Laporan Tugas Besar 1

IF3270 Pembelajaran Mesin

Feedforward Neural Network

Disusun Oleh

13522001 Mohammad Nugraha Eka Prawira

13522105 Fabian Radenta Bangun

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Semester VI - 2025

DAFTAR ISI

DESKRIPSI PERSOALAN

Implementasikan suatu modul FFNN yang memenuhi ketentuan-ketentuan berikut:

- FFNN yang diimplementasikan dapat **menerima jumlah neuron dari tiap layer** (termasuk input layer dan output layer)
- FFNN yang diimplementasikan dapat **menerima fungsi aktivasi dari tiap layer**. Pilihan fungsi aktivasi yang harus diimplementasikan adalah sebagai berikut:

Nama Fungsi Aktivasi	Definisi Fungsi
Linear	$Linear(x) = x$
ReLU	$ReLU(x) = \max(0, x)$
Sigmoid	$\sigma(x) = \frac{1}{1 + e^{-x}}$
Hyperbolic Tangent (tanh)	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Softmax	Untuk vector $\vec{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$, $softmax(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$

- FFNN yang diimplementasikan dapat **menerima fungsi loss** dari model tersebut. Pilihan loss function yang harus diimplementasikan adalah sebagai berikut:

Nama Fungsi Loss	Definisi Fungsi
<u>MSE</u>	$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

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

  - Catatan:
    - Binary cross-entropy merupakan kasus khusus categorical cross-entropy dengan kelas sebanyak 2
    - Log yang digunakan merupakan logaritma natural (logaritma dengan basis e)
- Terdapat mekanisme untuk **inisialisasi bobot** tiap neuron (termasuk bias). Pilihan metode inisialisasi bobot yang harus diimplementasikan adalah sebagai berikut:
  - **Zero initialization**
  - Random dengan distribusi **uniform**.
    - Menerima parameter lower bound (batas minimal) dan upper bound (batas maksimal)
    - Menerima parameter seed untuk reproducibility
  - Random dengan distribusi **normal**.
    - Menerima parameter mean dan variance
    - Menerima parameter seed untuk reproducibility
- Instance model yang diinisialisasikan harus bisa **menyimpan bobot** tiap neuron (termasuk bias)
- Instance model yang diinisialisasikan harus bisa **menyimpan gradien bobot** tiap neuron (termasuk bias)
- Instance model memiliki method untuk **menampilkan model** berupa **struktur jaringan** beserta **bobot** dan **gradien bobot** tiap neuron dalam bentuk graf. (Format graf dibebaskan)
- Instance model memiliki method untuk **menampilkan distribusi bobot** dari tiap layer.
  - Menerima masukan berupa list of integer (bisa disesuaikan ke struktur data lain sesuai kebutuhan) yang menyatakan layer mana saja yang distribusinya akan di-plot

- Instance model memiliki method untuk **menampilkan distribusi gradien bobot** dari tiap layer.
    - Menerima masukan berupa list of integer (bisa disesuaikan ke struktur data lain sesuai kebutuhan) yang menyatakan layer mana saja yang distribusinya akan di-plot
- Instance model memiliki method untuk ***save*** dan ***load***
- Model memiliki implementasi **forward propagation** dengan ketentuan sebagai berikut:
    - Dapat menerima input berupa **batch**.
- Model memiliki implementasi **backward propagation** untuk menghitung perubahan gradien:
    - Dapat menangani perhitungan perubahan gradien untuk input data **batch**.
    - Gunakan konsep **chain rule** untuk menghitung gradien tiap bobot terhadap loss function.
    - Berikut merupakan **turunan pertama** untuk setiap fungsi aktivasi:

Nama Fungsi Aktivasi	**Turunan Pertama**
Linear	$$\frac{d\left(Linear(x)\right)}{dx} = 1$$
ReLU	$$\frac{d\left(ReLU(x)\right)}{dx} = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$
Sigmoid	$$\frac{d\left(\sigma(x)\right)}{dx} = \sigma(x)\left(1 - \sigma(x)\right)$$
Hyperbolic Tangent (tanh)	$$\frac{d(\tanh(x))}{dx} = \left(\frac{2}{e^x - e^{-x}}\right)^2$$
Softmax	Untuk vector $\vec{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$, $$\frac{d(softmax(\vec{x})_i)}{d\vec{x}} = \begin{bmatrix} \frac{\partial(softmax(\vec{x})_1)}{\partial x_1} & \cdots & \frac{\partial(softmax(\vec{x})_1)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial(softmax(\vec{x})_n)}{\partial x_1} & \cdots & \frac{\partial(softmax(\vec{x})_n)}{\partial x_n} \end{bmatrix}$$ Dimana untuk $i, j \in \{1, \dots, n\}$, $$\frac{\partial(softmax(\vec{x})_i)}{\partial x_j} = softmax(\vec{x})_i\left(\delta_{i,j} - softmax(\vec{x})_j\right)$$ $$\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

- Berikut merupakan **turunan pertama** untuk setiap fungsi loss terhadap bobot suatu FFNN (lanjutkan sisanya menggunakan chain rule):

Nama Fungsi Loss	Definisi Fungsi
MSE	$\frac{\partial \mathcal{L}_{MSE}}{\partial W} = -\frac{2}{n}\sum_{i=1}^{n}\frac{\partial \mathcal{L}_{MSE}}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial W} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)\frac{\partial \hat{y}_i}{\partial W}$
Binary Cross-Entropy	$\frac{\partial \mathcal{L}_{BCE}}{\partial W} = -\frac{1}{n}\sum_{i=1}^{n}\frac{\partial \mathcal{L}_{BCE}}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial W} = -\frac{1}{n}\sum_{i=1}^{n}\frac{\hat{y}_i - y_i}{\hat{y}_i(1-\hat{y}_i)}\frac{\partial \hat{y}_i}{\partial W}$
Categorical Cross-Entropy	$\frac{\partial \mathcal{L}_{CCE}}{\partial W} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{C}\frac{\partial \mathcal{L}_{CCE}}{\partial \hat{y}_{ij}}\frac{\partial \hat{y}_{ij}}{\partial W} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{C}\frac{y_{ij}}{\hat{y}_{ij}}\frac{\partial \hat{y}_{ij}}{\partial W}$

- Model memiliki implementasi **weight update** dengan menggunakan **gradient descent** untuk memperbarui bobot berdasarkan gradien yang telah dihitung, berikut persamaannya:

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

α = Learning rate

- Implementasi untuk pelatihan model harus memenuhi ketentuan berikut:
  - Dapat menerima parameter berikut:
    - Batch size
    - Learning rate
    - Jumlah epoch
    - Verbose
      - Verbose 0 berarti tidak menampilkan apa-apa selama pelatihan
      - Verbose 1 berarti hanya menampilkan progress bar beserta dengan kondisi training loss dan validation loss saat itu
  - Proses pelatihan mengembalikan **histori dari proses pelatihan** yang berisi **training loss** dan **validation loss tiap epoch.**

- Lakukan **pengujian** terhadap implementasi FFNN dengan ketentuan sebagai berikut:
  - Analisis pengaruh beberapa hyperparameter sebagai berikut:
    - Pengaruh **depth (banyak layer)** dan **width (banyak neuron per layer)**
      - Pilih 3 variasi kombinasi width (depth tetap) dan 3 variasi depth (width semua layer tetap)
      - Bandingkan hasil akhir prediksinya
      - Bandingkan grafik loss pelatihannya
    - Pengaruh **fungsi aktivasi** hidden layer

        - Lakukan untuk setiap fungsi aktivasi yang diimplementasikan **kecuali softmax.**
        - Bandingkan hasil akhir prediksinya
        - Bandingkan grafik loss pelatihannya
        - Bandingkan distribusi bobot dan gradien bobot dari beberapa/semua layer pada model
    - Pengaruh **learning rate**
        - Lakukan 3 variasi learning rate (nilainya dibebaskan)
        - Bandingkan hasil akhir prediksinya
        - Bandingkan grafik loss pelatihannya
        - Bandingkan distribusi bobot dan gradien bobot dari beberapa/semua layer pada model
    - Pengaruh **inisialisasi bobot**
        - Lakukan untuk setiap metode inisialisasi bobot yang diimplementasikan
        - Bandingkan hasil akhir prediksinya
        - Bandingkan grafik loss pelatihannya
        - Bandingkan distribusi bobot dan gradien bobot dari beberapa/semua layer pada model
- Analisis perbandingan hasil prediksi dengan library sklearn MLP
    - Lakukan satu kali pelatihan dengan hyperparameter yang sama untuk kedua model
    - Hyperparameter yang digunakan dibebaskan
    - Bandingkan hasil akhir prediksinya saja
- Gunakan dataset berikut untuk menguji model: mnist_784
    - Gunakan method fetch_openml dari sklearn untuk memuat dataset
    - Berikut contoh untuk memuat dataset: Contoh
- ~~Akan ada beberapa ***test case*** yang akan diberikan oleh tim asisten (menyusul)~~ Note: Tetap akan ada test case untuk penilaian, akan dilakukan saat asisten memeriksa tugas~~.~~
- Pengujian dilakukan di file .ipynb terpisah

PEMBAHASAN

Penjelasan Implementasi

Deskripsi Kelas dan Atribut

File Utils.py

```
import numpy as np
import matplotlib.pyplot as plt

# ----------- ACTIVATION FUNCTIONS -----------
def linear(x) :
   return x

def relu(x) :
   return np.maximum(0,x)

def sigmoid(x) :
   return 1/(1+np.exp(-x))

def tanh(x) :
   return np.tanh(x)

def softmax(x) : #vektor dalam array
   x = x-np.max(x, axis=1, keepdims=True)
   exp_x = np.exp(x)
   return exp_x/np.sum(exp_x, axis=1, keepdims=True)

def leaky_relu(x, alpha=0.01):
   return np.where(x>0, x, alpha*x)

def elu(x, alpha=1.0):
   return np.where(x>0, x, alpha*(np.exp(x)-1))


# ----------- TURUNAN ACTIVATION FUNCTIONS -----------
def d_linear(x) :
   return np.ones_like(x)
```

```python
def d_relu(x) :
    return (x>0).astype(float)

def d_sigmoid(x) :
    s = sigmoid(x)
    return s*(1-s)

def d_tanh(x) :
    return 1-np.tanh(x)**2

def d_softmax(x) :
    return np.ones_like(x)

def d_leaky_relu(x, alpha=0.01):
    return np.where(x>0, 1, alpha)

def d_elu(x, alpha=1.0):
    return np.where(x>0, 1, alpha * np.exp(x))


# ----------- LOSS FUNCTIONS -----------
def mse(y_true, y_pred):
    return np.mean((y_true-y_pred)**2)

def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1-epsilon)
    return -np.mean(y_true*np.log(y_pred)+(1-y_true)*np.log(1-y_pred))

def categorical_cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1-epsilon)
    return -np.mean(np.sum(y_true*np.log(y_pred), axis=1))


# ----------- TURUNAN LOSS FUNCTIONS -----------
def d_mse(y_true, y_pred):
    return 2*(y_pred-y_true)/y_true.size

def d_bce(y_true, y_pred):
    return (y_pred-y_true)/(y_pred*(1-y_pred)+1e-9)
```

```python
def d_cce(y_true, y_pred):
   return y_pred-y_true

# ----------- INITIALIZE WEIGHT -----------
def initialize_weights(input_size, output_size, method="uniform", **kwargs):
   if method == "zero":
       return np.zeros((input_size, output_size))
   elif method == "uniform":
       low = kwargs.get("low", -0.1)
       high = kwargs.get("high", 0.1)
       return np.random.uniform(low, high, (input_size, output_size))
   elif method == "normal":
       mean = kwargs.get("mean", 0)
       std = kwargs.get("std", 0.01)
       return np.random.normal(mean, std, (input_size, output_size))
   elif method == "xavier":
       limit = np.sqrt(6/(input_size+output_size))
       return np.random.uniform(-limit, limit, (input_size, output_size))
   elif method=="he" :
       std = np.sqrt(2/input_size)
       return np.random.normal(0, std, (input_size, output_size))
```

File **utils.py** merupakan modul utilitas yang berisi implementasi berbagai fungsi penting untuk mendukung pembangunan model neural network. Modul ini mencakup tiga kategori utama: fungsi aktivasi, fungsi loss, serta metode inisialisasi bobot. Fungsi aktivasi yang disediakan meliputi Linear, ReLU, Sigmoid, Tanh, Softmax, Leaky ReLU, dan ELU, beserta turunannya. Fungsi loss yang tersedia antara lain Mean Squared Error (MSE), Binary Cross Entropy (BCE), dan Categorical Cross Entropy (CCE), lengkap dengan turunannya untuk menghitung gradien selama pelatihan. Selain itu, modul ini menyediakan berbagai metode inisialisasi bobot seperti inisialisasi nol, uniform, normal, Xavier, dan He, yang dapat dipilih sesuai kebutuhan.

Kelas FFNN

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as colors
import networkx as nx
import pickle
from utils import (
```

```python
    linear, relu, sigmoid, tanh, softmax, leaky_relu, elu,
    d_linear, d_relu, d_sigmoid, d_tanh, d_softmax,d_leaky_relu, d_elu,
    mse, binary_cross_entropy, categorical_cross_entropy,
    d_mse, d_bce, d_cce, initialize_weights
)

class FFNN :
    def __init__(self, layer_sizes, activation_func, loss_func="cce",
weight_init="uniform", learning_rate=0.01, l1_lambda=0, l2_lambda=0):
        self.layers = []
        for i in range (len(layer_sizes)-1) :
            layer = {
                "weights" : initialize_weights(layer_sizes[i],layer_sizes[i+1],
weight_init),
                "bias" : np.zeros((1, layer_sizes[i+1])),
                "activation" : activation_func[i],
                "grad_weight" : None,
                "grad_bias" : None
            }
            self.layers.append(layer)
        self.loss_func = loss_func
        self.learning_rate = learning_rate
        self.l1_lambda = l1_lambda
        self.l2_lambda = l2_lambda
        self.history = {"train_loss" : [], "val_loss" : []}

    def _apply_activation(self, x, activation) :
        if activation=="linear" :
            return linear(x)
        elif activation=="relu" :
            return relu(x)
        elif activation=="sigmoid" :
            return sigmoid(x)
        elif activation=="tanh" :
            return tanh(x)
        elif activation=="softmax" :
            return softmax(x)
        elif activation=="leaky_relu" :
            return leaky_relu(x)
        elif activation=="elu" :
            return elu(x)
```

```python
        else:
            raise ValueError(f"Unknown function: {activation}")

    def _activation_derivative(self, x, activation) :
        if activation=="linear" :
            return d_linear(x)
        elif activation=="relu" :
            return d_relu(x)
        elif activation=="sigmoid" :
            return d_sigmoid(x)
        elif activation=="tanh" :
            return d_tanh(x)
        elif activation=="softmax" :
            return d_softmax(x)
        elif activation=="leaky_relu" :
            return d_leaky_relu(x)
        elif activation=="elu" :
            return d_elu(x)
        else:
            raise ValueError(f"Unknown function: {activation}")

    def _get_loss_function(self) :
        if self.loss_func=="mse" :
            return mse
        elif self.loss_func=="bce" :
            return binary_cross_entropy
        elif self.loss_func=="cce" :
            return categorical_cross_entropy

    def _get_loss_derivative(self) :
        if self.loss_func=="mse" :
            return d_mse
        elif self.loss_func=="bce" :
            return d_bce
        elif self.loss_func=="cce" :
            return d_cce

    def forward(self, x) :
        self.input = x
        for layer in self.layers :
            layer["input"] = x
```

```python
            Z = np.dot(x, layer["weights"]) + layer["bias"]
            A = self._apply_activation(Z, layer["activation"])
            layer["output"] = A
            x = A
        return x

    def compute_regularization_loss(self):
        l1_loss = 0
        l2_loss = 0
        if self.l1_lambda > 0 or self.l2_lambda > 0:
            for layer in self.layers:
                if self.l1_lambda > 0:
                    l1_loss+=np.sum(np.abs(layer["weights"]))
                if self.l2_lambda > 0:
                    l2_loss+=np.sum(np.square(layer["weights"]))
        return self.l1_lambda*l1_loss+0.5*self.l2_lambda*l2_loss

    def backward(self, y_true, y_pred) :
        if self.layers[-1]["activation"] == "softmax" and self.loss_func == "cce":
            error = y_pred-y_true
        else:
            error = self._get_loss_derivative()(y_true, y_pred)
        for i in reversed(range(len(self.layers))) :
            layer = self.layers[i]
            A = layer["output"]
            dA = error*self._activation_derivative(A, layer["activation"])

            if i==0:
                prev_output = self.input
            else:
                prev_output = self.layers[i-1]["output"]

            layer["grad_weights"] = np.dot(prev_output.T, dA)
            if self.l1_lambda > 0:
                l1_grad = np.sign(layer["weights"])
                layer["grad_weights"]+=self.l1_lambda*l1_grad
            if self.l2_lambda > 0:
                l2_grad = layer["weights"]
                layer["grad_weights"]+=self.l2_lambda*l2_grad

            layer["grad_bias"] = np.sum(dA, axis=0, keepdims=True)
```

```
            error = np.dot(dA, layer["weights"].T)

    def update_weights(self) :
        for layer in self.layers:
            layer["grad_weights"] = np.clip(layer["grad_weights"], -1, 1)
            layer["grad_bias"] = np.clip(layer["grad_bias"], -1, 1)
            layer["weights"] -= self.learning_rate*layer["grad_weights"]
            layer["bias"] -= self.learning_rate*layer["grad_bias"]


    def plot_weight_distribution(self, layers=None):
        if layers is None:
            layers = range(len(self.layers))

        plt.figure(figsize=(12, 4*len(layers)))
        for i in layers:
            plt.subplot(len(layers), 1, i+1)
            plt.hist(self.layers[i]['weights'].flatten(), bins=50)
            plt.title(f'Weight Distribution - Layer {i}')
        plt.tight_layout()
        plt.show()

    def plot_weight_gradient_distribution(self, layers=None):
        if layers is None:
            layers = range(len(self.layers))

        plt.figure(figsize=(12, 4*len(layers)))
        for i in layers:
            plt.subplot(len(layers), 1, i+1)
            if self.layers[i].get('grad_weights') is not None:
                plt.hist(self.layers[i]['grad_weights'].flatten(), bins=50)
                plt.title(f'Weight Gradient Distribution - Layer {i}')
        plt.tight_layout()
        plt.show()

    def visualize_network_structure(self, highlight_weights=True,
highlight_gradients=False):
        G = nx.DiGraph()
        pos = {}
        edge_weights = []
        for layer_idx, layer in enumerate(self.layers):
```

```
            num_neurons = layer['weights'].shape[1]
            for neuron_idx in range(num_neurons):
                node_name = f'Layer {layer_idx} - Neuron {neuron_idx}'
                G.add_node(node_name)
                pos[node_name] = (layer_idx, neuron_idx - (num_neurons-1)/2)

        for layer_idx in range(len(self.layers)-1):
            current_layer_neurons = self.layers[layer_idx]['weights'].shape[1]
            next_layer_neurons = self.layers[layer_idx+1]['weights'].shape[1]

            for curr_neuron in range(current_layer_neurons):
                for next_neuron in range(next_layer_neurons):
                    curr_node = f'Layer {layer_idx} - Neuron {curr_neuron}'
                    next_node = f'Layer {layer_idx+1} - Neuron {next_neuron}'

                    if highlight_weights:
                        weight = abs(self.layers[layer_idx]['weights'][curr_neuron,
next_neuron])
                    elif highlight_gradients and
self.layers[layer_idx].get('grad_weights') is not None:
                        weight =
abs(self.layers[layer_idx]['grad_weights'][curr_neuron, next_neuron])
                    else:
                        weight = 1

                    G.add_edge(curr_node, next_node, weight=weight)
                    edge_weights.append(weight)

        fig, ax = plt.subplots(figsize=(15, 10))

        if edge_weights:
            norm = colors.Normalize(vmin=min(edge_weights), vmax=max(edge_weights))
            cmap = cm.coolwarm

            for (u, v, data) in G.edges(data=True):
                nx.draw_networkx_edges(
                    G, pos,
                    edgelist=[(u,v)],
                    edge_color=cmap(norm(data['weight'])),
                    width=max(0.1, 2 * data['weight']),
                    alpha=0.6,
```

```
                   arrows=True,
                   arrowsize=10,
                   ax=ax
               )

           sm = cm.ScalarMappable(cmap=cmap, norm=norm)
           sm.set_array([])

           plt.colorbar(sm, ax=ax, label='Weight/Gradient Magnitude')
       else:
           nx.draw_networkx_edges(
               G, pos,
               edge_color='blue',
               width=0.5,
               alpha=0.6,
               arrows=True,
               arrowsize=10,
               ax=ax
           )

       nx.draw_networkx_nodes(G, pos, node_color='lightblue', node_size=300,
alpha=0.8, ax=ax)
       nx.draw_networkx_labels(G, pos, font_size=8, font_weight="bold", ax=ax)

       ax.set_title("Neural Network Structure Visualization")
       ax.axis('off')

       plt.tight_layout()
       plt.show()

   def save(self, filename):
       model_data = {
           "layers": self.layers,
           "loss_func": self.loss_func,
           "learning_rate": self.learning_rate,
           "l1_lambda": self.l1_lambda,
           "l2_lambda": self.l2_lambda,
           "history": self.history
       }

       with open(filename, 'wb') as f:
```

```
            pickle.dump(model_data, f)
        print(f"Model saved to {filename}")

    def load(self, filename):
        with open(filename, 'rb') as f:
            model_data = pickle.load(f)

        self.layers = model_data["layers"]
        self.loss_func = model_data["loss_func"]
        self.learning_rate = model_data["learning_rate"]
        self.l1_lambda = model_data["l1_lambda"]
        self.l2_lambda = model_data["l2_lambda"]
        self.history = model_data["history"]

        print(f"Model loaded from {filename}")

    def train(self, X_train, y_train, X_val, y_val, epochs=100, batch_size=32,
verbose=1):
        for epoch in range(epochs):
            epoch_train_losses = []

            indices = np.arange(len(X_train))
            np.random.shuffle(indices)

            if verbose == 1:
                from tqdm import tqdm
                pbar = tqdm(total=len(X_train), desc=f'Epoch {epoch+1}/{epochs}',
unit='samples')

            for i in range(0, len(X_train), batch_size):
                batch_indices = indices[i:i+batch_size]
                X_batch = X_train[batch_indices]
                y_batch = y_train[batch_indices]

                y_pred = self.forward(X_batch)
                self.backward(y_batch, y_pred)
                self.update_weights()

                batch_loss = self._get_loss_function()(y_batch, y_pred)
                reg_loss = self.compute_regularization_loss()
                epoch_train_losses.append(batch_loss + reg_loss)
```

```
            if verbose == 1:
                pbar.update(len(X_batch))

        if verbose == 1:
            pbar.close()

        train_loss = np.mean(epoch_train_losses)
        val_loss = self.compute_loss(X_val, y_val)
        self.history["train_loss"].append(train_loss)
        self.history["val_loss"].append(val_loss)

        if verbose == 1:
            print(f'\tTrain Loss: {train_loss:.4f} - Val Loss: {val_loss:.4f}')

    return self.history

def compute_loss(self, X, y) :
    y_pred = self.forward(X)
    loss_func = self._get_loss_function()
    data_loss = loss_func(y, y_pred)
    reg_loss = self.compute_regularization_loss()
    return data_loss+reg_loss

def predict(self, X):
    return self.forward(X)
```

Kelas FFNN (Feed-Forward Neural Network) merupakan implementasi neural network sederhana. Class ini mendukung berbagai fungsi aktivasi, fungsi loss, inisialisasi bobot, regularisasi (L1/L2), serta visualisasi struktur jaringan.

Kelas ini memiliki beberapa atribut, yaitu :

1. layers (list) : Menyimpan informasi setiap layer, termasuk *weights*, bias, *activation function*, dan gradien (grad_weights, grad_bias).
2. loss_func (string) : Menyimpan informasi jenis fungsi loss yang digunakan.
3. learning_rate (float) : Learning rate untuk *update weight.*
4. l1_lambda, l2_lambda (float) : Koefisien regularisasi L1 dan L2.
5. history (dictionary) : Menyimpan *history* dari *training loss dan validation loss*.

Kelas ini juga memiliki beberapa *method*, yaitu :

1. forward() : Propagasi maju untuk menghitung output jaringan.
2. backward() : Propagasi mundur untuk menghitung gradien error.

3. update_weights() : Memperbarui bobot dan bias.
4. train() : Melatih model sesuai dengan ketentuan input pada spesifikasi.
5. predict() : Memprediksi output.
6. compute_loss() : Menghitung total loss.
7. plot_weight_distribution() : Memvisualisasikan distribusi bobot per layer.
8. plot_weight_gradient_distribution() : Memvisualisasikan distribusi gradien bobot.
9. visualize_network_structure() : Memvisualisasikan graf jaringan.
10. save() : Menyimpan model ke file.
11. load() : Memuat model dari file.

Penjelasan Forward Propagation

Forward propagation adalah proses di mana input data dilewatkan melalui setiap layer jaringan saraf. Pada implementasi ini, proses forward propagation dilakukan sebagai berikut:

1. Input data masuk ke layer pertama
2. Setiap neuron menghitung weighted sum (Z = X * W + b)
3. Fungsi aktivasi diterapkan pada weighted sum
4. Output dari satu layer menjadi input untuk layer berikutnya
5. Output akhir layer terakhir merupakan prediksi

ML - ffnn.py

```python
def forward(self, x) :
        self.input = x
        for layer in self.layers :
            layer["input"] = x
            Z = np.dot(x, layer["weights"]) + layer["bias"]
            A = self._apply_activation(Z, layer["activation"])
            layer["output"] = A
            x = A
        return x
```

Penjelasan Backward Propagation dan Weight Update

Backward propagation adalah proses menghitung gradien kesalahan dan memperbarui bobot:

1. Hitung kesalahan antara prediksi dan target
2. Hitung gradien untuk setiap layer dari belakang ke depan
3. Perbarui bobot menggunakan gradien dan learning rate

ML - ffnn.py

```python
def backward(self, y_true, y_pred) :
        if self.layers[-1]["activation"] == "softmax" and self.loss_func == "cce":
            error = y_pred-y_true
        else:
            loss_derivative_func = self._get_loss_derivative()
            error = loss_derivative_func(y_true, y_pred)
        for i in reversed(range(len(self.layers))) :
            layer = self.layers[i]
            A = layer["output"]
            dA = error*self._activation_derivative(A, layer["activation"])

            if i==0:
                prev_output = self.input
            else:
                prev_output = self.layers[i-1]["output"]

            layer["grad_weights"] = np.dot(prev_output.T, dA)
            layer["grad_bias"] = np.sum(dA, axis=0, keepdims=True)
            error = np.dot(dA, layer["weights"].T)
```

Metode Visualisasi

Distribusi Bobot Jaringan (plot_weight_distribution)

Metode ini memvisualisasikan distribusi bobot untuk setiap lapisan jaringan menggunakan histogram.

Cara Kerja

- Membuat plot dengan jumlah subplot sesuai jumlah lapisan yang dipilih
- Menggunakan plt.hist() untuk menampilkan distribusi bobot yang diratakan (flatten)
- Memberikan judul unik untuk setiap subplot berdasarkan indeks lapisan

Kegunaan

- Menganalisis sebaran statistik bobot di setiap lapisan
- Mendeteksi potensi masalah seperti vanishing/exploding gradients
- Memahami bagaimana bobot tersebar selama proses pelatihan

ML - ffnn.py

```python
def plot_weight_distribution(self, layers=None):
        """Plot weight distribution for specified layers"""
        if layers is None:
            layers = range(len(self.layers))

        plt.figure(figsize=(12, 4*len(layers)))
        for i in layers:
            plt.subplot(len(layers), 1, i+1)
            plt.hist(self.layers[i]['weights'].flatten(), bins=50)
            plt.title(f'Weight Distribution - Layer {i}')
        plt.tight_layout()
        plt.show()
```

Distribusi Gradien Bobot (plot_weight_gradient_distribution)

Metode serupa dengan distribusi bobot, namun fokus pada gradien bobot dari setiap lapisan.

Cara Kerja

- Membuat plot dengan subplot untuk setiap lapisan
- Memvisualisasikan histogram gradien bobot yang diratakan
- Hanya memplot lapisan yang memiliki informasi gradien

Kegunaan

- Mengukur perubahan bobot selama proses backpropagation
- Mendeteksi tingkat pembaruan bobot di setiap lapisan
- Membantu dalam debugging dan optimasi proses pelatihan

```python
ML - ffnn.py

def plot_weight_gradient_distribution(self, layers=None):
        """Plot weight gradient distribution for specified layers"""
        if layers is None:
            layers = range(len(self.layers))

        plt.figure(figsize=(12, 4*len(layers)))
        for i in layers:
            plt.subplot(len(layers), 1, i+1)
            if self.layers[i].get('grad_weights') is not None:
                plt.hist(self.layers[i]['grad_weights'].flatten(), bins=50)
                plt.title(f'Weight Gradient Distribution - Layer {i}')
        plt.tight_layout()
        plt.show()
```

Visualisasi Struktur Jaringan (visualize_network_structure)

Metode paling kompleks yang menciptakan representasi visual graf dari arsitektur jaringan saraf.

Cara Kerja

- Membangun graf berarah (DiGraph) menggunakan NetworkX
- Membuat node untuk setiap neuron di setiap lapisan
- Menghubungkan neuron antar lapisan dengan edge
- Memberikan warna dan ketebalan edge berdasarkan:
    1. Magnitude bobot (default)
    2. Magnitude gradien (opsional)

Parameter

- highlight_weights: Warna edge berdasarkan magnitude bobot
- highlight_gradients: Warna edge berdasarkan magnitude gradien

ML - ffnn.py

```python
def visualize_network_structure(self, highlight_weights=True, highlight_gradients=False):
        """
        Parameters:
        -----------
        highlight_weights : bool, optional (default=True)
            If True, color nodes and edges based on weight magnitudes
        highlight_gradients : bool, optional (default=False)
            If True, color nodes and edges based on gradient magnitudes
        """
        G = nx.DiGraph()
        pos = {}
        edge_weights = []
        for layer_idx, layer in enumerate(self.layers):
            num_neurons = layer['weights'].shape[1]
            for neuron_idx in range(num_neurons):
                node_name = f'Layer {layer_idx} - Neuron {neuron_idx}'
                G.add_node(node_name)
                pos[node_name] = (layer_idx, neuron_idx - (num_neurons-1)/2)

        for layer_idx in range(len(self.layers)-1):
            current_layer_neurons = self.layers[layer_idx]['weights'].shape[1]
            next_layer_neurons = self.layers[layer_idx+1]['weights'].shape[1]

            for curr_neuron in range(current_layer_neurons):
                for next_neuron in range(next_layer_neurons):
                    curr_node = f'Layer {layer_idx} - Neuron {curr_neuron}'
                    next_node = f'Layer {layer_idx+1} - Neuron {next_neuron}'

                    if highlight_weights:
                        weight = abs(self.layers[layer_idx]['weights'][curr_neuron, next_neuron])
                    elif highlight_gradients and self.layers[layer_idx].get('grad_weights') is not None:
                        weight = abs(self.layers[layer_idx]['grad_weights'][curr_neuron, next_neuron])
                    else:
                        weight = 1

                    G.add_edge(curr_node, next_node, weight=weight)
                    edge_weights.append(weight)

        fig, ax = plt.subplots(figsize=(15, 10))

        if edge_weights:
            norm = colors.Normalize(vmin=min(edge_weights), vmax=max(edge_weights))
            cmap = cm.coolwarm

            for (u, v, data) in G.edges(data=True):
                nx.draw_networkx_edges(
                    G, pos,
                    edgelist=[(u,v)],
                    edge_color=cmap(norm(data['weight'])),
                    width=max(0.1, 2 * data['weight']),
                    alpha=0.6,
                    arrows=True,
                    arrowsize=10,
                    ax=ax
                )

            sm = cm.ScalarMappable(cmap=cmap, norm=norm)
            sm.set_array([])

            plt.colorbar(sm, ax=ax, label='Weight/Gradient Magnitude')
        else:
            nx.draw_networkx_edges(
                G, pos,
                edge_color='blue',
                width=0.5,
                alpha=0.6,
                arrows=True,
                arrowsize=10,
                ax=ax
            )

        nx.draw_networkx_nodes(G, pos, node_color='lightblue', node_size=300, alpha=0.8, ax=ax)
        nx.draw_networkx_labels(G, pos, font_size=8, font_weight="bold", ax=ax)

        ax.set_title("Neural Network Structure Visualization")
        ax.axis('off')

        plt.tight_layout()
        plt.show()
```

\

Hasil Pengujian

Pengaruh Depth dan Width

- Desain Eksperimen
  - Activation function : ReLU untuk selain fungsi aktivasi terakhir dan menggunakan softmax untuk fungsi aktivasi terakhir
  - Loss function : Categorical Cross-Entropy
  - Learning Rate : 0,01
  - Jumlah epoch : 10
  - Inisialisasi bobot : Uniform
  - Variasi Depth : [[784, 128, 10], [784, 128, 128, 10], [784, 128, 128, 128, 10]]
  - Variasi Width : [[784, 128, 10], [784, 256, 10], [784, 512, 10]]
- Hasil :

```
Analyzing Depth and Width

Width Variation 1: [784, 128, 10]
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 97117.15samples/s]
        Train Loss: 0.2522 - Val Loss: 0.1521
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 107940.56samples/s]
        Train Loss: 0.1052 - Val Loss: 0.1092
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 100505.06samples/s]
        Train Loss: 0.0764 - Val Loss: 0.1195
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 100918.58samples/s]
        Train Loss: 0.0565 - Val Loss: 0.1130
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 99183.51samples/s]
        Train Loss: 0.0446 - Val Loss: 0.0927
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 118720.26samples/s]
        Train Loss: 0.0367 - Val Loss: 0.0908
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 103090.34samples/s]
        Train Loss: 0.0273 - Val Loss: 0.0925
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 110591.89samples/s]
        Train Loss: 0.0215 - Val Loss: 0.1004
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 101567.58samples/s]
        Train Loss: 0.0149 - Val Loss: 0.1008
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 113948.56samples/s]
        Train Loss: 0.0108 - Val Loss: 0.0899
Accuracy: 0.9768571428571429

Width Variation 2: [784, 256, 10]
Epoch 1/10: 100%|██████████| 56000/56000 [00:01<00:00, 52843.00samples/s]
        Train Loss: 0.2334 - Val Loss: 0.1376
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 56775.04samples/s]
        Train Loss: 0.0972 - Val Loss: 0.1006
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 57339.35samples/s]
        Train Loss: 0.0640 - Val Loss: 0.1090
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 57633.09samples/s]
        Train Loss: 0.0480 - Val Loss: 0.0851
```

```
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 57234.80samples/s]
	Train Loss: 0.0363 - Val Loss: 0.0852
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 58444.86samples/s]
	Train Loss: 0.0263 - Val Loss: 0.0867
Epoch 7/10: 100%|██████████| 56000/56000 [00:01<00:00, 40452.14samples/s]
	Train Loss: 0.0180 - Val Loss: 0.0825
Epoch 8/10: 100%|██████████| 56000/56000 [00:01<00:00, 55459.82samples/s]
	Train Loss: 0.0126 - Val Loss: 0.0825
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 58937.94samples/s]
	Train Loss: 0.0084 - Val Loss: 0.0847
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 62186.15samples/s]
	Train Loss: 0.0059 - Val Loss: 0.0810
Accuracy: 0.9792142857142857

Width Variation 3: [784, 512, 10]
Epoch 1/10: 100%|██████████| 56000/56000 [00:01<00:00, 29910.92samples/s]
	Train Loss: 0.2178 - Val Loss: 0.1410
Epoch 2/10: 100%|██████████| 56000/56000 [00:01<00:00, 29959.89samples/s]
	Train Loss: 0.0869 - Val Loss: 0.0973
Epoch 3/10: 100%|██████████| 56000/56000 [00:01<00:00, 29078.63samples/s]
	Train Loss: 0.0569 - Val Loss: 0.0859
Epoch 4/10: 100%|██████████| 56000/56000 [00:01<00:00, 29241.49samples/s]
	Train Loss: 0.0382 - Val Loss: 0.0930
Epoch 5/10: 100%|██████████| 56000/56000 [00:01<00:00, 29428.55samples/s]
	Train Loss: 0.0264 - Val Loss: 0.0736
Epoch 6/10: 100%|██████████| 56000/56000 [00:01<00:00, 29051.38samples/s]
	Train Loss: 0.0184 - Val Loss: 0.0727
Epoch 7/10: 100%|██████████| 56000/56000 [00:02<00:00, 26311.09samples/s]
	Train Loss: 0.0120 - Val Loss: 0.0749
Epoch 8/10: 100%|██████████| 56000/56000 [00:01<00:00, 32946.37samples/s]
	Train Loss: 0.0086 - Val Loss: 0.0759
Epoch 9/10: 100%|██████████| 56000/56000 [00:01<00:00, 30415.56samples/s]
	Train Loss: 0.0048 - Val Loss: 0.0710
Epoch 10/10: 100%|██████████| 56000/56000 [00:01<00:00, 33396.49samples/s]
	Train Loss: 0.0028 - Val Loss: 0.0707
Accuracy: 0.9815714285714285

Depth Variation 1: [784, 128, 10]
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 120989.31samples/s]
	Train Loss: 0.2544 - Val Loss: 0.1397
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 103365.45samples/s]
	Train Loss: 0.1076 - Val Loss: 0.1072
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 117536.19samples/s]
	Train Loss: 0.0755 - Val Loss: 0.1009
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 108805.44samples/s]
	Train Loss: 0.0564 - Val Loss: 0.0923
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 107750.11samples/s]
	Train Loss: 0.0443 - Val Loss: 0.0964
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 112013.41samples/s]
	Train Loss: 0.0358 - Val Loss: 0.0837
```

```
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 105916.43samples/s]
        Train Loss: 0.0279 - Val Loss: 0.0955
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 123553.77samples/s]
        Train Loss: 0.0208 - Val Loss: 0.0867
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 126197.21samples/s]
        Train Loss: 0.0154 - Val Loss: 0.0902
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 128434.51samples/s]
        Train Loss: 0.0131 - Val Loss: 0.0900
Accuracy: 0.977

Depth Variation 2: [784, 128, 128, 10]
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 98696.51samples/s]
        Train Loss: 0.2766 - Val Loss: 0.1630
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 96683.25samples/s]
        Train Loss: 0.1160 - Val Loss: 0.1396
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 92641.79samples/s]
        Train Loss: 0.0811 - Val Loss: 0.1357
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 77391.27samples/s]
        Train Loss: 0.0646 - Val Loss: 0.1078
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 86010.67samples/s]
        Train Loss: 0.0546 - Val Loss: 0.1048
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 93739.36samples/s]
        Train Loss: 0.0434 - Val Loss: 0.1148
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 97040.59samples/s]
        Train Loss: 0.0369 - Val Loss: 0.1070
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 92572.85samples/s]
        Train Loss: 0.0274 - Val Loss: 0.1346
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 87424.72samples/s]
        Train Loss: 0.0230 - Val Loss: 0.1079
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 90042.68samples/s]
        Train Loss: 0.0209 - Val Loss: 0.1180
Accuracy: 0.9759285714285715

Depth Variation 3: [784, 128, 128, 128, 10]
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 88663.29samples/s]
        Train Loss: 0.3175 - Val Loss: 0.1543
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 84177.12samples/s]
        Train Loss: 0.1325 - Val Loss: 0.1276
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 97527.49samples/s]
        Train Loss: 0.1078 - Val Loss: 0.1321
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 85573.00samples/s]
        Train Loss: 0.0986 - Val Loss: 0.1556
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 78979.59samples/s]
        Train Loss: 0.0982 - Val Loss: 0.1703
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 80664.68samples/s]
        Train Loss: 0.0933 - Val Loss: 0.1669
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 87149.26samples/s]
        Train Loss: 0.0827 - Val Loss: 0.1839
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 74821.20samples/s]
        Train Loss: 0.0895 - Val Loss: 0.1780
```

```
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 96206.73samples/s]
        Train Loss: 0.0869 - Val Loss: 0.1934
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 85781.42samples/s]
        Train Loss: 0.0847 - Val Loss: 0.2911
Accuracy: 0.9617142857142857
```

Pengaruh Fungsi Aktivasi

- Desain Eksperimen
  - Variasi activation function :
    - Linear
    - ReLU
    - Sigmoid
    - tanh
    - ELU
    - Leaky ReLU
  - Loss function : Categorical Cross-Entropy
  - Learning Rate : 0,01
  - Jumlah epoch : 10
  - Inisialisasi bobot : Uniform
- Hasil :

```
Analyzing Activation Functions

Activation Function: linear
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 123427.16samples/s]
        Train Loss: 0.4324 - Val Loss: 0.3845
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 122871.94samples/s]
        Train Loss: 0.3762 - Val Loss: 0.3673
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 116009.76samples/s]
        Train Loss: 0.3560 - Val Loss: 0.3425
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 125931.84samples/s]
        Train Loss: 0.3486 - Val Loss: 0.3571
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 127992.89samples/s]
        Train Loss: 0.3396 - Val Loss: 0.3375
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 128368.88samples/s]
        Train Loss: 0.3395 - Val Loss: 0.3459
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 128838.11samples/s]
        Train Loss: 0.3440 - Val Loss: 0.4052
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 119425.18samples/s]
        Train Loss: 0.3454 - Val Loss: 0.3862
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 127098.81samples/s]
        Train Loss: 0.3421 - Val Loss: 0.4034
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 130186.85samples/s]
        Train Loss: 0.3470 - Val Loss: 0.4304
Accuracy: 0.8908571428571429
```

```
Activation Function: relu
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 126662.97samples/s]
        Train Loss: 0.2515 - Val Loss: 0.1390
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 129184.62samples/s]
        Train Loss: 0.1072 - Val Loss: 0.1174
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 135944.77samples/s]
        Train Loss: 0.0773 - Val Loss: 0.0991
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 130822.11samples/s]
        Train Loss: 0.0589 - Val Loss: 0.1104
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 140215.59samples/s]
        Train Loss: 0.0454 - Val Loss: 0.1052
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 127363.95samples/s]
        Train Loss: 0.0387 - Val Loss: 0.0922
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 133673.56samples/s]
        Train Loss: 0.0301 - Val Loss: 0.0909
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 122468.07samples/s]
        Train Loss: 0.0231 - Val Loss: 0.0916
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 120952.99samples/s]
        Train Loss: 0.0172 - Val Loss: 0.1003
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 122342.66samples/s]
        Train Loss: 0.0131 - Val Loss: 0.0912
Accuracy: 0.9772142857142857

Activation Function: sigmoid
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 112329.14samples/s]
        Train Loss: 0.4731 - Val Loss: 0.3258
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 113182.96samples/s]
        Train Loss: 0.3129 - Val Loss: 0.3113
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 120247.51samples/s]
        Train Loss: 0.3104 - Val Loss: 0.3107
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 111707.94samples/s]
        Train Loss: 0.3118 - Val Loss: 0.3121
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 96682.09samples/s]
        Train Loss: 0.3072 - Val Loss: 0.3154
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 109568.66samples/s]
        Train Loss: 0.3089 - Val Loss: 0.3204
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 113945.74samples/s]
        Train Loss: 0.3297 - Val Loss: 0.3549
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 113583.56samples/s]
        Train Loss: 0.3631 - Val Loss: 0.3796
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 113495.03samples/s]
        Train Loss: 0.3791 - Val Loss: 0.3892
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 118677.49samples/s]
        Train Loss: 0.3932 - Val Loss: 0.4038
Accuracy: 0.8826428571428572

Activation Function: tanh
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 122255.74samples/s]
        Train Loss: 0.3218 - Val Loss: 0.2225
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 116427.94samples/s]
```

```
        Train Loss: 0.1846 - Val Loss: 0.1758
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 128638.07samples/s]
        Train Loss: 0.1429 - Val Loss: 0.1549
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 119638.69samples/s]
        Train Loss: 0.1150 - Val Loss: 0.1450
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 127704.58samples/s]
        Train Loss: 0.0956 - Val Loss: 0.1289
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 118490.16samples/s]
        Train Loss: 0.0834 - Val Loss: 0.1274
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 129523.50samples/s]
        Train Loss: 0.0732 - Val Loss: 0.1392
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 116725.51samples/s]
        Train Loss: 0.0643 - Val Loss: 0.1237
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 127182.50samples/s]
        Train Loss: 0.0546 - Val Loss: 0.1213
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 93072.39samples/s]
        Train Loss: 0.0508 - Val Loss: 0.1249
Accuracy: 0.9658571428571429

Activation Function: elu
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 100819.51samples/s]
        Train Loss: 0.3143 - Val Loss: 0.2131
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 96604.51samples/s]
        Train Loss: 0.1661 - Val Loss: 0.1646
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 93256.71samples/s]
        Train Loss: 0.1240 - Val Loss: 0.1607
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 101176.75samples/s]
        Train Loss: 0.1048 - Val Loss: 0.1615
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 104820.63samples/s]
        Train Loss: 0.0862 - Val Loss: 0.1270
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 109194.78samples/s]
        Train Loss: 0.0765 - Val Loss: 0.1097
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 95258.17samples/s]
        Train Loss: 0.0692 - Val Loss: 0.1193
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 109376.46samples/s]
        Train Loss: 0.0596 - Val Loss: 0.1100
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 101051.13samples/s]
        Train Loss: 0.0533 - Val Loss: 0.1416
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 100128.71samples/s]
        Train Loss: 0.0476 - Val Loss: 0.1192
Accuracy: 0.9692142857142857

Activation Function: leaky_relu
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 109953.55samples/s]
        Train Loss: 0.2570 - Val Loss: 0.1445
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 111717.61samples/s]
        Train Loss: 0.1078 - Val Loss: 0.1237
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 121001.28samples/s]
        Train Loss: 0.0750 - Val Loss: 0.0947
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 124816.15samples/s]
```

```
        Train Loss: 0.0578 - Val Loss: 0.0963
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 116836.35samples/s]
        Train Loss: 0.0471 - Val Loss: 0.0963
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 113968.02samples/s]
        Train Loss: 0.0374 - Val Loss: 0.1004
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 110886.88samples/s]
        Train Loss: 0.0290 - Val Loss: 0.0937
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 122513.36samples/s]
        Train Loss: 0.0220 - Val Loss: 0.0953
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 121681.22samples/s]
        Train Loss: 0.0170 - Val Loss: 0.0997
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 109748.21samples/s]
        Train Loss: 0.0132 - Val Loss: 0.0984
Accuracy: 0.9767142857142858
```

Pengaruh Learning Rate

- Desain Eksperimen
  - Activation function : ReLU untuk selain fungsi aktivasi terakhir dan menggunakan softmax untuk fungsi aktivasi terakhir
  - Loss function : Categorical Cross-Entropy
  - Variasi Learning Rate : 0,001; 0,01; 0,1
  - Jumlah epoch : 10
  - Inisialisasi bobot : Uniform
- Hasil :

```
Analyzing Learning Rates

Learning Rate: 0.001
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 110033.50samples/s]
        Train Loss: 0.5563 - Val Loss: 0.3177
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 113846.99samples/s]
        Train Loss: 0.2780 - Val Loss: 0.2563
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 117488.39samples/s]
        Train Loss: 0.2234 - Val Loss: 0.2203
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 114324.93samples/s]
        Train Loss: 0.1882 - Val Loss: 0.1894
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 115129.97samples/s]
        Train Loss: 0.1625 - Val Loss: 0.1707
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 126326.71samples/s]
        Train Loss: 0.1435 - Val Loss: 0.1586
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 136146.02samples/s]
        Train Loss: 0.1278 - Val Loss: 0.1446
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 109004.79samples/s]
        Train Loss: 0.1154 - Val Loss: 0.1386
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 129551.72samples/s]
        Train Loss: 0.1048 - Val Loss: 0.1310
```

```
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 129297.69samples/s]
        Train Loss: 0.0964 - Val Loss: 0.1239
Accuracy: 0.9642142857142857

Learning Rate: 0.01
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 128899.27samples/s]
        Train Loss: 0.2604 - Val Loss: 0.1490
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 132278.90samples/s]
        Train Loss: 0.1091 - Val Loss: 0.1026
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 127468.52samples/s]
        Train Loss: 0.0786 - Val Loss: 0.1203
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 129193.36samples/s]
        Train Loss: 0.0603 - Val Loss: 0.1077
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 128282.01samples/s]
        Train Loss: 0.0467 - Val Loss: 0.1025
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 135645.42samples/s]
        Train Loss: 0.0384 - Val Loss: 0.0961
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 129363.78samples/s]
        Train Loss: 0.0302 - Val Loss: 0.0919
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 129089.34samples/s]
        Train Loss: 0.0233 - Val Loss: 0.0941
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 129860.21samples/s]
        Train Loss: 0.0175 - Val Loss: 0.1078
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 125625.92samples/s]
        Train Loss: 0.0134 - Val Loss: 0.0945
Accuracy: 0.9757142857142858

Learning Rate: 0.1
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 127990.58samples/s]
        Train Loss: 0.8166 - Val Loss: 0.6134
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 130007.92samples/s]
        Train Loss: 0.5262 - Val Loss: 0.6100
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 130327.42samples/s]
        Train Loss: 0.4659 - Val Loss: 0.5608
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 137403.77samples/s]
        Train Loss: 0.4364 - Val Loss: 0.4741
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 131788.53samples/s]
        Train Loss: 0.4193 - Val Loss: 0.4381
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 123592.25samples/s]
        Train Loss: 0.3827 - Val Loss: 0.4305
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 128228.57samples/s]
        Train Loss: 0.3756 - Val Loss: 0.4454
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 126520.78samples/s]
        Train Loss: 0.3632 - Val Loss: 0.5007
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 111293.71samples/s]
        Train Loss: 0.3420 - Val Loss: 0.4803
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 83600.55samples/s]
        Train Loss: 0.3425 - Val Loss: 0.5103
Accuracy: 0.9168571428571428
```

Pengaruh Inisialisasi Bobot

- Desain Eksperimen
  - Activation function : ReLU untuk selain fungsi aktivasi terakhir dan menggunakan softmax untuk fungsi aktivasi terakhir
  - Loss function : Categorical Cross-Entropy
  - Variasi Learning Rate : 0,01
  - Jumlah epoch : 10
  - Variasi inisialisasi bobot :
    - Zero
    - Uniform
    - Normal
    - He
    - Xavier
- Hasil :

```
Analyzing Weight Initialization

Initialization Method: zero
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 99015.01samples/s]
        Train Loss: 2.3021 - Val Loss: 2.3015
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 100869.08samples/s]
        Train Loss: 2.3022 - Val Loss: 2.3012
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 99726.45samples/s]
        Train Loss: 2.3023 - Val Loss: 2.3017
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 118494.46samples/s]
        Train Loss: 2.3020 - Val Loss: 2.3013
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 72570.61samples/s]
        Train Loss: 2.3023 - Val Loss: 2.3020
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 129905.10samples/s]
        Train Loss: 2.3023 - Val Loss: 2.3029
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 120694.87samples/s]
        Train Loss: 2.3023 - Val Loss: 2.3012
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 107361.96samples/s]
        Train Loss: 2.3021 - Val Loss: 2.3019
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 117227.86samples/s]
        Train Loss: 2.3023 - Val Loss: 2.3010
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 118981.62samples/s]
        Train Loss: 2.3022 - Val Loss: 2.3018
Accuracy: 0.11428571428571428

Initialization Method: uniform
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 117797.02samples/s]
        Train Loss: 0.2575 - Val Loss: 0.1535
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 109344.02samples/s]
        Train Loss: 0.1092 - Val Loss: 0.1262
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 101127.70samples/s]
        Train Loss: 0.0766 - Val Loss: 0.1113
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 130647.69samples/s]
```

Train Loss: 0.0608 - Val Loss: 0.0934
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 120270.54samples/s]
Train Loss: 0.0463 - Val Loss: 0.0939
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 109755.95samples/s]
Train Loss: 0.0365 - Val Loss: 0.0959
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 125493.36samples/s]
Train Loss: 0.0288 - Val Loss: 0.1021
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 127627.20samples/s
Train Loss: 0.0222 - Val Loss: 0.0985
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 132206.08samples/s]
Train Loss: 0.0180 - Val Loss: 0.0925
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 123751.47samples/s]
Train Loss: 0.0126 - Val Loss: 0.0927
Accuracy: 0.9755714285714285

Initialization Method: normal
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 122405.97samples/s]
Train Loss: 0.2890 - Val Loss: 0.1463
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 128714.06samples/s]
Train Loss: 0.1101 - Val Loss: 0.1419
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 91120.64samples/s]
Train Loss: 0.0780 - Val Loss: 0.0966
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 124542.42samples/s]
Train Loss: 0.0589 - Val Loss: 0.1063
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 126397.14samples/s]
Train Loss: 0.0462 - Val Loss: 0.0995
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 136484.22samples/s]
Train Loss: 0.0371 - Val Loss: 0.0842
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 123244.85samples/s]
Train Loss: 0.0307 - Val Loss: 0.0893
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 121730.16samples/s]
Train Loss: 0.0222 - Val Loss: 0.0852
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 100196.79samples/s]
Train Loss: 0.0173 - Val Loss: 0.0885
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 94353.26samples/s]
Train Loss: 0.0148 - Val Loss: 0.0860
Accuracy: 0.9782142857142857

Initialization Method: he
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 100662.53samples/s]
Train Loss: 0.2445 - Val Loss: 0.1566
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 126434.83samples/s]
Train Loss: 0.1104 - Val Loss: 0.1208
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 91255.77samples/s]
Train Loss: 0.0769 - Val Loss: 0.0980
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 135206.59samples/s]
Train Loss: 0.0604 - Val Loss: 0.0957
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 132229.90samples/s]
Train Loss: 0.0464 - Val Loss: 0.1041
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 127607.10samples/s]

```
        Train Loss: 0.0366 - Val Loss: 0.1048
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 136343.28samples/s]
        Train Loss: 0.0278 - Val Loss: 0.1035
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 130212.47samples/s]
        Train Loss: 0.0217 - Val Loss: 0.0942
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 141132.24samples/s]
        Train Loss: 0.0182 - Val Loss: 0.0927
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 138041.88samples/s]
        Train Loss: 0.0128 - Val Loss: 0.0951
Accuracy: 0.9778571428571429

Initialization Method: xavier
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 150096.29samples/s]
        Train Loss: 0.2411 - Val Loss: 0.1435
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 135950.04samples/s]
        Train Loss: 0.1042 - Val Loss: 0.1141
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 137985.76samples/s]
        Train Loss: 0.0746 - Val Loss: 0.1021
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 137227.00samples/s]
        Train Loss: 0.0586 - Val Loss: 0.0912
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 122171.80samples/s]
        Train Loss: 0.0458 - Val Loss: 0.0964
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 139105.33samples/s]
        Train Loss: 0.0372 - Val Loss: 0.0912
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 139968.51samples/s]
        Train Loss: 0.0287 - Val Loss: 0.1033
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 130020.95samples/s]
        Train Loss: 0.0221 - Val Loss: 0.0930
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 137132.14samples/s]
        Train Loss: 0.0164 - Val Loss: 0.0977
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 135230.72samples/s]
        Train Loss: 0.0138 - Val Loss: 0.0992
Accuracy: 0.9747857142857143
```

Pengaruh Regularisasi

- Desain Eksperimen
    - Activation function : ReLU untuk selain fungsi aktivasi terakhir dan menggunakan softmax untuk fungsi aktivasi terakhir
    - Loss function : Categorical Cross-Entropy
    - Variasi Learning Rate : 0,01
    - Jumlah epoch : 10
    - Inisialisasi bobot : Uniform
    - Variasi percobaan :
        - Tanpa regularisasi
        - Dengan regularisasi L1
        - Dengan regularisasi L2
- Hasil :

```
Analyzing Regularization

Tanpa Regularisasi
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 115232.26samples/s]
        Train Loss: 0.2570 - Val Loss: 0.1572
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 133920.58samples/s]
        Train Loss: 0.1063 - Val Loss: 0.1082
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 121811.22samples/s]
        Train Loss: 0.0749 - Val Loss: 0.1005
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 127099.16samples/s]
        Train Loss: 0.0591 - Val Loss: 0.0921
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 126981.32samples/s]
        Train Loss: 0.0456 - Val Loss: 0.0868
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 140743.14samples/s]
        Train Loss: 0.0351 - Val Loss: 0.0912
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 121792.08samples/s]
        Train Loss: 0.0285 - Val Loss: 0.0903
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 119103.25samples/s]
        Train Loss: 0.0227 - Val Loss: 0.0879
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 122922.68samples/s]
        Train Loss: 0.0172 - Val Loss: 0.0920
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 136155.65samples/s]
        Train Loss: 0.0131 - Val Loss: 0.0898
Accuracy: 0.976

Dengan Regularisasi L1
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 57100.19samples/s]
        Train Loss: 5.5776 - Val Loss: 5.4266
Epoch 2/10: 100%|██████████| 56000/56000 [00:01<00:00, 55463.82samples/s]
        Train Loss: 5.2918 - Val Loss: 5.1749
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 58718.30samples/s]
        Train Loss: 5.0127 - Val Loss: 4.9159
Epoch 4/10: 100%|██████████| 56000/56000 [00:01<00:00, 49856.53samples/s]
        Train Loss: 4.7594 - Val Loss: 4.6704
Epoch 5/10: 100%|██████████| 56000/56000 [00:01<00:00, 49785.79samples/s]
        Train Loss: 4.5185 - Val Loss: 4.4573
Epoch 6/10: 100%|██████████| 56000/56000 [00:01<00:00, 55257.23samples/s]
        Train Loss: 4.3021 - Val Loss: 4.2482
Epoch 7/10: 100%|██████████| 56000/56000 [00:01<00:00, 54329.16samples/s]
        Train Loss: 4.1155 - Val Loss: 4.0890
Epoch 8/10: 100%|██████████| 56000/56000 [00:01<00:00, 54082.82samples/s]
        Train Loss: 3.9558 - Val Loss: 3.9627
Epoch 9/10: 100%|██████████| 56000/56000 [00:01<00:00, 55584.89samples/s]
        Train Loss: 3.8213 - Val Loss: 3.8330
Epoch 10/10: 100%|██████████| 56000/56000 [00:01<00:00, 54607.10samples/s]
        Train Loss: 3.7136 - Val Loss: 3.7302
Accuracy: 0.9772857142857143

Dengan Regularisasi L2
```

```
Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 114893.89samples/s]
        Train Loss: 0.5014 - Val Loss: 0.4684
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 82804.59samples/s]
        Train Loss: 0.4315 - Val Loss: 0.4865
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 112739.82samples/s]
        Train Loss: 0.4582 - Val Loss: 0.5151
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 107621.50samples/s]
        Train Loss: 0.4916 - Val Loss: 0.5600
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 113781.36samples/s]
        Train Loss: 0.5206 - Val Loss: 0.5960
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 112609.40samples/s]
        Train Loss: 0.5488 - Val Loss: 0.6268
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 112105.84samples/s]
        Train Loss: 0.5776 - Val Loss: 0.6566
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 115144.13samples/s]
        Train Loss: 0.6001 - Val Loss: 0.6771
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 116086.48samples/s]
        Train Loss: 0.6202 - Val Loss: 0.7014
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 117900.44samples/s]
        Train Loss: 0.6347 - Val Loss: 0.7232
Accuracy: 0.9751428571428571
```

Perbandingan dengan Library sklearn

- Desain Eksperimen
  - Activation function : ReLU untuk selain fungsi aktivasi terakhir dan menggunakan softmax untuk fungsi aktivasi terakhir
  - Loss function : Categorical Cross-Entropy
  - Learning Rate : 0,01
  - Jumlah epoch : 10
  - Inisialisasi bobot : Uniform
  - Tanpa regularisasi
- Hasil :

```
Comparing with Sklearn MLP

Epoch 1/10: 100%|██████████| 56000/56000 [00:00<00:00, 91754.09samples/s]
        Train Loss: 0.2577 - Val Loss: 0.1419
Epoch 2/10: 100%|██████████| 56000/56000 [00:00<00:00, 92417.03samples/s]
        Train Loss: 0.1116 - Val Loss: 0.1136
Epoch 3/10: 100%|██████████| 56000/56000 [00:00<00:00, 83203.37samples/s]
        Train Loss: 0.0809 - Val Loss: 0.1162
Epoch 4/10: 100%|██████████| 56000/56000 [00:00<00:00, 84904.42samples/s]
        Train Loss: 0.0622 - Val Loss: 0.1031
Epoch 5/10: 100%|██████████| 56000/56000 [00:00<00:00, 99172.07samples/s]
        Train Loss: 0.0468 - Val Loss: 0.0921
Epoch 6/10: 100%|██████████| 56000/56000 [00:00<00:00, 92374.00samples/s]
        Train Loss: 0.0367 - Val Loss: 0.0899
```

```
Epoch 7/10: 100%|██████████| 56000/56000 [00:00<00:00, 93310.17samples/s]
        Train Loss: 0.0284 - Val Loss: 0.0976
Epoch 8/10: 100%|██████████| 56000/56000 [00:00<00:00, 95033.28samples/s]
        Train Loss: 0.0241 - Val Loss: 0.0868
Epoch 9/10: 100%|██████████| 56000/56000 [00:00<00:00, 99312.80samples/s]
        Train Loss: 0.0184 - Val Loss: 0.0908
Epoch 10/10: 100%|██████████| 56000/56000 [00:00<00:00, 118654.41samples/s]
        Train Loss: 0.0142 - Val Loss: 0.0910
Custom FFNN Accuracy: 0.9757857142857143
Sklearn MLP Accuracy: 0.9715714285714285
```

KESIMPULAN DAN SARAN

Kesimpulan

1. Implementasi FFNN dari awal memberikan pemahaman mendalam tentang mekanisme *neural network*, mulai dari proses *forward propagation* hingga *backward propagation.*
2. Hyperparameter seperti learning rate, fungsi aktivasi, dan lainnya sangat mempengaruhi kinerja dan akurasi model.
3. Kombinasi hyperparameter yang tepat dapat memberikan akurasi yang lebih baik.
4. Setiap metode dan fungsi memiliki kelebihan dan kekurangannya masing-masing, tidak ada yang bisa dinyatakan sempurna.
5. Untuk mendapatkan hasil maksimal, pemilihan metode dan fungsi harus disesuaikan dengan karakter spesifik dari masalah yang dihadapi.

Saran

1. Eksplorasi dan eksperimen dengan fungsi aktivasi lain di luar spesifikasi soal.
2. Eksplorasi dan eksperimen dengan metode inisialisasi bobot lain di luar spesifikasi soal.
3. Eksplorasi dan eksperimen dengan *loss function* lain di luar spesifikasi soal.
4. Melakukan lebih banyak percobaan dengan berbagai angka pada parameter, baik itu jumlah epoch, learning rate, dan lainnya.

PEMBAGIAN TUGAS

13522001	Visualisasi (struktur jaringan, distribusi bobot, distribusi gradien). Dokumentasi dan perbandingan dengan scikit-learn.
13522105	Implementasi FFNN (forward pass, backward pass, update weights). Eksperimen parameter (jumlah layer, fungsi aktivasi, learning rate).

REFERENSI

The spelled-out intro to neural networks and backpropagation: building micrograd
https://www.jasonosajima.com/forwardprop
https://www.jasonosajima.com/backprop
https://numpy.org/doc/2.2/
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
https://math.libretexts.org/Bookshelves/Calculus/Calculus_(OpenStax)/14%3A_Differentiation_of_Functions_of_Several_Variables/14.05%3A_The_Chain_Rule_for_Multivariable_Functions
https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/
https://douglasorr.github.io/2021-11-autodiff/article.html
https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2022/tutorials/tut01.pdf

GITHUB:
https://github.com/fabianradenta/Tubes1_ML