LAPORAN TUGAS BESAR
IF3270 - PEMBELAJARAN MESIN
Feed Forward Neural Network

Disusun Oleh

M Zulfiansyah Bayu Pratama	13521028
Fahrian Afdholi	13521031
Brian Kheng	13521049

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024/2025

BAB I
PENDAHULUAN

1.1 Latar Belakang

Feedforward Neural Network (FFNN) adalah dasar pembelajaran mesin yang penting untuk memahami pola data kompleks. Tugas ini bertujuan agar mahasiswa mampu mengimplementasikan FFNN secara manual guna mendalami konsep forward propagation, backward propagation, dan analisis hyperparameter.

1.2 Tujuan

1. Memahami konsep dasar dan komponen dari FFNN
2. Melakukan eksperimen dengan berbagai hyperparameter
3. Melakukan perbandingan kinerja dengan library sklearn

BAB II
PEMBAHASAN

2.1 Penjelasan Implementasi

Pada bagian ini, kami melakukan implementasi kelas FFNN dengan kode yang dapat dilihat sebagai berikut:

2.1.1 FFNN

```
def __init__(self, layer_sizes, activation_functions, loss_function,
weight_init_method='random_uniform',
                 weight_init_params=None):
        """
        Initialize the FFNN model.

        Parameters:
        -----------
        layer_sizes : list
            Number of neurons in each layer (including input and output
layers)
        activation_functions : list
            Activation functions for each layer (except input layer)
        loss_function : str
            Loss function to use for training
        weight_init_method : str
            Method for weight initialization
        weight_init_params : dict
            Parameters for weight initialization
        """
        self.layer_sizes = layer_sizes
        self.num_layers = len(layer_sizes)

        # Validation
        if len(activation_functions) != self.num_layers - 1:
            raise ValueError("Number of activation functions must match
number of layers - 1")

        self.activation_functions = activation_functions
        self.loss_function = loss_function

        # Default weight initialization parameters
        if weight_init_params is None:
            if weight_init_method == 'random_uniform':
                weight_init_params = {'lower_bound': -0.5, 'upper_bound':
0.5, 'seed': 42}
            elif weight_init_method == 'random_normal':
```

```
                weight_init_params = {'mean': 0, 'variance': 0.1, 'seed': 42}

        # Initialize weights and biases
        self.weights = []
        self.biases = []
        self.weight_gradients = []
        self.bias_gradients = []
        self._initialize_weights(weight_init_method, weight_init_params)

        # For storing intermediate values during forward/backward pass
        self.z_values = []  # Pre-activation values
        self.a_values = []  # Post-activation values
```

Saat melakukan inisiasi kelas FFNN, terdapat beberapa parameter yang harus dimasukkan dalam kelas tersebut. Berikut adalah penjelasan masing-masing dari parameter yang ada:

- `layer_sizes`: sebuah array yang memiliki elemen berupa jumlah neuron pada masing masing layer termasuk input dan output layer itu sendiri
- `activation_functions`: sebuah array yang memiliki elemen bertipe string yang memiliki value tipe aktivasi pada masing masing hidden layer
- `loss_function`: sebuah string yang memiliki value berupa tipe loss function yang akan digunakan pada kelas FFNN
- *`weight_init_method`*: sebuah string yang memiliki value berupa tipe weight yang akan digunakan dalam kelas FFNN
- `weight_init_params`: sebuah objek yang menyimpan parameter yang diperlukan dalam tipe weight tertentu

2.1.2 Metode *_initialize_weights*

```
def _activate(self, z, activation_function):
        """
        Apply activation function.

        Parameters:
        -----------
        z : numpy.ndarray
            Pre-activation values
        activation_function : str
            Name of the activation function
```

```
        Returns:
        --------
        numpy.ndarray
            Activated values
        """
        match activation_function:
            case 'linear':
                return z
            case 'relu':
                return np.maximum(0, z)
            case 'sigmoid':
                z_safe = np.clip(z, -500, 500)  # Prevent overflow
                return 1 / (1 + np.exp(-z_safe))
            case 'tanh':
                return np.tanh(z)
            case 'softmax':
                shifted_z = z - np.max(z, axis=0, keepdims=True) # Prevent
overflow
                exp_z = np.exp(shifted_z)
                return exp_z / np.sum(exp_z, axis=0, keepdims=True)
            case _:
                raise ValueError(f"Unsupported activation function:
{activation_function}")
```

Pada metode ini dilakukan inisialisasi pada tipe *weight* masukan dengan memberikan *parameter method* dan *param*. Tipe dari *weight* yang ada sendiri terdapat 3 macam yaitu *zero*, *random uniform*, dan *random normal*. *Weight* dan *bias* yang diinisialisasi sendiri dimasukkan ke dalam atribut pada kelas FFNN yaitu `weight`, `weight_gradient`, dan `bias` yang memiliki elemen berupa *array*.

2.1.3 Metode *_activate*

```
def _activate(self, z, activation_function):
        """
        Apply activation function.

        Parameters:
        -----------
        z : numpy.ndarray
            Pre-activation values
        activation_function : str
            Name of the activation function

        Returns:
        --------
        numpy.ndarray
```

```
            Activated values
        """
        match activation_function:
            case 'linear':
                return z
            case 'relu':
                return np.maximum(0, z)
            case 'sigmoid':
                z_safe = np.clip(z, -500, 500)  # Prevent overflow
                return 1 / (1 + np.exp(-z_safe))
            case 'tanh':
                return np.tanh(z)
            case 'softmax':
                shifted_z = z - np.max(z, axis=0, keepdims=True) # Prevent
overflow
                exp_z = np.exp(shifted_z)
                return exp_z / np.sum(exp_z, axis=0, keepdims=True)
            case _:
                raise ValueError(f"Unsupported activation function:
{activation_function}")
```

Pada metode ini digunakan untuk melakukan aktivasi pada masing masing node. Pada method tersebut terdapat masukan berupa z yang dimana *array* yang berisi *pre-activation values* dan `activation_function` yang merupakan tipe dari aktifasi yang dipilih. Terdapat pilihan dalam aktivasi fungsi yaitu linear, relu, sigmoid, tanh, dan softmax dengan masing masing formula sebagai berikut:

$$Linear(x) = x$$

$$ReLU(x) = \max(0, x)$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$softmax(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

2.1.4 Metode *_activate_derivative*

```
def _activate_derivative(self, z, activation_function):
        """
        Compute the derivative of the activation function.

        Parameters:
        -----------
        z : numpy.ndarray
            Pre-activation values
        activation_function : str
            Name of the activation function

        Returns:
        --------
        numpy.ndarray
            Derivatives of the activation function
        """
        match activation_function:
            case 'linear':
                return np.ones_like(z)
            case 'relu':
                return (z > 0).astype(float)
            case 'sigmoid':
                activated_values = self._activate(z, 'sigmoid')
                return activated_values * (1 - activated_values)
            case 'tanh':
                return 1 - np.tanh(z)**2
            case 'softmax':
                activated_values = self._activate(z, 'softmax')
                return activated_values
            case _:
                raise ValueError(f"Unsupported activation function:
{activation_function}")
```

Pada *method* ini akan melakukan penurunan pada output dari `_activate` pada masing masing node yang dimana memiliki parameter yang sama dengan `_activate` yaitu `z` dan `activation_function`.

2.1.5 Metode *_compute_loss*

```
def _compute_loss(self, y_true, y_pred):
        """
        Compute the loss between true and predicted values.

        Parameters:
        -----------
        y_true : numpy.ndarray
            True values
        y_pred : numpy.ndarray
            Predicted values

        Returns:
        --------
        float
            Loss value
        """
        n_samples = y_true.shape[1]

        match self.loss_function:
            case 'mse':
                return np.mean(np.sum((y_true - y_pred)**2, axis=0)) / 2
            case 'binary_crossentropy':
                # Clip to avoid log(0)
                y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
                return -np.sum(y_true * np.log(y_pred) + (1 - y_true) *
np.log(1 - y_pred)) / n_samples
            case 'categorical_crossentropy':
                # Clip to avoid log(0)
                y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
                return -np.sum(y_true * np.log(y_pred)) / n_samples
            case _:
                raise ValueError(f"Unsupported loss function:
{self.loss_function}")
```

Method ini berfungsi untuk melakukan perhitungan *error* dengan menggunakan *prediction value* dan *actual value*. Terdapat beberapa macam *loss function* yang diimplementasi pada method ini diantaranya:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$\mathcal{L}_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right)$$

$$\mathcal{L}_{CCE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{C}\left(y_{ij} \log \hat{y}_{ij}\right)$$

2.1.6 Metode *_compute_loss_derivative*

```
def _compute_loss_derivative(self, y_true, y_pred):
        """
        Compute the derivative of the loss function.

        Parameters:
        -----------
        y_true : numpy.ndarray
            True values
        y_pred : numpy.ndarray
            Predicted values

        Returns:
        --------
        numpy.ndarray
            Derivative of the loss function
        """
        n_samples = y_true.shape[1]

        match self.loss_function:
            case 'mse':
                return (y_pred - y_true) / n_samples
            case 'binary_crossentropy':
                # Clip to avoid division by 0
                y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
                return -(y_true / y_pred - (1 - y_true) / (1 - y_pred)) /
n_samples
            case 'categorical_crossentropy':
                # For softmax activation function
                if self.activation_functions[-1] == 'softmax':
                    return (y_pred - y_true) / n_samples

                # For other activation functions
                y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
                return -y_true / y_pred / n_samples
            case _:
                raise ValueError(f"Unsupported loss function:
{self.loss_function}")
```

Pada method ini berfungsi untuk melakukan kalkulasi *loss function derivative* yang memiliki parameter aktual value dan prediksi value. Terdapat beberapa macam yang fungsi *derivative* yang diimplementasi yaitu diantaranya:

$$\frac{\partial \mathcal{L}_{MSE}}{\partial W} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)\frac{\partial \hat{y}_i}{\partial W}$$

$$\frac{\partial \mathcal{L}_{BCE}}{\partial W} = -\frac{1}{n}\sum_{i=1}^{n}\frac{\hat{y}_i - y_i}{\hat{y}_i(1-\hat{y}_i)}\frac{\partial \hat{y}_i}{\partial W}$$

$$\frac{\partial \mathcal{L}_{CCE}}{\partial W} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{C} y_{ij}\frac{\partial \hat{y}_{ij}}{\partial W}$$

2.1.7 Metode *forward*

```
def forward(self, X):
        """
        Perform forward propagation.

        Parameters:
        -----------
        X : numpy.ndarray
            Input data (shape: features x samples)

        Returns:
        --------
        numpy.ndarray
            Predictions from the output layer
        """
        # Reset stored intermediate values
        self.z_values = []
        self.a_values = []

        # First activation is the input
        a = X
        self.a_values.append(a)

        # Forward propagation through each layer
        for i in range(self.num_layers - 1):
            z = np.dot(self.weights[i], a) + self.biases[i]
            a = self._activate(z, self.activation_functions[i])

            self.z_values.append(z)
            self.a_values.append(a)

        return a  # Output of the last layer
```

Pada metode ini dilakukan kalkulasi untuk melakukan prediksi pada *output layer*. Cara kerjanya sendiri seperti kalkulasi *forward propagation* pada umumnya yang melakukan perkalian *weight* dengan layer dan ditambahkan dengan bias. Untuk mendapatkan output perlu dilakukan aktivasi dengan memanggil *activate method* dan akan mengembalikan *output* pada layer tersebut. Kalkulasi terus dilakukan

sampai mendapatkan *output* dari output layer yang akan digunakan sebagai return pada method ini.

Forward propagation sendiri berfungsi untuk memberikan perdiksi pada dengan melakukan perhitungan *output* masing masing layer sampai dengan *output layer*. Selain itu forward propagation juga memiliki fungsi untuk menyediakan data untuk perhitungan *loss function* dan *back propagation*.

2.1.8 Metode *backward*

```
def backward(self, X, y):
        """
        Perform backward propagation to compute gradients.

        Parameters:
        -----------
        X : numpy.ndarray
            Input data (shape: features x samples)
        y : numpy.ndarray
            Target values (shape: outputs x samples)
        """
        n_samples = X.shape[1]

        # Forward pass
        y_pred = self.forward(X)

        # Compute the initial error (delta) from the loss function
        delta = self._compute_loss_derivative(y, y_pred)

        # Handle the special case for softmax + categorical cross-entropy
        if self.activation_functions[-1] == 'softmax' and self.loss_function
== 'categorical_crossentropy':
            pass
        else:
            # For other combinations, multiply by the activation derivative
            delta = delta * self._activate_derivative(self.z_values[-1],

self.activation_functions[-1])

        # Backpropagate through each layer
        for i in range(self.num_layers - 2, -1, -1):
            # Compute gradients for this layer
            self.weight_gradients[i] = np.dot(delta, self.a_values[i].T) /
n_samples
            self.bias_gradients[i] = np.sum(delta, axis=1, keepdims=True) /
n_samples

            # Compute delta for the previous layer (if not the input layer)
```

```
            if i > 0:
                delta = np.dot(self.weights[i].T, delta)
                delta = delta * self._activate_derivative(self.z_values[i-1],

self.activation_functions[i-1])
```

Pada *method* ini dilakukan *backward propagation*, yaitu proses menghitung gradien dari *loss function* terhadap weight dan bias jaringan, yang nantinya akan digunakan untuk memperbarui selama proses *training*. Metode ini memiliki dua parameter, diantaranya adalah

- `X` : *Input data* dalam bentuk format `(jumlah_fitur, jumlah_data)`.
- `y` : Target/label dalam bentuk `(jumlah_output, jumlah_data)`.

Sebelum menghitung gradien, metode ini menjalankan *forward propagation* untuk menghitung prediksi `y_pred` dan menyimpan semua nilai `z` dan `a` antar *layer*. Setelah itu dilakukan perhitungan turunan *loss* terhadap *output* prediksi yang disimpan dalam variabel `delta`. Untuk kebanyakan kombinasi fungsi aktivasi dan *loss*, delta masih perlu dikalikan dengan turunan aktivasi. Namun untuk **kombinasi *softmax* + *categorical cross entropy*,** delta sudah sesuai dan tidak perlu dikalikan lagi. Karena rumus turunan gabungannya lebih sederhana.

Setelah dilakukan pengecekan fungsi aktivasi, akan dilakukan *backpropagation* ke layer sebelumnya dari layer terakhir ke layer pertama. Untuk tiap layer akan dilakukan perhitungan weight gradien, bias gradien, dan delta baru (jika bukan *input layer*). Setelah semua proses *backpropagation* selesai, maka hasil weight dan bias gradien akan disimpan.

2.1.9 Metode *update_weight*

```
def update_weights(self, learning_rate):
        """
        Update weights and biases using gradient descent.

        Parameters:
        -----------
        learning_rate : float
            Learning rate for gradient descent
```

```
        """
        for i in range(len(self.weights)):
            self.weights[i] -= learning_rate * self.weight_gradients[i]
            self.biases[i] -= learning_rate * self.bias_gradients[i]
```

Pada metode ini, dilakukan pembaruan weight dan bias pada setiap layer jaringan menggunakan metode *gradient descent*. Metode ini dijalankan setelah proses *backward* yang menghasilkan gradien dari weight dan bias. Metode ini memiliki satu parameter yaitu `learning rate`, yang merepresentasikan nilai pengatur besar langkah pembaruan weight dari bias. Semakin kecil *learning rate*, maka perubahan weight akan semakin kecil. Cara kerja dari metode ini adalah melakukan operasi pengurangan nilai weight dan bias dengan nilai gradiennya untuk setiap layer.

2.1.10 Metode *train*

```
def train(self, X_train, y_train, X_val=None, y_val=None, batch_size=32,
learning_rate=0.01,
              epochs=100, verbose=1):
        """
        Train the FFNN model.

        Parameters:
        -----------
        X_train : numpy.ndarray
            Training input data (shape: features x samples)
        y_train : numpy.ndarray
            Training target values (shape: outputs x samples)
        X_val : numpy.ndarray, optional
            Validation input data
        y_val : numpy.ndarray, optional
            Validation target values
        batch_size : int
            Size of batches for mini-batch gradient descent
        learning_rate : float
            Learning rate for gradient descent
        epochs : int
            Number of training epochs
        verbose : int
            Verbosity level (0: no output, 1: progress bar)

        Returns:
        --------
        dict
            Training history (loss and validation loss)
```

```
        """
        n_samples = X_train.shape[1]
        n_batches = int(np.ceil(n_samples / batch_size))

        history = {
            'train_loss': [],
            'val_loss': []
        }

        for epoch in range(epochs):
            # Shuffle data
            indices = np.random.permutation(n_samples)
            X_shuffled = X_train[:, indices]
            y_shuffled = y_train[:, indices]

            epoch_loss = 0

            # Mini-batch gradient descent
            for b in range(n_batches):
                start_idx = b * batch_size
                end_idx = min((b + 1) * batch_size, n_samples)

                X_batch = X_shuffled[:, start_idx:end_idx]
                y_batch = y_shuffled[:, start_idx:end_idx]

                # Forward pass
                y_pred = self.forward(X_batch)
                batch_loss = self._compute_loss(y_batch, y_pred)
                epoch_loss += batch_loss * (end_idx - start_idx) / n_samples

                # Backward pass
                self.backward(X_batch, y_batch)

                # Update weights
                self.update_weights(learning_rate)

            # Record training loss
            history['train_loss'].append(epoch_loss)

            # Compute validation loss if validation data is provided
            if X_val is not None and y_val is not None:
                y_val_pred = self.forward(X_val)
                val_loss = self._compute_loss(y_val, y_val_pred)
                history['val_loss'].append(val_loss)

            # Print progress
            if verbose == 1:
                if X_val is not None and y_val is not None:
                    print(f"Epoch {epoch+1}/{epochs} - loss: {epoch_loss:.4f}
- val_loss: {val_loss:.4f}")
                else:
                    print(f"Epoch {epoch+1}/{epochs} - loss:
```

```
{epoch_loss:.4f}")

        return history
```

Pada metode ini akan dilakukan training model menggunakan *dataset training*, dengan opsi *mini-batch training* dan *validasi*. Proses ini melibatkan *forward propagation*, *backward propagation*, *update* weight, yang diulang selama beberapa *epoch*. Terdapat beberapa parameter pada metode ini, diantaranya adalah :

- `X_train`: Input fitur (berbentuk `features × samples`)
- `y_train`: Label target dari *input training* `(output × samples)`
- `X_val`: Data validasi (fitur)
- `y_val`: Target validasi
- `batch_size`: Ukuran mini-batch yang digunakan saat training
- `learning_rate`: Nilai *learning rate* untuk `update_weights`
- `epochs`: Jumlah iterasi (ulang) *training* terhadap seluruh *dataset*
- `verbose`: Level *output*, jika 1 maka cetak progres setiap *epoch*

Untuk cara kerjanya, awalnya data diacak agar tidak ada urutan tetap dalam *batch*. Lalu data dibagi ke dalam batch kecil. Untuk setiap batch, akan dilakukan beberapa langkah sebagai berikut,

- Data diacak agar tidak ada urutan tetap dalam batch.
- Data dibagi ke dalam batch kecil.
- Untuk setiap batch:
    - Lakukan *forward* untuk mendapatkan output prediksi
    - Hitung loss
    - Lakukan *backward* untuk menghitung gradien
    - Perbarui weight dengan `update_weights()`
- Simpan nilai *loss* per epoch, juga validasi *loss* jika tersedia.
- Cetak log jika `verbose=1`.

2.1.11 Metode *predict*

```
def predict(self, X):
        """
        Make predictions for input data.

        Parameters:
        -----------
        X : numpy.ndarray
            Input data (shape: features x samples)

        Returns:
        --------
        numpy.ndarray
            Predictions
        """
        return self.forward(X)
```

Pada metode ini akan mengembalikan *output* prediksi dari jaringan terhadap data input `X` yang dihasilkan melalui metode `forward()`.

2.1.12 Metode *visualize_model*

```
def visualize_model(self):
        """
        Visualize the model structure with weights and gradients as a graph.
        """
        G = nx.DiGraph()

        # Add nodes for each layer
        layer_nodes = []
        for l in range(self.num_layers):
            layer_nodes.append([])
            for n in range(self.layer_sizes[l]):
                node_id = f"L{l}N{n}"
                G.add_node(node_id, layer=l, neuron=n)
                layer_nodes[l].append(node_id)

        # Add edges with weights and gradients
        for l in range(self.num_layers - 1):
            for i in range(self.layer_sizes[l]):
                for j in range(self.layer_sizes[l+1]):
                    weight = self.weights[l][j, i]
                    gradient = self.weight_gradients[l][j, i]
                    G.add_edge(
                        layer_nodes[l][i],
                        layer_nodes[l+1][j],
                        weight=weight,
                        gradient=gradient
                    )

        # Create positions for visualization
```

```
        pos = {}
        for l in range(self.num_layers):
            for n in range(self.layer_sizes[l]):
                pos[f"L{l}N{n}"] = (l, n - self.layer_sizes[l]/2)

        # Draw the graph
        plt.figure(figsize=(12, 8))

        # Draw nodes
        colors = []
        for node in G.nodes():
            layer = int(node[1])
            if layer == 0:
                colors.append('blue')  # Input layer
            elif layer == self.num_layers - 1:
                colors.append('red')   # Output layer
            else:
                colors.append('green') # Hidden layers

        nx.draw_networkx_nodes(G, pos, node_color=colors, node_size=500,
alpha=0.8)

        # Draw edges with color based on weight value
        edges = G.edges()
        weights = [G[u][v]['weight'] for u, v in edges]

        # Normalize weights for coloring
        weight_abs = [abs(w) for w in weights]
        max_weight = max(weight_abs) if weight_abs else 1.0
        norm_weights = [abs(w)/max_weight for w in weights]

        cmap = plt.cm.Blues
        nx.draw_networkx_edges(G, pos, width=2, edge_color=norm_weights,
edge_cmap=cmap)

        # Draw labels
        nx.draw_networkx_labels(G, pos)

        plt.title("Neural Network Structure with Weights and Gradients")
        plt.axis('off')
        plt.show()
```

Pada metode ini akan membuat visualisasi grafis struktur jaringan neural (termasuk node dan koneksi antar layer), serta menampilkan weight dan gradien dalam bentuk warna *edge*.

Metode ini menggunakan `networkx` untuk membangun graf dari setiap node neuron dan `matplotlib` untuk menggambar visualisasi jaringan secara

keseluruhan.Data dibagi ke dalam batch kecil.. Untuk Setiap neuron diwakili node unik, diberi label `L<layer>N<neuron>`. Setiap weight diwakili *edge* antar node, weight digunakan untuk pewarnaan. Warna *edge* mencerminkan nilai relatif weight (semakin besar maka semakin gelap).

2.1.13 Metode *plot_weight_distribution*

```
def plot_weight_distribution(self, layers=None):
        """
        Plot the distribution of weights for specified layers.

        Parameters:
        -----------
        layers : list, optional
            List of layer indices to plot. If None, all layers are plotted.
        """
        if layers is None:
            layers = list(range(len(self.weights)))

        n_layers = len(layers)
        fig, axes = plt.subplots(1, n_layers, figsize=(15, 5))

        # Handle case with only one layer
        if n_layers == 1:
            axes = [axes]

        for i, layer_idx in enumerate(layers):
            if layer_idx < 0 or layer_idx >= len(self.weights):
                print(f"Warning: Layer index {layer_idx} is out of range.
Skipping.")
                continue

            weights = self.weights[layer_idx].flatten()
            axes[i].hist(weights, bins=30, alpha=0.7)
            axes[i].set_title(f"Layer {layer_idx+1} Weights")
            axes[i].set_xlabel("Weight Value")
            axes[i].set_ylabel("Frequency")

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

Pada metode ini akan menampilkan distribusi nilai weight dalam layer tertentu dalam bentuk histogram. Metode ini memiliki satu parameter, yaitu `layers`, yang berupa daftar indeks layer yang ingin divisualisasikan. Jika `layers = None`, maka semua layer divisualisasi. Untuk setiap layer yang dipilih untuk ditampilkan pada metode ini, ambil semua lalu ratakan ke 1D.

2.1.14 Metode *plot_gradient_distribution*

```
def plot_gradient_distribution(self, layers=None):
        """
        Plot the distribution of weight gradients for specified layers.

        Parameters:
        -----------
        layers : list, optional
            List of layer indices to plot. If None, all layers are plotted.
        """
        if layers is None:
            layers = list(range(len(self.weight_gradients)))

        n_layers = len(layers)
        fig, axes = plt.subplots(1, n_layers, figsize=(15, 5))

        # Handle case with only one layer
        if n_layers == 1:
            axes = [axes]

        for i, layer_idx in enumerate(layers):
            if layer_idx < 0 or layer_idx >= len(self.weight_gradients):
                print(f"Warning: Layer index {layer_idx} is out of range.
Skipping.")
                continue

            gradients = self.weight_gradients[layer_idx].flatten()
            axes[i].hist(gradients, bins=30, alpha=0.7)
            axes[i].set_title(f"Layer {layer_idx+1} Gradients")
            axes[i].set_xlabel("Gradient Value")
            axes[i].set_ylabel("Frequency")

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

Pada metode ini akan menampilkan distribusi nilai gradien dalam layer tertentu dalam bentuk histogram. Metode ini memiliki satu parameter, yaitu `layers`, yang berupa daftar indeks layer yang ingin divisualisasikan. Jika `layers = None`, maka semua layer divisualisasi.

2.1.15 Metode *save_model*

```
def save_model(self, filepath):
        """
        Save the model to a file.

        Parameters:
```

```
        ----------
        filepath : str
            Path to save the model
        """
        model_data = {
            'layer_sizes': self.layer_sizes,
            'activation_functions': self.activation_functions,
            'loss_function': self.loss_function,
            'weights': [w.tolist() for w in self.weights],
            'biases': [b.tolist() for b in self.biases]
        }

        np.save(filepath, model_data, allow_pickle=True)
        print(f"Model saved to {filepath}")
```

Pada metode ini akan menyimpan struktur dan parameter model ke dalam file `.npy`, agar bisa digunakan kembali tanpa *training* ulang.

2.1.16 Metode *load_model*

```
@classmethod
    def load_model(cls, filepath):
        """
        Load a model from a file.

        Parameters:
        ----------
        filepath : str
            Path to load the model from

        Returns:
        --------
        FFNN
            Loaded model
        """
        model_data = np.load(filepath, allow_pickle=True).item()

        # Create a new model instance
        model = cls(
            layer_sizes=model_data['layer_sizes'],
            activation_functions=model_data['activation_functions'],
            loss_function=model_data['loss_function'],
            weight_init_method='zero'  # Will be overwritten
        )

        # Replace the weights and biases
        model.weights = [np.array(w) for w in model_data['weights']]
        model.biases = [np.array(b) for b in model_data['biases']]

        # Initialize gradients with zeros
```

```
        model.weight_gradients = [np.zeros_like(w) for w in model.weights]
        model.bias_gradients = [np.zeros_like(b) for b in model.biases]

        print(f"Model loaded from {filepath}")
        return model
```

Pada metode ini akan memuat model FFNN dari *file* `.npy` yang telah disimpan sebelumnya, dan mengembalikan objek FFNN yang telah dibentuk ulang.

2.2 Pengujian

Pada bagian ini akan dilakukan pengujian pada model FFNN yang sudah di implementasi. Berikut adalah pengujian yang akan dilakukan:

1. Pengaruh *depth* dan *width* pada model FFNN
2. Pengaruh fungsi aktivasi pada model FFNN
3. Pengaruh *learning rate* pada model FFNN
4. Pengaruh inisialisasi weight pada model FFNN
5. Perbandingan dengan *library* sklearn

Pada pengujiannya, kami menggunakan *dataset* mnst_784 untuk melakukan training pada model yang kami implementasi dan model dari *library* sklearn.

2.2.1 Pengaruh *Depth* dan *Width* Pada Model FFNN

Pada pengujian kali ini kami memberikan masukan sebagai berikut pada model FFNN yang diimplementasi:

- `loss_function` : *Categorical Crossentrophy*
- `Weight_init_method` : *Random Normal*
- `weight_init_params` :
  - `mean` : 0
  - `variance` : 0.1
  - `seed` : 42

Pada saat melakukan training berikut adalah masukan yang diberikan pada model FFNN yang diimplementasi:

- `batch_size` : 32
- `learning_rate` : 0.1
- `epochs` : 20
- `verbose` :1

2.2.1.1 *Depth*

Pada bagian ini dilakukan pengujian *depth* dengan melakukan pengujian jumlah layer yang berbeda beda. Berikut adalah spesifikasi pengujian pada masing masing *depth*.

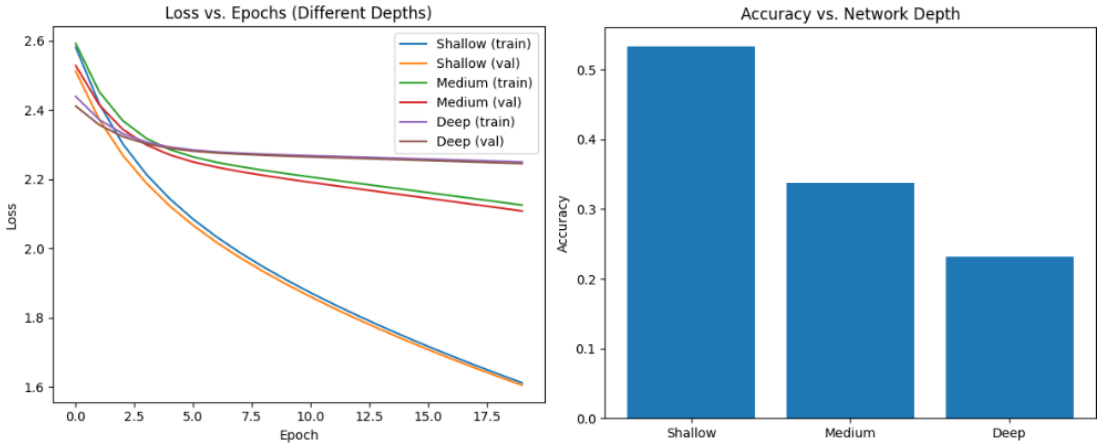
- `layers` : memiliki isi berupa jumlah neuron pada masing masing layer
- `activations` : memiliki isi berupa tipe aktivasi yang akan dilakukan pada tiap tiap layer

Pada pengujiannya kami menampilkan akurasi pada masing masing *depth* dan menampilkan distribusi *loss function* dan akurasi. Berikut adalah hasil dari pengujiannya,

```
Training Shallow network...
Epoch 1/20 - loss: 2.5812 - val_loss: 2.5120
Epoch 2/20 - loss: 2.4181 - val_loss: 2.3724
Epoch 3/20 - loss: 2.3024 - val_loss: 2.2695
Epoch 4/20 - loss: 2.2144 - val_loss: 2.1889
Epoch 5/20 - loss: 2.1436 - val_loss: 2.1226
Epoch 6/20 - loss: 2.0842 - val_loss: 2.0664
Epoch 7/20 - loss: 2.0329 - val_loss: 2.0170
Epoch 8/20 - loss: 1.9872 - val_loss: 1.9728
Epoch 9/20 - loss: 1.9458 - val_loss: 1.9325
Epoch 10/20 - loss: 1.9075 - val_loss: 1.8950
Epoch 11/20 - loss: 1.8718 - val_loss: 1.8599
Epoch 12/20 - loss: 1.8379 - val_loss: 1.8266
Epoch 13/20 - loss: 1.8057 - val_loss: 1.7949
Epoch 14/20 - loss: 1.7748 - val_loss: 1.7647
Epoch 15/20 - loss: 1.7452 - val_loss: 1.7356
Epoch 16/20 - loss: 1.7166 - val_loss: 1.7075
Epoch 17/20 - loss: 1.6890 - val_loss: 1.6804
Epoch 18/20 - loss: 1.6623 - val_loss: 1.6543
Epoch 19/20 - loss: 1.6365 - val_loss: 1.6290
Epoch 20/20 - loss: 1.6115 - val_loss: 1.6046
Shallow network accuracy: 0.5340

Training Medium network...
Epoch 1/20 - loss: 2.5921 - val_loss: 2.5282
Epoch 2/20 - loss: 2.4528 - val_loss: 2.4148
Epoch 3/20 - loss: 2.3693 - val_loss: 2.3442
Epoch 4/20 - loss: 2.3181 - val_loss: 2.2994
Epoch 5/20 - loss: 2.2858 - val_loss: 2.2703
Epoch 6/20 - loss: 2.2643 - val_loss: 2.2496
Epoch 7/20 - loss: 2.2486 - val_loss: 2.2344
Epoch 8/20 - loss: 2.2363 - val_loss: 2.2218
Epoch 9/20 - loss: 2.2255 - val_loss: 2.2108
Epoch 10/20 - loss: 2.2157 - val_loss: 2.2005
Epoch 11/20 - loss: 2.2063 - val_loss: 2.1908
Epoch 12/20 - loss: 2.1971 - val_loss: 2.1815
Epoch 13/20 - loss: 2.1881 - val_loss: 2.1722
```

```
Epoch 14/20 - loss: 2.1791 - val_loss: 2.1630
Epoch 15/20 - loss: 2.1702 - val_loss: 2.1539
Epoch 16/20 - loss: 2.1612 - val_loss: 2.1448
Epoch 17/20 - loss: 2.1522 - val_loss: 2.1356
Epoch 18/20 - loss: 2.1432 - val_loss: 2.1264
Epoch 19/20 - loss: 2.1342 - val_loss: 2.1173
Epoch 20/20 - loss: 2.1250 - val_loss: 2.1080
Medium network accuracy: 0.3380

Training Deep network...
Epoch 1/20 - loss: 2.4390 - val_loss: 2.4113
Epoch 2/20 - loss: 2.3720 - val_loss: 2.3570
Epoch 3/20 - loss: 2.3317 - val_loss: 2.3231
Epoch 4/20 - loss: 2.3073 - val_loss: 2.3023
Epoch 5/20 - loss: 2.2928 - val_loss: 2.2894
Epoch 6/20 - loss: 2.2841 - val_loss: 2.2812
Epoch 7/20 - loss: 2.2787 - val_loss: 2.2760
Epoch 8/20 - loss: 2.2751 - val_loss: 2.2721
Epoch 9/20 - loss: 2.2724 - val_loss: 2.2692
Epoch 10/20 - loss: 2.2700 - val_loss: 2.2667
Epoch 11/20 - loss: 2.2680 - val_loss: 2.2644
Epoch 12/20 - loss: 2.2660 - val_loss: 2.2622
Epoch 13/20 - loss: 2.2640 - val_loss: 2.2602
Epoch 14/20 - loss: 2.2620 - val_loss: 2.2581
Epoch 15/20 - loss: 2.2600 - val_loss: 2.2560
Epoch 16/20 - loss: 2.2579 - val_loss: 2.2539
Epoch 17/20 - loss: 2.2559 - val_loss: 2.2518
Epoch 18/20 - loss: 2.2538 - val_loss: 2.2495
Epoch 19/20 - loss: 2.2518 - val_loss: 2.2473
Epoch 20/20 - loss: 2.2496 - val_loss: 2.2451
Deep network accuracy: 0.2320
```

Berikut adalah distribusi *loss function* dan akurasi pada masing masing *depth*:



Gambar 2.2.1.1 Diagram distribusi *loss function* dan perbandingan antara akurasi dengan *network depth*

a) Shallow

- `layers : [784, 30, 10]`

- `activations : ["sigmoid", "softmax"]`

Pada kedalaman ini dalam distribusi *loss function* terlihat jika penurunan *loss*-nya lebih signifikan pada tiap epochnya dan menghasilkan *loss functio*n yang lebih rendah dibandingkan dengan kedalaman lainnya dan akurasi yang dihasilkan juga lebih baik dibandingkan dengan *Medium* dan *Deep*.

b) Medium

- `layers : [784, 30, 30, 10]`
- `activations : ["sigmoid", "sigmoid", "softmax"]`

Pada kedalaman ini *loss function* turun tidak lebih signifikan pada setiap epoch dibandingkan dengan *Shallow* dengan hasil akhir *loss function* yang lebih tinggi dari *Shallow* dan akurasi yang dihasilkan juga lebih rendah dibandingkan dengan *Shallow*.

c) Deep

- `layers : [784, 30, 30, 30, 10]`
- `activations : ["sigmoid","sigmoid","sigmoid","softmax"]`

Pada kedalaman ini memiliki *loss function* yang turunnya tidak lebih signifikan dibandingkan dengan kedalaman lainnya pada tiap epoch dengan hasil akhir loss function yang lebih tinggi dibandingkan dengan kedalaman lainnya dan akurasi yang dihasilkan juga lebih kecil dibandingkan dengan kedalaman *Shallow* dan *Medium*.

Selain itu semakin banyak epoch yang dilakukan data *train* dan *validation*-nya semakin overlap dan menghasilkan *loss function* yang tinggi dimana mengindikasikan bahwa pada model ini mengalami *underfitting*.

Pada pengujian *depth* dapat disimpulkan jika semakin banyak depth yang digunakan akan mengurangi akurasi dan menghasilkan *loss function* yang

lebih besar. Hal ini dapat disebabkan karena gradien (yang digunakan untuk *update weight*) menjadi sangat kecil (*vanishing*) atau sangat besar (*exploding*) saat melewati banyak lapisan selama backpropagation.

2.2.1.2 *Width*

Pada pengujian kali ini akan dilakukan pengujian width yang dimana akan terdapat tiga layer (input, 1 hidden, dan output) dengan memberikan jumlah width yang berbeda beda pada hidden layer.

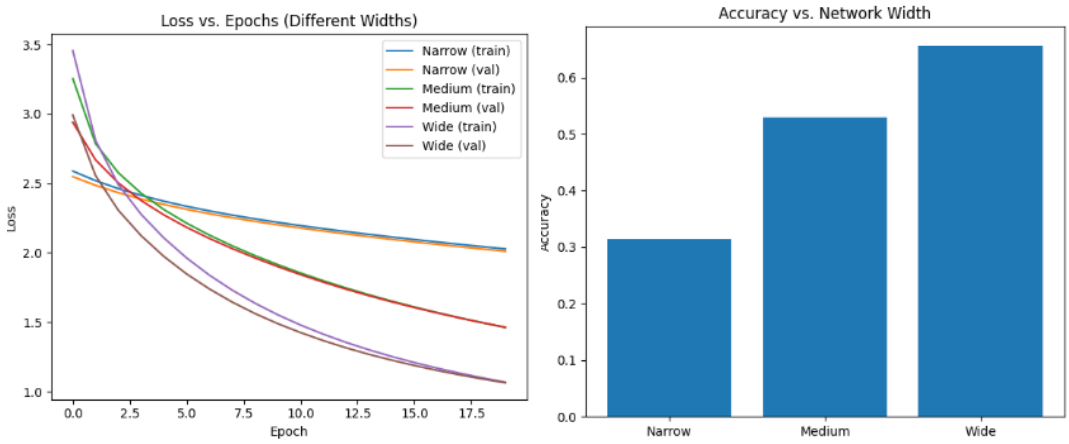
- `layers` : memiliki isi berupa jumlah neuron pada masing masing layer
- `activations` : memiliki isi berupa tipe aktivasi yang akan dilakukan pada tiap tiap layer

Pada pengujiannya kami menampilkan akurasi masing masing *width*, distribusi *loss function* tiap epoch, dan distribusi akurasi masing masing *width*. Berikut adalah hasil dari pengujian masing masing *width*:

```
Training Narrow network...
Epoch 1/20 - loss: 2.5885 - val_loss: 2.5485
Epoch 2/20 - loss: 2.5197 - val_loss: 2.4857
Epoch 3/20 - loss: 2.4623 - val_loss: 2.4328
Epoch 4/20 - loss: 2.4136 - val_loss: 2.3876
Epoch 5/20 - loss: 2.3714 - val_loss: 2.3481
Epoch 6/20 - loss: 2.3344 - val_loss: 2.3130
Epoch 7/20 - loss: 2.3014 - val_loss: 2.2814
Epoch 8/20 - loss: 2.2716 - val_loss: 2.2527
Epoch 9/20 - loss: 2.2443 - val_loss: 2.2262
Epoch 10/20 - loss: 2.2192 - val_loss: 2.2016
Epoch 11/20 - loss: 2.1957 - val_loss: 2.1785
Epoch 12/20 - loss: 2.1737 - val_loss: 2.1567
Epoch 13/20 - loss: 2.1528 - val_loss: 2.1360
Epoch 14/20 - loss: 2.1329 - val_loss: 2.1161
Epoch 15/20 - loss: 2.1139 - val_loss: 2.0971
Epoch 16/20 - loss: 2.0956 - val_loss: 2.0788
Epoch 17/20 - loss: 2.0778 - val_loss: 2.0611
Epoch 18/20 - loss: 2.0607 - val_loss: 2.0439
Epoch 19/20 - loss: 2.0441 - val_loss: 2.0272
Epoch 20/20 - loss: 2.0279 - val_loss: 2.0110
Narrow network accuracy: 0.3140

Training Medium network...
Epoch 1/20 - loss: 3.2550 - val_loss: 2.9400
Epoch 2/20 - loss: 2.7862 - val_loss: 2.6692
Epoch 3/20 - loss: 2.5764 - val_loss: 2.5031
Epoch 4/20 - loss: 2.4289 - val_loss: 2.3765
```

```
Epoch 5/20 - loss: 2.3116 - val_loss: 2.2727
Epoch 6/20 - loss: 2.2132 - val_loss: 2.1830
Epoch 7/20 - loss: 2.1273 - val_loss: 2.1032
Epoch 8/20 - loss: 2.0501 - val_loss: 2.0304
Epoch 9/20 - loss: 1.9797 - val_loss: 1.9633
Epoch 10/20 - loss: 1.9147 - val_loss: 1.9009
Epoch 11/20 - loss: 1.8545 - val_loss: 1.8428
Epoch 12/20 - loss: 1.7986 - val_loss: 1.7887
Epoch 13/20 - loss: 1.7465 - val_loss: 1.7382
Epoch 14/20 - loss: 1.6978 - val_loss: 1.6912
Epoch 15/20 - loss: 1.6523 - val_loss: 1.6471
Epoch 16/20 - loss: 1.6095 - val_loss: 1.6058
Epoch 17/20 - loss: 1.5694 - val_loss: 1.5669
Epoch 18/20 - loss: 1.5315 - val_loss: 1.5302
Epoch 19/20 - loss: 1.4959 - val_loss: 1.4958
Epoch 20/20 - loss: 1.4623 - val_loss: 1.4631
Medium network accuracy: 0.5300

Training Wide network...
Epoch 1/20 - loss: 3.4567 - val_loss: 2.9920
Epoch 2/20 - loss: 2.8093 - val_loss: 2.5559
Epoch 3/20 - loss: 2.4910 - val_loss: 2.3062
Epoch 4/20 - loss: 2.2772 - val_loss: 2.1234
Epoch 5/20 - loss: 2.1068 - val_loss: 1.9737
Epoch 6/20 - loss: 1.9627 - val_loss: 1.8475
Epoch 7/20 - loss: 1.8387 - val_loss: 1.7394
Epoch 8/20 - loss: 1.7310 - val_loss: 1.6451
Epoch 9/20 - loss: 1.6368 - val_loss: 1.5624
Epoch 10/20 - loss: 1.5535 - val_loss: 1.4897
Epoch 11/20 - loss: 1.4796 - val_loss: 1.4254
Epoch 12/20 - loss: 1.4138 - val_loss: 1.3675
Epoch 13/20 - loss: 1.3546 - val_loss: 1.3156
Epoch 14/20 - loss: 1.3013 - val_loss: 1.2687
Epoch 15/20 - loss: 1.2531 - val_loss: 1.2268
Epoch 16/20 - loss: 1.2093 - val_loss: 1.1884
Epoch 17/20 - loss: 1.1692 - val_loss: 1.1528
Epoch 18/20 - loss: 1.1325 - val_loss: 1.1205
Epoch 19/20 - loss: 1.0989 - val_loss: 1.0910
Epoch 20/20 - loss: 1.0678 - val_loss: 1.0635
Wide network accuracy: 0.6570
```

Berikut adalah distribusi dari *loss function* dan akurasi masing masing *width*:



Gambar 2.2.1.2 Diagram distribusi *loss function* dan perbandingan antara akurasi dengan *network width*

a) Narrow

- `layers` : `[784, 10, 10]`
- `activations` : `["sigmoid", "softmax"]`

Pada *width narrow*, *loss function* yang dihasilkan pada tiap tiap epoch tidak mengalami penurunan yang signifikan dan menghasilkan *loss function* yang sangat besar dibandingkan dengan pengujian *width* lainnya, akurasi yang dihasilkan juga sangat rendah dibandingkan dengan model lainnya. Hal ini dapat disebabkan karena model yang terlalu sederhana dan terlalu singkat dalam mempelajari pola yang diberikan.

b) Medium

- `layers` : `[784, 50, 10]`
- `activations` : `["sigmoid", "softmax"]`

Pada *width medium*, *loss function* yang dihasilkan pada tiap epoch mengalami penurunan yang lebih besar dibandingkan *width narrow*, hasil akhir dari *loss function* pada *width medium* juga lebih rendah dibandingkan dengan width narrow dan hasilnya akurasinya lebih tinggi.

c) Wide

- `layers` : `[784, 100, 10]`
- `activations` : `["sigmoid", "softmax"]`

Pada *width wide*, *loss function* yang dihasilkan tiap epoch mengalami penurunan yang lebih signifikan, hasil dari *loss function* di akhir epoch juga lebih rendah, dan akurasi yang dihasilkan lebih tinggi dibandingkan dengan pengujian width lainnya.

Pada pengujian *width*, dapat disimpulkan jika semakin besar *width* pada layer maka model akan semakin optimal dalam melakukan pembelajaran pada dataset yang diberikan, menghasilkan *loss function* yang lebih rendah dan memiliki akurasi yang lebih tinggi.

2.2.2 Pengaruh *Activation Function* Pada Model FFNN

Pada pengujian ini akan dilakukan perbandingan antara *activation function* pada masing masing model dengan melakukan perbandingan *loss function*, *akurasi*, dan *distribusi gradient weight* pada masing masing layer. Berikut adalah spesifikasi yang dilakukan dalam pengujian

- `loss_function` : *Categorical Crossentrophy*
- `weight_init_method` : *Random Normal*
- `layer_sizes` : [784, 50, 10]
- `weight_init_params` :
  - `mean` : 0
  - `variance` : 0.1
  - `seed` : 42

Pada saat melakukan training berikut adalah masukan yang diberikan pada model FFNN yang diimplementasi:

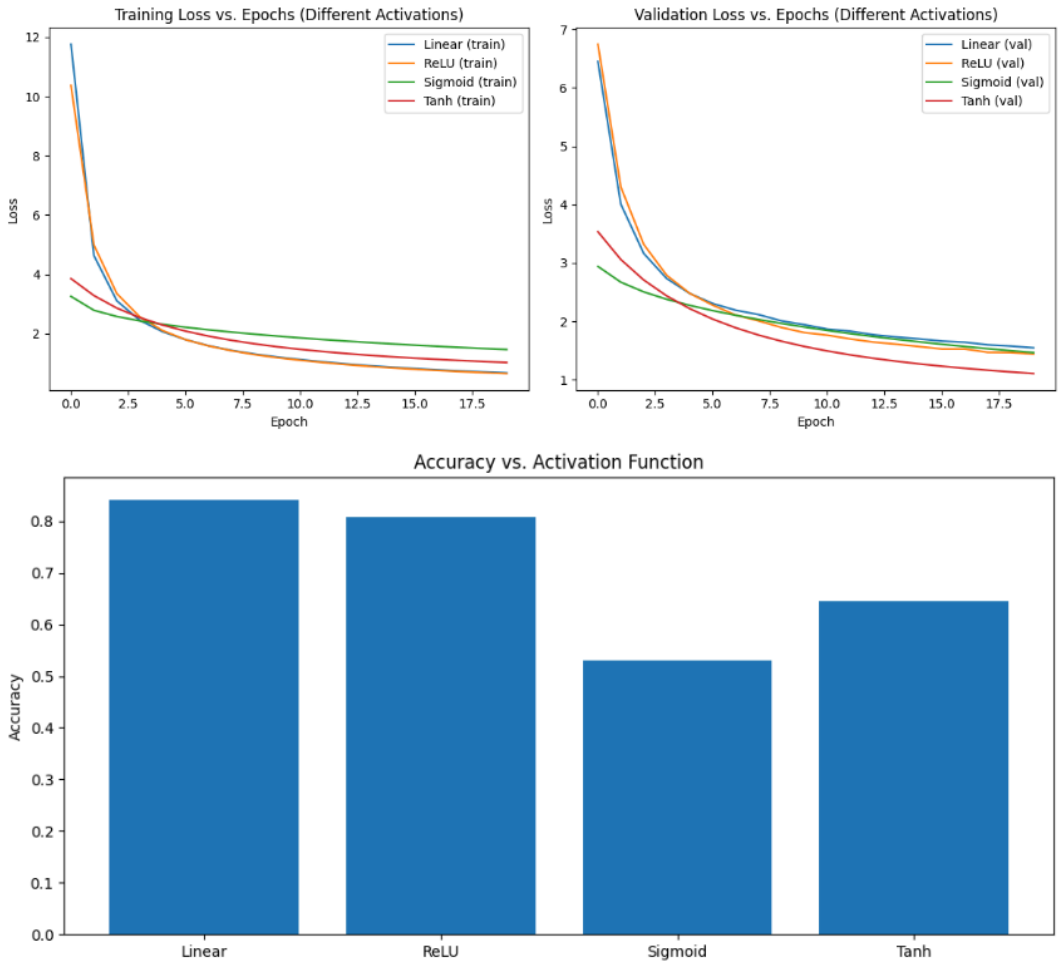
- `batch_size` : 32
- `learning_rate` : 0.1
- `epochs` : 20
- `verbose` : 1

Berikut adalah hasil dari pengujian berupa loss function tiap tiap epoch:

```
Training network with Linear activation...
Epoch 1/20 - loss: 11.7655 - val_loss: 6.4498
Epoch 2/20 - loss: 4.6335 - val_loss: 4.0060
Epoch 3/20 - loss: 3.1049 - val_loss: 3.1609
Epoch 4/20 - loss: 2.4470 - val_loss: 2.7319
Epoch 5/20 - loss: 2.0627 - val_loss: 2.4774
Epoch 6/20 - loss: 1.7955 - val_loss: 2.3043
Epoch 7/20 - loss: 1.5902 - val_loss: 2.1900
Epoch 8/20 - loss: 1.4324 - val_loss: 2.1146
Epoch 9/20 - loss: 1.3101 - val_loss: 2.0065
Epoch 10/20 - loss: 1.2083 - val_loss: 1.9410
Epoch 11/20 - loss: 1.1220 - val_loss: 1.8660
Epoch 12/20 - loss: 1.0450 - val_loss: 1.8301
Epoch 13/20 - loss: 0.9772 - val_loss: 1.7693
Epoch 14/20 - loss: 0.9155 - val_loss: 1.7306
Epoch 15/20 - loss: 0.8670 - val_loss: 1.6982
Epoch 16/20 - loss: 0.8183 - val_loss: 1.6599
Epoch 17/20 - loss: 0.7746 - val_loss: 1.6401
Epoch 18/20 - loss: 0.7389 - val_loss: 1.5963
Epoch 19/20 - loss: 0.7033 - val_loss: 1.5744
Epoch 20/20 - loss: 0.6732 - val_loss: 1.5451
Linear activation accuracy: 0.8420

Training network with ReLU activation...
Epoch 1/20 - loss: 10.3765 - val_loss: 6.7469
Epoch 2/20 - loss: 4.9896 - val_loss: 4.2996
Epoch 3/20 - loss: 3.3493 - val_loss: 3.3158
Epoch 4/20 - loss: 2.5637 - val_loss: 2.7872
Epoch 5/20 - loss: 2.1005 - val_loss: 2.4792
Epoch 6/20 - loss: 1.7924 - val_loss: 2.2769
Epoch 7/20 - loss: 1.5833 - val_loss: 2.1069
Epoch 8/20 - loss: 1.4215 - val_loss: 2.0070
Epoch 9/20 - loss: 1.2913 - val_loss: 1.8939
Epoch 10/20 - loss: 1.1863 - val_loss: 1.8058
Epoch 11/20 - loss: 1.0999 - val_loss: 1.7618
Epoch 12/20 - loss: 1.0235 - val_loss: 1.6972
Epoch 13/20 - loss: 0.9549 - val_loss: 1.6434
Epoch 14/20 - loss: 0.8940 - val_loss: 1.6074
Epoch 15/20 - loss: 0.8435 - val_loss: 1.5665
Epoch 16/20 - loss: 0.7960 - val_loss: 1.5249
Epoch 17/20 - loss: 0.7546 - val_loss: 1.5250
Epoch 18/20 - loss: 0.7166 - val_loss: 1.4675
Epoch 19/20 - loss: 0.6809 - val_loss: 1.4647
Epoch 20/20 - loss: 0.6522 - val_loss: 1.4382
ReLU activation accuracy: 0.8080

Training network with Sigmoid activation...
Epoch 1/20 - loss: 3.2550 - val_loss: 2.9400
Epoch 2/20 - loss: 2.7862 - val_loss: 2.6692
Epoch 3/20 - loss: 2.5764 - val_loss: 2.5031
Epoch 4/20 - loss: 2.4289 - val_loss: 2.3765
Epoch 5/20 - loss: 2.3116 - val_loss: 2.2727
Epoch 6/20 - loss: 2.2132 - val_loss: 2.1830
Epoch 7/20 - loss: 2.1273 - val_loss: 2.1032
Epoch 8/20 - loss: 2.0501 - val_loss: 2.0304
Epoch 9/20 - loss: 1.9797 - val_loss: 1.9633
```

```
Epoch 10/20 - loss: 1.9147 - val_loss: 1.9009
Epoch 11/20 - loss: 1.8545 - val_loss: 1.8428
Epoch 12/20 - loss: 1.7986 - val_loss: 1.7887
Epoch 13/20 - loss: 1.7465 - val_loss: 1.7382
Epoch 14/20 - loss: 1.6978 - val_loss: 1.6912
Epoch 15/20 - loss: 1.6523 - val_loss: 1.6471
Epoch 16/20 - loss: 1.6095 - val_loss: 1.6058
Epoch 17/20 - loss: 1.5694 - val_loss: 1.5669
Epoch 18/20 - loss: 1.5315 - val_loss: 1.5302
Epoch 19/20 - loss: 1.4959 - val_loss: 1.4958
Epoch 20/20 - loss: 1.4623 - val_loss: 1.4631
Sigmoid activation accuracy: 0.5300

Training network with Tanh activation...
Epoch 1/20 - loss: 3.8551 - val_loss: 3.5342
Epoch 2/20 - loss: 3.2831 - val_loss: 3.0577
Epoch 3/20 - loss: 2.8603 - val_loss: 2.7064
Epoch 4/20 - loss: 2.5401 - val_loss: 2.4347
Epoch 5/20 - loss: 2.2874 - val_loss: 2.2169
Epoch 6/20 - loss: 2.0825 - val_loss: 2.0379
Epoch 7/20 - loss: 1.9136 - val_loss: 1.8890
Epoch 8/20 - loss: 1.7731 - val_loss: 1.7639
Epoch 9/20 - loss: 1.6548 - val_loss: 1.6583
Epoch 10/20 - loss: 1.5542 - val_loss: 1.5682
Epoch 11/20 - loss: 1.4676 - val_loss: 1.4909
Epoch 12/20 - loss: 1.3924 - val_loss: 1.4242
Epoch 13/20 - loss: 1.3263 - val_loss: 1.3662
Epoch 14/20 - loss: 1.2682 - val_loss: 1.3154
Epoch 15/20 - loss: 1.2166 - val_loss: 1.2703
Epoch 16/20 - loss: 1.1704 - val_loss: 1.2301
Epoch 17/20 - loss: 1.1288 - val_loss: 1.1940
Epoch 18/20 - loss: 1.0908 - val_loss: 1.1613
Epoch 19/20 - loss: 1.0560 - val_loss: 1.1317
Epoch 20/20 - loss: 1.0241 - val_loss: 1.1046
Tanh activation accuracy: 0.6450
```

Berikut adalah hasil akurasi, distribusi *loss function*, dan distribusi akurasi pada masing masing fungsi aktivasi,



Gambar 2.2.2.1 Diagram distribusi *loss function* dan perbandingan antara akurasi dengan *network depth*

Terdapat empat jenis *activation function yang* diuji pada *hidden layer*, yaitu: `Linear`, `ReLU`, `Sigmoid`, dan `Tanh`. Hasil menunjukkan bahwa semua *activation function* mampu menurunkan nilai *loss* selama proses *training*, walaupun terdapat beberapa yang memiliki karakteristik yang berbeda. Berikut adalah penjelasan mengenai hasil dari setiap *activation function*.

a) Linear

Model dengan fungsi aktivasi `Linear` menunjukkan penurunan *training loss* dan *validation loss* yang cukup konsisten dalam 20 epoch. Model ini mencapai akurasi tertinggi di antara semua aktivasi, yaitu sebesar **84.20%** pada data *test*. Hal ini menunjukkan bahwa untuk *dataset* dan arsitektur tertentu, model sederhana dengan aktivasi linear tetap bisa bekerja cukup

efektif. Distribusi weight dan gradien dari model linear juga menunjukkan penyebaran yang tidak terlalu ekstrem.

b) ReLU (Rectified Linear Unit)

Model dengan aktivasi `ReLU` menunjukkan karakteristik awal yang mirip dengan linear, yaitu penurunan *loss* yang cepat, baik pada data training maupun validasi. Namun, pada akhir training, validation loss terlihat mulai stagnan, bahkan sedikit meningkat. Meskipun demikian, model masih berhasil mencapai akurasi tinggi sebesar **80.80%** pada data uji. Dari sisi distribusi gradien, `ReLU` memiliki penyebaran yang lebih baik dibanding `Sigmoid` dan `Tanh`.

c) Sigmoid

Model dengan fungsi aktivasi `Sigmoid` mengalami kesulitan dalam menurunkan *loss* secara agresif, baik *training loss* maupun *validation loss* turun sangat lambat. Akurasinya juga rendah, hanya **53.00%.** Ini menandakan bahwa model gagal belajar representasi yang baik.

d) Tanh

Model dengan fungsi aktivasi `Tanh` memiliki performa sedikit lebih baik dibanding `Sigmoid`. *Loss* turun lebih cepat, dan *validation loss* secara konsisten membaik selama epoch. Akurasi akhirnya adalah **64.50%**, lebih tinggi dari `Sigmoid` tetapi tetap jauh di bawah `Linear` dan `ReLU`.

Pengujian ini membuktikan bahwa pemilihan *activation function* memiliki dampak besar terhadap performa FFNN. Pada kasus pengujian ini, `Linear` *activation function* memberikan performa lebih bagus bila dibandingkan dengan *activation function* lainnya, apalagi dengan *activation function* seperti `Sigmoid`. Beberapa *activation function* seperti `Sigmoid` dan `Tanh` memiliki akurasi yang rendah disebabkan karena efek *vanishing gradient***,** dimana nilai gradien dari fungsi *loss* terhadap weight menjadi sangat kecil ketika backpropagation dilakukan dari output layer ke layer-layer sebelumnya. Akibatnya, *update* terhadap weight di layer awal menjadi sangat kecil.

2.2.3 Pengaruh *Learning Rate* pada Model FFNN

Pada pengujian ini, akan dilakukan perbandingan dengan memasukkan tiga *learning rate* yang dipilih untuk mengetahui pengaruh *learning rate* pada model yang dilatih. Berikut adalah spesifikasi model yang akan digunakan pada pengujian ini:

- `loss_function` : *Categorical Crossentrophy*
- `weight_init_method` : *Random Normal*
- `layer_sizes` : [784, 50, 10]
- `activation_function` : ["*sigmoid*", "*softmax*"]
- `weight_init_params` :
  - `mean` : 0
  - `variance` : 0.1
  - `seed` : 42

Pada saat melakukan *training* berikut adalah masukan yang diberikan pada model FFNN yang diimplementasi:

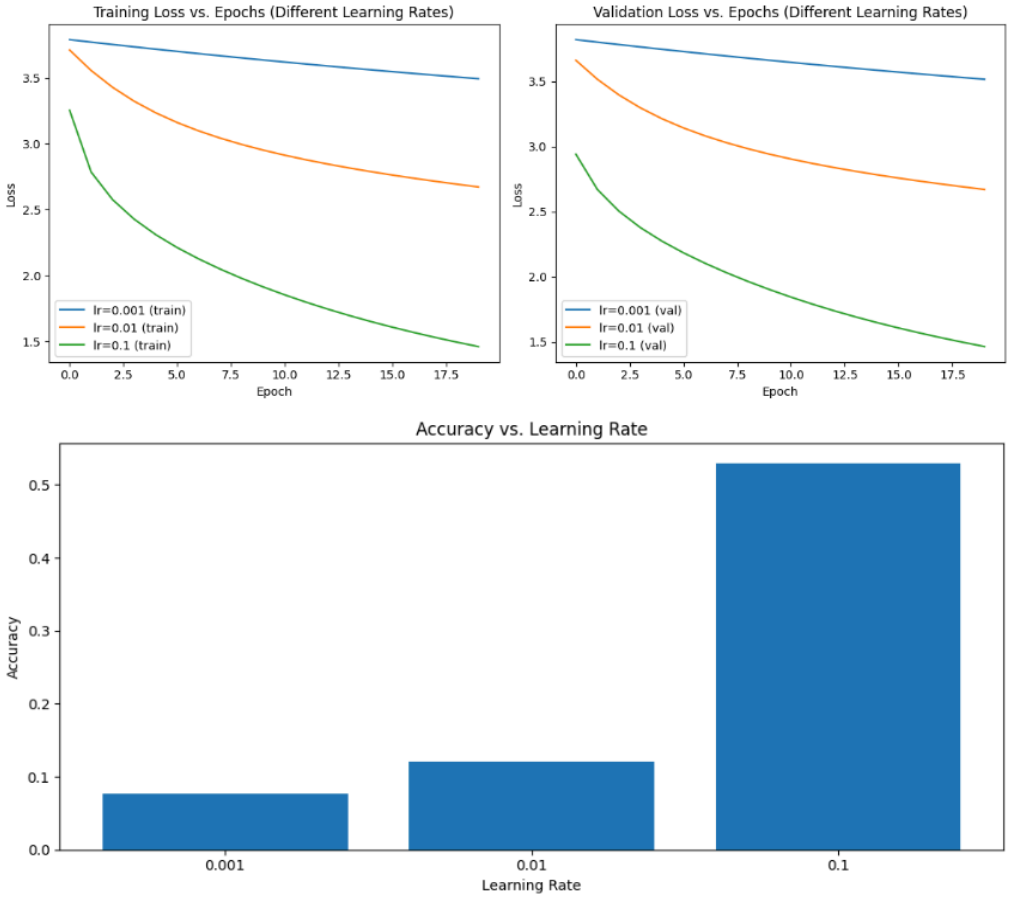
- `batch_size` : 32
- `learning_rate` : 0.1, 0.01, 0.001 (tiga *learning rate* yang diuji)
- `epochs` : 20
- `verbose` : 1

Berikut adalah hasil pengujian yang dilakukan berupa hasil dari *loss function* pada tiap epoch pada masing masing learning rate,

```
Training network with learning rate = 0.001...
Epoch 1/20 - loss: 3.7901 - val_loss: 3.8197
Epoch 2/20 - loss: 3.7716 - val_loss: 3.8007
Epoch 3/20 - loss: 3.7534 - val_loss: 3.7820
Epoch 4/20 - loss: 3.7356 - val_loss: 3.7638
Epoch 5/20 - loss: 3.7181 - val_loss: 3.7458
Epoch 6/20 - loss: 3.7010 - val_loss: 3.7283
Epoch 7/20 - loss: 3.6842 - val_loss: 3.7110
Epoch 8/20 - loss: 3.6678 - val_loss: 3.6942
Epoch 9/20 - loss: 3.6517 - val_loss: 3.6776
Epoch 10/20 - loss: 3.6359 - val_loss: 3.6614
Epoch 11/20 - loss: 3.6204 - val_loss: 3.6454
```

```
Epoch 12/20 - loss: 3.6052 - val_loss: 3.6298
Epoch 13/20 - loss: 3.5903 - val_loss: 3.6145
Epoch 14/20 - loss: 3.5757 - val_loss: 3.5995
Epoch 15/20 - loss: 3.5614 - val_loss: 3.5848
Epoch 16/20 - loss: 3.5474 - val_loss: 3.5704
Epoch 17/20 - loss: 3.5336 - val_loss: 3.5563
Epoch 18/20 - loss: 3.5202 - val_loss: 3.5424
Epoch 19/20 - loss: 3.5069 - val_loss: 3.5288
Epoch 20/20 - loss: 3.4940 - val_loss: 3.5155
Learning rate 0.001 accuracy: 0.0760

Training network with learning rate = 0.01...
Epoch 1/20 - loss: 3.7118 - val_loss: 3.6612
Epoch 2/20 - loss: 3.5561 - val_loss: 3.5153
Epoch 3/20 - loss: 3.4285 - val_loss: 3.3954
Epoch 4/20 - loss: 3.3233 - val_loss: 3.2959
Epoch 5/20 - loss: 3.2356 - val_loss: 3.2127
Epoch 6/20 - loss: 3.1616 - val_loss: 3.1420
Epoch 7/20 - loss: 3.0983 - val_loss: 3.0813
Epoch 8/20 - loss: 3.0434 - val_loss: 3.0285
Epoch 9/20 - loss: 2.9952 - val_loss: 2.9820
Epoch 10/20 - loss: 2.9523 - val_loss: 2.9404
Epoch 11/20 - loss: 2.9137 - val_loss: 2.9031
Epoch 12/20 - loss: 2.8787 - val_loss: 2.8690
Epoch 13/20 - loss: 2.8466 - val_loss: 2.8378
Epoch 14/20 - loss: 2.8169 - val_loss: 2.8090
Epoch 15/20 - loss: 2.7893 - val_loss: 2.7821
Epoch 16/20 - loss: 2.7633 - val_loss: 2.7570
Epoch 17/20 - loss: 2.7389 - val_loss: 2.7333
Epoch 18/20 - loss: 2.7158 - val_loss: 2.7109
Epoch 19/20 - loss: 2.6939 - val_loss: 2.6896
Epoch 20/20 - loss: 2.6729 - val_loss: 2.6692
Learning rate 0.01 accuracy: 0.1210

Training network with learning rate = 0.1...
Epoch 1/20 - loss: 3.2550 - val_loss: 2.9400
Epoch 2/20 - loss: 2.7862 - val_loss: 2.6692
Epoch 3/20 - loss: 2.5764 - val_loss: 2.5031
Epoch 4/20 - loss: 2.4289 - val_loss: 2.3765
Epoch 5/20 - loss: 2.3116 - val_loss: 2.2727
Epoch 6/20 - loss: 2.2132 - val_loss: 2.1830
Epoch 7/20 - loss: 2.1273 - val_loss: 2.1032
Epoch 8/20 - loss: 2.0501 - val_loss: 2.0304
Epoch 9/20 - loss: 1.9797 - val_loss: 1.9633
Epoch 10/20 - loss: 1.9147 - val_loss: 1.9009
Epoch 11/20 - loss: 1.8545 - val_loss: 1.8428
Epoch 12/20 - loss: 1.7986 - val_loss: 1.7887
Epoch 13/20 - loss: 1.7465 - val_loss: 1.7382
Epoch 14/20 - loss: 1.6978 - val_loss: 1.6912
Epoch 15/20 - loss: 1.6523 - val_loss: 1.6471
Epoch 16/20 - loss: 1.6095 - val_loss: 1.6058
Epoch 17/20 - loss: 1.5694 - val_loss: 1.5669
Epoch 18/20 - loss: 1.5315 - val_loss: 1.5302
Epoch 19/20 - loss: 1.4959 - val_loss: 1.4958
Epoch 20/20 - loss: 1.4623 - val_loss: 1.4631
Learning rate 0.1 accuracy: 0.5300
```

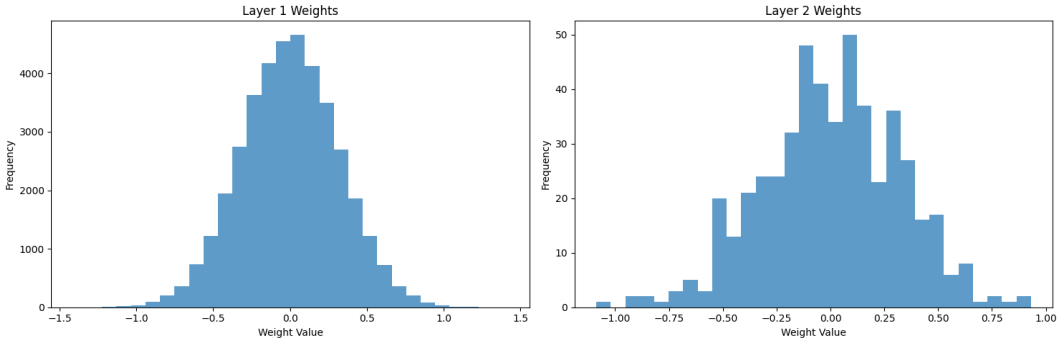
Berikut adalah hasil dari distribusi dari *loss function* pada data *training* dan valid serta distribusi akurasi pada masing masing *learning rate* yang diujikan.

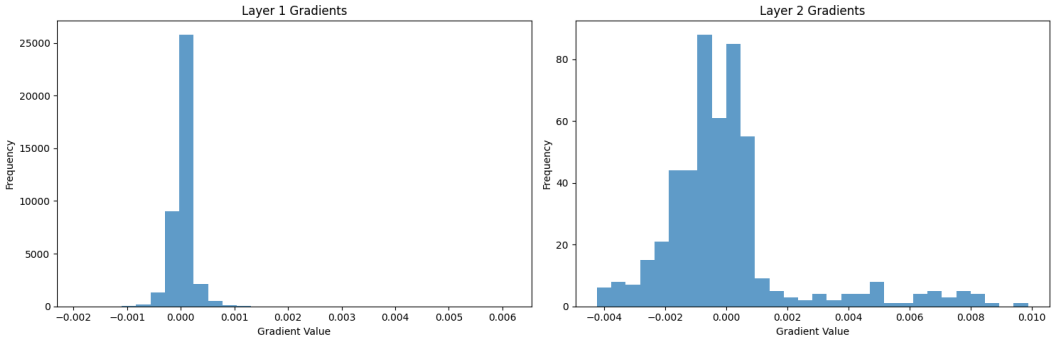


Gambar 2.2.3.1 Diagram distribusi *loss function* dan perbandingan antara akurasi dengan *learning rate*

Berikut adalah distribusi *weight* dan *gradient weight* dari masing masing *learning rate* dan penjelasan dari masing masing data yang ada pada hasil uji:

a) *Learning Rate* 0.001

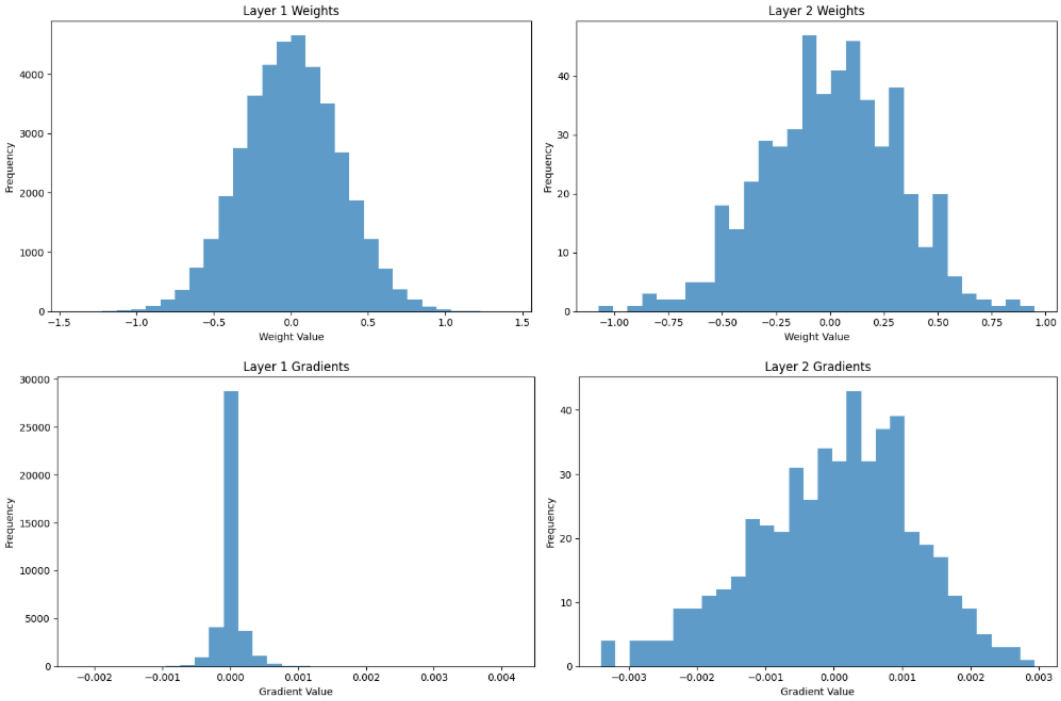




Gambar 2.2.3.2 Diagram distribusi *weight* dan *gradient weight* dengan *learning rate* 0.001

Dari hasil diagram diatas, model menunjukkan penurunan loss yang sangat lambat baik pada data *training* maupun *validation*. *Training loss* hanya turun dari 3.79 ke 3.49 dalam 20 epoch, dan *validation loss* dari 3.82 ke 3.51, yang artinya progres *training* hampir stagnan. Akurasi pada data uji pun sangat rendah, yaitu hanya 7.60%. Bisa disimpulkan bahwa *learning rate* ini **terlalu kecil** untuk memicu proses *learning* yang efektif.

b) *Learning Rate* 0.01

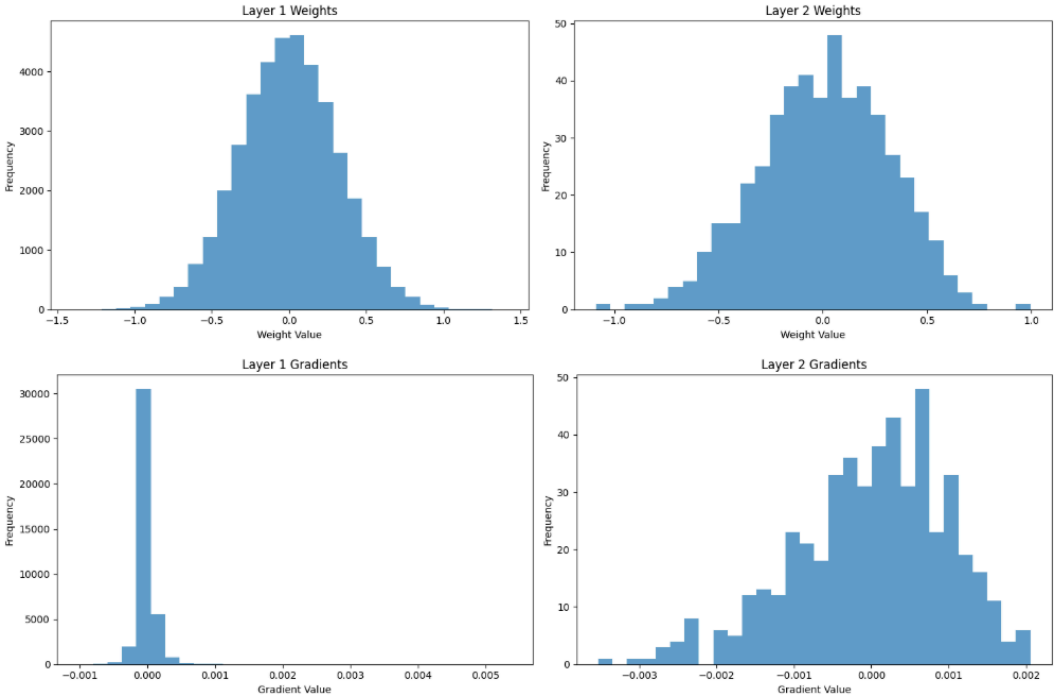


Gambar 2.2.3.3 Diagram distribusi *weight* dan *gradient weight* dengan *learning rate* 0.01

Dari hasil diagram diatas, model menunjukkan penurunan loss yang konsisten dari awal sampai akhir *training*. Akurasi pada data uji adalah 12.10%, yang

masih tergolong rendah, tetapi lebih baik daripada learning rate 0.001. Distribusi weight terlihat mulai menyebar, dan gradien sudah lebih bervariasi dibanding sebelumnya. Hal ini menunjukkan bahwa meskipun pembaruan weight terjadi, masih relatif lambat. Dengan demikian, *learning rate* ini **cukup stabil**, namun masih belum cukup besar untuk memicu *learning* yang optimal dalam waktu training 20 epoch.

c) *Learning Rate* 0.1



Gambar 2.2.3.4 Diagram distribusi *weight* dan *gradient weight* dengan *learning rate* 0.1

Dari hasil diagram diatas, model menunjukkan performa terbaik dari ketiga konfigurasi. *Training loss* menurun dengan cepat dari 3.25 ke 1.46, dan *validation loss* dari 2.94 ke 1.46. Akurasi pada data uji sangat tinggi bila dibandingkan kedua konfigurasi *learning rate* lainnya, yaitu sekitar 53.00%. Ini menunjukkan bahwa model berhasil mempelajari representasi data yang berguna untuk klasifikasi. Distribusi weight menunjukkan perubahan yang sehat, dan gradien tersebar luas, terutama di layer kedua, menandakan pembaruan weight berjalan secara aktif. Tidak terlihat gejala *overfitting* meskipun *learning rate* cukup besar, sehingga nilai ini menjadi nilai yang ideal untuk *training* dalam pengujian ini.

Dari hasil pengujian di atas, bisa kita simpulkan bahwa *learning rate* yang besar mendorong pembaruan weight lebih signifikan, sementara *learning rate* yang kecil menyebabkan model sangat lambat bahkan hampir stagnan.

2.2.4 Pengaruh *Weight Initialization* Pada Model FFNN

Pada pengujian ini akan dilakukan perbandingan *loss function*, akurasi, serta distribusi weight dan gradient pada masing masing tipe *weight* yang akan diujikan (*zero*, *random uniform*, *random normal*). Berikut adalah spesifikasi model yang digunakan pada pengujian ini:

- `loss_function` : *Categorical Crossentrophy*
- `layer_sizes` : [784, 50, 10]
- `activation_function` : [“sigmoid”, “softmax”]

Pada saat melakukan training berikut adalah masukan yang diberikan pada model FFNN yang diimplementasi:

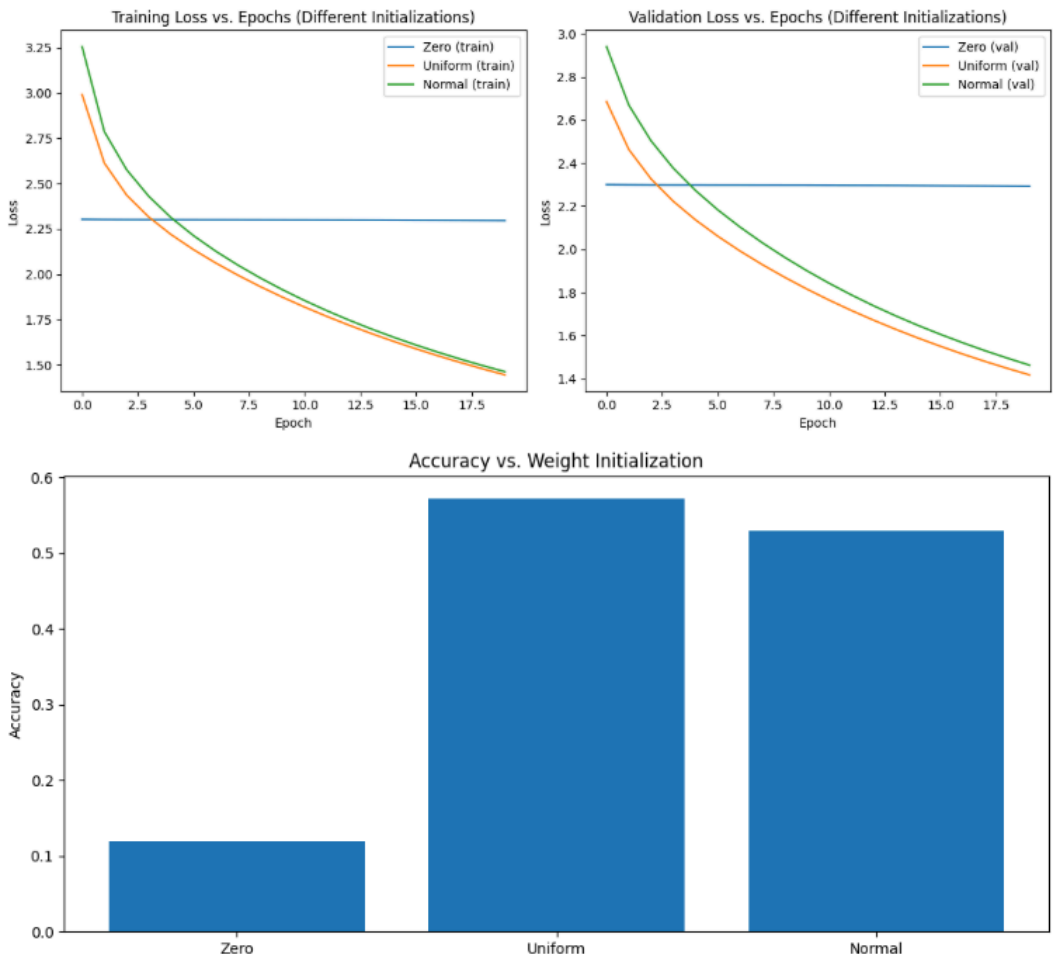
- `batch_size` : 32
- `learning_rate` : 0.1
- `epochs` : 20
- `verbose` : 1

Berikut adalah hasil pengujian berupa *loss function* tiap epoch dan akurasi pada masing masing *weight* yang diuji,

```
Experiment: Weight Initialization

Training network with Zero initialization...
Epoch 1/20 - loss: 2.3023 - val_loss: 2.3004
Epoch 2/20 - loss: 2.3013 - val_loss: 2.2993
Epoch 3/20 - loss: 2.3009 - val_loss: 2.2987
Epoch 4/20 - loss: 2.3007 - val_loss: 2.2983
Epoch 5/20 - loss: 2.3005 - val_loss: 2.2980
Epoch 6/20 - loss: 2.3003 - val_loss: 2.2980
Epoch 7/20 - loss: 2.3002 - val_loss: 2.2978
Epoch 8/20 - loss: 2.3001 - val_loss: 2.2976
Epoch 9/20 - loss: 2.2999 - val_loss: 2.2974
Epoch 10/20 - loss: 2.2997 - val_loss: 2.2972
Epoch 11/20 - loss: 2.2995 - val_loss: 2.2970
Epoch 12/20 - loss: 2.2992 - val_loss: 2.2967
```

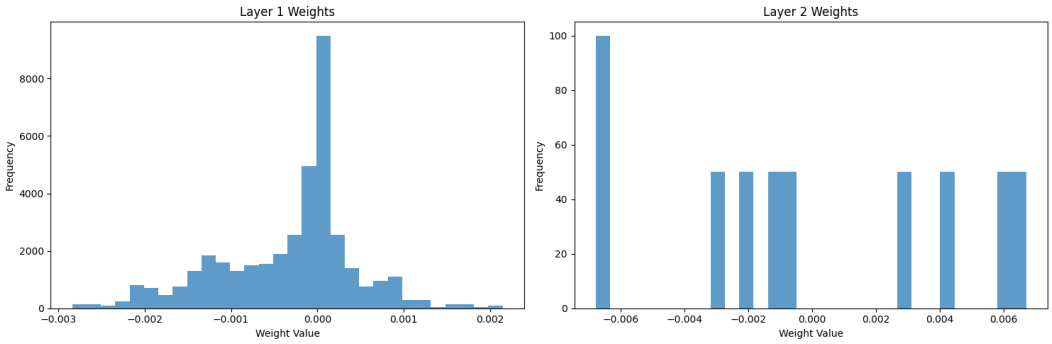
```
Epoch 13/20 - loss: 2.2990 - val_loss: 2.2964
Epoch 14/20 - loss: 2.2987 - val_loss: 2.2960
Epoch 15/20 - loss: 2.2984 - val_loss: 2.2959
Epoch 16/20 - loss: 2.2980 - val_loss: 2.2956
Epoch 17/20 - loss: 2.2976 - val_loss: 2.2950
Epoch 18/20 - loss: 2.2972 - val_loss: 2.2945
Epoch 19/20 - loss: 2.2967 - val_loss: 2.2938
Epoch 20/20 - loss: 2.2961 - val_loss: 2.2933
Zero initialization accuracy: 0.1200

Training network with Uniform initialization...
Epoch 1/20 - loss: 2.9903 - val_loss: 2.6847
Epoch 2/20 - loss: 2.6127 - val_loss: 2.4633
Epoch 3/20 - loss: 2.4363 - val_loss: 2.3265
Epoch 4/20 - loss: 2.3149 - val_loss: 2.2228
Epoch 5/20 - loss: 2.2185 - val_loss: 2.1365
Epoch 6/20 - loss: 2.1360 - val_loss: 2.0608
Epoch 7/20 - loss: 2.0622 - val_loss: 1.9921
Epoch 8/20 - loss: 1.9946 - val_loss: 1.9291
Epoch 9/20 - loss: 1.9320 - val_loss: 1.8706
Epoch 10/20 - loss: 1.8735 - val_loss: 1.8159
Epoch 11/20 - loss: 1.8186 - val_loss: 1.7646
Epoch 12/20 - loss: 1.7671 - val_loss: 1.7165
Epoch 13/20 - loss: 1.7185 - val_loss: 1.6713
Epoch 14/20 - loss: 1.6727 - val_loss: 1.6288
Epoch 15/20 - loss: 1.6294 - val_loss: 1.5885
Epoch 16/20 - loss: 1.5885 - val_loss: 1.5505
Epoch 17/20 - loss: 1.5496 - val_loss: 1.5145
Epoch 18/20 - loss: 1.5129 - val_loss: 1.4805
Epoch 19/20 - loss: 1.4779 - val_loss: 1.4483
Epoch 20/20 - loss: 1.4447 - val_loss: 1.4178
Uniform initialization accuracy: 0.5730

Training network with Normal initialization...
Epoch 1/20 - loss: 3.2550 - val_loss: 2.9400
Epoch 2/20 - loss: 2.7862 - val_loss: 2.6692
Epoch 3/20 - loss: 2.5764 - val_loss: 2.5031
Epoch 4/20 - loss: 2.4289 - val_loss: 2.3765
Epoch 5/20 - loss: 2.3116 - val_loss: 2.2727
Epoch 6/20 - loss: 2.2132 - val_loss: 2.1830
Epoch 7/20 - loss: 2.1273 - val_loss: 2.1032
Epoch 8/20 - loss: 2.0501 - val_loss: 2.0304
Epoch 9/20 - loss: 1.9797 - val_loss: 1.9633
Epoch 10/20 - loss: 1.9147 - val_loss: 1.9009
Epoch 11/20 - loss: 1.8545 - val_loss: 1.8428
Epoch 12/20 - loss: 1.7986 - val_loss: 1.7887
Epoch 13/20 - loss: 1.7465 - val_loss: 1.7382
Epoch 14/20 - loss: 1.6978 - val_loss: 1.6912
Epoch 15/20 - loss: 1.6523 - val_loss: 1.6471
Epoch 16/20 - loss: 1.6095 - val_loss: 1.6058
Epoch 17/20 - loss: 1.5694 - val_loss: 1.5669
Epoch 18/20 - loss: 1.5315 - val_loss: 1.5302
Epoch 19/20 - loss: 1.4959 - val_loss: 1.4958
Epoch 20/20 - loss: 1.4623 - val_loss: 1.4631
Normal initialization accuracy: 0.5300
```

Berikut adalah hasil dari distribusi akurasi dan *loss function* pada setiap model yang diujikan beserta distribusi *weight* pada masing masing layer model,



Gambar 2.2.4.1 Diagram distribusi *loss function* dan perbandingan antara akurasi dengan *weight initialization*

a) *Zero Initialization*

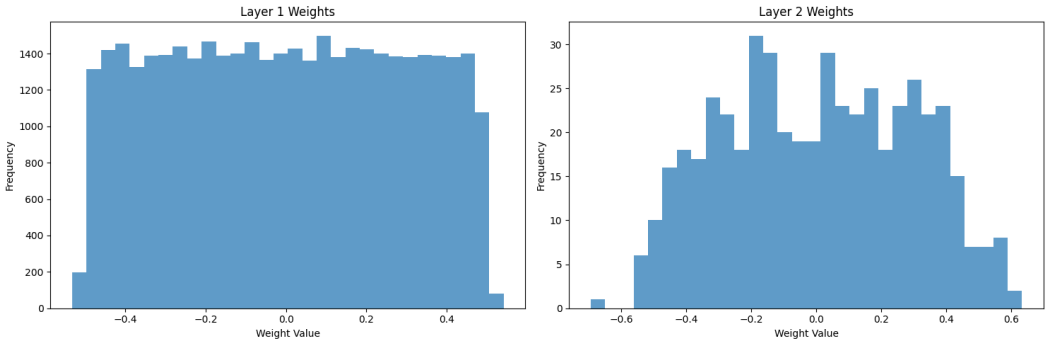


Gambar 2.2.4.2 Diagram distribusi *weight* pada *Zero initialization*

Model dengan *zero initialization* sepenuhnya gagal belajar. Hal ini tampak dari grafik *training* dan *validation loss* yang nyaris datar, tidak menunjukkan

penurunan yang signifikan dari epoch ke epoch. *Loss* hanya turun dari 2.3023 menjadi 2.2961 dalam 20 epoch. Akurasinya sangat rendah yaitu hanya 12.00%. Distribusi *weight* juga menunjukkan nilai-nilai yang terkonsentrasi di 0. Hal ini menunjukkan bahwa *zero initialization* menyebabkan semua neuron belajar hal yang sama, sehingga membuat jaringan tidak bisa belajar pola yang kompleks.

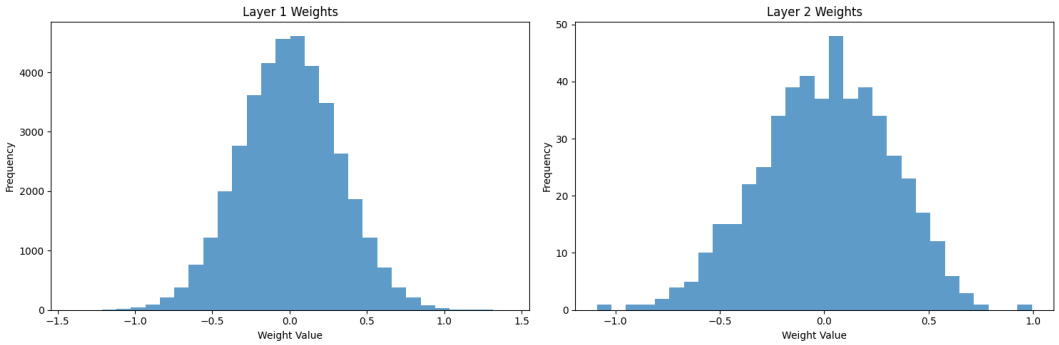
b) *Random Uniform Initialization*



Gambar 2.2.4.3 Diagram distribusi *weight* pada *Random Uniform Initialization*

Model dengan *random uniform initialization* memberikan hasil yang paling optimal bila dibandingkan dengan kedua model lainnya. *Training loss* turun tajam dari 2.99 ke 1.44, dan *validation loss* dari 2.68 ke 1.41. Model mencapai akurasi tertinggi yaitu 57.30% pada data uji. Distribusi *weight* tersebar merata dalam rentang yang luas di awal *training*, dan model berhasil menggunakan variasi *weight* tersebut untuk memperbarui parameter secara efektif.

c) *Random Normal Initialization*



Gambar 2.2.4.4 Diagram distribusi *weight* pada *Random Uniform Initialization*

Model dengan *random normal initialization* menghasilkan performa yang cukup baik, meskipun sedikit di bawah *uniform*. *Training loss* turun dari 3.25 ke 1.46, dan *validation loss* dari 2.94 ke 1.46. Akurasi akhir model adalah 53.00%. Meskipun lebih lambat dari *uniform* di awal training, model tetap mampu mempelajari representasi data yang baik. Distribusi *weight* berbentuk simetris dan cenderung stabil, cocok digunakan terutama jika jaringan lebih dalam.

Pengujian ini menunjukkan bahwa metode inisialisasi bobot sangat berpengaruh terhadap keberhasilan *training* FFNN. *Zero initialization* membuat model tidak belajar karena tidak ada asimetri antar neuron. *Uniform initialization* terbukti paling efektif untuk jaringan ini karena memberikan penyebaran bobot yang sehat dan stabil. *Normal initialization* juga bekerja cukup baik, tetapi sedikit lebih lambat dan rentan terhadap saturasi aktivasi *sigmoid*.

2.2.5 Perbandingan FFNN Model Custom dengan MLPClassifier

Pada pengujian kali ini akan dilakukan perbandingan hasil antara dua model yaitu FFNN *model custom* yang sudah di implementasi dan model `MLPClassifier` dari library `sklearn`. Berikut adalah spesifikasi yang digunakan pada masing masing model,

a) FFNN Custom

- `loss_function` : *Categorical Crossentrophy*
- `layer_sizes` : [784, 50, 10]
- `activation_function` : [“relu”, “softmax”]

Pada saat melakukan *training* berikut adalah masukan yang diberikan pada model FFNN yang diimplementasi:

- `batch_size` : 32
- `learning_rate` : 0.1
- `epochs` : 20
- `verbose` : 1

b) MLPClassifier

- `activation_function` : ["relu"]

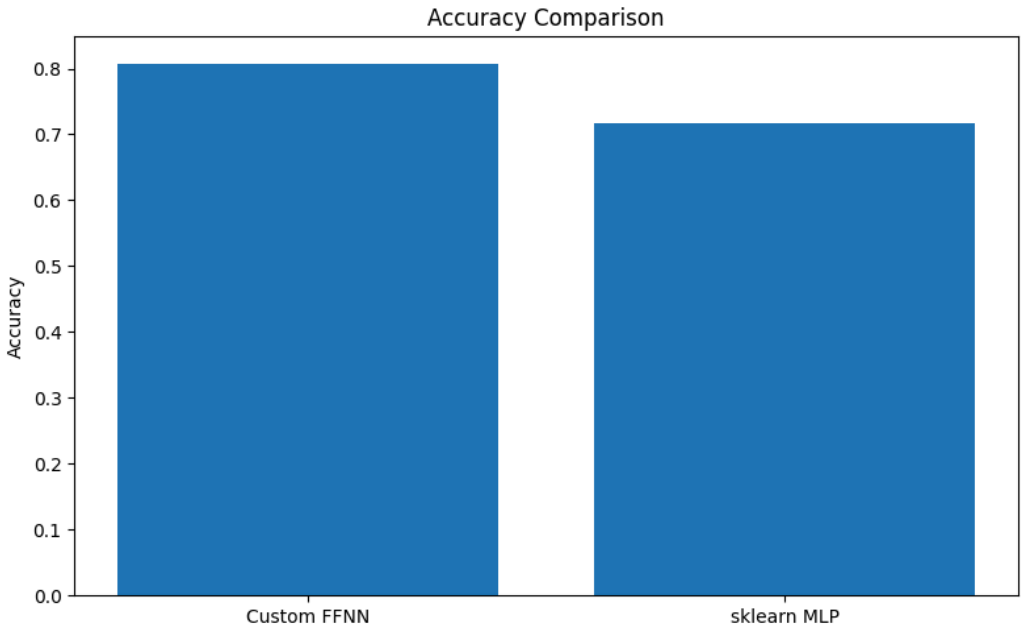
Pada saat melakukan *training* berikut adalah masukan yang diberikan pada model FFNN yang diimplementasi:

- `batch_size` : 32
- `learning_rate` : 0.1
- `epochs` : 20

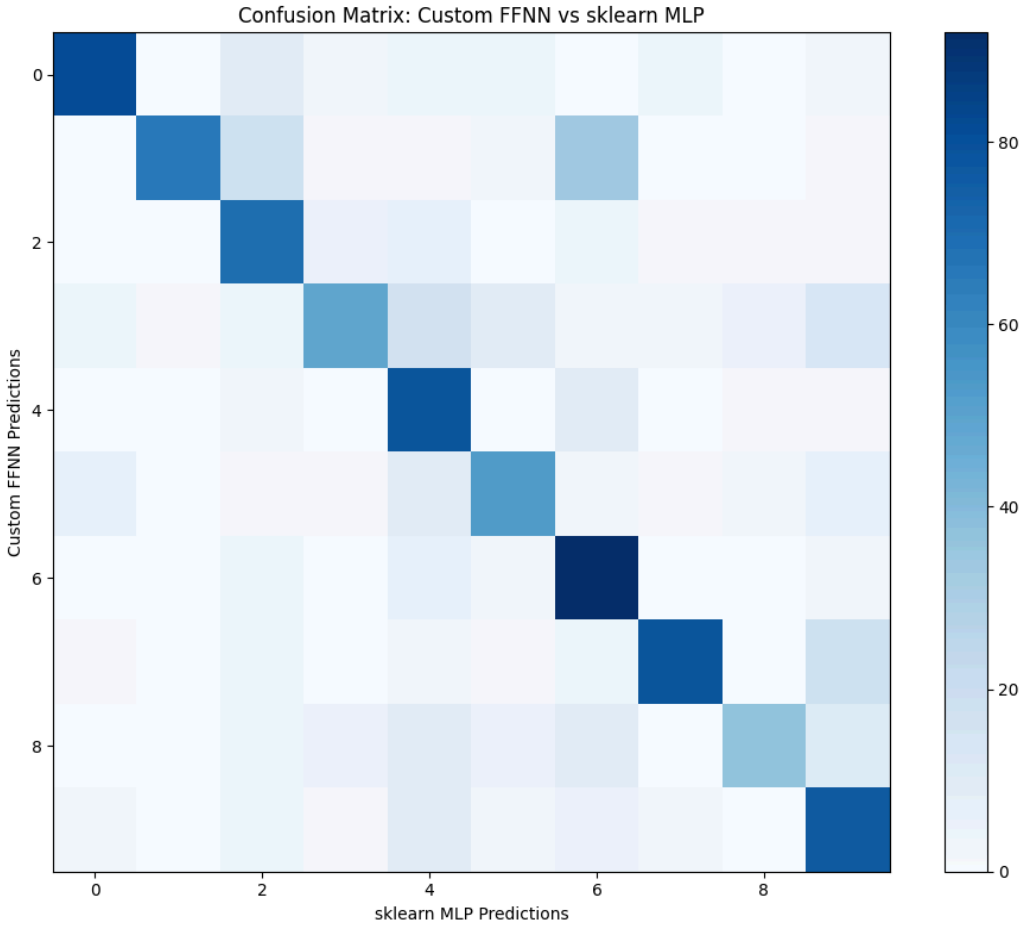
Berikut adalah hasil pengujian berupa *loss function* pada masing masing epoch model:

```
Experiment: Comparison with sklearn MLPClassifier
Training custom FFNN...
Epoch 1/20 - loss: 10.3765 - val_loss: 6.7469
Epoch 2/20 - loss: 4.9896 - val_loss: 4.2996
Epoch 3/20 - loss: 3.3493 - val_loss: 3.3158
Epoch 4/20 - loss: 2.5637 - val_loss: 2.7872
Epoch 5/20 - loss: 2.1005 - val_loss: 2.4792
Epoch 6/20 - loss: 1.7924 - val_loss: 2.2769
Epoch 7/20 - loss: 1.5833 - val_loss: 2.1069
Epoch 8/20 - loss: 1.4215 - val_loss: 2.0070
Epoch 9/20 - loss: 1.2913 - val_loss: 1.8939
Epoch 10/20 - loss: 1.1863 - val_loss: 1.8058
Epoch 11/20 - loss: 1.0999 - val_loss: 1.7618
Epoch 12/20 - loss: 1.0235 - val_loss: 1.6972
Epoch 13/20 - loss: 0.9549 - val_loss: 1.6434
Epoch 14/20 - loss: 0.8940 - val_loss: 1.6074
Epoch 15/20 - loss: 0.8435 - val_loss: 1.5665
Epoch 16/20 - loss: 0.7960 - val_loss: 1.5249
Epoch 17/20 - loss: 0.7546 - val_loss: 1.5250
Epoch 18/20 - loss: 0.7166 - val_loss: 1.4675
Epoch 19/20 - loss: 0.6809 - val_loss: 1.4647
Epoch 20/20 - loss: 0.6522 - val_loss: 1.4382
Custom FFNN accuracy: 0.8080
Model saved to ../model/ffnn

Training sklearn MLPClassifier...
sklearn MLPClassifier accuracy: 0.7160
```

Berikut adalah hasil distribusi akurasi dan *confusion matrix* masing masing model:



Gambar 2.2.5.1 Diagram distribusi akurasi antara *custom FFNN* dengan *sklearn MLP*



Gambar 2.2.5.2 Diagram *confusion matrix* antara *custom FFNN* dengan *sklearn MLP*

Berdasarkan grafik *Accuracy Comparison*, akurasi yang dicapai oleh `Custom FFNN` adalah 80.80%, sedangkan `sklearn MLPClassifier` hanya mencapai 71.60%. Ini menunjukkan bahwa model *FFNN custom* memberikan performa yang lebih baik, bahkan ketika menggunakan arsitektur dan konfigurasi yang identik. *Training loss* dan *validation loss* pada `Custom FFNN` menurun stabil dari epoch 1 hingga 20, menunjukkan bahwa model berhasil belajar secara bertahap. Tidak ada indikasi *overfitting* karena *validation loss* juga turun mengikuti *training loss*. Hal ini menandakan bahwa proses training stabil dan efektif. *Confusion matrix* yang diberikan membandingkan prediksi antara kedua model, sehingga menampilkan kesesuaian antar prediksi. Hasilnya, sebagian besar nilai diagonal cukup tinggi, artinya banyak prediksi dari kedua model berkorelasi.

Pengujian ini menunjukkan bahwa `Custom FFNN` yang dibuat dari awal mampu mengungguli `MLPClassifier` dari `sklearn` dalam hal akurasi dan kestabilan training, meskipun menggunakan arsitektur dan parameter training yang serupa.

BAB III
PENUTUP

3.1 Kesimpulan

Berdasarkan seluruh rangkaian pengujian yang dilakukan terhadap model *Feedforward Neural Network* (FFNN), dapat disimpulkan bahwa kinerja model sangat dipengaruhi oleh konfigurasi arsitektur dan parameter yang digunakan. Berikut adalah kesimpulan dari setiap pengujian,

- Penambahan *depth* dan *width* jaringan terbukti meningkatkan kemampuan representasi model, namun harus diimbangi dengan mekanisme pengendalian kompleksitas agar tidak terjadi *overfitting*.
- Dari segi *activation function*, `ReLU` dan `linear` menunjukkan hasil training yang lebih cepat dan akurasi lebih tinggi dibandingkan `sigmoid` dan `tanh`. Hal ini disebabkan oleh sifat `sigmoid` dan `tanh` yang cenderung menimbulkan masalah *vanishing gradient* pada jaringan yang dalam.
- Pada pengujian *learning rate*, diketahui bahwa pemilihan nilai yang terlalu kecil menyebabkan proses training lambat, sedangkan *learning rate* yang terlalu besar berisiko menyebabkan ketidakstabilan. Dalam pengujian ini, *learning rate* sebesar 0.1 terbukti memberikan hasil terbaik.
- Inisialisasi *weight* juga menjadi faktor penting, di mana metode *zero initialization* gagal memberikan *training* yang efektif akibat masalah simetri. Sebaliknya, *inisialisasi random uniform* dan random normal mampu memberikan distribusi bobot yang baik dan mendukung proses training.
- Terakhir, jika dibandingkan dengan model `MLPClassifier` dari `sklearn`, model FFNN yang diimplementasikan secara mandiri menunjukkan performa lebih baik dari sisi akurasi dan stabilitas *training*. Hal ini menunjukkan bahwa *model custom* dapat dioptimalkan untuk mencapai performa tinggi, asalkan desain dan parameter dikonfigurasi dengan tepat.

3.2 Saran

Sebagai saran dari pengujian FFNN ini, terdapat beberapa saran yang dapat dilakukan untuk pengembangan pengujian ini.

1. Untuk mengatasi *overfitting* yang mungkin terjadi pada model yang lebih kompleks, disarankan untuk menambahkan regularisasi seperti L2 *regularization* atau teknik *dropout*.
2. Implementasi strategi *early stopping* dan penyesuaian *learning rate* selama *training* bisa meningkatkan efisiensi *training*.

3.3 Pembagian Tugas

NIM	Nama	Tugas
13521028	Muhammad Zulfiansyah Bayu Pratama	Kode FFNN, Laporan
13521031	Fahrian Afdholi	Kode FFNN, Laporan
13521049	Brian Kheng	Kode FFNN, Laporan