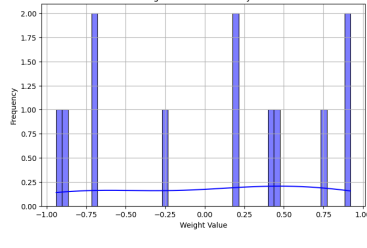
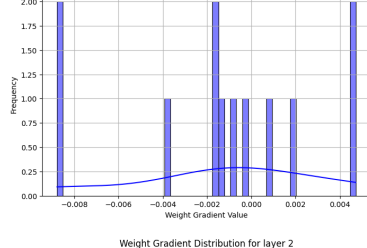
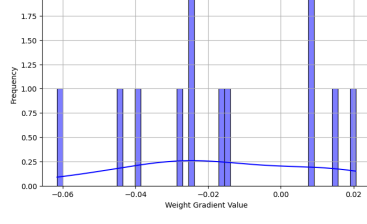


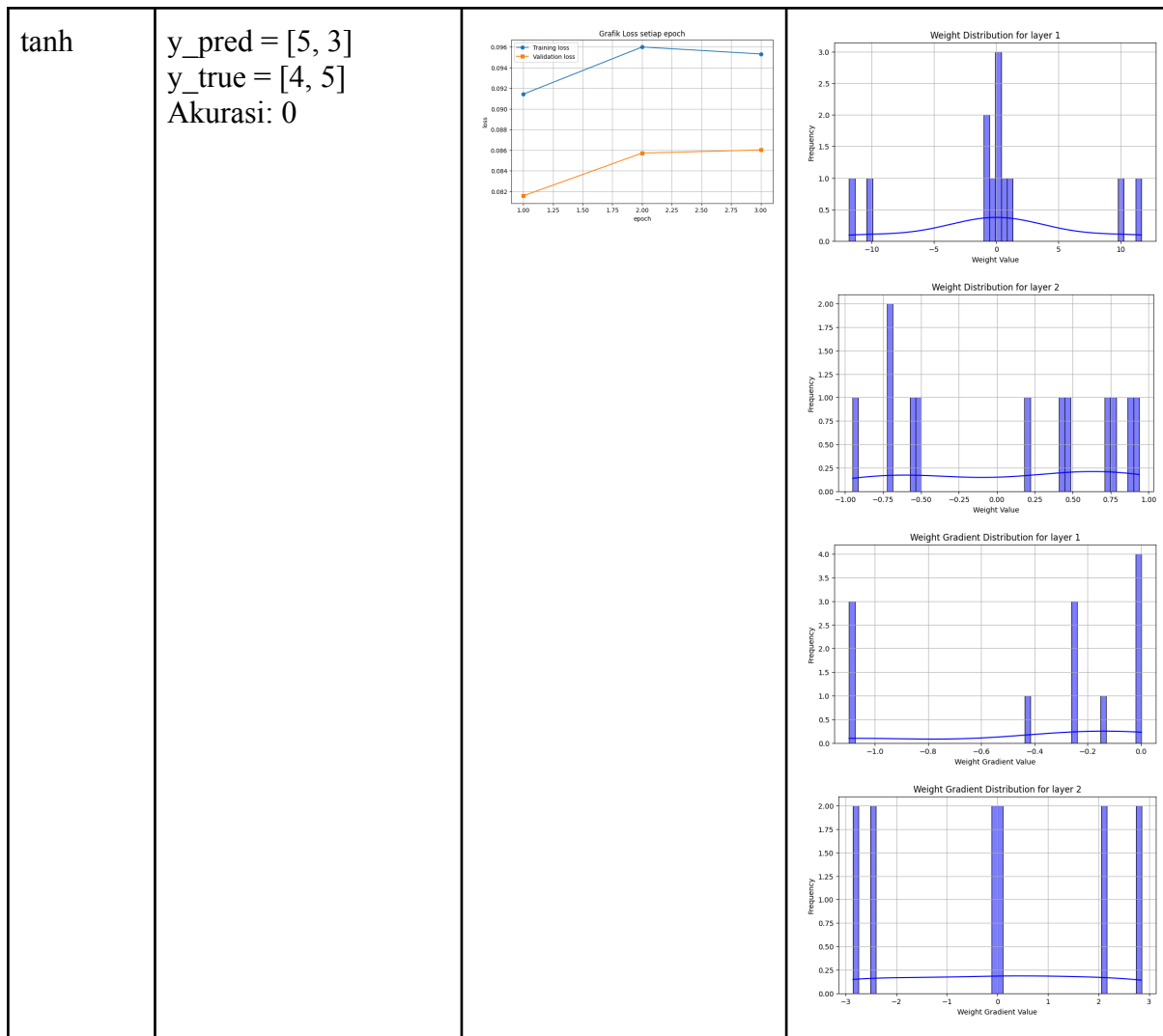
Berdasarkan hasil pengujian yang sudah dilakukan, terlihat bahwa pengaruh jumlah layer tidak terlalu berpengaruh secara signifikan yang ditandai dengan nilai loss yang cenderung sama terhadap model lain dengan jumlah layer yang berbeda. Seiring bertambahnya jumlah layer, nilai loss sangat sedikit menurun pada loss training dan validasi. Hal ini menandakan bahwa banyaknya layer pada implementasi model ini tidak terlalu berpengaruh terhadap hasil prediksi model. Selain itu, jumlah neuron dalam satu layer terlihat secara sekilas bahwa pengaruh ini cukup signifikan untuk setiap ukuran neuron yang berbeda. Seiring bertambahnya jumlah neuron, nilai loss training dan validasi meningkat di setiap epoch. Hal ini dapat terjadi karena semakin banyak hidden neuron yang digunakan, menyebabkan optimisasi seperti SGD, sulit menemukan solusi yang optimal, sehingga dapat menyebabkan gradien terlalu kecil atau terlalu besar, membuat training tidak stabil. Selain itu, pengaruh jumlah layer tidak memberikan dampak yang terlalu signifikan pada loss model, hal ini disebabkan oleh kompleksitas data yang relatif rendah, sehingga model dengan arsitektur sederhana mampu mempelajari pola data yang ada secara cukup efektif.

3.2. Pengaruh Fungsi Aktivasi

Fungsi aktivasi yang digunakan untuk pengujian beserta hasilnya adalah sebagai berikut dengan jumlah layer tetap 3 layer, jumlah hidden size tetap 3 neuron, fungsi loss MSE, ukuran input 784, ukuran output 10, dan inisialisasi bobot uniform random dari -1 sampai 1 (dengan random state 42). Selain itu, ukuran batch yang digunakan adalah 2, learning rate 0.1 (0.01 untuk linear dan tanh), dan jumlah epoch 3.

Fungsi Aktivasi	Hasil prediksi	Grafik Loss Training dan Validasi	Distribusi Bobot dan Gradiennya
linear	$y_{pred} = [3, 3]$ $y_{true} = [4, 5]$ Akurasi: 0		   
ReLU	$y_{pred} = [3, 3]$ $y_{true} = [4, 5]$ Akurasi: 0		 

			 
sigmoid	$y_{pred} = [3, 3]$ $y_{true} = [4, 5]$ Akurasi: 0	    	

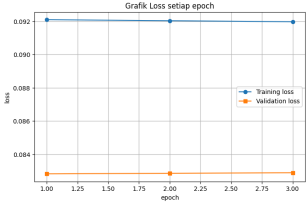
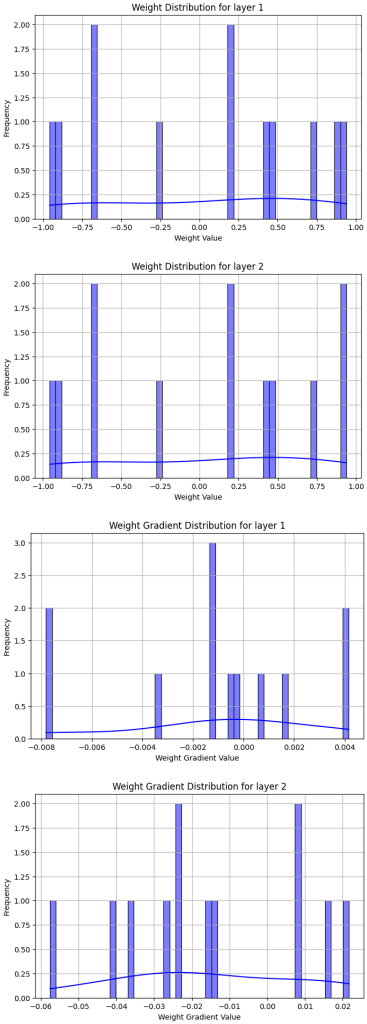
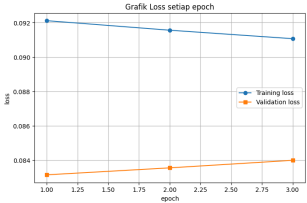
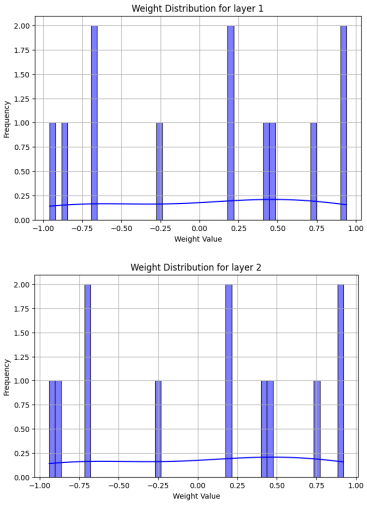


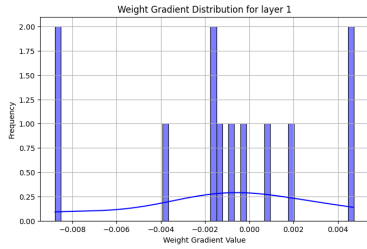
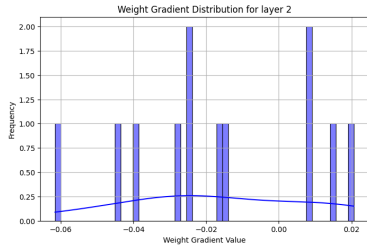
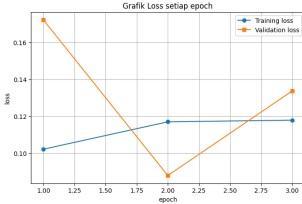
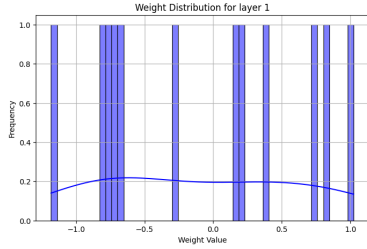
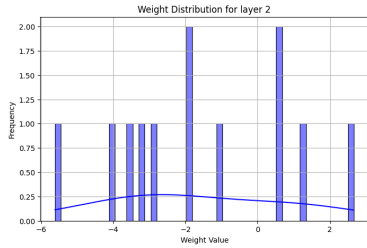
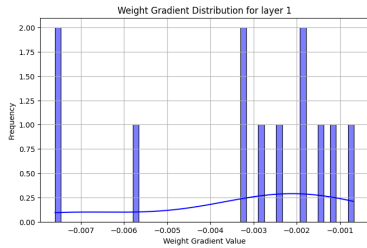
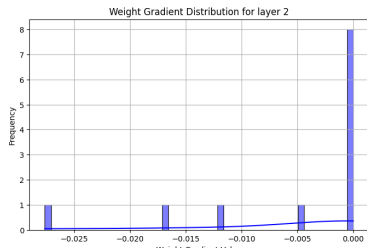
Berdasarkan hasil pengujian pada pengaruh fungsi aktivasi, terlihat bahwa seluruh variasi memberikan hasil yang berbeda-beda, tetapi terjadi peningkatan loss validasi yang sama. Pada fungsi aktivasi linear, model cenderung tidak mampu menangkap kompleksitas pola non-linear dalam data, sehingga performa validasi terus menurun. Meskipun fungsi aktivasi non-linear lain seperti ReLU, sigmoid, dan tanh telah digunakan, tren peningkatan loss validasi tetap terjadi, yang mengindikasikan bahwa permasalahan utama kemungkinan terletak pada aspek lain seperti struktur arsitektur model, kurangnya regularisasi, atau ketidaksesuaian parameter training.

3.3. Pengaruh *Learning Rate*

Learning rate yang digunakan untuk pengujian beserta hasilnya adalah sebagai berikut dengan jumlah layer tetap 3 layer, jumlah hidden size tetap 3 neuron, fungsi aktivasi sigmoid, fungsi loss MSE, ukuran input 784, ukuran output 10, dan inisialisasi bobot uniform random

dari -1 sampai 1 (dengan random state 42). Selain itu, ukuran batch yang digunakan adalah 2 dan jumlah epoch 3.

Learning Rate	Hasil prediksi	Grafik Loss Training dan Validasi	Distribusi Bobot dan Gradiennya
0.01	$y_{pred} = [3, 3]$ $y_{true} = [4, 5]$ Akurasi: 0		
0.1	$y_{pred} = [3, 3]$ $y_{true} = [4, 5]$ Akurasi: 0		

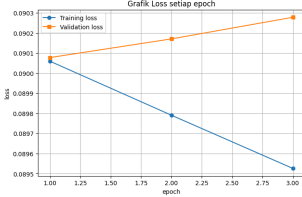
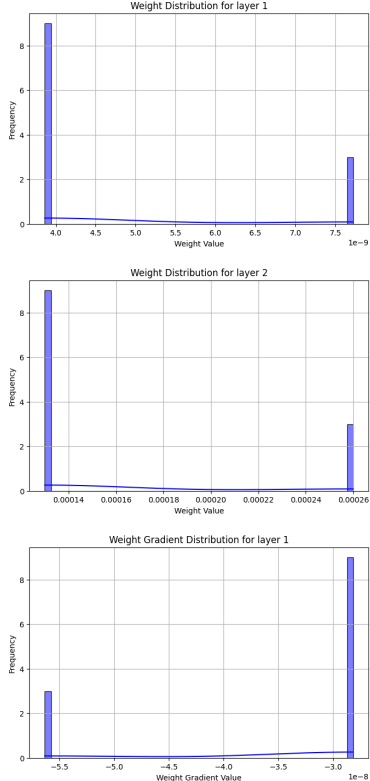
			 
10	$y_{pred} = [3, 3]$ $y_{true} = [4, 5]$ Akurasi: 0		   

Berdasarkan hasil yang diperoleh pada pengaruh learning rate, didapatkan bahwa model menunjukkan performa yang bervariasi. Pada learning rate 10, sempat terjadi penurunan loss validasi pada epoch kedua, tetapi kembali meningkat pada epoch berikutnya, yang

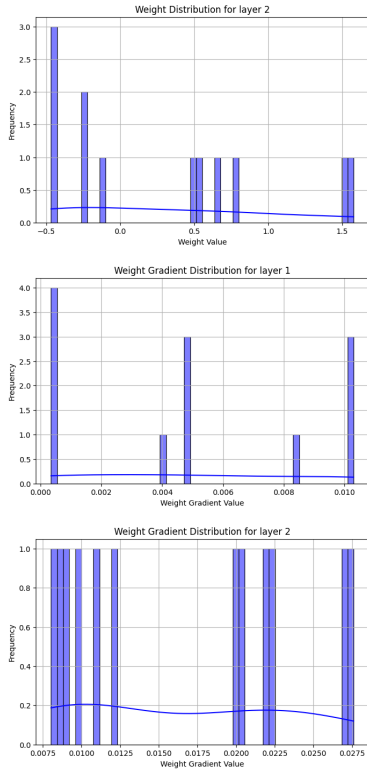
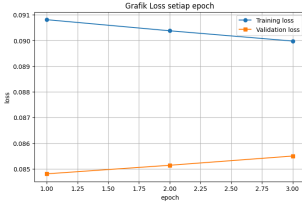
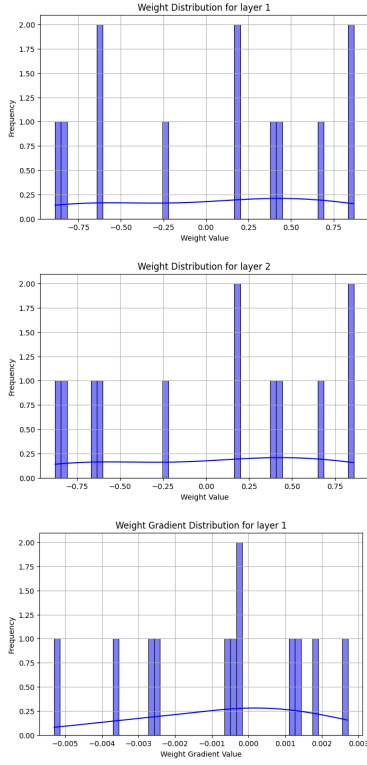
mengindikasikan bahwa proses pelatihan tidak stabil. Sementara itu, pada learning rate 0.01 dan 0.1, loss validasi terus meningkat dari awal hingga akhir pelatihan. Hal ini menunjukkan bahwa pemilihan learning rate yang terlalu kecil ataupun terlalu besar belum mampu menghasilkan proses pelatihan yang optimal, dan pelatihan lebih lanjut diperlukan untuk menemukan nilai learning rate yang tepat. Nilai learning rate yang terlalu kecil dapat menyebabkan model membutuhkan waktu lama untuk mencapai local optima, sedangkan learning rate yang terlalu besar, dapat menyebabkan model melewati local optima, sehingga learning rate yang cocok sulit ditemukan dan proses pelatihan menjadi tidak stabil. Oleh karena itu, pemilihan nilai learning rate yang tepat menjadi hal yang sangat penting agar model dapat belajar secara efisien dan dapat mencapai performa yang optimal.

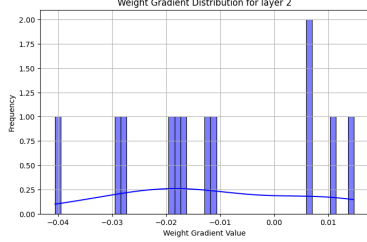
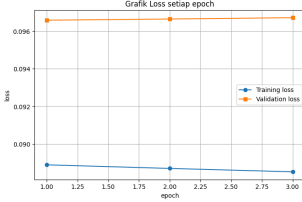
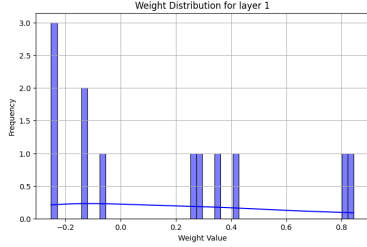
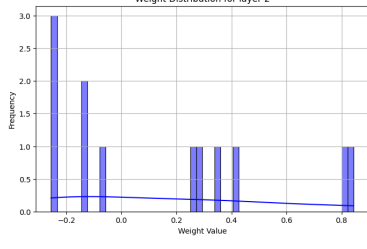
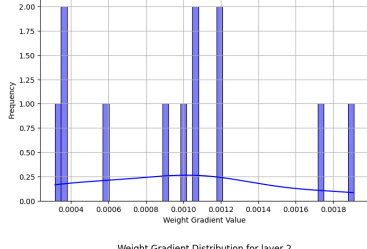
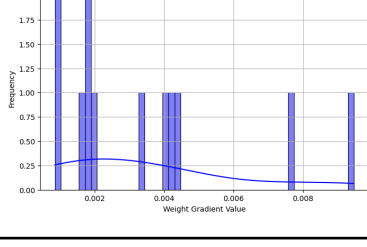
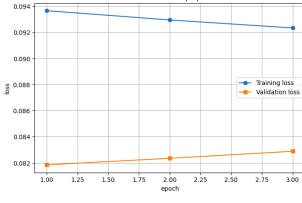
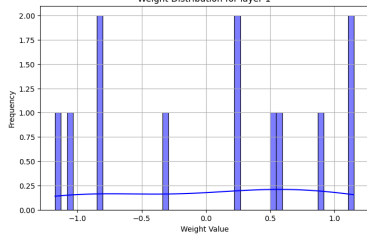
3.4. Pengaruh Inisialisasi Bobot

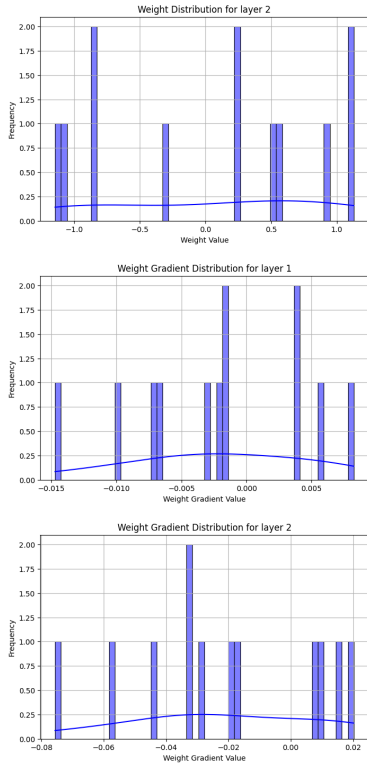
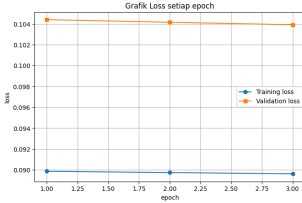
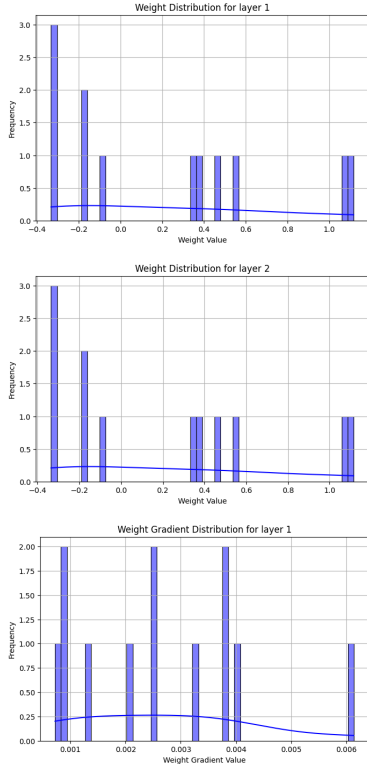
Inisialisasi bobot yang digunakan untuk pengujian beserta hasilnya adalah sebagai berikut dengan jumlah layer tetap 3 layer, jumlah hidden size tetap 3 neuron, fungsi aktivasi sigmoid, fungsi loss MSE, ukuran input 784, dan ukuran output 10. Selain itu, ukuran batch yang digunakan adalah 2, learning rate 0.1, dan jumlah epoch 3.

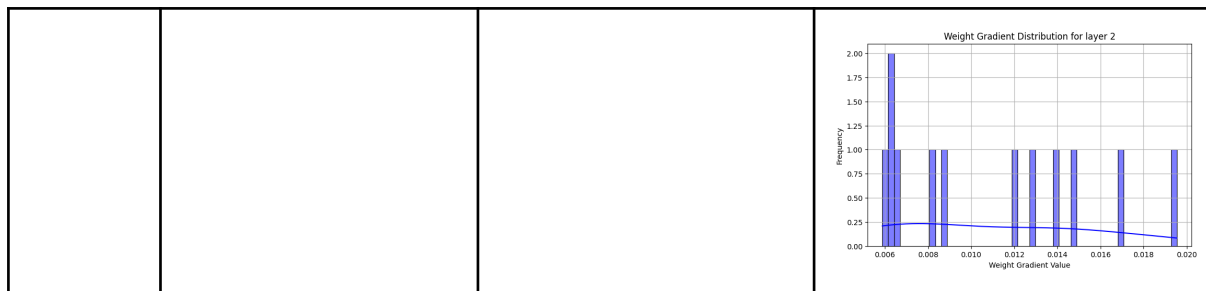
Inisialisasi Bobot	Hasil prediksi	Grafik Loss Training dan Validasi	Distribusi Bobot dan Gradiennya
zero	$y_{pred} = [1, 1]$ $y_{true} = [4, 5]$ Akurasi: 0		

uniform	$y_{\text{pred}} = [3, 3]$ $y_{\text{true}} = [4, 5]$ Akurasi: 0		
normal	$y_{\text{pred}} = [1, 1]$ $y_{\text{true}} = [4, 5]$ Akurasi: 0		

			
<p>Xavier uniform</p>	<p>$y_pred = [3, 3]$ $y_true = [4, 5]$ Akurasi: 0</p>		

			
Xavier normal	$y_{\text{pred}} = [1, 1]$ $y_{\text{true}} = [4, 5]$ Akurasi: 0		   
He uniform	$y_{\text{pred}} = [3, 3]$ $y_{\text{true}} = [4, 5]$ Akurasi: 0		

			
He normal	$y_pred = [1, 1]$ $y_true = [4, 5]$ Akurasi: 0		

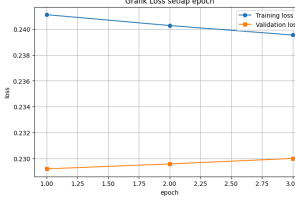
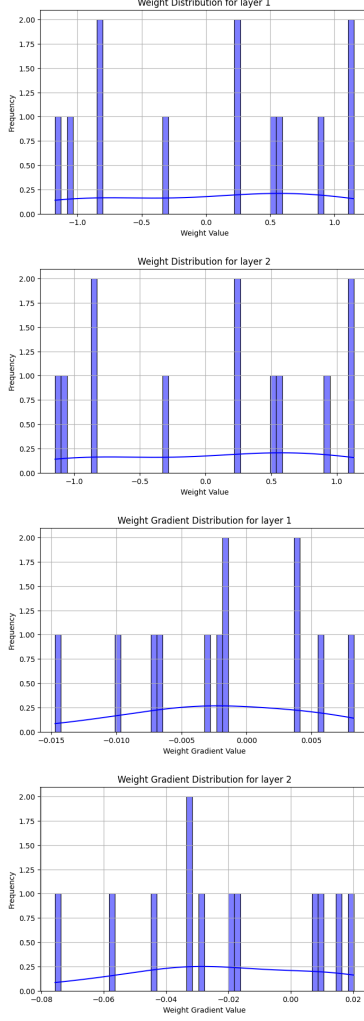
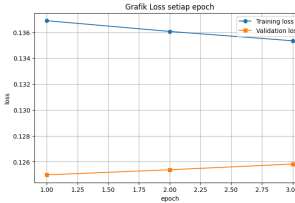
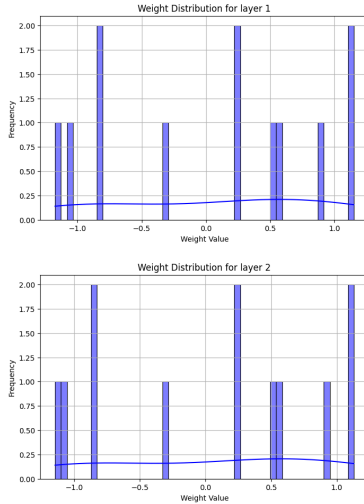


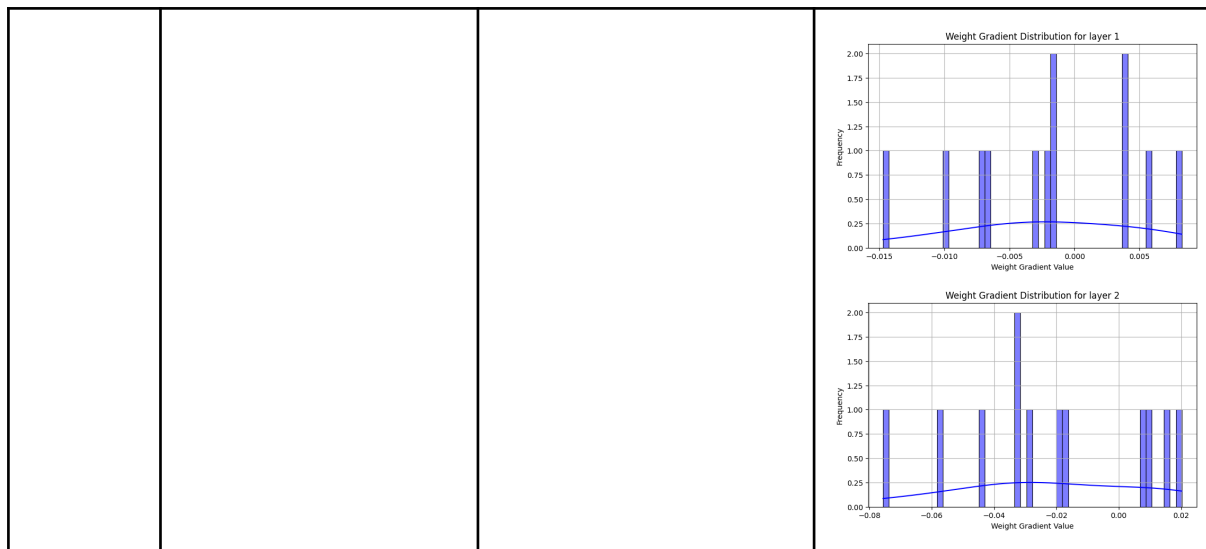
Berdasarkan hasil pengujian pada variasi inisialisasi bobot, didapatkan bahwa jenis inisialisasi bobot memiliki pengaruh yang cukup signifikan terhadap performa model. Dari seluruh variasi yang diuji, hanya inisialisasi normal dan He normal yang menunjukkan tren penurunan loss validasi. Sedangkan, metode inisialisasi yang lain, seperti zero, uniform, Xavier, dan He uniform menghasilkan tren peningkatan loss validasi selama proses pelatihan. Hal ini mengindikasikan bahwa pemilihan strategi inisialisasi bobot yang sesuai dapat membantu mempercepat konvergensi nilai loss dan meningkatkan generalisasi model. Hasil pengujian menunjukkan bahwa hanya metode normal dan He normal yang dapat menghasilkan penurunan loss validasi. Hal ini dapat terjadi karena kedua metode tersebut memberikan inisialisasi bobot awal yang relatif seimbang dan stabil untuk digunakan pada model ini. Berdasarkan visualisasi distribusi bobot awal pada layer pertama dan kedua, baik untuk inisialisasi bobot normal maupun He normal, terlihat bahwa keduanya memberikan bobot yang lebih acak dan tidak terlalu ekstrem. Sebagian besar bobot ada di sekitar nol, tetapi tetap memiliki variasi bobot negatif dan positif yang cukup. Hal ini menunjukkan bahwa bobot awal yang dihasilkan bersifat lebih seimbang. Distribusi ini berkontribusi pada penurunan loss validasi yang memungkinkan proses pelatihan model lebih efektif dibandingkan metode inisialisasi yang lain.

3.5. Pengaruh Regularisasi

Regularisasi yang digunakan untuk pengujian beserta hasilnya adalah sebagai berikut dengan jumlah layer tetap 3 layer, jumlah hidden size tetap 3 neuron, fungsi aktivasi sigmoid, fungsi loss MSE, ukuran input 784, ukuran output 10, dan inisialisasi bobot uniform random dari -1 sampai 1 (dengan random state 42). Selain itu, ukuran batch yang digunakan adalah 2, learning rate 0.1, dan jumlah epoch 3.

Regularisasi	Hasil prediksi	Grafik Loss Training dan Validasi	Distribusi Bobot dan Gradiennya
--------------	----------------	-----------------------------------	---------------------------------

<p>L1 ($\lambda = 0.01$)</p>	<p>$y_{\text{pred}} = [3, 3]$ $y_{\text{true}} = [4, 5]$ Akurasi: 0</p>		
<p>L2 ($\lambda = 0.01$)</p>	<p>$y_{\text{pred}} = [3, 3]$ $y_{\text{true}} = [4, 5]$ Akurasi: 0</p>		



Berdasarkan hasil eksperimen regularisasi yang telah dilakukan, terlihat bahwa regularisasi L1 dan L2 dengan nilai lambda 0.01 tidak mampu menurunkan nilai loss validasi. Keduanya menunjukkan tren peningkatan loss validasi selama proses training. Hal ini dapat disebabkan oleh nilai lambda yang relatif kecil sehingga efek regularisasinya kurang signifikan, atau model mungkin belum terlalu kompleks sehingga penambahan regularisasi hanya menghambat proses training. Regularisasi ini berguna dalam mengatasi overfitting atau model yang terlalu banyak mempelajari data training, sehingga apabila pengaruhnya (lambda) kecil atau parameternya terlalu sedikit (model kurang kompleks), maka pengaruh regularisasi tidak akan signifikan pada performa model saat training. Oleh karena itu, pemilihan nilai lambda dan kompleksitas model menjadi faktor penting dalam penerapan regularisasi.

3.6. Perbandingan dengan Scikit-Learn

Hyperparameter yang digunakan untuk pengujian dan perbandingan dengan sklearn tertera pada tabel berikut

Neuron	5
Hidden Layer	3
Fungsi Aktivasi	sigmoid
Learning Rate	0.1

Berdasarkan hyperparameter yang digunakan, berikut hasil pengujian pada kedua model, FFNN dan sklearn.

Model	Hasil prediksi
FFNN	<pre> Epoch 1/7: 21% ███████ 3/14 [02:38<09:42, 52.96s/batch, Batch Loss=0.0898] Epoch 1/7: 100% ██████████ 14/14 [01:11<00:00, 5.14s/batch, Batch Loss=0.0987] Epoch 1: Train Loss = 0.09271511245739096, Val Loss = 0.0894888900831522 Epoch 2/7: 100% ██████████ 14/14 [03:29<00:00, 14.99s/batch, Batch Loss=0.0967] Epoch 2: Train Loss = 0.09181082396874517, Val Loss = 0.08834854561843034 Epoch 3/7: 100% ██████████ 14/14 [06:00<00:00, 25.73s/batch, Batch Loss=0.095] Epoch 3: Train Loss = 0.09118387694915357, Val Loss = 0.08753884144012497 Epoch 4/7: 100% ██████████ 14/14 [06:51<00:00, 29.37s/batch, Batch Loss=0.0935] Epoch 4: Train Loss = 0.09072833644735279, Val Loss = 0.08695233215019565 Epoch 5/7: 100% ██████████ 14/14 [11:51<00:00, 50.81s/batch, Batch Loss=0.0923] Epoch 5: Train Loss = 0.09039063738166779, Val Loss = 0.08652546155438262 Epoch 6/7: 100% ██████████ 14/14 [11:17<00:00, 48.38s/batch, Batch Loss=0.0913] Epoch 6: Train Loss = 0.09013656975310137, Val Loss = 0.0862143975810194 Epoch 7/7: 100% ██████████ 14/14 [12:58<00:00, 55.63s/batch, Batch Loss=0.0905] Epoch 7: Train Loss = 0.089941471654204, Val Loss = 0.08598683570470383 Prediksi: [1 1 1 1] Kelas sebenarnya: [4 2 1 0] Akurasi: 0.25 </pre>
sklearn	<pre> Prediksi: [3 1 1 1] Kelas sebenarnya: [4 2 1 0] Akurasi: 0.25 C:\Users\devin\AppData\Local warnings.warn(</pre>

Berdasarkan hasil pengujian, ada perbedaan antara hasil implementasi sendiri dan hasil dengan menggunakan library sklearn. Ini disebabkan oleh beberapa faktor seperti cara menginisialisasi weight yang berbeda, atau cara menghitung gradien loss yang mungkin berbeda. Pengujian dengan sklearn juga menggunakan `learning_rate_init` untuk learning rate awal, yang artinya learning rate bisa berubah-ubah saat pelatihan.

Bab 4

Kesimpulan dan Saran

4.1. Kesimpulan

Pada tugas ini, penulis berhasil membuat implementasi model FFNN dengan menggunakan autodiff pada saat hitung gradien dan inisialisasi bobot tambahan (Xavier dan He) pada tahap inisialisasi. Berdasarkan model yang sudah dibuat dan dilakukan eksperimen pada beberapa hyperparameter, didapatkan bahwa pengaruh jumlah layer, hidden size, fungsi aktivasi, learning rate (0.01 dan 0.1), regularisasi, dan beberapa inisialisasi bobot (zero, uniform, Xavier normal dan uniform, serta He uniform) menunjukkan peningkatan loss validasi di setiap epoch-nya. Konfigurasi yang mampu menurunkan loss validasi di setiap epoch-nya hanya learning rate bernilai 10 dan inisialisasi bobot normal dan He normal. Selain itu, model yang digunakan juga memiliki arsitektur yang sangat sederhana, sehingga model kurang mengenali data yang kompleks. Faktor lain yang menyebabkan nilai loss validasi yang terus meningkat adalah penggunaan dataset yang sangat sedikit. Hal ini dilakukan karena waktu yang dibutuhkan model untuk training bisa sangat lama dalam satu epoch. Dengan jumlah dataset yang sangat sedikit, model sangat rentan terhadap overfitting atau underfitting, dan hasil validasi juga menjadi sangat fluktuatif. Oleh karena itu eksperimen yang dilakukan untuk mengenali sensitivitas hyperparameter pada model FFNN. Meskipun terdapat berbagai keterbatasan, penulis telah berhasil mengimplementasikan model FFNN menggunakan bahasa pemrograman python.

4.2. Saran

Saran untuk kelompok ini adalah sebagai berikut.

1. Meningkatkan komunikasi antar anggota kelompok
2. Pembagian tugas yang lebih jelas dan merata antar anggota kelompok
3. Mengerjakan tugas lebih awal
4. Menambahkan parallelization menggunakan GPU agar *training* lebih efisien.
5. Memanfaatkan bahasa pemrograman lain dalam perhitungan untuk menambah efisiensi dalam melakukan perhitungan.

PEMBAGIAN TUGAS

NIM	Nama	Pembagian Tugas
12821046	Fardhan Indrayesa	<ul style="list-style-type: none">- Value- ValueTensor- Regularization- Bonus automatic differentiation- Testing- Laporan
13522064	Devinzen	<ul style="list-style-type: none">- Layer- Model- Bonus Xavier dan He initialization- Testing

REFERENSI

<https://www.geeksforgeeks.org/feedforward-neural-network/>

<https://www.youtube.com/watch?v=VMj-3S1tku0>