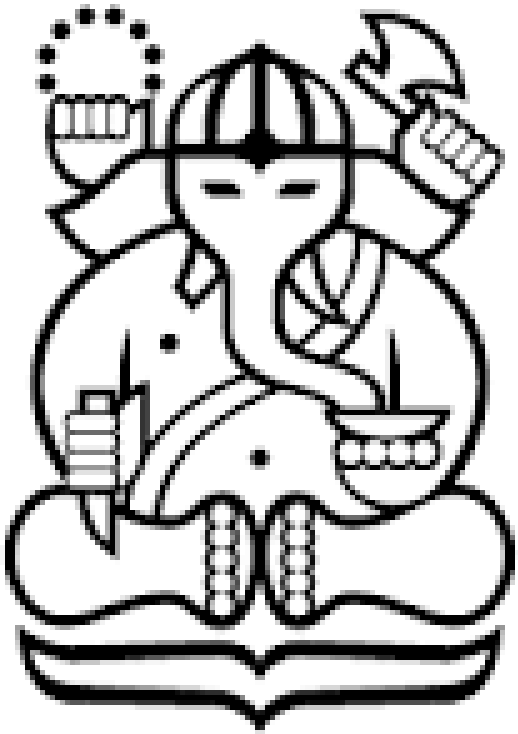
LAPORAN FEED FORWARD NEURAL NETWORK (FFNN)

IF3270 - Pembelajaran Mesin



Nama Kelompok:

1. Adril Putra Merin (13522068)
2. Marvin Scifo Y. Hutahaean (13522110)
3. Berto Richardo Togatorop (13522118)

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

2025

Daftar Isi

BAB I
DESKRIPSI PERSOALAN

Machine Learning (ML) adalah salah satu bentuk dari Artificial Intelligence (AI) yang melakukan pembelajaran terhadap dataset. Salah satu jenis dari Machine Learning adalah Supervised Learning. Neural Network adalah salah satu model yang merupakan Supervised Learning. Feed Forward Neural Network (FFNN) adalah salah satu jenis Neural Network yang paling sederhana dan banyak digunakan dalam berbagai bidang AI. Arsitektur FFNN terdiri dari tiga jenis *layer* yaitu *input layer*, *hidden layer*, *output layer*. Setiap neuron dalam satu lapisan terhubung secara langsung ke neuron di lapisan berikutnya melalui bobot yang disesuaikan selama proses pelatihan menggunakan algoritma seperti backpropagation. FFNN banyak digunakan untuk menyelesaikan berbagai jenis masalah antara lain:

1. Klasifikasi: FFNN dapat mengklasifikaskan data ke dalam beberapa kategori seperti identifikasi objek dalam gambar, dll
2. Regresi – FFNN dapat digunakan untuk memprediksi nilai kontinu, seperti perkiraan harga saham, analisis tren pasar, atau prediksi suhu berdasarkan data cuaca historis.
3. Pengenalan Pola – Jaringan ini dapat mengenali pola dalam data, yang berguna dalam aplikasi seperti pengenalan tulisan tangan dan analisis sentimen teks.
4. Sistem Rekomendasi – FFNN dapat membantu membangun sistem rekomendasi, seperti yang digunakan dalam e-commerce atau layanan streaming untuk menyarankan produk atau konten kepada pengguna berdasarkan preferensi mereka.
5. Pemrosesan Data Sensor – Dalam aplikasi industri dan IoT, FFNN digunakan untuk menganalisis data dari sensor dan membuat keputusan, seperti mendeteksi kerusakan mesin berdasarkan pola getaran atau suara.

Pada tugas ini, akan dibuat sebuah model FFNN *from scratch* menggunakan bahasa Python. FFNN ini diharapkan bisa menjalankan berbagai fungsi aktivasi saat melakukan pencarian di Neural Network tersebut yaitu sigmoid, ReLU, linear, hyperbolic tangent, dan softmax. FFNN ini juga akan menggunakan *loss function* yang terdiri dari MSE, Binary Cross-Entropy, dan Categorical Cross-Entropy.

BAB II
PEMBAHASAN

2.1. Penjelasan Implementasi

A. Deskripsi Kelas, Atribut, dan Metode

Pada tugas ini, kami membuat model *feed forward neural network* (FFNN) *from scratch* dengan hanya menggunakan library perhitungan, grafis, dan utilitas lain seperti *numpy, tqdm, networkx,* dan *adjustText*. Pada implementasinya, kami membuat beberapa kelas seperti kelas FFNN, ActivationFunctions, Layer, RMSNorm dan LossFunctions.

1. Kelas ActivationFunctions

   Kelas ActivationFunctions adalah sebuah kelas yang berisi fungsi aktivasi yang digunakan dalam proses pelatihan dan prediksi model. Secara umum, metode-metode pada kelas ini bersifat *static* sehingga tidak ada instantiasi objek untuk kelas ini. Berikut adalah metode-metode yang terdapat dalam kelas ini:

   - Linear Activation Function

     Fungsi identitas yang mengembalikan inputnya tanpa perubahan.

```
@staticmethod
    def linear(x, output=None, derivative=False):
        return x if not derivative else np.ones_like(x)
```

   - ReLU Activation Function

     ReLU digunakan untuk menghindari vanishing gradient pada jaringan dalam.

```
@staticmethod
    def relu(x, output=None, derivative=False):
        if derivative:
            return np.where(x > 0, 1, 0)
        return np.maximum(0, x)
```

   - Sigmoid Activation Function

     Sigmoid cocok untuk output antara 0 dan 1 tetapi rawan mengalami vanishing gradient.

```
@staticmethod
    def sigmoid(x, output=None, derivative=False):
        if derivative:
            return output * (1 - output)
        sig = 1 / (1 + np.exp(-x))
        return sig
```

   - Hyperbolic Tangent (tanh) Activation Function

Mengubah input menjadi nilai antara -1 dan 1.

```
@staticmethod
    def tanh(x, output=None, derivative=False):
        if derivative:
            return 1 - output**2
        t = np.tanh(x)
        return t
```

- Softmax Activation Function
  Softmax menghasilkan distribusi probabilitas dan memerlukan matriks Jacobian untuk perhitungan gradien.

```
@staticmethod
    def softmax(x, output=None, derivative=False):
        if derivative:
            batch_size, n_classes = output.shape
            jacobians = np.empty((batch_size, n_classes,
n_classes))
            for i in range(batch_size):
                s = output[i].reshape(-1, 1)
                jacobians[i] = np.diagflat(s) - np.dot(s,
s.T)
            return jacobians

        exp_x = np.exp(x - np.max(x, axis=-1,
keepdims=True))
        softmax_x = exp_x / np.sum(exp_x, axis=-1,
keepdims=True)
        return softmax_x
```

- Swish Activation Function
  Fungsi aktivasi yang merupakan kombinasi dari sigmoid dan input, yaitu $x * sigmoid(x)$.

```
@staticmethod
    def swish(x, output=None, derivative=False):
        if derivative:
            sig = 1 / (1 + np.exp(-x))
            return sig + x * sig * (1 - sig)
        sig = 1 / (1 + np.exp(-x))
        return x * sig
```

2. Kelas LossFunctions
   Kelas ini berisi beberapa fungsi loss yang digunakan dalam proses *backward propagation* dengan *gradient descent*. Fungsi-fungsi ini digunakan untuk mengukur seberapa baik prediksi model dibandingkan dengan nilai sebenarnya.
   - MSE

MSE digunakan untuk regresi dan menghitung selisih kuadrat antara prediksi dan nilai sebenarnya.

```
@staticmethod
def mse(y_true, y_pred, derivative=False):
    if derivative:
        return 2 * (y_pred - y_true) / y_true.size
    return np.mean((y_true - y_pred) ** 2)
```

- Binary Cross Entropy
  BCE digunakan untuk klasifikasi biner dan lebih sensitif terhadap perbedaan kecil dalam probabilitas.

```
@staticmethod
def binary_cross_entropy(y_true, y_pred,
derivative=False):
    epsilon = 1e-9
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    if derivative:
        return (y_pred - y_true) / (y_pred * (1 -
y_pred)) / y_true.size
    return -np.mean(y_true * np.log(y_pred) + (1 -
y_true) * np.log(1 - y_pred))
```

- Categorical Cross Entropy
  CCE digunakan dalam klasifikasi multi-kelas dengan output softmax.

```
@staticmethod
def categorical_cross_entropy(y_true, y_pred,
derivative=False):
    epsilon = 1e-9
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    if derivative:
        return -(y_true / y_pred) / y_true.shape[0]
    return -np.mean(np.sum(y_true * np.log(y_pred),
axis=1))
```

3. RMSNorm (Bonus)
   Kelas ini adalah implementasi dari *root mean square normalization* pada suatu layer. RMS Normalization sendiri adalah salah satu metode *layer normalization* yang melakukan normalisasi terhadap transformasi linear $W.x$ sebelum dijumlahkan dengan bias dan diberikan fungsi aktivasi. Secara matematis, bentuk umum RMSNorm adalah sebagai berikut (Zhang & Sennrich, 2019):

$$y = f\left(\frac{xW}{RMS(a)} \odot g + b\right)$$

Adapun RMS(a) dapat dituliskan sebagai berikut:

$$RMS(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$$

- Konstruktor kelas RMSNorm
  Pada konstruktor ini, kita menginisialisasi banyak dimensi (fitur) pada layer. Nantinya, kelas RMSNorm akan diinstansiasi oleh layer yang menggunakan normalisasi ini.

```
def __init__(self, dim, eps=1e-8):
    self.dim = dim
    self.eps = eps
    self.g = np.ones(dim)
    self.grad_g = np.zeros_like(self.g)
    self.cache = {}
```

- Forward Propagation pada RMSNorm
  Fungsi ini melakukan normalisasi terhadap Wx sebelum ditambahkan dengan bias dan diberikan fungsi aktivasi. Cara kerja fungsi ini adalah dengan mengkalkulasi persamaan berikut:

$$\widehat{a} = \frac{xW}{RMS(a)} \odot g$$

```
def forward(self, x):
    self.cache['x'] = x
    rms = np.sqrt(np.mean(x ** 2, axis=-1, keepdims=True)
+ self.eps)
    self.cache['rms'] = rms
    x_norm = x / rms
    self.cache['x_norm'] = x_norm
    out = x_norm * self.g
    return out
```

- Backward Propagation pada RMSNorm
  Fungsi ini melakukan backward propagation untuk melakukan pembelajaran terhadap g yang merupakan *learnable parameter* dalam RMSNorm. Sama halnya dengan *weight* dan *bias*, kita melakukan *backward propagation* menggunakan *gradient descent*. Diberikan loss function L, kita dapat menemukan gradiennya terhadap parameter g (Zhang & Sennrich, 2019):

$$\frac{\partial L}{\partial g} = \frac{\partial L}{\partial v}\frac{Wx}{RMS(a)}$$

  dengan v diberikan oleh persamaan berikut:

$$v = \frac{xW}{RMS(a)} \odot g + b$$

```
def backward(self, grad_output):
```

```
    x = self.cache['x']
    x_norm = self.cache['x_norm']
    rms = self.cache['rms']
    dim = self.dim
    self.grad_g = np.sum(grad_output * x_norm, axis=0)
    grad_x_norm = grad_output * self.g
    sum_x_grad = np.sum(x * grad_x_norm, axis=-1,
keepdims=True)
    dx = grad_x_norm / rms - (x * sum_x_grad) / (dim *
rms**3)
    return dx
```

4. Kelas Layer
   - Konstruktor Kelas Layer

     Konstruktor kelas layer akan menginisialisasikan bobot dan bias untuk setiap *neuron* pada layer ini berdasarkan parameter input. Kelas ini juga akan kelas RMSNorm jika layer ini dinormalisasikan menggunakan RMS.

```
def __init__(
        self,
        input_size: int,
        output_size: int,
        activation: Literal["linear", "relu", "sigmoid",
"tanh", "softmax", "swish", "elu"],
        weight_init: Literal["zero", "uniform", "normal",
"he", "xavier"],
        lower: float,
        upper: float,
        mean: float,
        variance: float,
        seed,
        use_rmsnorm: bool=True
    ):
        self.activation_name = activation
        self.activation = getattr(ActivationFunctions,
activation)

        if seed is not None:
            np.random.seed(seed)

        if weight_init == "zero":
            self.weights = np.zeros((input_size,
output_size))
            self.biases = np.zeros((1, output_size))
        elif weight_init == "uniform":
            self.weights = np.random.uniform(lower,
upper, (input_size, output_size))
            self.biases = np.random.uniform(lower, upper,
```

```
(1, output_size))
        elif weight_init == "normal":
            self.weights = np.random.normal(
                mean, np.sqrt(variance), (input_size,
output_size)
            )
            self.biases = np.random.normal(mean,
np.sqrt(variance), (1, output_size))
        elif weight_init == "he":
            self.weights = np.random.normal(
                0, np.sqrt(2 / input_size), (input_size,
output_size)
            )
            self.biases = np.zeros((1, output_size))
        elif weight_init == "xavier":
            limit = np.sqrt(6 / (input_size +
output_size))
            self.weights = np.random.uniform(-limit,
limit, (input_size, output_size))
            self.biases = np.zeros((1, output_size))
        else:
            self.weights = np.zeros((input_size,
output_size))
            self.biases = np.zeros((1, output_size))

        self.input = None
        self.output = None
        self.z = None
        self.grad_weights = np.zeros((input_size,
output_size))
        self.grad_biases = np.zeros((1, output_size))

        self.use_rmsnorm = use_rmsnorm
        if self.use_rmsnorm:
            self.rmsnorm = RMSNorm(output_size)
        else:
            self.rmsnorm = None
```

- Forward Propagation Kelas Layer

  Forward propagation yang dilakukan pada kelas layer, menggunakan forward propagation standard dimana output yang dihasilkan adalah:

  $$y = f(xW + b)$$

  Jika menggunakan RMS normalization, output yang dihasilkan adalah sebagai berikut:

  $$y = f\left(\frac{xW}{RMS(a)} \odot g + b\right)$$

  dimana $\odot$ merupakan *hadamard product* atau *element-wise multiplication*.

```
def forward(self, x):
        self.input = x
        z = x @ self.weights
        if self.use_rmsnorm:
            z = self.rmsnorm.forward(z)
        z = z + self.biases
        self.z = z
        self.output = self.activation(z)
        return self.output
```

- Backward Propagation Kelas Layer

  Fungsi backward melakukan backward propagation terhadap layer ini. Fungsi ini menerima parameter input berupa *grad_output* yang merupakan turunan *loss function* terhadap output dari layer ini:

$$\frac{\partial L}{\partial a^{(k)}}$$

  dimana $a^{(k)}$ adalah output dari layer ke-k. Secara matematis, dapat dituliskan:

$$a^{(k)} = f\left(x^{(k)}W^{(k)} + b^{(k)}\right)$$

  dengan f adalah fungsi aktivasi yang digunakan.

  Untuk memudahkan perhitungan, kita memisahkan cara kalkulasi backward propagation layer saat ini berdasarkan apakah fungsi aktivasi yang digunakan berupa *softmax* atau tidak. Hal ini dilakukan karena turunan softmax berupa matriks jacobian untuk setiap sampel pada input batch sehingga proses perhitungan tidak bisa di vektorisasi atau dijadikan matriks (harus dalam bentuk tensor). Adapun tujuan akhir dari fungsi ini adalah untuk menghitung

  1. Gradien bobot: $\frac{\partial L}{\partial W^{(k)}}$
  2. Gradien bias: $\frac{\partial L}{\partial b^{(k)}}$

  - **Gradien Bobot**

    Secara umum, proses backward propagation menggunakan prinsip turunan berantai:

$$\frac{\partial L}{\partial W^{(k)}} = \frac{\partial L}{\partial a^{(k)}}\frac{\partial a^{(k)}}{\partial z^{(k)}}\frac{\partial z^{(k)}}{\partial W^{(k)}}$$

    Perhatikan bahwa, $\frac{\partial L}{\partial a^{(k)}}$ merupakan masukan dari fungsi backward, yaitu *grad_output*. Jika layer saat ini merupakan layer output, maka $\frac{\partial L}{\partial a^{(k)}}$ merupakan turunan dari *loss function* yang digunakan terhadap output. Jika bukan merupakan layer output, maka persamaan akan menjadi:

$$\frac{\partial L}{\partial a^{(k)}} = \frac{\partial L}{\partial a^{(k+1)}}\frac{\partial a^{(k+1)}}{\partial z^{(k+1)}}\frac{\partial z^{(k+1)}}{\partial a^{(k)}} = \frac{\partial L}{\partial z^{(k+1)}}\frac{\partial z^{(k+1)}}{\partial a^{(k)}}$$

Misalkan $\delta^{(k)} = \frac{\partial L}{\partial z^{(k)}}$ maka persamaan diatas menjadi:

$$\frac{\partial L}{\partial a^{(k)}} = \delta^{(k+1)} \cdot \left(W^{(k+1)}\right)^T$$

Karena itu, hasil perhitungan $\delta^{(k+1)} \cdot \left(W^{(k+1)}\right)^T$ untuk setiap layer akan dikembalikan dan dijadikan parameter input *grad_output* untuk proses *backward propagation* layer sebelumnya.

Selanjutnya, tinjau bahwa $\frac{\partial a^{(k)}}{\partial z^{(k)}} = f'\left(z^{(k)}\right)$ yang menyatakan turunan fungsi aktivasi terhadap $z^{(k)}$ dan $\frac{\partial z^{(k)}}{\partial W^{(k)}} = x^{(k)}$. Dengan melakukan vektorisasi dan melakukan perhitungan dalam bentuk matriks, kita memperoleh gradien bobot sebagai berikut:

$$\delta^{(k)} = \frac{\partial L}{\partial z^{(k)}} = \frac{\partial L}{\partial a^{(k)}} \odot f'\left(z^{(k)}\right)$$

$$\Rightarrow \frac{\partial L}{\partial W^{(k)}} = \left(x^{(k)}\right)^T \cdot \delta^{(k)}$$

- **Gradien Bias**

Untuk gradien bias, kita juga menggunakan aturan rantai dan menghasilkan persamaan berikut:

$$\frac{\partial L}{\partial b^{(k)}} = \frac{\partial L}{\partial a^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial b^{(k)}} = \delta^{(k)} = \sum_{i=1}^{n} \delta_i^{(k)}$$

Perhatikan bahwa kita mengambil jumlah dari tiap sampel i pada batch untuk setiap fitur output layer j sehingga $\frac{\partial L}{\partial b^{(k)}}$ akan menghasilkan matriks $(1, k)$ dimana k adalah jumlah fitur output pada layer tersebut. Jadi, perhitungan $\delta_i^{(k)}$ untuk sampel i pada batch diberikan oleh:

$$\delta_i^{(k)} = \frac{\partial L}{\partial z_i^{(k)}} = \frac{\partial L}{\partial a_i^{(k)}} \frac{\partial a_i^{(k)}}{\partial z_i^{(k)}} = \frac{\partial L}{\partial a_i^{(k)}} \bullet J_i$$

dengan J_i adalah *jacobian matrix* untuk sampel ke-i pada batch.

- **Fungsi Aktivasi Softmax**

Secara umum, proses perhitungan pada fungsi aktivasi softmax tidak jauh berbeda dengan yang lainnya. Perbedaan utama terletak pada perbedaan cara perhitungan $\delta^{(k)}$. Untuk softmax, kita tidak dapat menggunakan operasi *element-wise* multiplication:

```
grad = grad_output * activation_grad
```

Hal ini karena turunan softmax berada dalam bentuk *jacobian matrix* yang saling bergantung antar output. Akibatnya, proses perhitungan harus dilakukan satu per satu untuk setiap sampel. Alternatif lain adalah

menggunakan tensor, tetapi lebih kompleks. Penulis memilih untuk melakukan perhitungan untuk setiap sampel dalam *batch*.

```
def backward(self, grad_output):
        if self.activation_name == "softmax":
            jacobians = self.activation(self.z,
self.output, derivative=True)
            grad_list = []
            for i in range(self.output.shape[0]):
                grad_i = np.dot(grad_output[i : i + 1],
jacobians[i])
                grad_list.append(grad_i)
            grad = np.concatenate(grad_list, axis=0)
        else:
            activation_grad = self.activation(self.z,
self.output, derivative=True)
            grad = grad_output * activation_grad

        # calculate the bias grad
        self.grad_biases = np.sum(grad, axis=0,
keepdims=True)

        # calculate the weight grad and g grad if
applicable
        if self.use_rmsnorm:
            grad = self.rmsnorm.backward(grad)

        self.grad_weights = self.input.T @ grad
        return grad @ self.weights.T
```

5. FFNN
   - Konstruktor Kelas FFNN

```
def __init__(
    self,
    layer_sizes: List[int],
    activations: List[Literal["linear", "relu",
"sigmoid", "tanh", "softmax", "swish", "elu"]],
    weight_init: Literal["zero", "uniform", "normal",
"he", "xavier"]="uniform",
    lower=-0.5,
    upper=0.5,
    mean=0,
    variance=1,
    seed=None,
    use_rmsnorm=False
):
    self.layers = []
    self.layer_sizes = layer_sizes
```

```
    self.activations = activations
    self.num_classes = layer_sizes[len(layer_sizes) - 1]
    for i in range(len(layer_sizes) - 1):
        self.layers.append(
            Layer(
                layer_sizes[i],
                layer_sizes[i + 1],
                activations[i],
                weight_init,
                lower,
                upper,
                mean,
                variance,
                seed,
                use_rmsnorm
            )
        )
```

- Method _forward pada Kelas FFNN

```
def __forward(self, x):
    for layer in self.layers:
        x = layer.forward(x)
    return x
```

- Method _backward pada Kelas FFNN

```
def __backward(self, loss_grad):
    for layer in reversed(self.layers):
        loss_grad = layer.backward(loss_grad)
```

- Methode Update Weight pada Kelas FFNN

```
def __update_weights(
    self, learning_rate: float, regularization=None,
reg_lambda=0.0
):
    for layer in self.layers:
        if regularization == "l1":
            reg_l1 = reg_lambda * np.sign(layer.weights)
            layer.weights -= learning_rate *
(layer.grad_weights + reg_l1)
            layer.biases -= learning_rate *
layer.grad_biases
            if layer.use_rmsnorm:
                layer.rmsnorm.g -= learning_rate *
layer.rmsnorm.grad_g
        elif regularization == "l2":
            reg_l2 = reg_lambda * 2 * layer.weights
```

```
            layer.weights -= learning_rate *
(layer.grad_weights + reg_l2)
            layer.biases -= learning_rate *
layer.grad_biases
            if layer.use_rmsnorm:
                layer.rmsnorm.g -= learning_rate *
layer.rmsnorm.grad_g
        else:
            layer.weights -= learning_rate *
layer.grad_weights
            layer.biases -= learning_rate *
layer.grad_biases
            if layer.use_rmsnorm:
                layer.rmsnorm.g -= learning_rate *
layer.rmsnorm.grad_g
```

- Metode Train Kelas FFNN

```
def train(
      self,
      x_train,
      y_train,
      x_val,
      y_val,
      loss_function:Literal["mse",
"binary_cross_entropy",
"categorical_cross_entropy"]="mse",
      learning_rate=0.01,
      epochs=100,
      batch_size=32,
      verbose=1,
      regularization:Union[None, Literal["l1",
"l2"]]=None,
      reg_lambda=0.0,
  ):
      loss_fn = getattr(LossFunctions, loss_function)
      history = {
          "training_loss": [],
          "val_loss": [],
      }
      num_samples = x_train.shape[0]

      if loss_function == "categorical_cross_entropy":
          if len(y_train.shape) == 1:
              y_train = np.eye(self.num_classes)[y_train]
          if len(y_val.shape) == 1:
              y_val = np.eye(self.num_classes)[y_val]
      else:
          if len(y_train.shape) == 1:
              y_train = y_train.reshape(len(y_train), 1)
```

```
            if len(y_val.shape) == 1:
                y_val = y_val.reshape(len(y_val), 1)

        for epoch in range(epochs):
            # shuffle the dataset for each epoch
            indices = np.arange(num_samples)
            np.random.shuffle(indices)
            x_shuffled, y_shuffled = x_train[indices],
y_train[indices]

            # train process
            epoch_loss = 0
            num_batches = (num_samples + batch_size - 1) //
batch_size
            if verbose:
                pbar = tqdm(
                    total=num_batches, desc=f"Epoch
{epoch+1}/{epochs}", unit="batch"
                )
            for start in range(0, num_samples, batch_size):
                end = min(start + batch_size, num_samples -
1)
                X_batch, y_batch = x_shuffled[start:end],
y_shuffled[start:end]

                preds = self.__forward(X_batch)
                loss = loss_fn(y_batch, preds)
                loss_grad = loss_fn(y_batch, preds,
derivative=True)

                # add regularization to the loss function
                if regularization == "l1":
                    for layer in self.layers:
                        loss += reg_lambda *
(np.sum(np.abs(layer.weights)))
                elif regularization == "l2":
                    for layer in self.layers:
                        loss += reg_lambda *
(np.sum(layer.weights**2))

                self.__backward(loss_grad)
                self.__update_weights(learning_rate,
regularization, reg_lambda)
                epoch_loss += loss
                if verbose:
                    pbar.update(1)
            if verbose:
                pbar.close()

            epoch_loss /= num_samples // batch_size
            history["training_loss"].append(epoch_loss)
```

```
            # compute validation loss
            val_preds = self.__forward(x_val)
            val_loss = loss_fn(y_val, val_preds)
            history["val_loss"].append(val_loss)

            if verbose:
                print(
                    f"Epoch {epoch+1}/{epochs} - Training
Loss: {epoch_loss:.4f} - Validation Loss: {val_loss:.4f}"
                )
        return history["training_loss"],
history["val_loss"]
```

- Method Predict Kelas FFNN

```
def predict_class(self, x):
        return np.argmax(self.__forward(x), axis=1)
```

- Method Save Model Kelas FFNN

```
def save_model(self, filename: str):
    with open(filename, "wb") as f:
        pickle.dump(self, f)
```

- Method Load Model pada Kelas FFNN

```
@staticmethod
def load_model(filename: str):
    with open(filename, "rb") as f:
        return pickle.load(f)
```

- Method Plot Weight Distribution pada Kelas FFNN

```
def plot_weight_distribution(self, layers):
    for i in layers:
        weights = self.layers[i].weights.flatten()
        plt.hist(weights, bins=30, alpha=0.7,
label=f"Layer {i}")
    plt.legend()
    plt.title("Weight Distribution")
    plt.xlabel("Weight Values")
    plt.ylabel("Frequency")
    plt.show()
```

- Method Display Graph pada Kelas FFNN

```

```

```
def display_graph(self):
    G = nx.DiGraph()
    pos = {}
    node_cnt = 0
    layer_x_offset = 1

    for layer_idx, num_neurons in
enumerate(self.layer_sizes):
        for neuron_idx in range(num_neurons):
            G.add_node(node_cnt, Layer=layer_idx)
            pos[node_cnt] = (layer_x_offset * layer_idx,
-neuron_idx)
            node_cnt += 1

    node_cnt = 0
    for layer_idx in range(len(self.layers)):
        num_neurons_curr = self.layer_sizes[layer_idx]
        num_neurons_next = self.layer_sizes[layer_idx +
1]
        for i in range(num_neurons_curr):
            for j in range(num_neurons_next):
                weight =
self.layers[layer_idx].weights[i, j]
                G.add_edge(
                    node_cnt + i, node_cnt +
num_neurons_curr + j, weight=weight
                )
        node_cnt += num_neurons_curr

    plt.figure(figsize=(8, 6))
    labels = {node: f"N{node}" for node in G.nodes()}
    edge_labels = {(i, j): f'{d["weight"]:.2f}' for i, j,
d in G.edges(data=True)}

    nx.draw(
        G,
        pos,
        with_labels=True,
        labels=labels,
        node_size=700,
        node_color="lightblue",
    )
    text_labels = []
    for (i, j), label in edge_labels.items():
        x, y = (pos[i][0] + pos[j][0]) / 2, (pos[i][1] +
pos[j][1]) / 2  # Midpoint
        text_labels.append(plt.text(x, y, label,
fontsize=8))
    adjust_text(
        text_labels,
    )
```

```
    plt.title("Feed Forward Neural Network
Visualization")
    plt.show()
```

B. Penjelasan Forward Propagation

Forward propagation adalah proses di dalam sebuah neural network yang melibatkan *input layer* yang menerima data untuk diproses ke *layer* selanjutnya. *Input data* akan diteruskan ke *hidden layer*. *Hidden layer* ini memiliki lebih dari 1 input dan untuk setiap *hidden layer*, input akan terkena fungsi aktivasi sebelum dikirim ke *layer* selanjutnya. Terakhir, *input data* akan mendapatkan data di *output layer* untuk mengaplikasikan fungsi aktivasi untuk terakhir kalinya supaya prediksi bisa dilakukan.

Pada implementasi forward propagation, setiap layer melakukan langkah berikut:

```
def __forward(self, x):
    for layer in self.layers:
        x = layer.forward(x)
    return
```

Di dalam kelas Layer, forward propagation dilakukan dengan:

- Menyimpan input: Input yang diterima oleh lapisan disimpan dalam atribut self.input untuk digunakan nanti dalam proses backpropagation.
- Kalkulasi z (pra-aktivasi): Input dikalikan dengan bobot (self.weights) menggunakan operasi matriks (x @ self.weights). Jika normalisasi RMS digunakan (self.use_rmsnorm), hasil tersebut dinormalisasi menggunakan metode self.rmsnorm.forward(z). Selanjutnya, bias (self.biases) ditambahkan ke hasil tersebut.
- Penerapan fungsi aktivasi: Hasil pra-aktivasi (z) kemudian dilewatkan melalui fungsi aktivasi yang telah ditentukan (self.activation) untuk menghasilkan output lapisan. Output ini disimpan dalam atribut self.output dan juga dikembalikan sebagai output dari metode ini.

Di dalam kelas rmsnorm, forward propagation dilakukan dengan:

- Penyimpanan Input: Input x disimpan dalam self.cache untuk digunakan nanti pada tahap backward.
- Perhitungan RMS: Menghitung rata-rata kuadrat dari elemen-elemen x pada sumbu terakhir, menambahkan nilai epsilon kecil (self.eps) untuk mencegah pembagian oleh nol, mengambil akar kuadrat dari hasil tersebut untuk mendapatkan nilai RMS.
- Normalisasi Input: Membagi setiap elemen x dengan nilai RMS yang telah dihitung, menghasilkan x_norm.

- Skalasi dengan Parameter g: Mengalikan x_norm dengan parameter skala self.g yang dapat dilatih, menghasilkan output akhir.
- Semua variabel perantara (x, rms, x_norm) disimpan dalam self.cache untuk digunakan pada proses backward.

C. Penjelasan Backward Propagation dan Weight Update

Algoritma ini digunakan untuk menghitung gradien dari fungsi loss terhadap bobot jaringan dengan menerapkan aturan rantai (chain rule) dalam diferensiasi. Hasil dari backpropagation kemudian digunakan untuk memperbarui bobot dan bias dengan metode optimisasi seperti Gradient Descent. Backward propagation bertujuan untuk mengurangi kesalahan yang terjadi pada proses prediksi dengan cara menyebarkan error dari output ke setiap layer sebelumnya.

1. Proses backward propagation

   Pada implementasi backward propagation, setiap layer melakukan langkah berikut:

```
def __backward(self, loss_grad):
    for layer in reversed(self.layers):
        loss_grad = layer.backward(loss_grad)
```

Di dalam kelas Layer, method backward propagation dilakukan dengan:

- Perhitungan Gradien Aktivasi: Jika fungsi aktivasi yang digunakan adalah softmax, maka gradien dihitung dengan membentuk matriks Jacobian dari output softmax dan mengalikan dengan gradien output. Hal ini dilakukan untuk memastikan bahwa gradien yang dihitung sesuai dengan sifat fungsi softmax dalam konteks backpropagation. Sebaliknya, jika fungsi aktivasi bukan softmax, gradien dihitung dengan mengalikan gradien output dengan turunan fungsi aktivasi yang sesuai.
- Perhitungan Gradien Bias: Gradien bias dihitung dengan menjumlahkan gradien di sepanjang axis 0 (baris) dan menjaga dimensi agar konsisten. Hal ini memastikan bahwa gradien bias dapat digunakan untuk memperbarui nilai bias selama proses optimisasi.
- Backward RMSNorm (Opsional): Jika layer menggunakan RMSNorm, gradien akan diproses melalui metode backward dari objek RMSNorm terkait. RMSNorm adalah teknik normalisasi yang dapat membantu meningkatkan stabilitas dan kinerja pelatihan jaringan saraf.
- Perhitungan Gradien Bobot: Gradien bobot dihitung dengan melakukan perkalian matriks antara input (transpos) dan gradien yang telah dihitung sebelumnya. Hal ini sesuai dengan aturan rantai dalam kalkulus, yang

memungkinkan kita untuk menghitung bagaimana perubahan pada bobot akan mempengaruhi fungsi loss.

- Propagasi Error ke Layer Sebelumnya: Fungsi backward mengembalikan hasil perkalian antara gradien dan transpos bobot. Nilai ini akan digunakan sebagai gradien input untuk layer sebelumnya, memungkinkan proses backpropagation untuk terus berjalan mundur melalui jaringan.

Di dalam kelas rmsnorm, backward propagation dilakukan dengan:

- Pengambilan Variabel dari Cache: Mengambil x, x_norm, dan rms dari self.cache.
- Perhitungan Gradien terhadap g: Menghitung gradien dari loss terhadap g dengan mengalikan grad_output dengan x_norm dan menjumlahkannya sepanjang sumbu batch. Hasilnya disimpan di self.grad_g.
- Perhitungan Gradien terhadap x_norm: Menghitung gradien dari loss terhadap x_norm dengan mengalikan grad_output dengan g.
- Perhitungan Gradien terhadap x:
  - Menghitung jumlah elemen-wise dari x dikalikan dengan grad_x_norm sepanjang sumbu terakhir.
  - Menghitung gradien akhir dx dengan rumus:
    - Membagi grad_x_norm dengan rms
    - Mengurangkan hasil perkalian x dengan sum_x_grad, dibagi dengan $dim \times rms^3$

2. Proses update weight

Setelah gradien dihitung, bobot dan bias diperbarui untuk meminimalkan error. Proses pembaruan ini melibatkan beberapa langkah. Implementasinya adalah sebagai berikut:

- Tanpa Regularisasi: Bobot diperbarui dengan mengurangi produk antara learning rate dan gradien bobot. Bias diperbarui dengan mengurangi produk antara learning rate dan gradien bias.
- Dengan Regularisasi L1 (Lasso): Selain gradien bobot, ditambahkan juga komponen yang proporsional terhadap tanda dari bobot, dikalikan dengan faktor regularisasi (reg_lambda).
- Dengan Regularisasi L2 (Ridge): Ditambahkan komponen yang proporsional terhadap bobot itu sendiri, dikalikan dengan faktor regularisasi.
- Pembaruan Parameter RMSNorm (Opsional): Jika layer menggunakan RMSNorm, parameter 'g' juga diperbarui dengan mengurangi produk antara learning rate dan gradien 'g' yang dihitung selama backpropagation.

2.2 Hasil Pengujian

1. Pengaruh Depth Hidden Layer

A. Deskripsi Pengujian

Kedalaman hidden layer adalah salah satu parameter dalam model Feed Forward Neural Network (FFNN) yang menentukan jumlah hidden layer dalam jaringan. Semakin dalam jaringan, semakin kompleks fitur yang dapat dipelajari oleh model, karena setiap lapisan bertanggung jawab untuk mengekstraksi pola dari lapisan sebelumnya. Jaringan dengan sedikit hidden layer cenderung lebih mudah dilatih dan membutuhkan lebih sedikit data, tetapi mungkin tidak cukup kuat untuk menangkap pola yang kompleks. Sebaliknya, jaringan yang lebih dalam dapat mempelajari representasi fitur yang lebih abstrak dan kompleks, tetapi juga lebih rentan terhadap overfitting, membutuhkan lebih banyak data, serta lebih sulit untuk dikonvergensikan.

Untuk memastikan bahwa variabel yang terlibat hanya kedalaman, kami membuat parameter lainnya sama untuk setiap variasi:

- **Aktivasi hidden layer** : hyperbolic tangent (tanh)
- **Banyak neuron** : 64
- **Inisialisasi bobot** : uniform
- **Loss function** : categorical cross entropy
- **RMSNorm** : false
- **Learning rate** : 0.1
- **Epoch** : 20
- **Regularization** : None
- **Batch Size** : 64

Adapun untuk variasi yang digunakan adalah sebagai berikut:

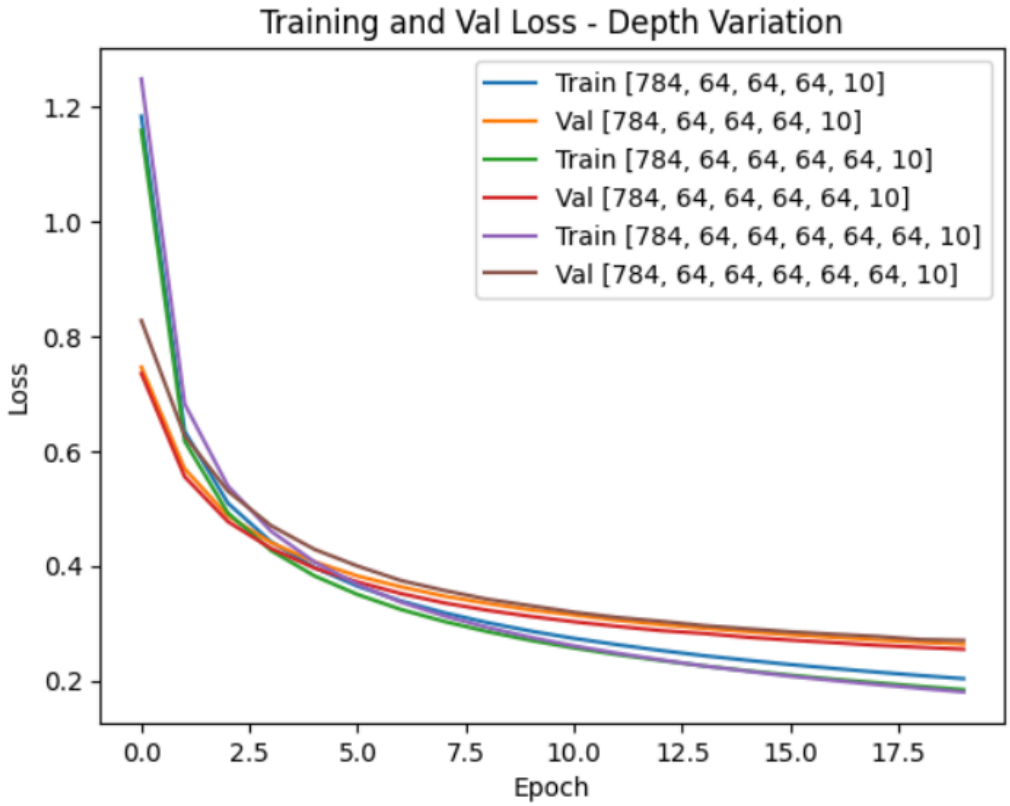
- **Variasi 1** : [784, 64, 64, 64, 10]
- **Variasi 2** : [784, 64, 64, 64, 64, 10]
- **Variasi 3** : [784, 64, 64, 64, 64, 64, 10]

B. Hasil Pengujian

Variasi	Akurasi	Log Loss
Variasi 1	0.9225	0.2637966101283638

Variasi 2	0.9236428571428571	0.2547632397029418
Variasi 3	0.9197857142857143	0.27023476257005286

Tabel 2.2.1 Tabel hasil pengujian terhadap kedalaman *hidden layer*



Gambar 2.2.1 Plot perbandingan training loss dan validation loss

Berdasarkan Tabel 2.2.1, terlihat bahwa akurasi mengalami sedikit penurunan seiring bertambahnya kedalaman hidden layer, sementara log loss cenderung meningkat. Meskipun demikian, perbedaannya tidak terlalu signifikan antara variasi tersebut.

Penurunan akurasi dan peningkatan log loss ini menunjukkan bahwa penambahan kedalaman layer tidak selalu meningkatkan performa model. Hal ini dapat terjadi karena layer yang lebih dalam membutuhkan lebih banyak data dan iterasi untuk mencapai konvergensi yang optimal. Selain itu, jaringan yang lebih dalam berpotensi mengalami overfitting. Meskipun demikian, kita tidak dapat menyimpulkan bahwa semakin sedikit layer maka akurasi juga lebih baik.

Berdasarkan Gambar 2.2.1, semua variasi menunjukkan penurunan training loss seiring bertambahnya epoch. Hal ini menunjukkan bahwa model berhasil melakukan pembelajaran dari data. Selain itu, dapat dilihat bahwa model dengan kedalaman variasi 3 mengalami peningkatan validation loss setelah beberapa epoch. Hal ini mengindikasikan

adanya *overfitting*. Di sisi lain, model dengan kedalaman yang lebih rendah memiliki validation loss yang cenderung lebih stabil dan lebih rendah.

2. Pengaruh Width Hidden Layer

A. Deskripsi Pengujian

Width adalah salah satu parameter penting dalam *neural network* yang menyatakan banyak *neuron* dalam satu *hidden layer*. Banyaknya neuron suatu layer menandakan banyaknya kapasitas representasi model dalam menangkap pola dari data. Semakin besar width, semakin banyak fitur kompleks yang dapat dipelajari model. Namun, jika neuron terlalu banyak, model berisiko mengalami overfitting. Selain itu, proses komputasi juga semakin mahal. Di sisi lain, model yang memiliki terlalu sedikit neuron mungkin tidak mampu menangkap pola kompleks dari suatu dataset sehingga pemilihan *width* yang optimal harus mempertimbangkan kompleksitas model dan kemampuan generalisasi.

Untuk memastikan bahwa variabel yang terlibat hanya *width*, kami membuat parameter lainnya sama untuk setiap variasi:

- **Aktivasi hidden layer** : hyperbolic tangent (tanh)
- **Kedalaman** : 3
- **Inisialisasi bobot** : uniform
- **Loss function** : categorical cross entropy
- **RMSNorm** : false
- **Learning rate** : 0.1
- **Epoch** : 20
- **Regularization** : None
- **Batch Size** : 64

Adapun untuk variasi yang digunakan adalah sebagai berikut:

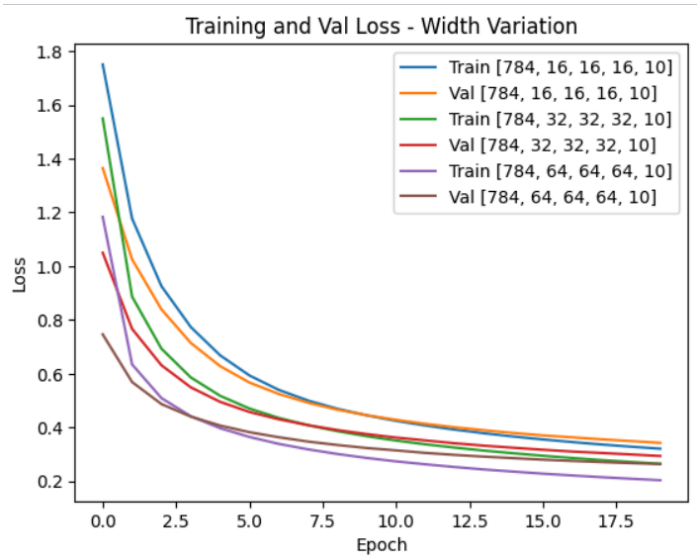
- **Variasi 1** : [784, 16, 16, 16, 10]
- **Variasi 2** : [784, 32, 32, 32, 10]
- **Variasi 3** : [784, 64, 64, 64, 10]

B. Hasil Pengujian

Variasi	Akurasi	Log Loss
Variasi 1	0.9023571428571429	0.3426240893971599

Variasi 2	0.9137142857142857	0.29366597573139797
Variasi 3	0.9225	0.2637966101283638

Tabel 2.2.2 Tabel hasil pengujian terhadap kedalaman *hidden layer*



Gambar 2.2.2 Plot perbandingan training loss dan validation loss

Berdasarkan Tabel 2.2.2, dapat dilihat bahwa semakin banyak neuron dalam hidden layer, semakin tinggi akurasi dan semakin rendah log loss. Hal ini mendukung pernyataan sebelumnya bahwa semakin besar *width*, kemampuan model untuk mempelajari pola kompleks juga meningkat. Hal ini terlihat jelas dari perbedaan akurasi dan log loss yang sangat signifikan.

Gambar 2.2.2 menunjukkan bahwa variasi ke-3 mengalami penurunan log loss yang cenderung stabil. Variasi ke-2 juga mengalami penurunan yang cukup stabil, tetapi mengalami sedikit kenaikan pada validation loss di akhir epoch. Di sisi lain, variasi ke-1 malah mengalami kenaikan yang cukup tinggi, baik di training loss maupun di validation loss setelah mengalami penurunan di epoch awal. Hal ini mungkin karena variasi ke-1 mengalami *underfitting*, dimana model tidak memiliki kapasitas yang cukup untuk menangkap pola dalam data sehingga performa menjadi turun. Variasi ke-2 menunjukkan performa yang lebih baik dengan penurunan log loss yang lebih stabil, tetapi sedikit peningkatan pada validation loss di akhir epoch dapat mengindikasikan mulai terjadinya overfitting. Sementara itu, variasi ke-3 dengan jumlah neuron terbanyak menunjukkan penurunan log loss yang paling stabil, yang menandakan bahwa model mampu belajar dengan baik dan menangkap pola yang lebih kompleks tanpa mengalami overfitting yang signifikan. Namun, kita tetap harus mempertimbangkan bahwa menambah jumlah neuron secara berlebihan juga dapat meningkatkan biaya komputasi dan risiko overfitting.

3. Pengaruh Fungsi Aktivasi Hidden Layer

A. Deskripsi Pengujian

Fungsi aktivasi berfungsi untuk memperkenalkan non-linearitas ke dalam model, sehingga jaringan dapat mempelajari pola yang kompleks dan tidak hanya terbatas pada hubungan linear. Fungsi ini diaplikasikan pada setiap neuron di hidden layer dan output layer untuk menentukan apakah suatu neuron akan aktif atau tidak. Pada tugas besar ini terdapat 7 jenis fungsi aktivasi yang diimplementasikan. Berikut adalah fungsi aktivasi yang digunakan:

1. Sigmoid
2. ReLU
3. Linear
4. Hyperbolic Tangent
5. Swish
6. ELU

Di pengujian ini, kami akan mencoba melakukan prediksi dari sebuah dataset menggunakan FFNN dengan semua fungsi aktivasi diatas, kecuali softmax. Untuk memastikan bahwa variabel yang terlibat hanya *activation function*, kami membuat parameter lainnya sama untuk setiap variasi:

- **Banyak neuron** : 64
- **Kedalaman** : 3
- **Inisialisasi bobot** : uniform
- **Loss function** : categorical cross entropy
- **RMSNorm** : false
- **Learning rate** : 0.1
- **Epoch** : 20
- **Regularization** : None
- **Batch Size** : 64

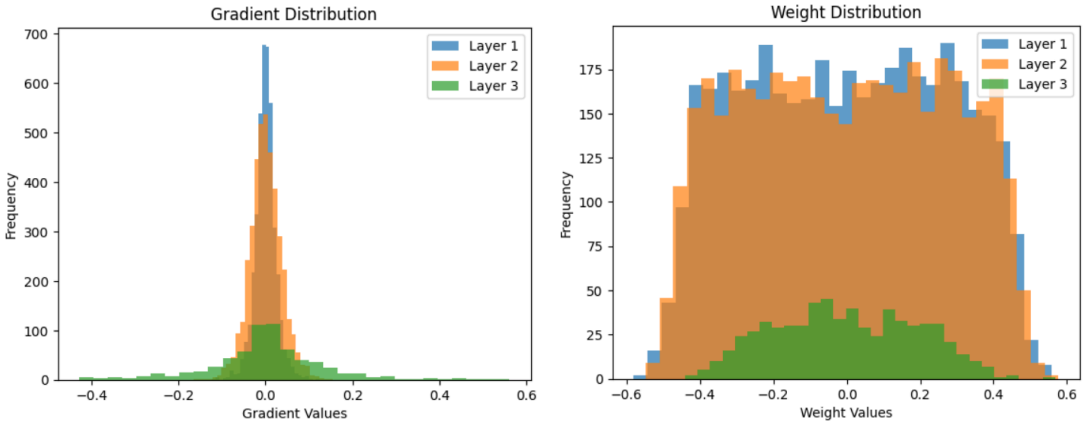
Adapun untuk variasi yang digunakan adalah sebagai berikut:

- **Variasi 1** : Linear
- **Variasi 2** : ReLU
- **Variasi 3** : Sigmoid
- **Variasi 4** : Tanh
- **Variasi 5** : Swish
- **Variasi 6** : ELU

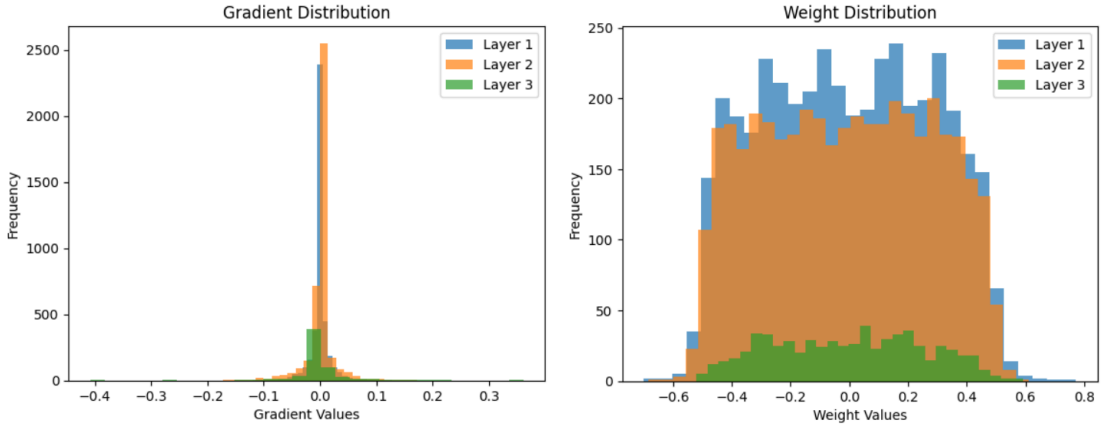
B. Hasil Pengujian

Variasi	Akurasi	Log Loss
Variasi 1	0.9022142857142857	0.35518232338891986
Variasi 2	0.9408571428571428	0.21055937630683624
Variasi 3	0.8841428571428571	0.4121789748556495
Variasi 4	0.9225	0.2637966101283638
Variasi 5	0.9379285714285714	0.21548986146553345
Variasi 6	0.9403571428571429	0.20157426750406082

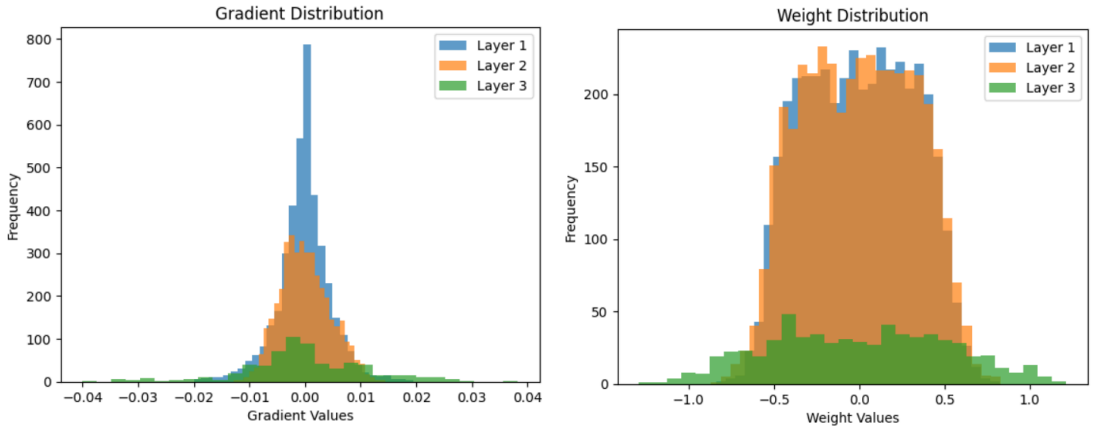
Tabel 2.2.3 Tabel hasil pengujian terhadap fungsi aktivasi



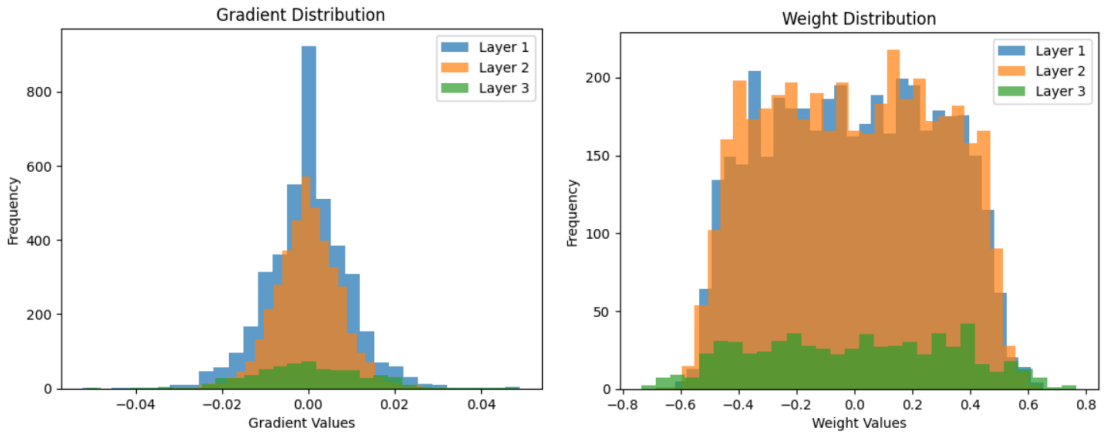
Gambar 2.2.3 Weight dan Gradient Distribution dari fungsi aktivasi linear



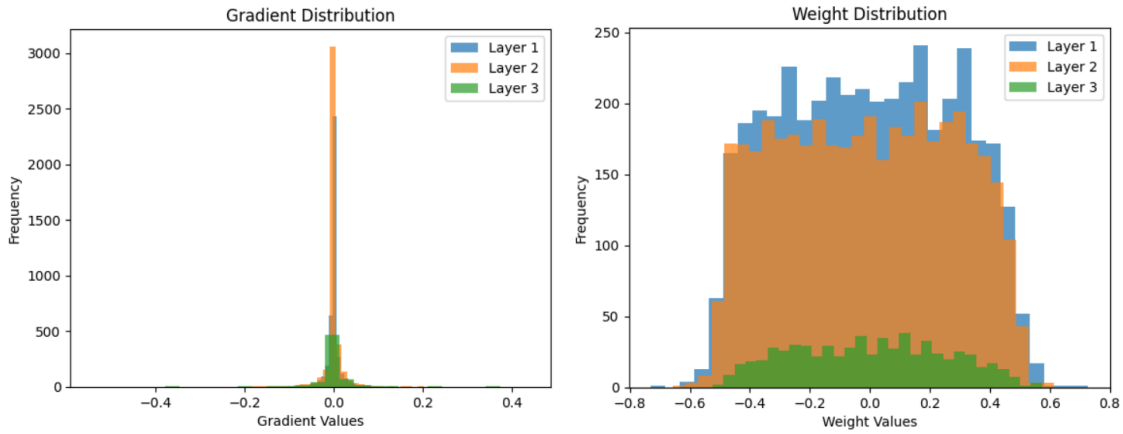
Gambar 2.2.4 Weight dan Gradient Distribution dari fungsi aktivasi ReLU



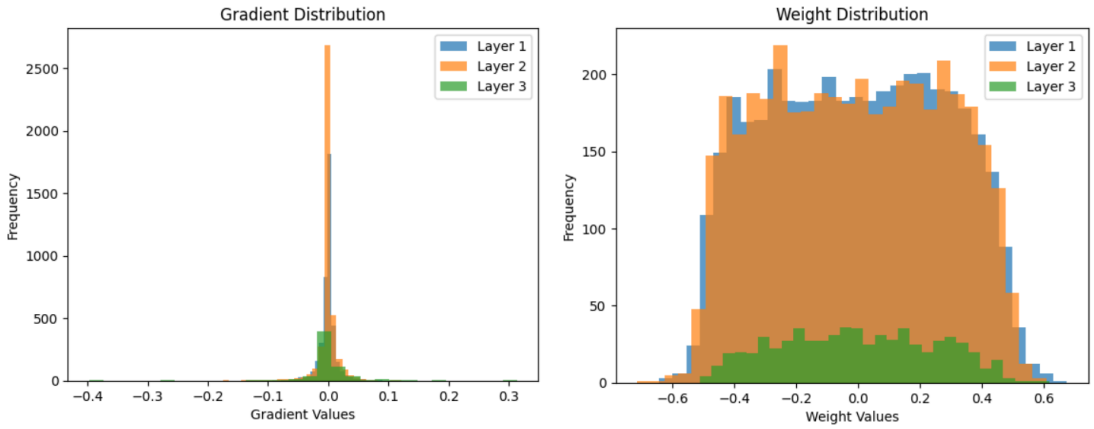
Gambar 2.2.5 Weight dan Gradient Distribution dari fungsi aktivasi sigmoid



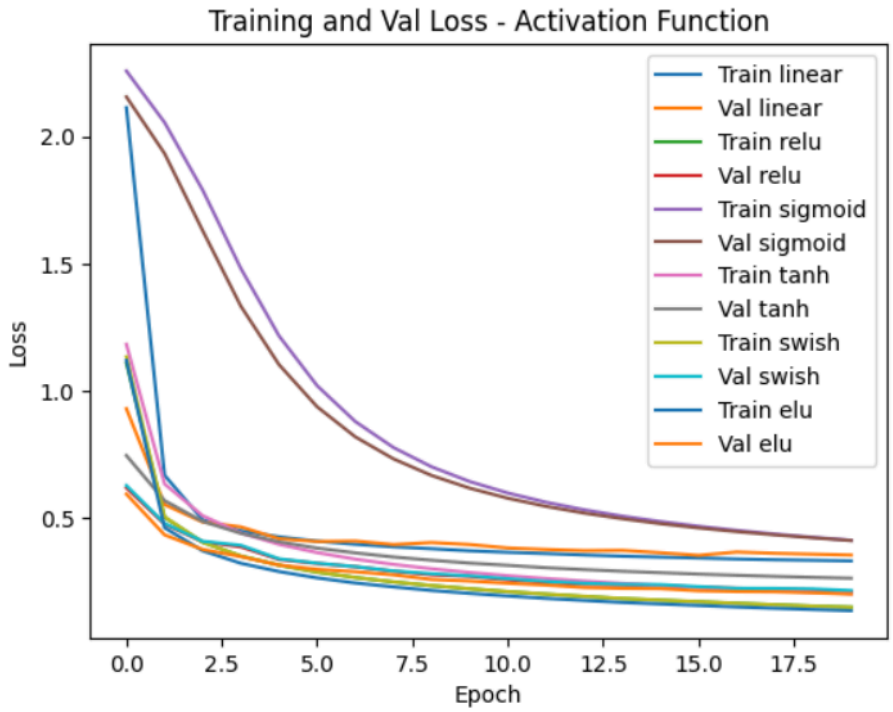
Gambar 2.2.6 Weight dan Gradient Distribution dari fungsi aktivasi tanh



Gambar 2.2.7 Weight dan Gradient Distribution dari fungsi aktivasi swish



Gambar 2.2.8 Weight dan Gradient Distribution dari fungsi aktivasi ELU



Gambar 2.2.9 Plot perbandingan training loss dan validation loss pada varian fungsi aktivasi

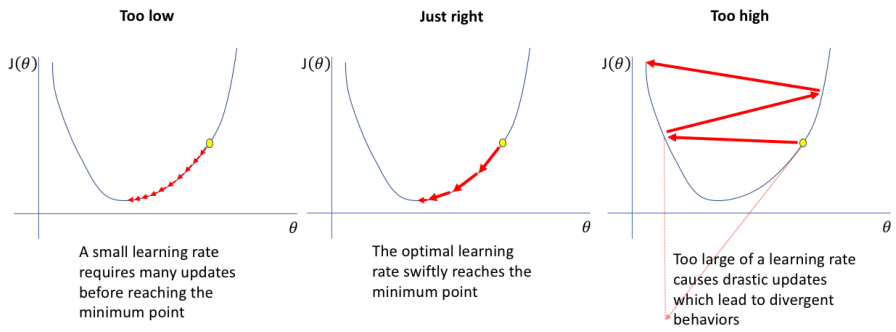
Tabel 2.2.3 menunjukkan bahwa fungsi aktivasi reLU memiliki akurasi yang paling tinggi dan ELU memiliki nilai log loss yang paling kecil. Ini menunjukkan bahwa kedua fungsi aktivasi inilah yang paling cocok dalam FFNN. Tingkat keakuratan dari fungsi aktivasi yang lain tidaklah jauh yaitu ELU -> Swish -> Tanh -> Linear -> Sigmoid sedangkan tingkat log loss dari fungsi yang lain adalah reLU -> Swish -> Tanh -> Linear -> Sigmoid. Dengan itu, fungsi aktivasi yang terbaik dalam pembuatan FFNN ini dengan dataset dan parameter yang diberikan adalah reLU atau ELU.

Pada Gambar 2.2.3 - Gambar 2.2.8, diberikan gambar plot weight dan gradient distribution dari model FFNN dengan berbagai fungsi aktivasi. Semua fungsi aktivasi memiliki distribusi gradien dan bobot yang normal namun persebarannya berbeda. Pada Sigmoid dan Tanh terdapat vanishing gradient problem yang artinya FFNN melakukan pembelajaran yang sangat lama sedangkan pada linear, Swish, reLU, dan ELU mereka tidak memiliki hal tersebut sehingga menunjukkan bahwa layer yang lebih dalam bisa belajar. Pada Gambar 2.2.9, ditunjukkan nilai loss yang agak mirip jalan grafnya antar semua fungsi aktivasinya sehingga menunjukan model belajar dengan seharusnya setelah beberapa epoch. Swish, ELU, dan ReLU memiliki loss yang paling kecil sehingga generalisasi model mereka lebih baik dan jarak yang pendek antara training dan validation loss mengimplikasikan *low overfitting*. Tidak seperti sigmoid dan tanh yang memiliki kurva loss yang lebih tinggi dan pada sigmoid memiliki validation loss yang paling tinggi yang menunjukkan *vanishing gradient issue* pada fungsi. Pada fungsi linear, performanya biasa-biasa saja dan terlalu sederhana.

4. Pengaruh Learning Rate

A. Deskripsi Pengujian

Learning rate adalah salah satu parameter penting dalam *neural network* yang menentukan seberapa besar perubahan *weight* model setiap kali terjadi pembaruan selama proses training. Learning rate yang terlalu besar dapat menyebabkan model tidak konvergen atau melompati solusi optimal, sedangkan learning rate yang terlalu kecil dapat menyebabkan proses training menjadi lambat dan model terjebak dalam lokal minimum.



Gambar 2.2.10 Ilustrasi learning rate

Untuk memastikan bahwa variabel yang terlibat hanya *learning rate*, kami membuat parameter lainnya sama untuk setiap variasi:

- **Aktivasi hidden layer** : hyperbolic tangent (tanh)
- **Kedalaman** : 3
- **Width** : 64
- **Inisialisasi bobot** : uniform
- **Loss function** : categorical cross entropy
- **RMSNorm** : false
- **Epoch** : 20
- **Regularization** : None
- **Batch Size** : 64

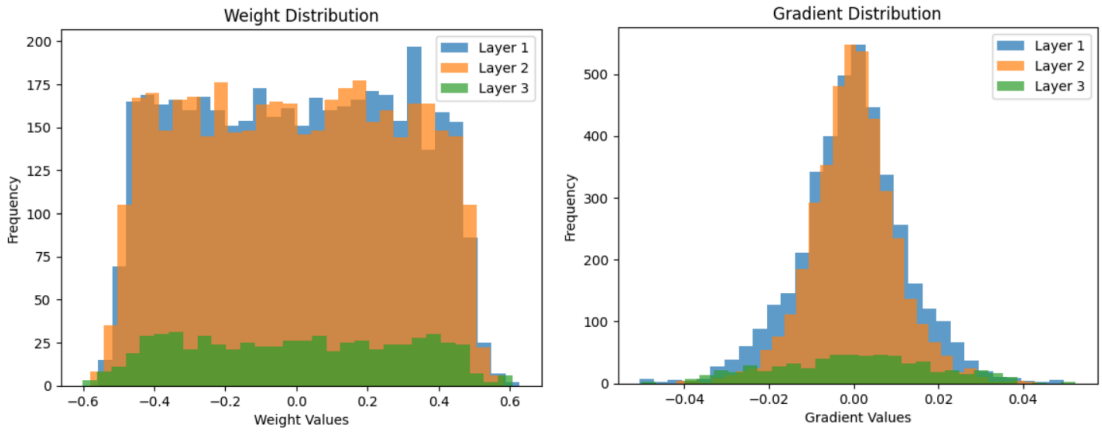
Adapun untuk variasi learning rate yang digunakan adalah sebagai berikut:

- **Variasi 1** : 0.001
- **Variasi 2** : 0.01
- **Variasi 3** : 0.1

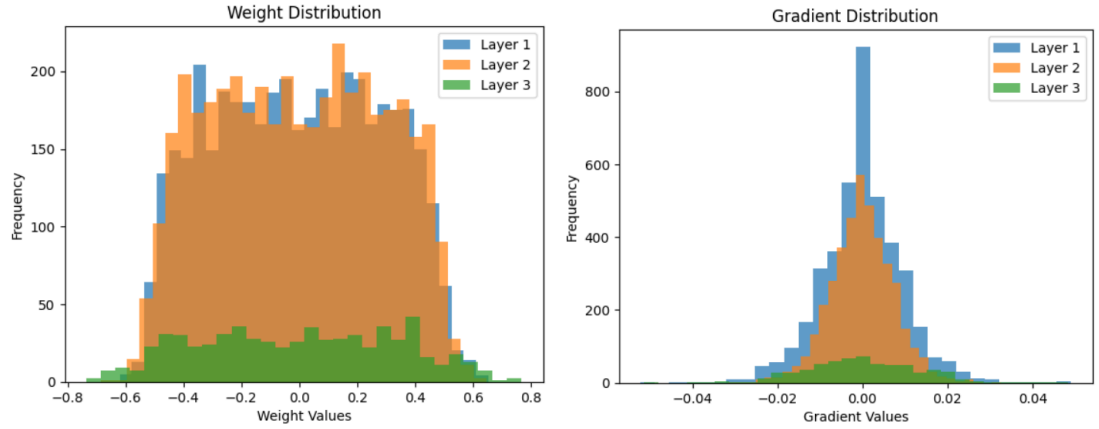
B. Hasil Pengujian

Variasi	Akurasi	Log Loss
Variasi 1	0.8289285714285715	0.5655044418814473
Variasi 2	0.9225	0.2637966101283638
Variasi 3	0.9527142857142857	0.18627334577135668

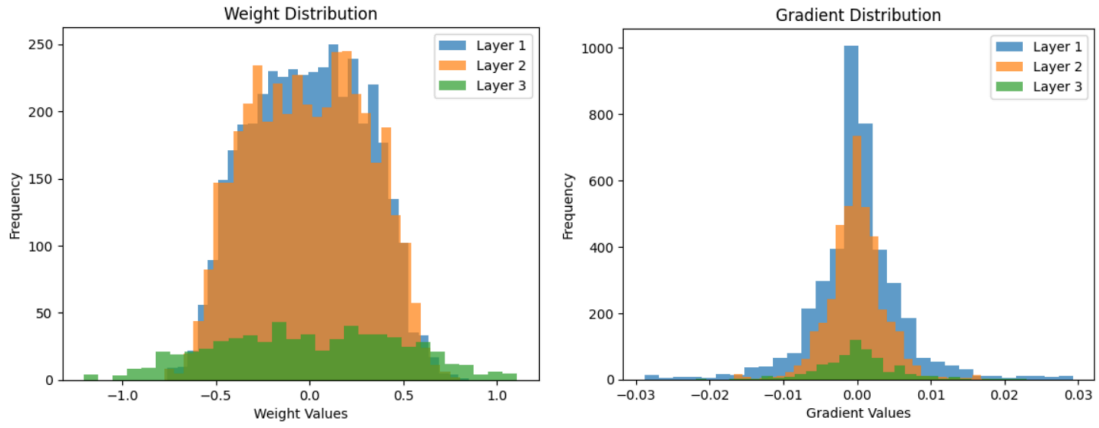
Tabel 2.2.4 Tabel hasil pengujian terhadap kedalaman *hidden layer*



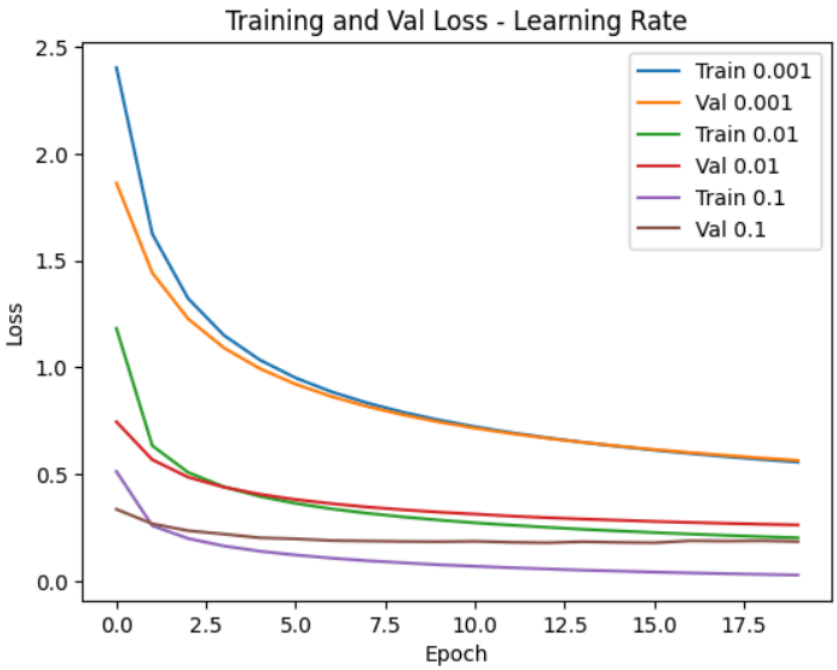
Gambar 2.2.11 Plot weight dan gradient distribution dari variasi 1



Gambar 2.2.12 Plot weight dan gradient distribution dari variasi 2



Gambar 2.2.13 Plot weight dan gradient distribution dari variasi 3



Gambar 2.2.14 Plot perbandingan training loss dan validation loss pada varian learning rate

Tabel 2.2.4 menunjukkan bahwa semakin semakin tinggi learning rate, semakin tinggi akurasi dan semakin rendah log loss. Hal ini mungkin disebabkan karena dalam konteks model dan dataset yang digunakan, learning rate 0.1 adalah learning rate yang lebih mendekati optimal sehingga dapat mendekati titik minimum dengan lebih cepat dalam 20 epochs. Variasi 0.01 dan 0.001 memiliki nilai akurasi dan log loss yang sebenarnya tidak jauh berbeda dari 0.1, tetapi varian ini membutuhkan lebih banyak update atau epoch sebelum mencapai titik minimum.

Selain itu, distribusi bobot pada learning rate 0.1, sebagaimana ditunjukkan oleh Gambar 2.2.13, lebih stabil dan menyerupai distribusi normal. Hal ini mengindikasikan bahwa model berhasil melakukan update bobot dengan baik dan mencapai keseimbangan optimal antara generalisasi dan training. Di sisi lain, distribusi weight pada learning rate lainnya masih menyerupai distribusi uniform.

Secara umum, dapat disimpulkan bahwa learning rate 0.1 memiliki performa yang paling baik dibandingkan learning rate lain dalam pengujian ini.

5. Pengaruh Inisialisasi Bobot

A. Deskripsi Pengujian

Inisialisasi bobot merupakan tahap awal dalam proses training *neural network* yang berperan penting dalam menentukan konvergensi dan stabilitas training. Inisialisasi yang baik dapat membantu model mencapai konvergensi dengan lebih cepat, sedangkan inisialisasi yang buruk dapat menyebabkan model terjebak dalam suatu kondisi sehingga membutuhkan waktu lebih banyak menuju konvergensi.

Untuk memastikan bahwa hanya inisialisasi bobot yang bervariasi, parameter lain dalam model tetap dijaga konstan, yaitu:

- **Aktivasi hidden layer** : hyperbolic tangent (tanh)
- **Kedalaman** : 3
- **Banyak neuron** : 64
- **Learning rate** : 0.1
- **Loss function** : categorical cross entropy
- **RMSNorm** : true
- **Epoch** : 20
- **Regularization** : None
- **Batch Size** : 64

Adapun metode inisialisasi bobot yang digunakan dalam pengujian ini adalah sebagai berikut:

1. **Zero Initialization** => semua bobot diinisialisasi dengan nilai nol
2. **Uniform Initialization** => bobot diinisialisasi secara acak dalam range tertentu dengan distribusi uniform
3. **Normal Initialization** => bobot diambil dari distribusi normal dengan mean dan varians tertentu
4. **Xavier Initialization** => Bobot diinisialisasi dengan skala tertentu dengan mempertimbangkan jumlah neuron pada layer sekarang dan layer sebelumnya. Secara matematis, Xavier Initialization dapat dituliskan sebagai berikut untuk bobot W_{ij}

$$W_{ij} \sim U\left[-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}\right]$$

   Dengan U adalah distribusi uniform dan fan_{in} adalah ukuran layer sebelumnya dan fan_{out} adalah ukuran layer saat ini.
5. **He Initialization** => Metode initialization yang mempertimbangkan non-linearitas dari fungsi aktivasi seperti ReLU. Secara matematis, dapat dituliskan sebagai:

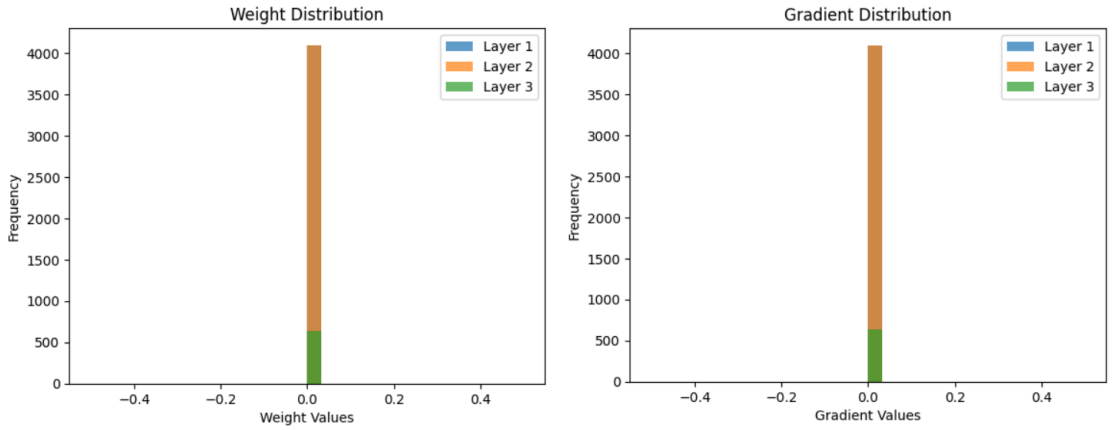
$$w_l \sim \mathcal{N}(0, 2/n_l)$$

Dengan N adalah distribusi normal dengan mean 0 dan varians sebesar $\frac{2}{n_i}$.

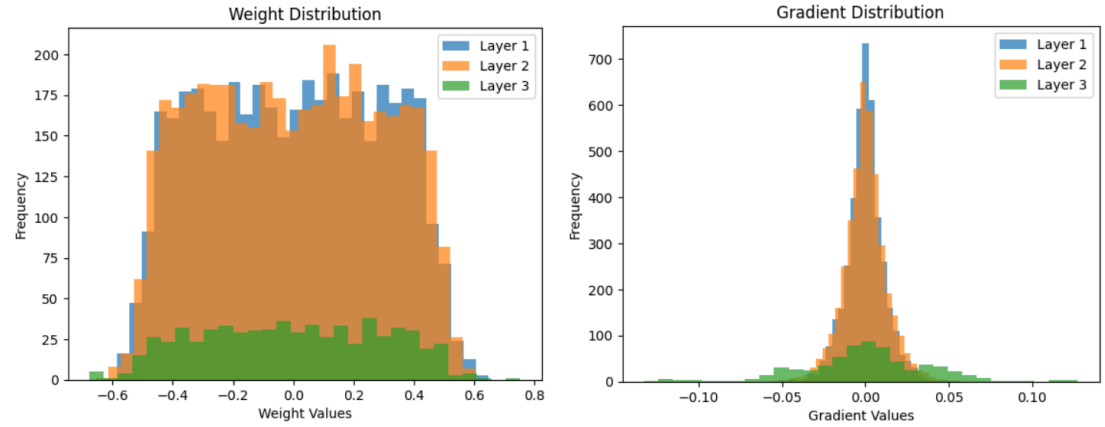
B. Hasil Pengujian

Variasi	Akurasi	Log Loss
Variasi 1	0.11428571428571428	2.3008875238773236
Variasi 2	0.9349285714285714	0.22389526420409042
Variasi 3	0.8392857142857143	0.529284478291713
Variasi 4	0.9652857142857143	0.12228876534211003
Variasi 5	0.9716428571428571	0.10317917701035213

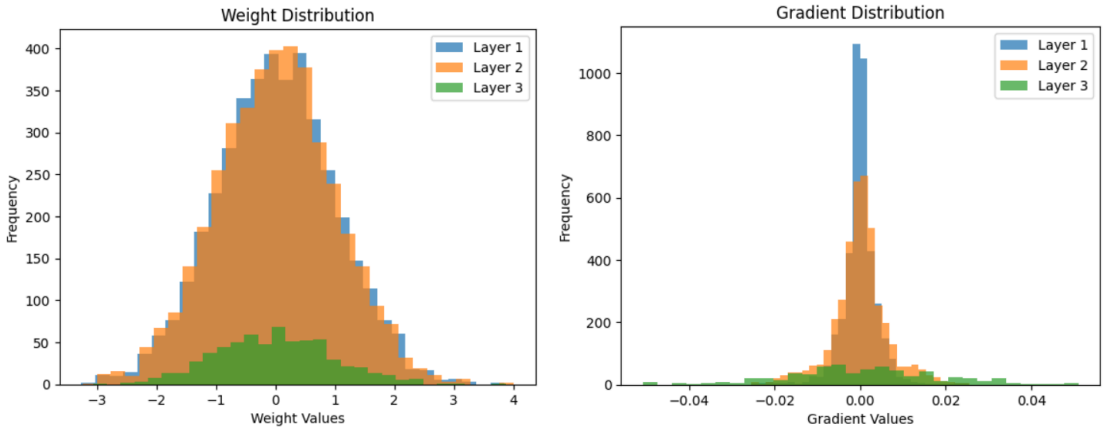
Tabel 2.2.5 Tabel hasil pengujian terhadap inisialisasi bobot



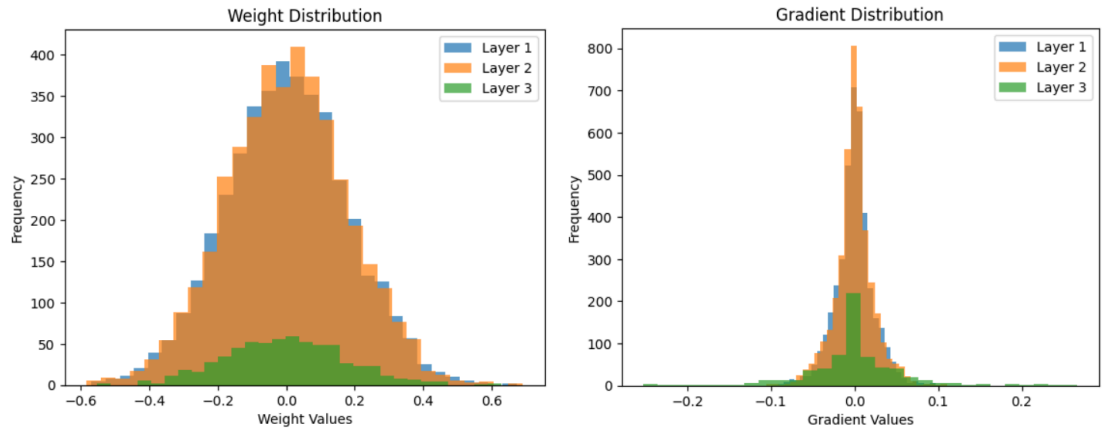
Gambar 2.2.15 Plot weight dan gradient distribution dari variasi inisialisasi bobot zero



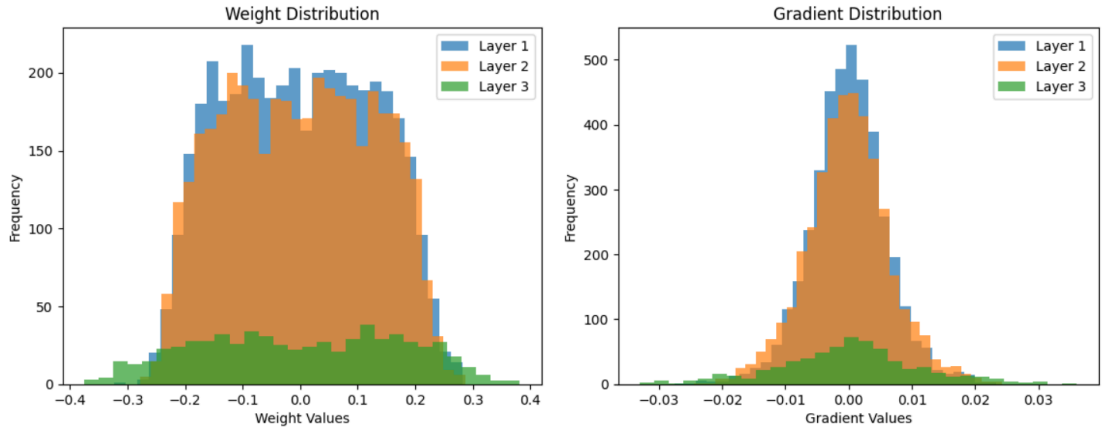
Gambar 2.2.16 Plot weight dan gradient distribution dari variasi inisialisasi bobot uniform



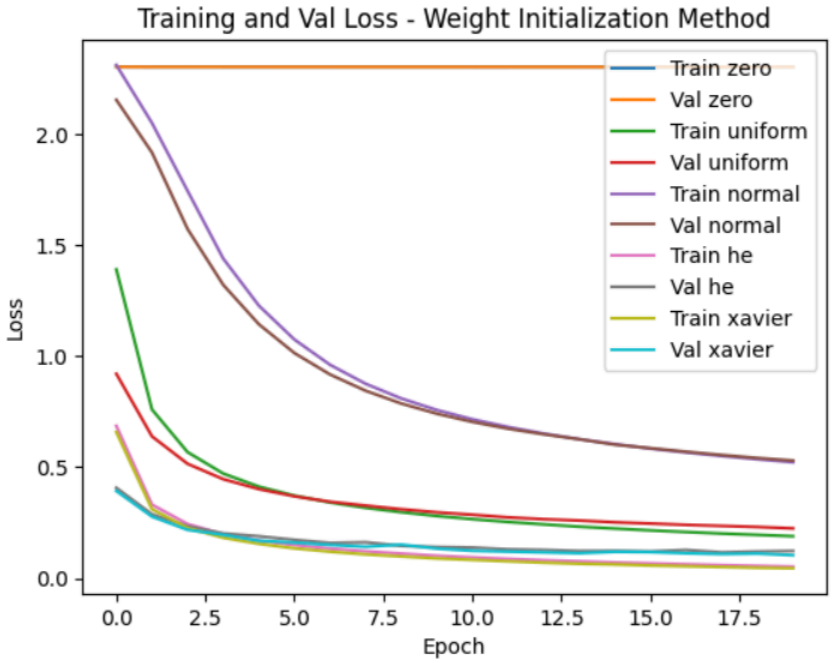
Gambar 2.2.17 Plot weight dan gradient distribution dari variasi inisialisasi bobot normal



Gambar 2.2.18 Plot weight dan gradient distribution dari variasi inisialisasi bobot he



Gambar 2.2.19 Plot weight dan gradient distribution dari variasi inisialisasi bobot xavier



Gambar 2.2.20 Training dan Val Loss dari metode inisialisasi bobot

Pada **Tabel 2.2.5** menunjukkan terkait hasil pengujian terhadap metode inisialisasi bobot. Terlihat hanya model dengan inisialisasi bobot zero memiliki akurasi rendah dan log loss yang tinggi artinya inisialisasi bobot zero ini tidak terlalu baik untuk digunakan dengan parameter yang ada di FFNN. Di sisi lain, model dengan inisialisasi bobot uniform, normal, he, dan xavier memiliki akurasi yang relatif sama di yaitu kisaran antara 0.922 - 0.975 dengan log loss memiliki kisaran yaitu 0.133 - 0.273 sehingga keempat inisialisasi bobot ini bisa memberikan hasil yang baik pada FFNN yang dibuat.

Pada **Gambar 2.2.15 - 2.2.19**, hanya inisialisasi bobot zerolah yang memiliki weight dan gradient distribution yang terlalu berpatok pada 0 saja yang artinya inisialisasi bobot zero tidak melakukan update weight yang baik sehingga hasil akurasi yang diberikan kecil. Berbeda dengan inisialisasi bobot uniform, normal, he, dan xavier, kedua weight dan gradient distribution memiliki distribusi normal sehingga variasi-variasi ini memiliki akurasi-akurasi yang relatif baik.

Terkait performa dari kelima variasi diperkuat lagi oleh grafik training dan val loss. Terlihat bahwa variasi inisialisasi bobot zero memiliki training dan val loss yang relatif tinggi dan tidak pernah turun seiring bertambahnya epochnya sehingga variasi ini kemungkinan *underfitting*. Berbeda dengan variasi lainnya yang memiliki penurunan training dan val loss yang optimal setiap epochnya sehingga variasi ini memiliki *fitting* yang cocok.

6. Pengaruh Root Mean Square (RMS) Normalization

A. Deskripsi Pengujian

RMSNorm (Root Mean Square Normalization) adalah alternatif ke LayerNorm yang menormalisasikan aktivasi tanpa menghitung mean. Ini dibuat untuk meningkatkan efisiensi dan stabilitas pada neural network. Beginilah rumus dari RMSNorm:

$$RMS(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$$

Output yang dinormalisasikan akan dihitung dengan rumus sebagai berikut:

$$y = \frac{x}{RMS(x)} \cdot \gamma$$

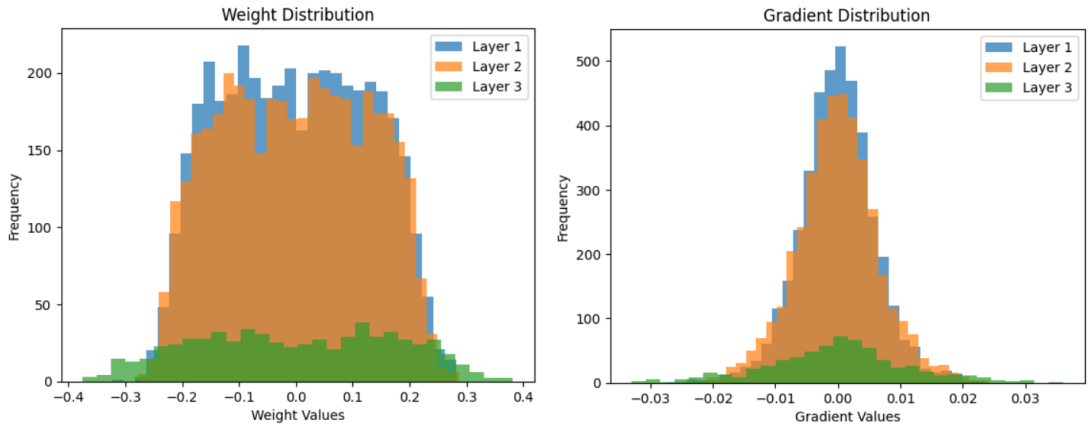
Untuk memastikan bahwa hanya inisialisasi bobot yang bervariasi, parameter lain dalam model tetap dijaga konstan, yaitu:

- **Aktivasi hidden layer** : hyperbolic tangent (tanh)
- **Kedalaman** : 3
- **Banyak neuron** : 64
- **Learning rate** : 0.1
- **Loss function** : categorical cross entropy
- **Inisialisasi bobot** : xavier
- **Epoch** : 20
- **Regularization** : None
- **Batch Size** : 64

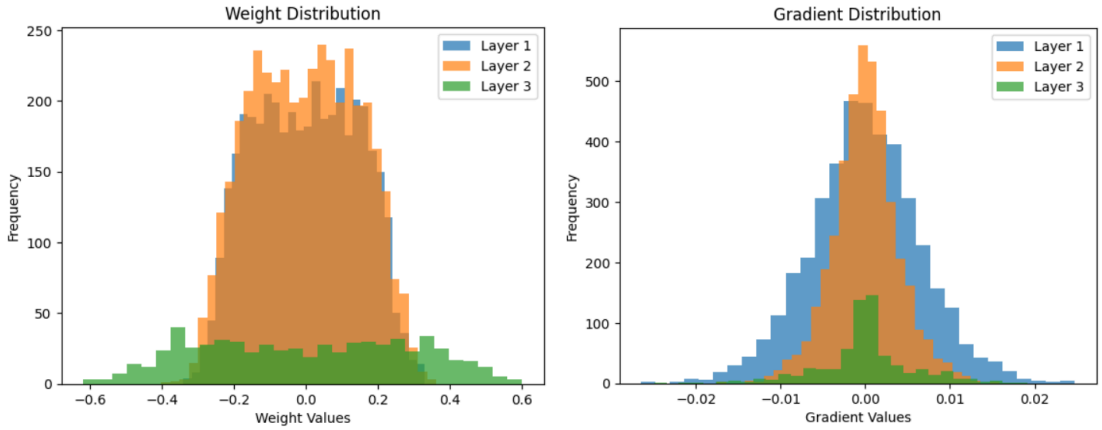
B. Hasil Pengujian

Variasi	Akurasi	Log Loss
Tanpa RMS	0.9587857142857142	0.14398589337499101
Dengan RMS	0.9716428571428571	0.10317917701035213

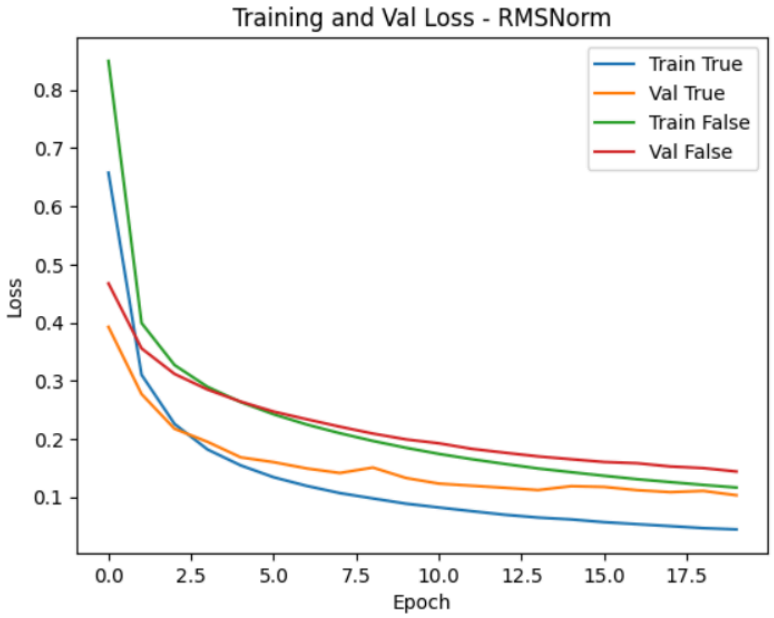
Tabel 2.2.6 Tabel hasil pengujian terhadap RMS Normalization



Gambar 2.2.21 Plot weight dan gradient distribution dari variasi dengan RMSNorm



Gambar 2.2.22 Plot weight dan gradient distribution dari variasi tanpa RMSNorm



Gambar 2.2.23 Perbandingan training dan validation loss dari variasi RMSNorm

Pada **Tabel 2.2.6**, terlihat variasi yang menggunakan RMS memiliki akurasi yang lebih tinggi dibandingkan dengan yang tidak menggunakan RMS. Ini artinya, RMS bisa meningkatkan akurasi dari sebuah model dengan data yang diberikan meskipun hanya meningkat sekitar 0.02 saja.

Pada **Gambar 2.2.21 - 2.2.22**, kedua weight dan gradient distribution sama-sama memiliki distribusi normal sehingga bisa disimpulkan kedua variasi bisa melakukan update bobot yang baik dan bisa menghasilkan akurasi yang baik juga. Hal ini didukung dengan **Gambar 2.2.23** yang menunjukkan training dan val loss dari variasi yang menggunakan RMS atau tidak. Kedua variasi mengalami penurunan nilai loss setiap bertambahnya epoch.

7. Pengaruh Regularisasi L1 dan L2

A. Deskripsi Pengujian

Regularisasi digunakan dalam machine learning untuk mencegah overfitting dengan menambahkan penalti pada bobot model. Berikut adalah perbedaan antara L1, L2, dan tanpa regularisasi.

Untuk memastikan bahwa hanya inisialisasi bobot yang bervariasi, parameter lain dalam model tetap dijaga konstan, yaitu:

- **Aktivasi hidden layer** : hyperbolic tangent (tanh)
- **Kedalaman** : 3
- **Banyak neuron** : 64
- **Learning rate** : 0.1
- **Loss function** : categorical cross entropy
- **Inisialisasi bobot** : xavier
- **Epoch** : 20
- **RMSNorm** : true
- **Batch Size** : 64

Pada regularisasi, terdapat 2 varian yang bisa digunakan dan akan diujikan beserta variasi yang tidak memiliki regularisasi:

1. Regularisasi L1: Regularisasi L1 menambahkan penalti terhadap jumlah nilai absolut bobot dalam fungsi loss:

$$L = L_{original} + \lambda \Sigma |w_i|$$

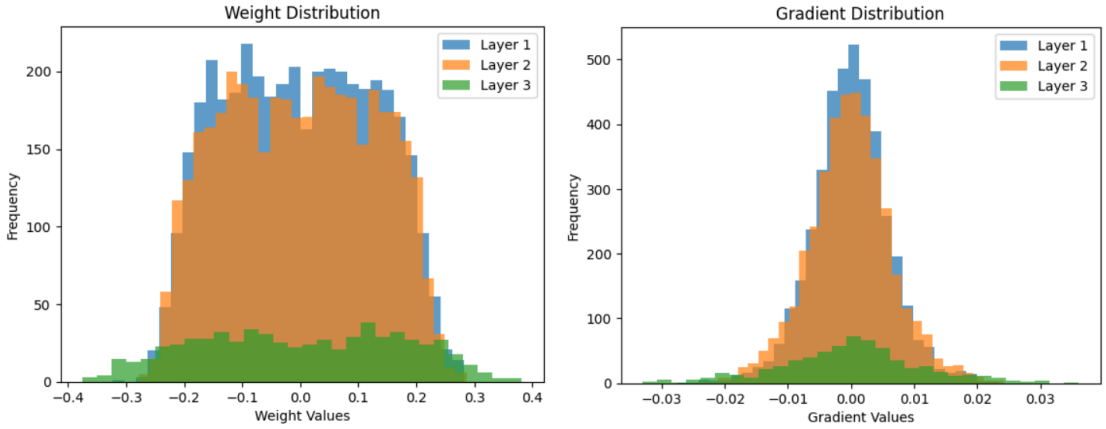
2. Regularisasi L2: Regularisasi L2 menambahkan penalti terhadap jumlah kuadrat bobot dalam fungsi loss:

$$L = L_{original} + \lambda \Sigma w_i^2$$

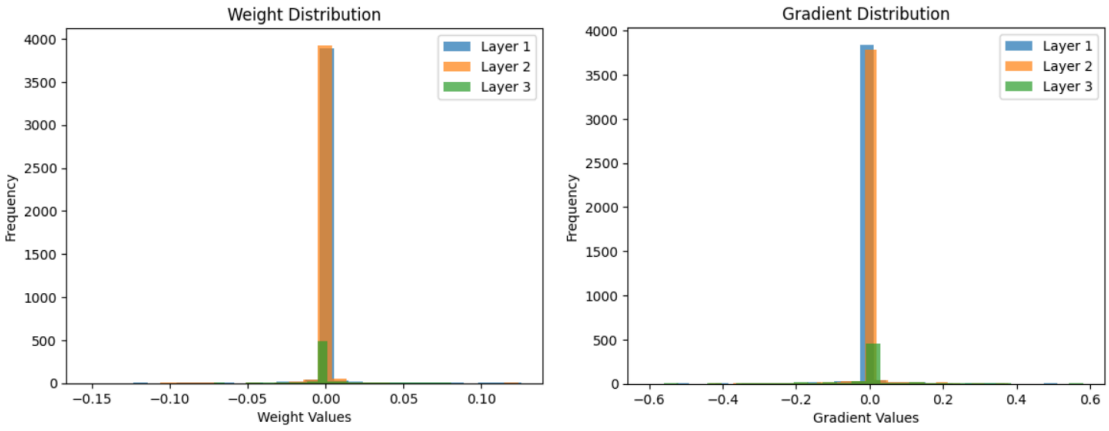
B. Hasil Pengujian

Variasi	Akurasi	Log Loss
Tanpa Regularisasi	0.9716428571428571	0.10317917701035213
Regularisasi L1	0.8897857142857143	0.39877523519350416
Regularisasi L2	0.9690714285714286	0.10591564191009085

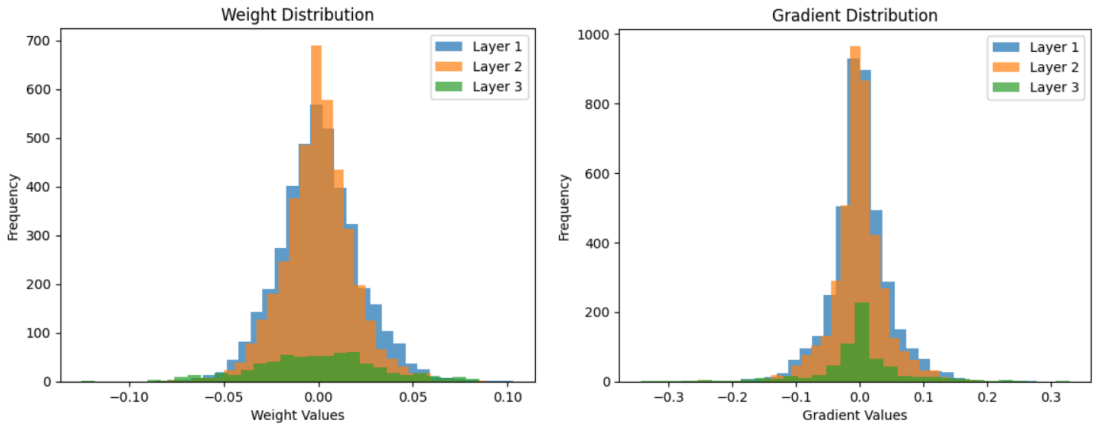
Tabel 2.2.7 Tabel hasil pengujian terhadap regularisasi l1 dan l2



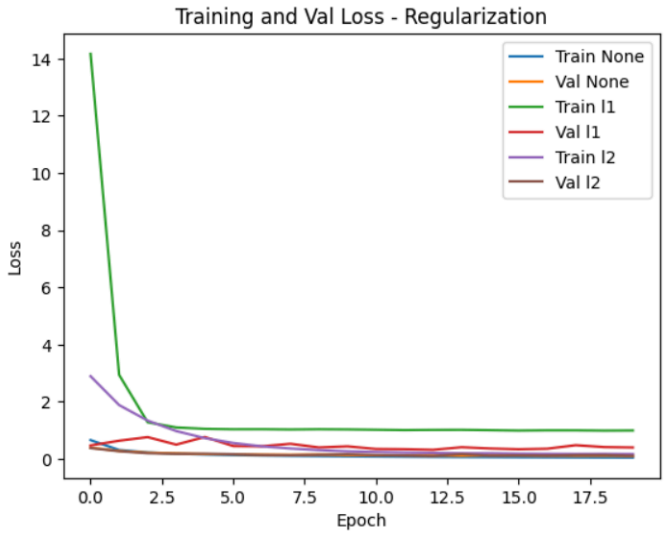
Gambar 2.2.24 Plot weight and gradient distribution dari variasi tanpa regularisasi



Gambar 2.2.25 Plot weight dan gradient distribution dari variasi regularisasi L1



Gambar 2.2.26 Plot weight dan gradient distribution dari variasi regularisasi L2



Gambar 2.2.27 Training dan Val Loss dari metode regularisasi L1 dan L2

Pada **Tabel 2.2.7**, akurasi yang besar terdapat di variasi tanpa regularisasi dan regularisasi l2 yaitu 0.971 dan 0.969 dengan log loss yang kecil juga yaitu 0.103 dan 0.105. Namun, melihat variasi tanpa regularisasi memiliki akurasi yang paling besar dibandingkan variasi yang lain, terlihat bahwa regularisasi tidak terlalu diperlukan dengan dataset yang diberikan. Regularisasi l1 memiliki akurasi 0.889 dan log loss sebesar 0.398 yang merupakan nilai yang cukup baik tetapi tidak sebagus l2 dan non-regularisasi.

Pada **Gambar 2.2.24 - 2.2.26**, bisa dilihat variasi tanpa regularisasi memiliki distribusi normal sehingga menunjukkan baiknya akurasi yang dihasilkan. Regularisasi L2 memiliki distribusi yang cukup normal tetapi terlalu berpusat di 0 karena aturan regularisasi L2 yang memerlukan adanya penalti pada bobotnya. Regularisasi memiliki distribusi yang terlalu berpusat di 0 sehingga terbukti variasi ini kesulitan untuk melakukan update pada bobot.

Pada **Gambar 2.2.27** Ttrain loss dan val loss lainnya memiliki penurunan yang relatif dibandingkan epoch lainnya. Tetapi train loss pada l1 memiliki nilai yang besar dibandingkan yang lain pada awalnya. Karena loss-nya kecil, seharusnya tidak ada masalah terkait *fitting*.

8. Perbandingan dengan MLPClassifier

A. Deskripsi Pengujian

MLPClassifier adalah singkatan dari Multi-Layer Perceptron Classifier yang digunakan untuk klasifikasi dalam library Scikit-Learn. Ini adalah jenis Feed Forward Neural Network (FFNN) yang dilatih menggunakan backpropagation.

Untuk MLPClassifier akan menggunakan parameter sebagai berikut:

1. Hidden layer width = 64
2. Hidden layer depth = 3
3. Activation function = tanh
4. Learning rate = 0.01
5. Batch size = 64
6. Max iteration = 20

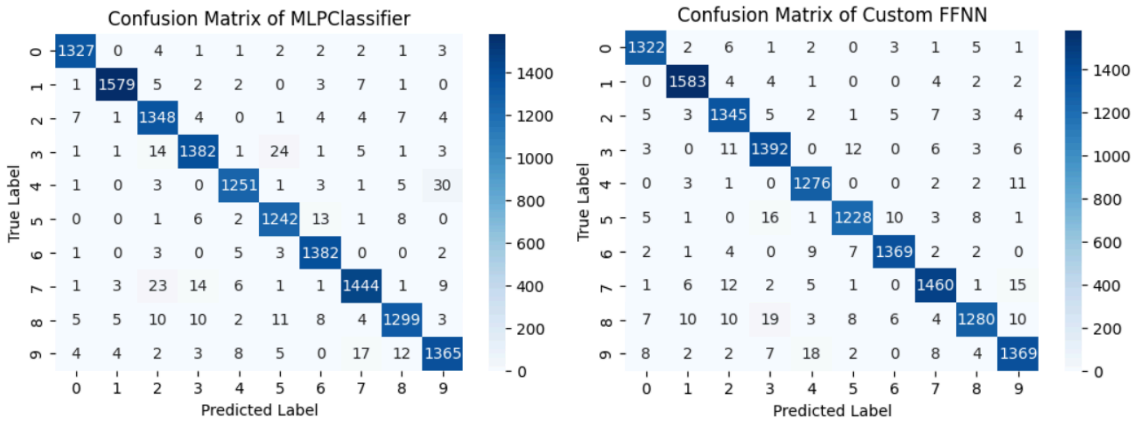
Untuk Custom FFNN akan menggunakan parameter sebagai berikut:

1. Hidden layer width = 64
2. Hidden layer depth = 3
3. Hidden layer activation function = tanh
4. Weight initialization = xavier
5. RMS normalization = true
6. Loss function = categorical_cross_entropy
7. Learning rate = 0.01
8. Batch size = 64
9. Epochs = 20

B. Hasil Pengujian

Variasi	Akurasi	Log Loss
MLPClassifier	0.9727857142857143	0.1404845370461712
Custom FFNN	0.9731428571428572	0.10247088387757484

Tabel 2.2.8 Tabel hasil pengujian terhadap model sklearn dan custom FFNN



Gambar 2.2.28 Confusion Matrix dari MLPClassifier dan Custom FFNN

Pada **Tabel 2.2.8**, akurasi yang dimiliki oleh Custom FFNN lebih besar dibandingkan dengan akurasi yang dimiliki oleh model yang disediakan oleh library sklearn sehingga bisa disimpulkan Custom FFNN memiliki performa yang lebih baik dengan dataset dan parameter yang diberikan. Pada **Gambar 2.2.28**, MLPClassifier memiliki nilai Confusion Matrix yang hampir sama persebarannya dibandingkan dengan Confusion Matrix yang dimiliki oleh Custom FFNN.

Kesimpulan & Saran

Pembagian Tugas

NIM	Tugas
13522068	Implementasi Program FFNN, Pengujian Program FFNN, Laporan
13522110	Laporan
13522118	Laporan

Referensi