Tugas Besar 2 IF3270 Pembelajaran Mesin

Convolutional Neural Network dan Recurrent Neural Network

Disusun oleh Kelompok 37:

Filbert	13522021
Benardo	13522055
William Glory Henderson	13522113

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

BAB I
DESKRIPSI MASALAH

Tugas Besar 2 IF3270 Pembelajaran Mesin bertujuan untuk memberikan pemahaman yang komprehensif mengenai implementasi dan analisis model-model deep learning fundamental, yaitu Convolutional Neural Network (CNN) dan Recurrent Neural Network (RNN), termasuk variannya seperti Long Short-Term Memory (LSTM). Fokus utama tugas ini adalah pada pengembangan modul propagasi maju (forward propagation) untuk ketiga jenis jaringan tersebut dari awal (from scratch) menggunakan bahasa Python. Untuk komponen CNN, mahasiswa ditugaskan untuk melatih model klasifikasi gambar menggunakan dataset CIFAR-10 dengan bantuan pustaka Keras. Pelatihan ini melibatkan serangkaian analisis pengaruh berbagai hyperparameter, seperti jumlah lapisan konvolusi, jumlah filter per lapisan, ukuran filter, dan jenis lapisan pooling. Model yang dilatih dengan Keras kemudian akan disimpan bobotnya untuk digunakan dalam modul propagasi maju from scratch. Pengujian dilakukan dengan membandingkan hasil prediksi (macro F1-score) antara implementasi from scratch dan Keras pada data uji. Pada komponen RNN dan LSTM, tugas serupa akan dilakukan untuk masalah klasifikasi teks menggunakan dataset NusaX-Sentiment (Bahasa Indonesia). Proses ini mencakup tahap preprocessing data teks, termasuk tokenisasi dan embedding, di mana mahasiswa dapat memanfaatkan lapisan TextVectorization dan Embedding dari Keras. Seperti pada CNN, model RNN dan LSTM akan dilatih menggunakan Keras dengan melakukan analisis variasi hyperparameter, yaitu jumlah lapisan RNN/LSTM, jumlah sel per lapisan, dan jenis lapisan berdasarkan arah (unidirectional vs. bidirectional). Bobot dari model terlatih juga akan disimpan dan modul propagasi maju from scratch akan diimplementasikan dan diuji performanya (menggunakan macro F1-score) terhadap hasil dari Keras pada data uji.

BAB II

PEMBAHASAN

2.1 Penjelasan Implementasi

2.1.1 Deskripsi Kelas Beserta Deskripsi Atribut dan Methodnya

2.1.1.1 CNN

2.1.1.1.1 Class CNNScratch

```
class CNNScratch:
    def __init__(self, keras_model):
        layers = keras_model.layers
        self.layers = []
        for l in layers:
            if isinstance(l, Conv2D):
                W, b = l.get_weights()
                activation_name = l.activation.__name__ if
hasattr(l, 'activation') else None
                self.layers.append(ConvolutionalLayer(W, b,
activation=activation_name))
            elif isinstance(l, MaxPooling2D):
                self.layers.append(PoolingLayer(size=2,
type='max'))
            elif isinstance(l, AveragePooling2D):
                self.layers.append(PoolingLayer(size=2,
type='avg'))
            elif isinstance(l, Flatten):
                self.layers.append(FlattenLayer())
            elif isinstance(l, Dense):
                W, b = l.get_weights()
                activation_name = l.activation.__name__ if
hasattr(l, 'activation') else None
                if activation_name == 'softmax':
                    act = 'softmax'
                elif activation_name == 'relu':
                    act = 'relu'
                elif activation_name == 'sigmoid':
                    act = 'sigmoid'
                elif activation_name == 'tanh':
                    act = 'tanh'
                elif activation_name == 'leaky_relu':
                    act = 'leaky_relu'
                else:
                    act = None
                self.layers.append(DenseLayer(W, b,
activation=act))
```

```
    def forward(self, x, batch_size=32):
        results = []
        for i in range(0, x.shape[0], batch_size):
            batch = x[i:i+batch_size]
            out = batch
            for layer in self.layers:
                out = layer.forward(out)
            results.append(out)
        return np.vstack(results)
```

Kelas CNNScratch merupakan kelas utama untuk membangun dan melakukan forward propagation pada model Convolutional Neural Network (CNN) secara modular dari awal (from scratch). Kelas ini mengekstrak arsitektur dan bobot model dari objek Keras yang telah dilatih, kemudian membangunnya ulang dalam bentuk urutan layer custom (Conv, Pooling, Flatten, Dense) yang diimplementasikan secara manual.

Atribut:

- layers: list berisi objek layer custom (ConvolutionalLayer, PoolingLayer, FlattenLayer, DenseLayer) yang telah diinisialisasi berdasarkan arsitektur dan bobot model Keras yang diberikan.

Method:

- forward(self, x, batch_size=32): melakukan proses forward propagation terhadap data input (x) secara batch, melalui semua layer yang ada di self.layers. Fungsi ini digunakan untuk melakukan inferensi, sehingga didapatkan prediksi model terhadap data.

2.1.1.1.2 Class Convolutional Layer

```
class ConvolutionalLayer:
    def __init__(self, weights, bias, stride=1, padding='same',
activation='relu'):
        self.weights = weights
        self.bias = bias
        self.stride = stride
        self.padding = padding
        self.activation = activation_functions[activation]

    def forward(self, x):
        batch_size, h_in, w_in, c_in = x.shape
        f_h, f_w, _, c_out = self.weights.shape
```

```
        if self.padding == 'same':
            pad_h = (f_h - 1) // 2
            pad_w = (f_w - 1) // 2
            x_padded = np.pad(x,
((0,0),(pad_h,pad_h),(pad_w,pad_w),(0,0)), mode='constant')
        else:
            x_padded = x
        h_out = (h_in + 2*pad_h - f_h)//self.stride + 1
        w_out = (w_in + 2*pad_w - f_w)//self.stride + 1
        out = np.zeros((batch_size, h_out, w_out, c_out))
        for b in range(batch_size):
            for i in range(h_out):
                for j in range(w_out):
                    for f in range(c_out):
                        h_start = i*self.stride
                        w_start = j*self.stride
                        window = x_padded[b, h_start:h_start+f_h,
w_start:w_start+f_w, :]
                        out[b, i, j, f] = np.sum(window *
self.weights[:,:,:,f]) + self.bias[f]
        return self.activation(out)
```

Kelas ConvolutionalLayer merepresentasikan layer konvolusi (Conv2D) pada jaringan CNN yang diimplementasikan secara manual (from scratch). Kelas ini bertanggung jawab untuk melakukan operasi konvolusi pada input menggunakan bobot (filter), bias, stride, dan padding serta menerapkan fungsi aktivasi tertentu.

Atribut:

- weights: bobot filter konvolusi
- bias: bias tiap filter
- stride: besar langkah pergeseran filter
- padding: jenis padding pada konvolusi
- activation: fungsi aktivasi yang dipakai

Method:

- forward(self, x): proses konvolusi 2D dan aktivasi terhadap input (tergantung fungsi aktivasi), output berupa feature map baru

2.1.1.1.3 Class Pooling Layer

```
class PoolingLayer:
    def __init__(self, size=2, type='max'):
        self.size = size
```

```
        self.type = type

    def forward(self, x):
        batch, h_in, w_in, c = x.shape
        h_out = h_in // self.size
        w_out = w_in // self.size
        out = np.zeros((batch, h_out, w_out, c))
        for b in range(batch):
            for i in range(h_out):
                for j in range(w_out):
                    window = x[b, i*self.size:(i+1)*self.size,
j*self.size:(j+1)*self.size, :]
                    if self.type == 'max':
                        out[b,i,j,:] = np.max(window, axis=(0,1))
                    else:
                        out[b,i,j,:] = np.mean(window, axis=(0,1))
        return out
```

Kelas PoolingLayer merepresentasikan layer pooling pada jaringan CNN yang diimplementasikan dari awal (from scratch). Layer ini digunakan untuk menurunkan dimensi spasial (tinggi dan lebar) dari feature map dan meningkatkan ketahanan model terhadap translasi dan distorsi kecil pada input. Pooling yang digunakan dapat berupa max pooling atau average pooling.

Atribut:

- size: ukuran window pooling (default: 2), misalnya 2x2 pooling
- type: jenis pooling yang digunakan ('max' untuk max pooling, 'avg' untuk average pooling)

Method:

- forward(self, x): melakukan operasi pooling pada input x (berbentuk batch feature map), sesuai ukuran window dan tipe pooling yang ditentukan.

2.1.1.1.4 Class FlattenLayer

```
class FlattenLayer:
    def forward(self, x):
        batch_size = x.shape[0]
        return x.reshape(batch_size, -1)
```

Kelas FlattenLayer merepresentasikan operasi flattening pada jaringan CNN, yaitu mengubah output feature map berdimensi banyak (3D) menjadi vektor satu dimensi

(1D) untuk setiap data dalam batch. Operasi ini diperlukan agar output feature map dari layer konvolusi/pooling dapat diproses oleh layer fully connected (Dense).

Atribut: -

Method:

- forward(self, x): melakukan operasi flatten pada input x, sehingga setiap data pada batch diubah menjadi vektor 1 dimensi.

2.1.1.1.5 Class DenseLayer

```
class DenseLayer:
    def __init__(self, weights, bias, activation='relu'):
        self.weights = weights
        self.bias = bias
        self.activation = activation_functions[activation]

    def forward(self, x):
        out = np.dot(x, self.weights) + self.bias
        if self.activation == softmax:
            return self.activation(out)
        else:
            return self.activation(out)
```

Kelas DenseLayer adalah implementasi layer fully connected (dense/linear layer) secara manual pada jaringan saraf tiruan. Layer ini melakukan operasi linier (dot product antara input dan bobot, lalu ditambah bias) dan menerapkan fungsi aktivasi sesuai dengan parameter yang diberikan.

Atribut:

- weights: matriks bobot layer dense
- bias: vektor bias layer dense
- activation: fungsi aktivasi yang digunakan

Method:

- forward(self, x): melakukan operasi linier pada input x (dot product dengan bobot dan penjumlahan bias), kemudian menerapkan fungsi aktivasi.

2.1.1.1.6 activation.py

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

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

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

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

def softmax(x):
    e = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e / np.sum(e, axis=1, keepdims=True)

activation_functions = {
    'relu': relu,
    'sigmoid': sigmoid,
    'tanh': tanh,
    'leaky_relu': leaky_relu,
    'softmax': softmax,
    None: lambda x: x
}
```

Kumpulan fungsi aktivasi ini digunakan pada layer dalam implementasi from scratch. Fungsi aktivasi berperan penting untuk memperkenalkan non-linearitas pada jaringan sehingga mampu mempelajari pola kompleks.

List fungsi:

- relu(x)
- sigmoid(x)
- tanh(x)
- leaky_relu(x)
- softmax(x)

2.1.1.2 RNN

2.1.1.2.1 Class EmbeddingLayer

```
import numpy as np


class EmbeddingLayer:
    def __init__(self, embedding_weights: np.ndarray):
```

```
        self.embedding_weights = embedding_weights
        self.vocab_size, self.embedding_dim =
embedding_weights.shape

    def forward(self, token_indices: np.ndarray) -> np.ndarray:
        return self.embedding_weights[token_indices]
```

Kelas *EmbeddingLayer* bertanggung jawab untuk mengubah urutan indeks token (hasil dari proses vektorisasi teks) menjadi urutan vektor embedding yang padat (dense). Representasi vektor ini menangkap informasi semantik dari token.

Atribut:

- embedding_weights (np.ndarray): Matriks 2D yang berisi bobot embedding. Setiap baris merepresentasikan vektor embedding untuk satu token dalam vocabulary. Dimensinya adalah (vocab_size, embedding_dim).
- vocab_size (int): Jumlah token unik dalam vocabulary (jumlah baris dalam embedding_weights).
- embedding_dim (int): Dimensi dari setiap vektor embedding (jumlah kolom dalam embedding_weights).

Method:

- forward(self, token_indices: np.ndarray) -> np.ndarray: Menerima input berupa array NumPy berisi urutan indeks token (token_indices) dan mengembalikan array NumPy berisi urutan vektor embedding yang sesuai.

2.1.1.2.2 Class SimpleRNNLayer

```
import numpy as np
from activation import tanh

class SimpleRNNLayer:
    def __init__(self,
                 W_xh: np.ndarray,
                 W_hh: np.ndarray,
                 b_h: np.ndarray,
                 rnn_units: int):
        self.W_xh = W_xh
        self.W_hh = W_hh
        self.b_h = b_h
```

```
        self.rnn_units = rnn_units

    def forward(self, X: np.ndarray, return_sequences: bool =
False) -> np.ndarray:
        batch_size, seq_len, _ = X.shape
        # Inisialisasi hidden state awal
        h = np.zeros((batch_size, self.rnn_units))
        outputs = []

        for t in range(seq_len):
            x_t = X[:, t, :]
            # Perhitungan hidden state
            h = tanh(np.dot(x_t, self.W_xh) + np.dot(h, self.W_hh)
+ self.b_h)
            if return_sequences:
                outputs.append(h.copy())

        if return_sequences:
            return np.stack(outputs, axis=1)
        else:
            return h
```

Kelas SimpleRNNLayer mengimplementasikan satu lapis Recurrent Neural Network (RNN) sederhana yang bersifat unidirectional. Layer ini memproses sekuens input langkah demi langkah (timestep), menghitung hidden state pada setiap langkah berdasarkan input saat itu dan hidden state dari langkah sebelumnya.

Atribut:

- W_xh (np.ndarray): Matriks bobot yang menghubungkan input pada timestep saat ini (x_t) ke hidden state (h_t).
- W_hh (np.ndarray): Matriks bobot yang menghubungkan hidden state dari timestep sebelumnya (h_t-1) ke hidden state saat ini (h_t).
- b_h (np.ndarray): Vektor bias yang ditambahkan dalam perhitungan hidden state.
- rnn_units (int): Jumlah unit (neuron) dalam layer RNN ini, yang juga menentukan dimensi dari hidden state.

Method:

- forward(self, X: np.ndarray, return_sequences: bool = False) -> np.ndarray: Melakukan proses forward propagation untuk layer RNN.

1. X: Input berupa array NumPy dengan bentuk (batch_size, sequence_length, input_dim).
2. return_sequences: Boolean yang menentukan output. Jika True, mengembalikan seluruh urutan hidden state dari semua timestep dengan bentuk (batch_size, sequence_length, rnn_units). Jika False, hanya mengembalikan hidden state terakhir dengan bentuk (batch_size, rnn_units).

2.1.1.2.3 Class BidirectionalLayer

```
import numpy as np
from SimpleRnnLayer import SimpleRNNLayer

class BidirectionalLayer:
    def __init__(self,
                 forward_layer: SimpleRNNLayer,
                 backward_layer: SimpleRNNLayer):
        self.forward_layer = forward_layer
        self.backward_layer = backward_layer

    def forward(self, X: np.ndarray,
                return_sequences: bool = False) -> np.ndarray:

        out_fw = self.forward_layer.forward(X,

return_sequences=return_sequences)

        X_reversed = X[:, ::-1, :]
        out_bw = self.backward_layer.forward(X_reversed,

return_sequences=return_sequences)

        if return_sequences:
            out_bw_reversed = out_bw[:, ::-1, :]
            return np.concatenate([out_fw, out_bw_reversed],
                                  axis=-1)
        else:
            return np.concatenate([out_fw, out_bw],
                                  axis=-1)
```

Kelas BidirectionalLayer membungkus dua instance SimpleRNNLayer untuk menciptakan fungsionalitas RNN bidirectional. Satu layer memproses sekuens dari arah

depan (forward), dan yang lainnya memproses sekuens dari arah belakang (backward). Output dari kedua arah kemudian digabungkan.

Atribut:

- forward_layer (SimpleRNNLayer): Instance dari SimpleRNNLayer yang memproses sekuens input dalam arah maju (dari awal ke akhir).
- backward_layer (SimpleRNNLayer): Instance dari SimpleRNNLayer yang memproses sekuens input dalam arah mundur (dari akhir ke awal).

Method:

- forward(self, X: np.ndarray, return_sequences: bool = False) -> np.ndarray: Melakukan forward propagation untuk kedua layer (forward dan backward).
    1. X: Input berupa array NumPy dengan bentuk (batch_size, sequence_length, input_dim).
    2. return_sequences: Boolean yang menentukan output. Jika True, output dari kedua arah untuk setiap timestep digabungkan, menghasilkan bentuk (batch_size, sequence_length, rnn_units * 2). Jika False, hanya hidden state terakhir dari arah forward dan hidden state pertama (setelah pembalikan) dari arah backward yang digabungkan, menghasilkan bentuk (batch_size, rnn_units * 2).

2.1.1.2.4 Class DropoutLayer

```
import numpy as np


class DropoutLayer:
    def __init__(self, rate: float):
        self.rate = rate

    def forward(self,
                X: np.ndarray,
                training: bool = False) -> np.ndarray:
        if not training or self.rate == 0.0:
            return X

        # Inverted dropout
```

```
        keep_prob = 1.0 - self.rate
        mask = np.random.binomial(1, keep_prob, size=X.shape)
        return X * mask / keep_prob
```

Kelas DropoutLayer mengimplementasikan mekanisme dropout, sebuah teknik regularisasi yang membantu mencegah overfitting pada model neural network. Selama training, dropout secara acak menonaktifkan (meng-nol-kan) sejumlah neuron pada layer dengan probabilitas tertentu.

Atribut:

- rate (float): Probabilitas sebuah neuron akan di-nol-kan selama proses training. Nilainya antara 0 dan 1.

Method:

- forward(self, X: np.ndarray, training: bool = False) -> np.ndarray: Menerapkan dropout pada input X.
  1. X: Input array NumPy.
  2. training: Boolean yang mengindikasikan apakah model sedang dalam mode training. Dropout hanya aktif jika training bernilai True dan rate > 0. Jika training adalah False (mode inferensi), input dikembalikan tanpa perubahan.

2.1.1.2.5 Class DenseLayer

```
from typing import Optional
import numpy as np
from activation import softmax, tanh

class DenseLayer:
    def __init__(self, weights: np.ndarray,
                 bias: np.ndarray,
                 activation: Optional[str] = None):
        self.weights = weights
        self.bias = bias
        self.activation = activation

    def forward(self, X: np.ndarray) -> np.ndarray:
        output = np.dot(X, self.weights) + self.bias
        if self.activation == 'softmax':
            return softmax(output)
```

```
        elif self.activation == 'tanh':
            return tanh(output)
        return output
```

Kelas DenseLayer mengimplementasikan layer fully-connected (juga dikenal sebagai layer linear atau affine). Layer ini melakukan transformasi linear pada input (perkalian matriks dengan bobot dan penambahan bias), yang kemudian dapat diikuti oleh fungsi aktivasi.

Atribut:

- weights (np.ndarray): Matriks bobot 2D dari layer dense.
- bias (np.ndarray): Vektor bias 1D dari layer dense.
- activation (Optional[str]): String yang menandakan nama fungsi aktivasi yang akan diterapkan setelah transformasi linear (misalnya, 'softmax', 'relu', 'tanh'). Jika None, tidak ada aktivasi yang diterapkan (output linear).

Method:

- forward(self, X: np.ndarray) -> np.ndarray: Melakukan operasi forward propagation. Input X dikalikan dengan weights, bias ditambahkan, dan hasilnya dilewatkan melalui fungsi aktivasi yang ditentukan (jika ada).

2.1.1.2.6 Class ScratchRNNModel

```
from SimpleRnnLayer import SimpleRNNLayer
from BidirectionalLayer import BidirectionalLayer
from DropoutLayer import DropoutLayer
import numpy as np
from EmbeddingLayer import EmbeddingLayer
from typing import List, Any
class ScratchRNNModel:
    def __init__(self, layers: List[Any]):
        self.layers = layers
        rec_types = (SimpleRNNLayer, BidirectionalLayer)
        rec_indices = [i for i,
                       layer in enumerate(self.layers)
                       if isinstance(layer, rec_types)]
        if not rec_indices:
            raise ValueError("Tidak ada layer rekuren
(SimpleRNNLayer/BidirectionalLayer) yang ditemukan.")
```

```
        self._last_recurrent_idx = max(rec_indices)

    def forward(self, X: np.ndarray,
                training: bool = False) -> np.ndarray:
        output = X
        rec_types = (SimpleRNNLayer, BidirectionalLayer)

        for i, layer in enumerate(self.layers):
            if isinstance(layer, rec_types):
                return_seq = (i < self._last_recurrent_idx)
                output = layer.forward(output,
                                       return_sequences=return_seq)
            elif isinstance(layer, DropoutLayer):
                output = layer.forward(output,
                                       training=training)
            else:
                output = layer.forward(output)
        return output

    def predict(self, X: np.ndarray) -> np.ndarray:
        return self.forward(X, training=False)
```

Kelas ScratchRNNModel adalah kelas utama yang berfungsi sebagai container untuk seluruh arsitektur model RNN modular. Kelas ini menyimpan urutan layer-layer (seperti EmbeddingLayer, SimpleRNNLayer, BidirectionalLayer, DropoutLayer, DenseLayer) dan mengatur alur data (forward propagation) melalui semua layer tersebut secara berurutan.

Atribut:

- layers (List[Any]): Sebuah list yang berisi instance dari objek-objek layer yang membentuk keseluruhan arsitektur model. Urutan layer dalam list ini menentukan urutan pemrosesan data.
- _last_recurrent_idx (int): Atribut internal yang menyimpan indeks dari layer rekuren (RNN/Bidirectional) terakhir dalam list layers. Ini digunakan untuk menentukan kapan parameter return_sequences pada layer rekuren harus disetel ke False (yaitu, hanya untuk layer rekuren terakhir sebelum layer Dense).

Method:

- forward(self, X: np.ndarray, training: bool = False) -> np.ndarray: Melakukan proses forward propagation lengkap melalui semua layer yang ada dalam atribut layers.
    1. X: Input data, biasanya berupa array NumPy berisi indeks token.
    2. training: Boolean yang menandakan apakah model sedang dalam mode training. Ini akan diteruskan ke layer yang perilakunya berbeda saat training dan inferensi (seperti DropoutLayer).
- predict(self, X: np.ndarray) -> np.ndarray: Sebuah method pintas (shortcut) yang memanggil self.forward(X, training=False). Digunakan untuk melakukan inferensi atau prediksi pada data baru.

2.1.1.3 LSTM

2.1.1.3.1 Class EmbeddingLayer

```
class EmbeddingLayer:
    def __init__(self, embedding_weights: np.ndarray):
        self.embedding_weights = embedding_weights
        self.vocab_size, self.embedding_dim =
embedding_weights.shape

    def forward(self, token_indices: np.ndarray) -> np.ndarray:
        return self.embedding_weights[token_indices]
```

Kelas ini berfungsi sebagai representasi layer embedding yang mengubah token indeks (misal: hasil vektorisasi kata) menjadi vektor berdimensi padat (dense vector), sesuai matriks bobot embedding yang dimiliki.

Atribut:

- embedding_weights: Matriks bobot embedding, berbentuk array 2D (jumlah kata x dimensi embedding).
- vocab_size: Ukuran kosakata (jumlah baris pada matriks embedding).
- embedding_dim: Dimensi setiap vektor embedding (jumlah kolom pada matriks embedding).

Method:

- forward(token_indices): Mengubah urutan indeks token menjadi urutan vektor embedding sesuai tabel embedding.

2.1.1.3.2 Class LSTMLayer

```
class LSTMLayer:
    def __init__(self, lstm_weights: Dict[str, np.ndarray]):
        self.input_weights_input =
lstm_weights['input_weights_input']
        self.input_weights_forget =
lstm_weights['input_weights_forget']
        self.input_weights_candidate =
lstm_weights['input_weights_candidate']
        self.input_weights_output =
lstm_weights['input_weights_output']

        self.hidden_weights_input =
lstm_weights['hidden_weights_input']
        self.hidden_weights_forget =
lstm_weights['hidden_weights_forget']
        self.hidden_weights_candidate =
lstm_weights['hidden_weights_candidate']
        self.hidden_weights_output =
lstm_weights['hidden_weights_output']

        self.bias_input = lstm_weights['bias_input']
        self.bias_forget = lstm_weights['bias_forget']
        self.bias_candidate = lstm_weights['bias_candidate']
        self.bias_output = lstm_weights['bias_output']

        self.hidden_units = self.input_weights_forget.shape[1]
        self.input_dim = self.input_weights_forget.shape[0]

    def forward(self,
                input_sequence: np.ndarray,
                mask: Optional[np.ndarray] = None,
                initial_hidden_state: Optional[np.ndarray] = None,
                initial_cell_state: Optional[np.ndarray] = None,
                return_sequences: bool = True) -> Tuple[np.ndarray,
Tuple[np.ndarray, np.ndarray]]:
        batch_size, sequence_length, _ = input_sequence.shape

        if mask is None:
            mask = np.ones((batch_size, sequence_length),
dtype=bool)

        if initial_hidden_state is None:
            hidden_state = np.zeros((batch_size,
self.hidden_units))
        else:
            hidden_state = initial_hidden_state.copy()
```

```
        if initial_cell_state is None:
            cell_state = np.zeros((batch_size, self.hidden_units))
        else:
            cell_state = initial_cell_state.copy()

        if return_sequences:
            all_hidden_states = np.zeros((batch_size,
sequence_length, self.hidden_units))

        for time_step in range(sequence_length):
            current_input = input_sequence[:, time_step, :]
            mask_t = mask[:, time_step]

            h_prev = hidden_state.copy()
            c_prev = cell_state.copy()

            input_gate = sigmoid(
                current_input @ self.input_weights_input +
                hidden_state @ self.hidden_weights_input +
                self.bias_input
            )

            forget_gate = sigmoid(
                current_input @ self.input_weights_forget +
                hidden_state @ self.hidden_weights_forget +
                self.bias_forget
            )

            candidate_values = tanh(
                current_input @ self.input_weights_candidate +
                hidden_state @ self.hidden_weights_candidate +
                self.bias_candidate
            )

            output_gate = sigmoid(
                current_input @ self.input_weights_output +
                hidden_state @ self.hidden_weights_output +
                self.bias_output
            )

            cell_state = forget_gate * cell_state + input_gate *
candidate_values

            new_c = cell_state
            new_h = output_gate * tanh(new_c)

            hidden_state = np.where(mask_t[:, None], new_h, h_prev)
            cell_state   = np.where(mask_t[:, None], new_c, c_prev)

            if return_sequences:
                all_hidden_states[:, time_step, :] = hidden_state
```

```
        if return_sequences:
            return all_hidden_states, (hidden_state, cell_state)
        else:
            return hidden_state, (hidden_state, cell_state)
```

Kelas ini adalah implementasi satu layer LSTM (Long Short-Term Memory) standar, lengkap dengan bobot-bobot dan bias untuk keempat gate (input, forget, candidate, output) serta metode propagasi maju (forward).

Atribut:

- input_weights_input, input_weights_forget, input_weights_candidate, input_weights_output: Matriks bobot input untuk masing-masing gate.
- hidden_weights_input, hidden_weights_forget, hidden_weights_candidate, hidden_weights_output: Matriks bobot state tersembunyi untuk masing-masing gate.
- bias_input, bias_forget, bias_candidate, bias_output: Bias masing-masing gate.
- hidden_units: Jumlah unit LSTM (ukuran state tersembunyi).
- input_dim: Dimensi input ke layer.

Method:

- forward(input_sequence, mask, initial_hidden_state, initial_cell_state, return_sequences): Melakukan perhitungan forward propagation pada seluruh timestep sequence, dengan dukungan masking (misal padding) dan pilihan output (semua state atau hanya state akhir).

2.1.1.3.3 Class BidirectionalLSTMLayer

```
class BidirectionalLSTMLayer:
    def __init__(self,
                 fw_weights: Dict[str, np.ndarray],
                 bw_weights: Dict[str, np.ndarray]):

        self.fw = LSTMLayer(fw_weights)
        self.bw = LSTMLayer(bw_weights)
        self.hidden_units = self.fw.hidden_units

    def forward(self,
                input_sequence: np.ndarray,
```

```
                mask: Optional[np.ndarray] = None,
                initial_hidden_state=None,
                initial_cell_state=None,
                return_sequences: bool = False
               ):
        out_fw, (h_fw, c_fw) = self.fw.forward(
            input_sequence, mask,
            initial_hidden_state, initial_cell_state,
            return_sequences=return_sequences
        )
        rev_x   = input_sequence[:, ::-1, :]
        rev_mask= mask[:, ::-1] if mask is not None else None
        out_bw, (h_bw, c_bw) = self.bw.forward(
            rev_x, rev_mask,
            initial_hidden_state, initial_cell_state,
            return_sequences=return_sequences
        )

        if return_sequences:
            out_bw = out_bw[:, ::-1, :]
            return np.concatenate([out_fw, out_bw], axis=-1),
(None, None)
        else:
            h = np.concatenate([h_fw, h_bw], axis=-1)
            return h, (h, None)
```

Kelas ini membungkus dua buah LSTM (forward dan backward) menjadi satu layer bidirectional. Outputnya adalah gabungan hasil kedua arah, baik pada seluruh sequence maupun hanya hidden state terakhir.

Atribut:

- fw: LSTMLayer arah forward.
- bw: LSTMLayer arah backward.
- hidden_units: Jumlah unit LSTM per arah.

Method:

- forward(input_sequence, mask, initial_hidden_state, initial_cell_state, return_sequences): Melakukan forward propagation dua arah (maju dan mundur), lalu menggabungkan hasil sesuai konfigurasi output.

2.1.1.3.4 Class DropoutLayer

```
class DropoutLayer:
    def __init__(self, dropout_rate: float):
        self.dropout_rate = dropout_rate

    def forward(self, inputs: np.ndarray, training: bool = False)
-> np.ndarray:
        if not training or self.dropout_rate == 0.0:
            return inputs

        keep_probability = 1.0 - self.dropout_rate
        dropout_mask = (np.random.rand(*inputs.shape) <
keep_probability)

        return inputs * dropout_mask / keep_probability
```

Kelas ini merepresentasikan dropout layer untuk regularisasi, dengan tujuan mencegah overfitting. Dropout dilakukan hanya saat training.

Atribut:

- dropout_rate: Rasio dropout (persentase neuron yang di-nol-kan setiap batch).

Method:

- forward(inputs, training): Mengaplikasikan dropout mask pada input saat training; pada inferensi hanya meneruskan input tanpa modifikasi.

2.1.1.3.5 Class DenseLayer

```
class DenseLayer:
    def __init__(self,
                 weights: np.ndarray,
                 bias: np.ndarray,
                 activation: Optional[str] = None):
        self.weights = weights
        self.bias = bias
        self.activation = activation
        self.input_dim, self.output_dim = weights.shape

    def forward(self, inputs: np.ndarray) -> np.ndarray:
        linear_output = inputs @ self.weights + self.bias

        if self.activation == 'softmax':
            return softmax(linear_output)
        elif self.activation == 'relu':
            return np.maximum(0, linear_output)
        elif self.activation == 'tanh':
            return tanh(linear_output)
        elif self.activation == 'sigmoid':
            return sigmoid(linear_output)
```

```
        else:
            return linear_output
```

Kelas ini merepresentasikan fully-connected (dense) layer. Layer ini biasanya digunakan sebagai layer terakhir untuk klasifikasi.

Atribut:

- weights: Matriks bobot layer (input_dim x output_dim).
- bias: Vektor bias.
- activation: Fungsi aktivasi yang digunakan (softmax, relu, sigmoid, tanh, atau None).
- input_dim, output_dim: Dimensi input dan output layer.

Method:

- forward(inputs): Menghitung output linear lalu mengaplikasikan fungsi aktivasi yang dipilih pada outputnya.

2.1.1.3.6 Class LSTMModel

```
class LSTMModel:
    def __init__(self, layers: List[Any]):
        self.layers = layers
        rec_types = (LSTMLayer, BidirectionalLSTMLayer)
        rec_idxs = [i for i, L in enumerate(layers) if
isinstance(L, rec_types)]
        if not rec_idxs:
            raise ValueError("No recurrent layers found!")
        self._last_lstm_idx = max(rec_idxs)

    def forward(self, token_indices: np.ndarray, training: bool =
False) -> np.ndarray:
        mask = (token_indices != 0)
        current = token_indices
        rec_types = (LSTMLayer, BidirectionalLSTMLayer)

        for idx, layer in enumerate(self.layers):
            if isinstance(layer, rec_types):
                return_seq = (idx != self._last_lstm_idx)
                current, _ = layer.forward(current,
                                           mask=mask,

return_sequences=return_seq)
            elif isinstance(layer, DropoutLayer):
                current = layer.forward(current, training=training)
            else:
                current = layer.forward(current)
        return current
```

```
def predict(self, token_indices: np.ndarray) -> np.ndarray:
    return self.forward(token_indices, training=False)
```

Kelas ini merupakan wrapper/model komposit, terdiri dari urutan layer-layer lain (embedding, LSTM/bidirectional LSTM, dropout, dense). Ini adalah titik masuk utama untuk proses forward/predict pada seluruh arsitektur model LSTM.

Atribut:

- layers: List berisi urutan layer (bisa kombinasi EmbeddingLayer, LSTMLayer, BidirectionalLSTMLayer, DropoutLayer, DenseLayer).
- _last_lstm_idx: Indeks layer LSTM/bidirectional terakhir pada list, untuk menentukan kapan mengambil output last hidden state.

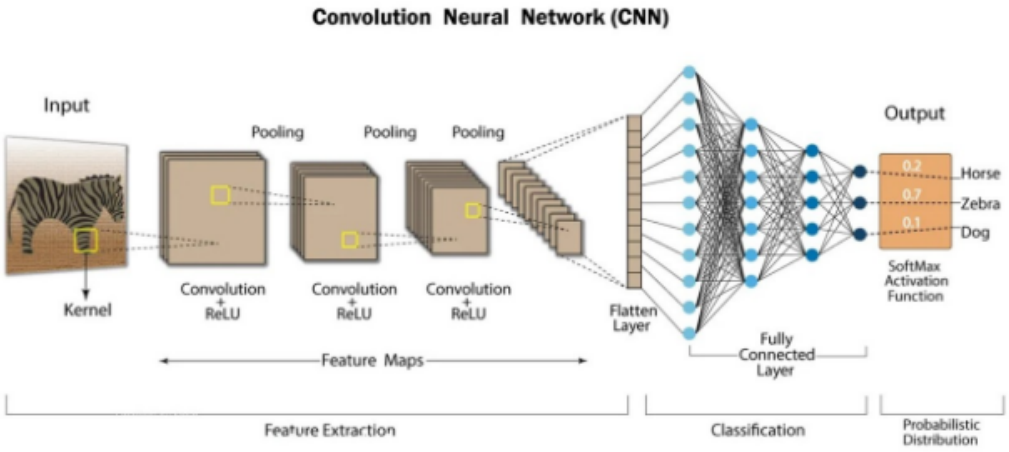
Method:

- forward(token_indices, training): Melakukan propagasi maju ke seluruh layer sesuai urutan, dengan handling khusus untuk layer LSTM/bidirectional (hanya last hidden state pada layer terakhir).
- predict(token_indices): Shortcut untuk melakukan prediksi tanpa training mode.

2.1.2 Penjelasan Forward Propagation

2.1.2.1 CNN

Propagasi maju (forward propagation) pada Convolutional Neural Network (CNN) adalah proses ketika data input (misal gambar) dimasukkan ke dalam jaringan, kemudian diproses secara berurutan atau sekuensial melalui setiap layer untuk menghasilkan output prediksi (misal label kelas). Proses ini dilakukan tanpa mengubah bobot (weights), hanya untuk inference (prediksi). Berdasarkan implementasi kode from scratch yang dibuat, berikut adalah langkah-langkah dalam forward propagation:



1. Input Layer

Proses dimulai ketika data input, contohnya gambar dari dataset CIFAR-10 (berukuran 32x32 piksel dengan 3 kanal warna), disajikan ke jaringan. Data ini umumnya diorganisir dalam format array atau sebagai batch (kumpulan data) untuk efisiensi pemrosesan.

2. Convolutional Layer (Conv2D)

Setiap Conv2D layer menerima input berupa gambar atau feature map dari lapisan sebelumnya, kemudian mengaplikasikan filter/kernel untuk mengekstrak fitur-fitur spasial. Operasi konvolusi dilakukan dengan cara menggeser setiap filter ke seluruh bagian input. Untuk setiap posisi filter, dihitung hasil perkalian elemen-demi-elemen antara filter dan bagian input yang bersesuaian, yang kemudian dijumlahkan dan ditambahkan dengan nilai bias. Hasilnya adalah feature map baru sebanyak jumlah filter. Setelah convolution, biasanya diterapkan fungsi aktivasi ReLU (Rectified Linear Unit), yaitu mengubah semua nilai negatif menjadi nol. Untuk fungsi aktivasi bisa berbeda tergantung yang dipakai.

3. Pooling Layer (Max/Average Pooling)

Setelah convolution, dilakukan pooling untuk mereduksi dimensi spatial feature map. Pengurangan dimensi ini membantu mengurangi jumlah parameter dalam jaringan, menurunkan beban komputasi, dan meningkatkan ketahanan model terhadap variasi kecil atau pergeseran posisi fitur pada input. MaxPooling mengambil nilai maksimum di tiap window, AveragePooling mengambil rata-rata dari semua nilai. Hasil pooling digunakan sebagai input ke layer berikutnya.

4. Flatten Layer

Setelah melalui serangkaian lapisan konvolusi dan pooling, feature map yang berbentuk multidimensi diratakan menjadi vektor satu dimensi (flatten), agar dapat diproses oleh dense layer (fully connected).

5. Dense Layer

Dense layer adalah layer fully connected yang menerima input berupa vektor hasil dari lapisan Flatten. Pada layer ini dilakukan transformasi linier, yaitu perkalian dot product antara vektor input dengan matriks bobot lapisan, yang kemudian ditambahkan dengan vektor bias. Hasil dari operasi linier ini kemudian dilewatkan ke fungsi aktivasi. Dense hidden biasanya menggunakan aktivasi ReLU, sedangkan output layer biasanya menggunakan softmax (untuk klasifikasi multikelas) sehingga output berupa probabilitas untuk setiap kelas.

6. Output

Output akhir dari keseluruhan proses propagasi maju adalah sebuah vektor yang merepresentasikan probabilitas untuk setiap kelas yang mungkin. Kelas yang diprediksi oleh model adalah kelas yang memiliki nilai probabilitas tertinggi dalam vektor tersebut.

Setiap layer diimplementasikan sebagai class modular dengan method forward(), sehingga data bisa mengalir dari satu layer ke layer berikutnya secara terstruktur. Batch processing artinya forward propagation dilakukan per-batch untuk efisiensi memori dan kecepatan. Bobot diambil langsung dari model Keras hasil training, sehingga output dari-scratch dan Keras bisa dibandingkan langsung.

Visualisasi Alur dari kode yang dibuat

Input → [Conv2D + ReLU] → [Pooling] → [Conv2D + ReLU] → [Pooling] → [Flatten] → [Dense + ReLU] → [Dense + Softmax] → Output Kelas

2.1.2.2 RNN

Forward propagation pada arsitektur Recurrent Neural Network (RNN) modular kami adalah proses di mana jaringan memproses data sekuensial langkah demi langkah. "Memori" (hidden state) diperbarui pada setiap timestep, hingga akhirnya menghasilkan

output prediksi. Dalam implementasi modular kami, alur ini diatur oleh kelas *ScratchRNNModel* yang mengelola urutan layer-layer individual. Proses ini melibatkan tahap-tahap berikut yang masing-masing ditangani oleh kelas layer spesifik: representasi input melalui Embedding (*EmbeddingLayer*), pemrosesan sekuens oleh satu atau lebih layer RNN (*SimpleRNNLayer* atau *BidirectionalLayer*), regularisasi Dropout (*DropoutLayer* - yang diabaikan saat inferensi), dan akhirnya proyeksi ke kelas output melalui layer Dense dengan aktivasi Softmax (*DenseLayer*). Keseluruhan proses ini dijalankan secara berurutan tanpa mengubah bobot model.

Kelas *ScratchRNNModel* dalam implementasi kami menerima input berupa urutan indeks token. Method *forward()*-nya kemudian secara sekuensial meneruskan output dari satu layer ke layer berikutnya.

1. Representasi Input melalui Embedding

   Sebelum data teks masuk ke layer RNN inti, setiap token (yang direpresentasikan sebagai nomor indeks) diubah menjadi vektor berdimensi tetap melalui *EmbeddingLayer*. Misalnya, jika kita memiliki batch kalimat dengan ukuran *(batch_size, sequence_length)*, maka setelah melewati *EmbeddingLayer*, bentuk datanya menjadi *(batch_size, sequence_length, embedding_dim)*. Vektor embedding ini bersifat padat (dense) sehingga mampu menangkap hubungan semantik antar-token secara lebih kaya dibandingkan representasi one-hot.

2. Iterasi Per‑Timestep di RNN Layer

   Inti dari pemrosesan RNN terjadi di dalam *SimpleRNNLayer* (atau *BidirectionalLayer* yang menggunakan dua *SimpleRNNLayer*). Pada setiap timestep *t*, hidden state dihitung berdasarkan input pada timestep tersebut xt dan hidden state dari timestep sebelumnya ht−1. Secara matematis, untuk satu sel RNN sederhana:

   $$ht = tanh(X_t W_{xh} + h_{t-1} W_{hh} + b_h)$$

   Proses ini diulang dari t = 1 hingga t = seq_len. Jika terdapat beberapa lapis RNN (num_layers > 1) dalam *ScratchRNNModel*, output dari lapis pertama menjadi input xt untuk lapis kedua, dan seterusnya. *ScratchRNNModel* secara otomatis mengatur parameter *return_sequences* pada setiap layer rekuren, di mana

hanya layer rekuren terakhir sebelum layer Dense yang akan memiliki *return_sequences=False* untuk hanya mengambil output state terakhir.

3. Mode Bidirectional

   Pada konfigurasi bidirectional, setiap lapis RNN menjalankan dua aliran:

   - Forward pass memproses urutan dari t = 1 ke t = seq_len,
   - Backward pass memproses urutan terbalik dari t = seq_len ke t = 1.

   Hasil hidden state dari arah maju dan arah mundur kemudian digabungkan (concatenate). Jika return_sequences=True (untuk layer rekuren yang bukan terakhir), seluruh rangkaian output gabungan dikembalikan. Jika return_sequences=False (untuk layer rekuren terakhir), hanya state terakhir dari arah maju dan state pertama (setelah pembalikan) dari arah mundur yang digabungkan.

4. Dropout dan Inferensi

   Setelah melewati semua lapis RNN, outputnya dilewatkan ke *DropoutLayer*. Lapisan ini berfungsi untuk mencegah overfitting selama proses training dengan secara acak menonaktifkan sebagian neuron. Namun, selama forward propagation untuk inferensi (seperti yang dilakukan saat pengujian atau prediksi), *DropoutLayer* tidak melakukan operasi apapun dan hanya meneruskan data tanpa perubahan. Hal ini memastikan bahwa output model bersifat deterministik dan konsisten saat inferensi.

5. Proyeksi ke Kelas dan Softmax

   Hidden state akhir (h_last) dari layer rekuren terakhir, baik yang berdimensi *(batch_size, rnn_units)* (untuk unidirectional) atau *(batch_size, rnn_units * 2)* (untuk bidirectional), kemudian dipetakan ke ruang kelas melalui *DenseLayer*. Operasi ini dihitung sebagai:

$$logits = h_{last} W_{dense} + b_{dense}$$

   Diakhiri dengan fungsi **Softmax**:

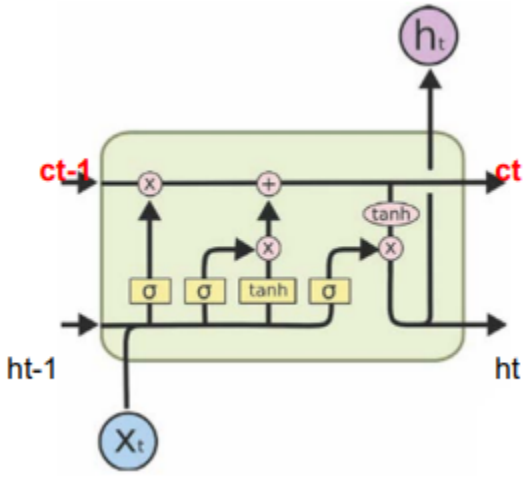
$$prob_i = \frac{exp(logits_i)}{\Sigma_j exp(logits_j)}$$

Hasilnya adalah distribusi probabilitas untuk setiap kelas. Prediksi akhir model dipilih sebagai kelas dengan probabilitas tertinggi dari distribusi ini.

Dengan demikian, forward propagation pada implementasi RNN modular kami mengikuti alur yang terstruktur melalui serangkaian layer independen, dimulai dari embedding input, pemrosesan sekuensial oleh layer RNN, regularisasi opsional, dan diakhiri dengan proyeksi ke output probabilitas kelas. Semua ini dilakukan tanpa mengubah bobot yang telah dipelajari selama pelatihan.

2.1.2.3 LSTM

Long Short-Term Memory (LSTM) merupakan varian Recurrent Neural Network (RNN) yang dirancang untuk mengatasi masalah vanishing gradient pada RNN klasik dengan cara menyimpan memori jangka panjang di dalam *cell state*. Pada forward propagation, LSTM memproses satu langkah waktu t dalam lima tahap utama: menerima input baru dan keadaan sebelumnya, menghitung tiga buah *gate* (forget, input, dan output), menghitung kandidat isi sel, memperbarui cell state, dan akhirnya menghasilkan hidden state (output) untuk langkah selanjutnya. Semua operasi ini dilakukan secara berurutan dalam satu blok LSTM, memungkinkan jaringan memilih informasi mana yang dipertahankan, diubah, atau diabaikan.



Pertama-tama, LSTM menerima vektor input saat ini x□ serta hidden state (kondisi keluaran) h□-1 dan cell state (kondisi memori) c□-1 dari langkah waktu sebelumnya. Kedua vektor keadaan tersebut h□-1 dan x□ digabungkan (concatenate) dan

diproses oleh berbagai lapisan affine (perkalian bobot dan penjumlahan bias) di setiap gate. Hasil affine kemudian dilewatkan melalui fungsi aktivasi sigmoid (σ) atau tanh, menghasilkan vektor nilai antara 0 dan 1 (pada sigmoid) atau antara –1 dan 1 (pada tanh). Nilai-nilai inilah yang menentukan sejauh mana informasi lama dibuang, informasi baru ditambahkan, dan informasi dalam sel dihasilkan.

Gate pertama adalah forget gate, yang menghitung vektor $f_t = \sigma(W_{xf} \cdot x_t + W_{hf} \cdot h_{t-1} + b_f)$. Setiap elemen $f_{t,i}$ berkisar antara 0 (buang sama sekali) hingga 1 (simpan sepenuhnya), dan diterapkan ke c_{t-1} dengan operasi perkalian elemen sehingga LSTM dapat melupakan sebagian informasi yang sudah tidak relevan lagi. Dengan cara ini, cell state lama yang penting tetap terjaga, sedangkan bagian yang tidak perlu dihapus sedikit demi sedikit.

Selanjutnya, input gate dan cell candidate bekerja sama untuk menambah informasi baru ke cell state. Input gate menghitung $i_t = \sigma(W_{xi} \cdot x_t + W_{hi} \cdot h_{t-1} + b_i)$, yang mengatur seberapa besar informasi baru boleh masuk. Paralel dengan itu, sebuah lapisan affine lain diikuti fungsi tanh menghasilkan kandidat isi sel, $\sim c_t = \tanh(W_{xc} \cdot x_t + W_{hc} \cdot h_{t-1} + b\_c)$. Nilai $\sim c_t$ ini berkisar antara –1 hingga 1 dan kemudian dikalikan dengan i_t, sehingga hanya informasi terpilih yang benar-benar ditambahkan ke memori.

Tahap berikutnya adalah memperbarui cell state

$$c_t = f_t * c_{t-1} + i_t * \sim c_t$$

Kombinasi ini menjadikan cell state c_t membawa gabungan memori lama yang dipertahankan dan memori baru yang relevan, sambil menjaga skala nilai tetap stabil.

Akhirnya, output gate menentukan apa yang akan dikeluarkan sebagai hidden state. Output gate menghitung $o_t = \sigma(W_{xo} \cdot x_t + W_{ho} \cdot h_{t-1} + b_o)$, lalu hidden state baru dihitung sebagai $h_t = o_t * \tanh(c_t)$. Dengan cara ini, LSTM dapat memilih bagian mana dari memori (cell state) yang paling representatif untuk diteruskan ke langkah berikutnya atau ke lapisan output akhir.

Secara ringkas, dalam satu langkah waktu forward propagation, LSTM melakukan :

(1) pengambilan input dan keadaan lama,

(2) “melupakan” sebagian memori lewat forget gate,

(3) menyeleksi dan menghasilkan memori baru lewat input gate dan kandidat isi sel,

(4) memperbarui cell state sesuai perpaduan memori lama dan baru, serta

(5) mengeluarkan hidden state terpilih lewat output gate. Alur ini memungkinkan LSTM menyimpan informasi penting dalam jangka panjang sembari fleksibel menyesuaikan dengan data baru yang masuk.

2.2 Hasil Pengujian

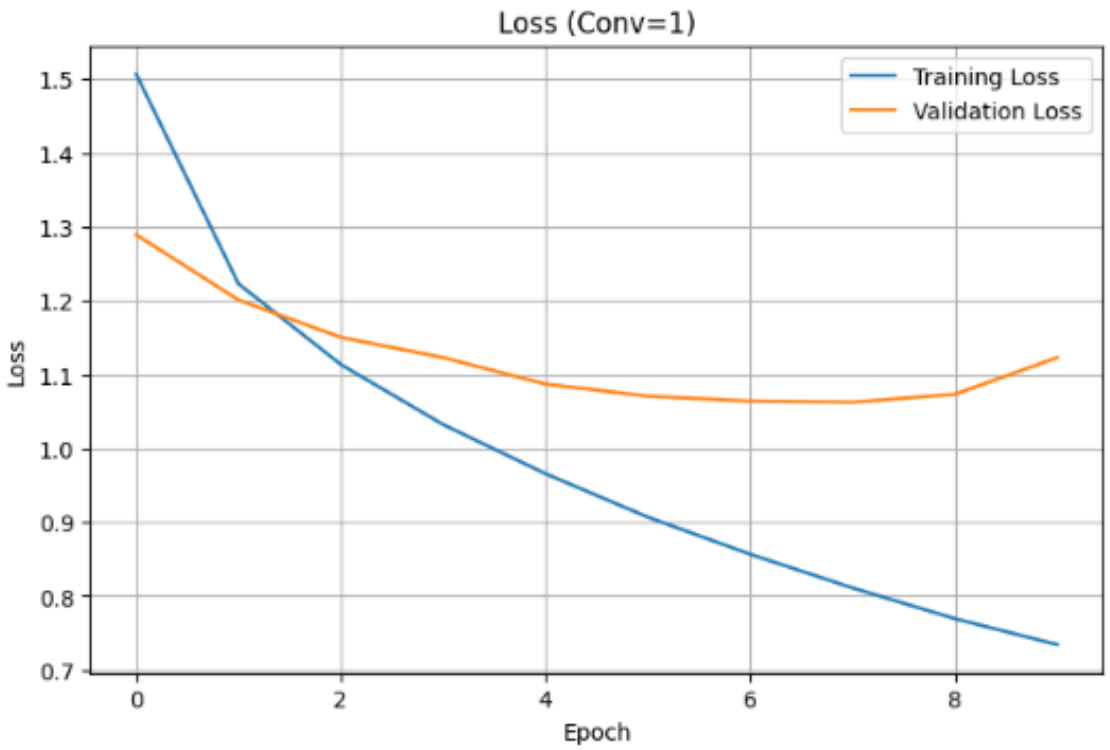
2.2.1 CNN

2.2.1.1 Pengaruh jumlah layer konvolusi

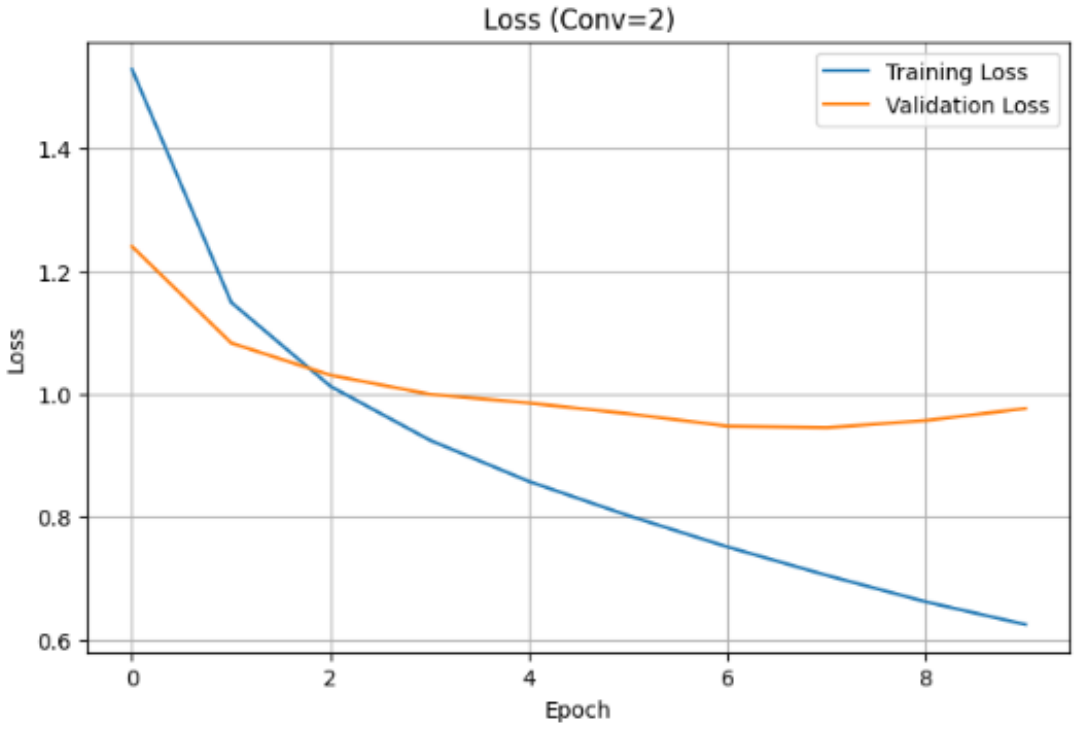
Pada pengujian jumlah layer konvolusi dilakukan 3 percobaan yaitu jumlah layer konvolusi 1, 2, dan 3. Pada pengujian ini, digunakan banyak filter ([32, 64, 128]), ukuran filter [(3,3), (3,3), (3,3)], dan jenis pooling (Max Pooling). Berdasarkan pengujian didapatkan hasil f1-score sebagai berikut:

Jumlah Layer	F1-Score
1	0.6270
2	0.6827
3	0.7080

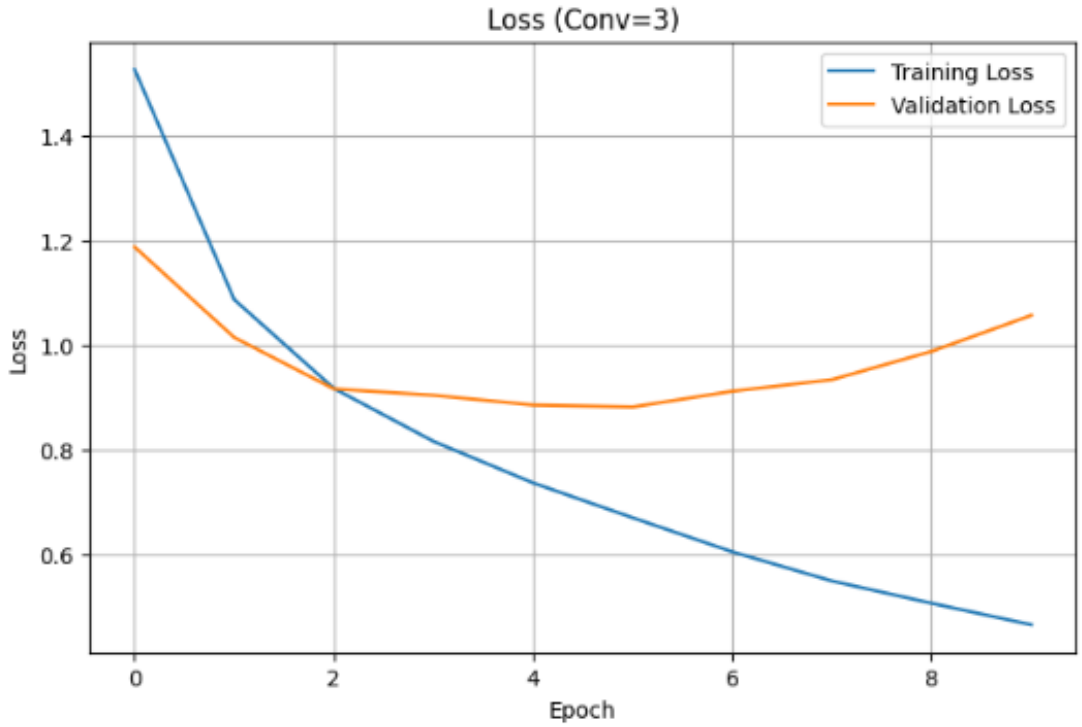
Berdasarkan tabel hasil eksperimen, model CNN dengan 3 konfigurasi jumlah layer konvolusi menunjukkan tren peningkatan performa seiring bertambahnya jumlah layer pada kasus klasifikasi gambar CIFAR-10. Konfigurasi dengan 3 layer konvolusi mencapai performa terbaik dengan macro F1-score sebesar 0.7080 dan akurasi tertinggi, diikuti oleh model dengan 2 layer (F1-score 0.6827) dan model dengan 1 layer (F1-score 0.6270). Hal ini menandakan bahwa perubahan jumlah layer konvolusi mempengaruhi keakuratan dari model. Kenaikan performa ini juga dapat diamati secara konsisten pada grafik training loss dan validation loss tiap epoch.



Pada model dengan 1 layer konvolusi, baik training loss maupun validation loss mengalami penurunan yang cukup konsisten tanpa fluktuasi ekstrem, menandakan proses pembelajaran berjalan stabil. Validation loss memang sedikit lebih tinggi dibanding training loss di akhir, namun gap antara keduanya masih dalam batas wajar dan tidak menunjukkan gejala overfitting yang berarti. Nilai macro F1-score yang diperoleh (0.6270) menunjukkan bahwa model sederhana ini sudah mampu mengenali pola dasar pada gambar, namun terbatas dalam mengekstraksi fitur yang lebih kompleks.



Penambahan layer menjadi 2 layer konvolusi menghasilkan penurunan training loss yang lebih cepat dan konsisten, serta validation loss yang cenderung lebih stabil. Gap antara training dan validation loss masih relatif kecil, menandakan generalisasi model tetap terjaga meski kapasitas model bertambah. Hal ini dibuktikan dengan kenaikan macro F1-score ke 0.6827, memperlihatkan bahwa model lebih mampu dalam mengekstraksi fitur menengah yang lebih relevan untuk tugas klasifikasi citra.



Pada model dengan 3 layer konvolusi, penurunan training loss menjadi paling signifikan, bahkan mencapai nilai terendah di antara semua konfigurasi. Validation loss juga menunjukkan tren penurunan pada awal epoch dan mulai sedikit naik di epoch akhir, sebuah indikasi bahwa model mulai mengalami sedikit overfitting. Namun, secara umum, validation loss tetap terkendali dan gap dengan training loss tidak sebesar pada suatu kasus overfitting yang tinggi. Hasil F1-score tertinggi (0.7080) menunjukkan bahwa model dengan tiga layer konvolusi paling optimal dalam menangkap berbagai tingkat kompleksitas fitur pada dataset, sehingga mampu membedakan kelas-kelas citra dengan lebih baik.

Analisis dari grafik juga menguatkan bahwa dengan bertambahnya jumlah layer, model memiliki kapasitas lebih besar untuk mengekstrak fitur kompleks pada data citra, sehingga mampu menangkap pola yang sebelumnya tidak terdeteksi oleh model lebih dangkal. Namun, perlu diingat bahwa penambahan layer secara berlebihan pada dataset yang lebih kecil atau kurang bervariasi bisa meningkatkan risiko overfitting, yang biasanya terlihat dari validation loss yang naik drastis dan gap yang semakin besar

dengan training loss. Pada eksperimen ini, penambahan layer hingga tiga masih memberikan manfaat tanpa menyebabkan overfitting yang signifikan.

Secara praktis, hasil ini menegaskan bahwa penambahan jumlah layer konvolusi pada CNN akan meningkatkan kemampuan model dalam mengekstraksi fitur citra dan memperbaiki performa klasifikasi, selama masih proporsional dengan ukuran serta variasi data. Pada CIFAR-10, model dengan tiga layer konvolusi terbukti paling optimal, sedangkan penambahan lebih banyak layer (di luar eksperimen ini) mungkin perlu diuji kembali dengan teknik tambahan.

Kesimpulannya, pemilihan jumlah layer konvolusi harus mempertimbangkan keseimbangan antara kapasitas model dan kompleksitas data. Untuk CIFAR-10, tiga layer konvolusi adalah pilihan yang cukup optimal dibanding 1 dan 2 layer. 3 Layer memberikan keseimbangan antara performa dan generalisasi model, tanpa menyebabkan overfitting yang merugikan.

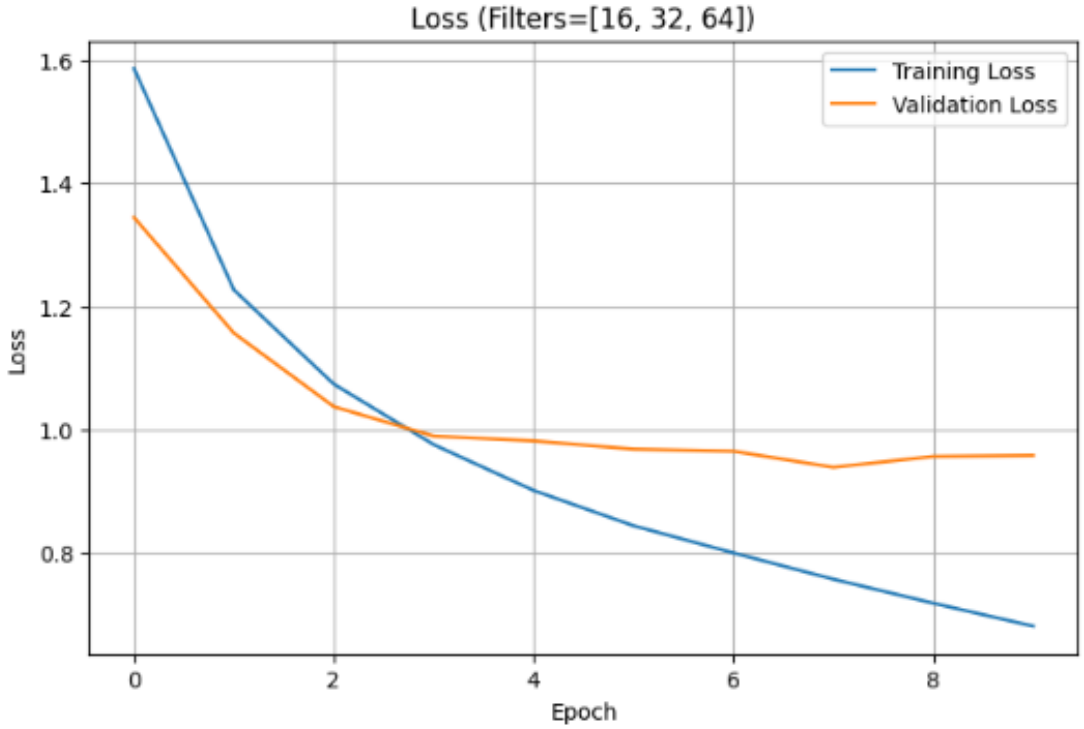
2.2.1.2 Pengaruh banyak filter per layer konvolusi

Pada pengujian banyak filter per layer konvolusi dilakukan 3 percobaan yaitu banyak filter konvolusi [16, 32, 64], [32, 64, 128], dan [64, 128, 256]. Pada pengujian ini, digunakan jumlah layer (3), ukuran filter [(3,3), (3,3), (3,3)], dan jenis pooling (Max Pooling). Berdasarkan pengujian didapatkan hasil f1-score sebagai berikut:

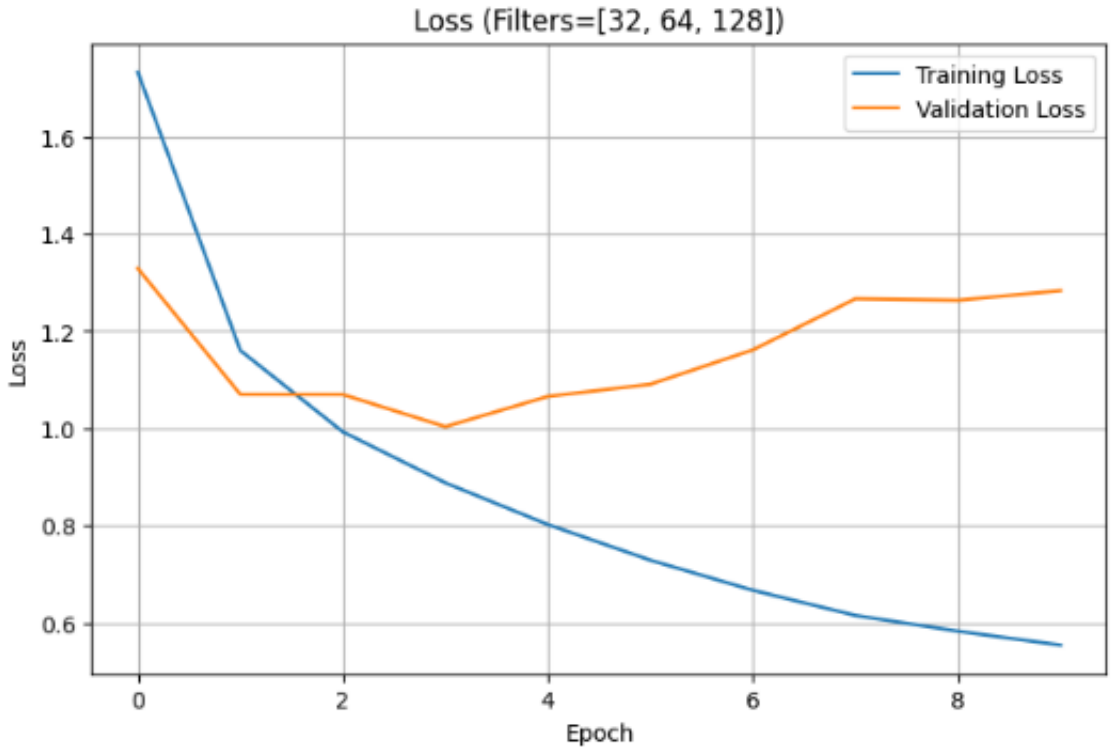
Banyak Filter per Layer Konvolusi	F1-Score
16, 32, 64	0.6792
32, 64, 128	0.7080
64, 128, 256	0.7039

Berdasarkan tabel hasil eksperimen, model CNN dengan tiga variasi jumlah filter per layer konvolusi menunjukkan bahwa performa model cenderung meningkat seiring bertambahnya jumlah filter, namun hanya sampai titik tertentu. Konfigurasi dengan 32, 64, dan 128 filter pada setiap layer konvolusi mencapai performa terbaik dengan macro F1-score sebesar 0.7080, sedikit lebih tinggi dibanding konfigurasi dengan filter lebih kecil (16, 32, 64 dengan F1-score 0.6792) maupun lebih besar (64, 128, 256 dengan

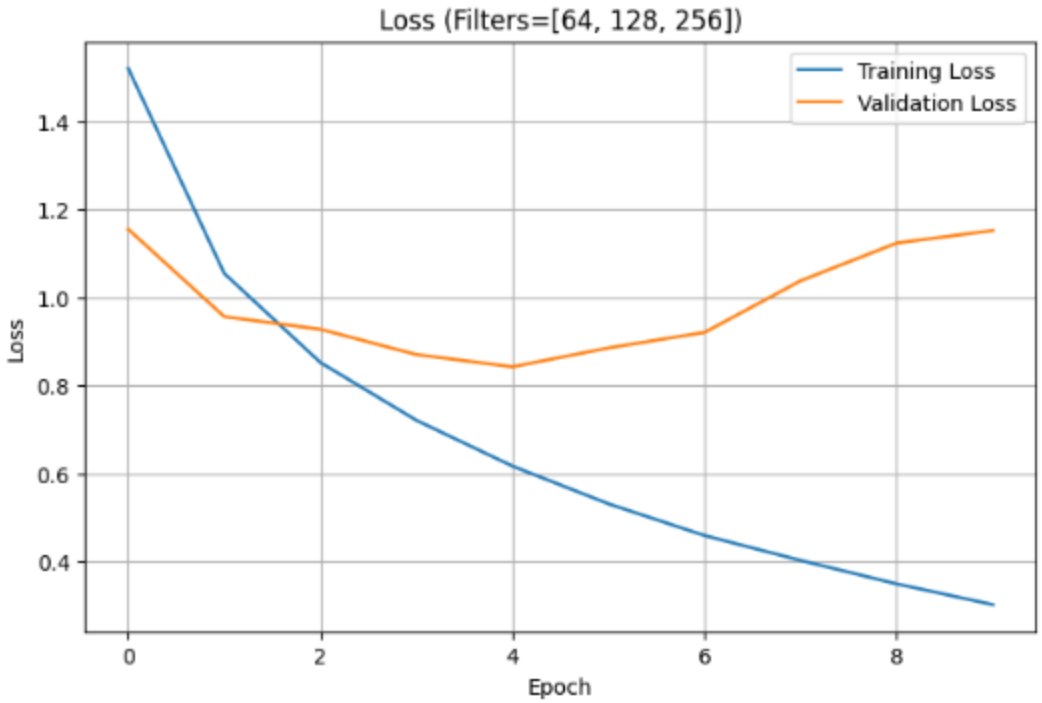
F1-score 0.7039). Hal ini menandakan bahwa perubahan jumlah filter per layer konvolusi mempengaruhi keakuratan dari model. Peningkatan performa ini juga dapat diamati secara konsisten pada grafik training loss dan validation loss setiap epoch, khususnya ketika filter dinaikkan dari konfigurasi kecil ke sedang.



Pada model dengan filter yang lebih kecil [16, 32, 64], baik training loss maupun validation loss mengalami penurunan stabil sepanjang epoch kecuali di epoch awal (0-2), dengan gap yang relatif kecil di antara keduanya. Hal ini menandakan proses pembelajaran yang sehat atau aman dan risiko overfitting yang minimal, namun terbatas pada kemampuan model untuk menangkap fitur yang lebih kompleks. Macro F1-score yang dicapai cukup baik (0.6792), namun masih kalah dibandingkan konfigurasi dengan lebih banyak filter.



Penambahan filter ke konfigurasi [32, 64, 128] menghasilkan penurunan training loss yang lebih cepat dan gap dengan validation loss masih dalam batas wajar. Validation loss memang mulai sedikit naik pada epoch akhir, namun tetap dalam rentang stabil sehingga risiko overfitting masih relatif rendah. Peningkatan macro F1-score menjadi 0.7080 menandakan model dengan jumlah filter ini mampu mengekstraksi fitur visual yang lebih kaya dan relevan untuk klasifikasi gambar pada CIFAR-10.



Sementara itu, pada model dengan jumlah filter terbesar [64, 128, 256], penurunan training loss berlangsung sangat cepat, tetapi validation loss justru mulai stagnan dan cenderung meningkat pada epoch-epoch akhir. Gap antara training dan validation loss juga menjadi semakin lebar, yang merupakan indikasi awal dari overfitting. Nilai macro F1-score sedikit menurun menjadi 0.7039, meskipun kapasitas model jauh lebih besar. Hal ini menandakan bahwa penambahan filter yang terlalu banyak dapat meningkatkan risiko overfitting, dan tidak selalu memberikan keuntungan performa yang signifikan.

Analisis grafik loss dari setiap konfigurasi juga menguatkan bahwa dengan menambah jumlah filter secara moderat, model lebih mampu mengenali pola kompleks pada data citra, sehingga menghasilkan prediksi yang lebih baik. Namun, menambah filter secara berlebihan akan memperbesar kapasitas model melebihi kebutuhan data, sehingga meningkatkan risiko overfitting yang ditandai dengan naiknya validation loss di epoch akhir meskipun training loss terus turun.

Secara praktis, hasil ini menegaskan bahwa penambahan jumlah filter per layer konvolusi harus dilakukan secara proporsional dengan mempertimbangkan kompleksitas

data dan ukuran dataset. Pada kasus CIFAR-10, konfigurasi [32, 64, 128] terbukti paling optimal dalam menghasilkan model dibandingkan [16, 32, 64] dan [64, 128, 256]. Konfigurasi [32, 64, 128] ini seimbang antara kapasitas ekstraksi fitur dan kemampuan generalisasi, tanpa meningkatkan risiko overfitting secara berlebihan. Penambahan filter di atas konfigurasi ini hanya menambah kompleksitas dan beban komputasi, tanpa memberikan manfaat berarti pada performa klasifikasi.

Kesimpulannya, pemilihan jumlah filter yang tepat sangat penting untuk memperoleh performa CNN yang optimal. Jumlah filter yang terlalu sedikit akan membatasi kemampuan model dalam mengenali pola penting, sementara terlalu banyak filter justru bisa menimbulkan overfitting dan inefisiensi komputasi.

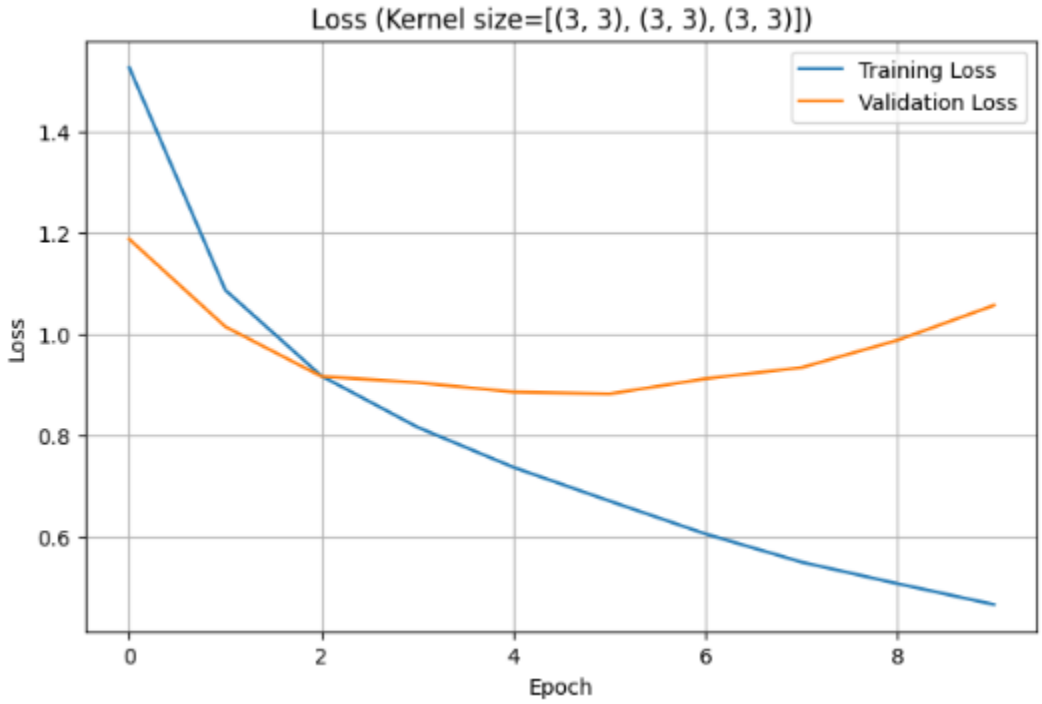
2.2.1.3 Pengaruh ukuran filter per layer konvolusi

Pada pengujian ukuran filter per layer konvolusi dilakukan 3 percobaan yaitu banyak filter konvolusi [(3, 3), (3, 3), (3, 3)], [(5, 5), (5, 5), (5, 5)], dan [(7, 7), (7, 7), (7, 7)]. Pada pengujian ini, digunakan jumlah filter (3), banyak filter ([32, 64, 128]), dan jenis pooling (Max Pooling). Berdasarkan pengujian didapatkan hasil f1-score sebagai berikut:

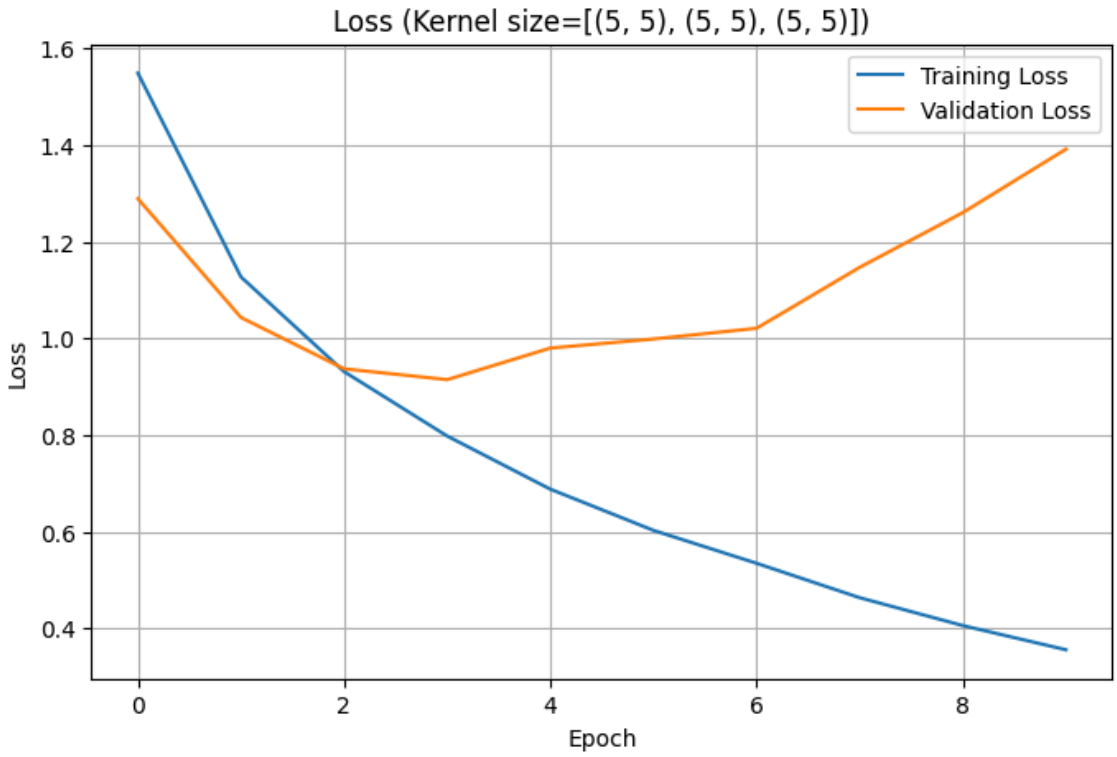
Ukuran Filter per Layer Konvolusi	F1-Score
(3, 3), (3, 3), (3, 3)	0.7080
(5, 5), (5, 5), (5, 5)	0.6630
(7, 7), (7, 7), (7, 7)	0.6093

Berdasarkan tabel hasil eksperimen, model CNN dengan tiga konfigurasi ukuran filter pada setiap layer konvolusi menunjukkan tren penurunan performa seiring membesarnya ukuran filter, pada kasus klasifikasi gambar CIFAR-10. Konfigurasi dengan filter (3, 3) di semua layer mencapai performa terbaik dengan macro F1-score sebesar 0.7080, diikuti oleh model dengan filter (5, 5) di semua layer (F1-score 0.6630), dan model dengan filter (7, 7) di semua layer (F1-score 0.6093). Hal ini menandakan bahwa perubahan ukuran filter per layer konvolusi mempengaruhi keakuratan dari model.

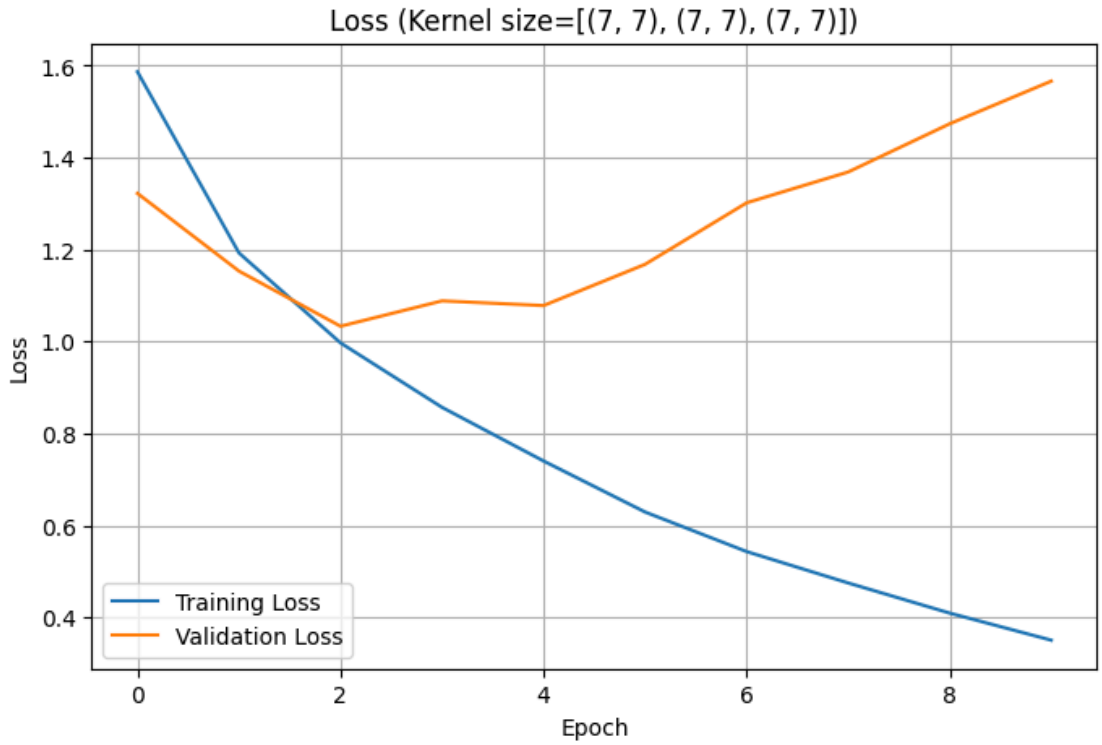
Pola penurunan performa ini juga tercermin secara konsisten pada grafik training loss dan validation loss tiap epoch.



Pada model dengan filter (3, 3) di setiap layer, baik training loss maupun validation loss menurun secara konsisten tanpa adanya fluktuasi yang ekstrem. Validation loss memang sedikit lebih tinggi dibandingkan training loss di akhir, tetapi gap di antara keduanya masih dalam batas wajar dan tidak menunjukkan gejala overfitting yang signifikan. Nilai macro F1-score tertinggi pada konfigurasi ini menunjukkan bahwa filter kecil seperti (3, 3) mampu menangkap detail penting pada gambar CIFAR-10, sehingga model dapat mengenali pola secara efektif.



Ketika ukuran filter diperbesar menjadi (5, 5) di setiap layer, terlihat bahwa training loss masih terus menurun dengan cepat, namun validation loss mulai stagnan dan bahkan meningkat di epoch-epoch akhir. Gap antara training dan validation loss semakin melebar, menandakan adanya gejala overfitting. Penurunan F1-score menjadi 0.6630 pada konfigurasi ini menunjukkan bahwa filter yang lebih besar mulai kehilangan kemampuan menangkap detail lokal, sehingga kemampuan model dalam membedakan kelas gambar pun menurun.



Pada model dengan filter (7, 7) di semua layer, performa model menurun secara signifikan. Training loss masih turun secara drastis, tetapi validation loss justru naik tajam, terutama pada epoch-epoch akhir. Gap antara training loss dan validation loss menjadi sangat besar, sebuah indikasi jelas adanya overfitting. Macro F1-score pada model ini juga turun drastis menjadi 0.6093, mengindikasikan bahwa filter besar membuat model kehilangan banyak informasi penting yang diperlukan untuk klasifikasi gambar.

Analisis grafik loss menguatkan bahwa dengan bertambahnya ukuran filter, training loss memang dapat ditekan lebih rendah, namun kemampuan model untuk melakukan generalisasi pada data validasi menjadi semakin buruk. Hal ini terjadi karena filter yang terlalu besar cenderung melewatkan detail-detail lokal yang penting, sehingga model hanya belajar dari pola global yang tidak cukup untuk membedakan objek-objek pada gambar CIFAR-10 yang berukuran kecil dan penuh detail.

Secara praktis, hasil eksperimen ini menegaskan bahwa penggunaan filter kecil, seperti (3, 3), merupakan pilihan optimal untuk image classification pada CIFAR-10 dibandingkan ukuran filter (5, 5) dan (7, 7). Penggunaan filter yang lebih besar

meningkatkan risiko overfitting dan menurunkan generalisasi model. Oleh karena itu, pemilihan ukuran filter harus disesuaikan dengan ukuran input data dan kompleksitas fitur yang ingin diekstrak oleh model.

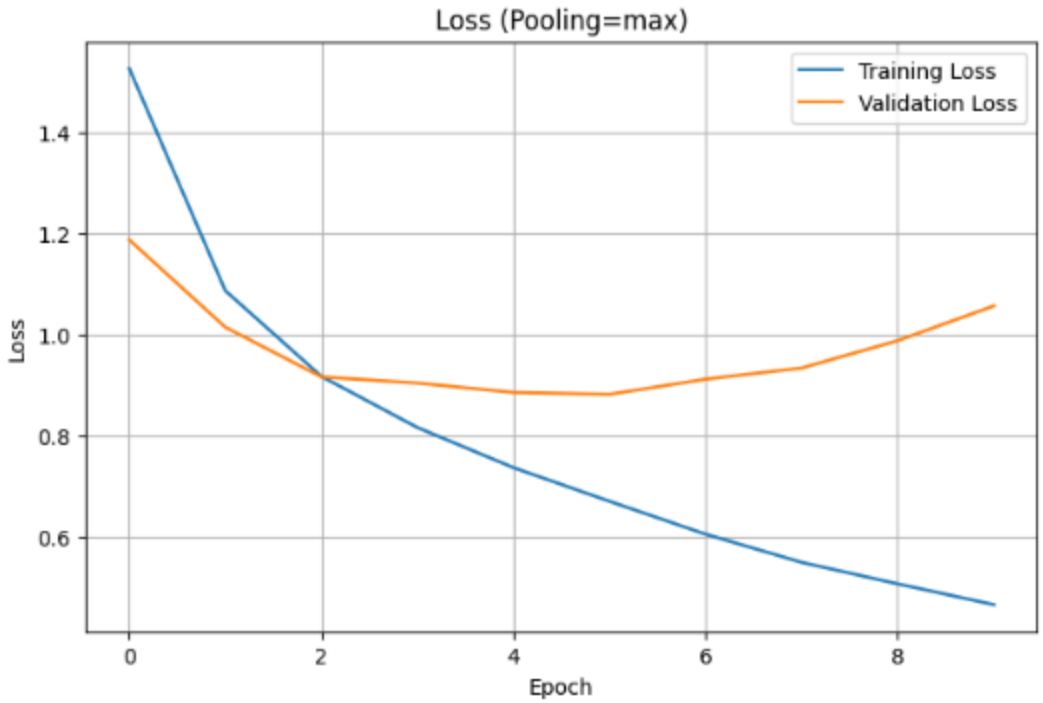
Kesimpulannya, semakin besar ukuran filter konvolusi, performa model CNN pada CIFAR-10 justru menurun, baik dari sisi F1-score maupun kestabilan loss. Filter (3, 3) memberikan hasil terbaik dan menjaga keseimbangan antara detail lokal dan generalisasi untuk klasifikasi gambar pada dataset ini.

2.2.1.4 Pengaruh jenis pooling layer

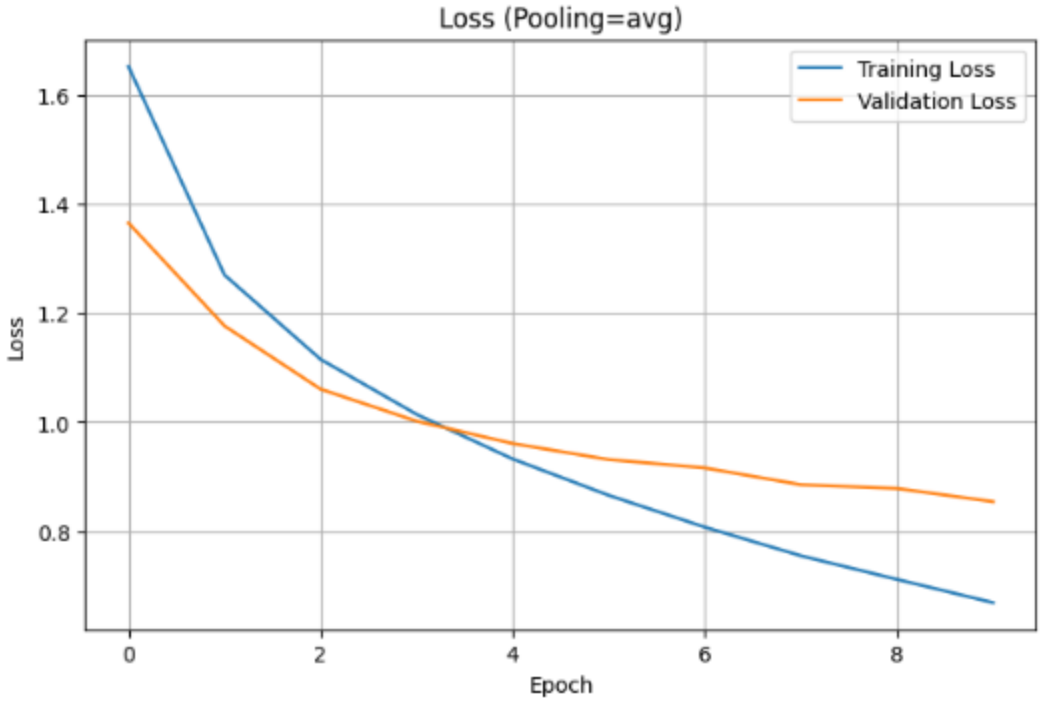
Pada pengujian jenis pooling layer konvolusi dilakukan 2 percobaan yaitu Max Pooling (mengambil nilai tertinggi) dan Average Pooling (mengambil rata-rata). Pada pengujian ini, digunakan jumlah layer (3), banyak filter ([32, 64, 128]), dan ukuran filter [(3,3), (3,3), (3,3)]. Berdasarkan pengujian didapatkan hasil f1-score sebagai berikut:

Jenis Pooling	F1-Score
Max Pooling	0.7080
Average Pooling	0.7060

Berdasarkan tabel hasil eksperimen, pemilihan jenis pooling layer yaitu antara max pooling dan average pooling berpengaruh pada performa akhir model CNN untuk klasifikasi gambar CIFAR-10, meskipun perbedaannya tidak terlalu signifikan. Model dengan max pooling menghasilkan macro F1-score tertinggi sebesar 0.7080, sedikit lebih baik dibanding model dengan average pooling yang menghasilkan macro F1-score sebesar 0.7060. Hasil ini juga tergambar pada grafik training loss dan validation loss tiap epoch, yang menunjukkan pola penurunan loss yang mirip pada kedua jenis pooling, namun dengan sedikit keunggulan pada max pooling.



Pada model dengan max pooling, baik training loss maupun validation loss mengalami penurunan stabil sepanjang proses pelatihan. Validation loss memang sedikit naik pada epoch-epoch akhir, namun gap antara training dan validation loss masih tergolong wajar. Nilai F1-score yang diperoleh pun menjadi yang tertinggi pada eksperimen ini, menandakan bahwa max pooling lebih efektif dalam mengekstraksi fitur-fitur paling dominan dari setiap patch gambar, sehingga model mampu membedakan pola-pola penting dengan lebih tajam.



Sementara pada model dengan average pooling, grafik training loss dan validation loss juga menunjukkan penurunan yang stabil dan gap yang kecil, bahkan validation loss cenderung lebih stabil dibanding max pooling di epoch-epoch akhir. Namun, nilai F1-score yang diperoleh sedikit lebih rendah. Hal ini mengindikasikan bahwa average pooling, yang mengambil nilai rata-rata dari setiap patch, cenderung menghasilkan fitur yang lebih halus dan kurang responsif terhadap pola-pola menonjol pada gambar. Meskipun model tetap dapat belajar dan melakukan generalisasi dengan baik, sensitivitas terhadap fitur yang benar-benar penting sedikit berkurang dibandingkan dengan max pooling.

Analisis dari grafik loss menguatkan bahwa baik max pooling maupun average pooling mampu menjaga proses pembelajaran yang stabil dan menghindari overfitting berlebihan. Namun, keunggulan tipis dari max pooling berasal dari kemampuannya untuk mempertahankan fitur yang paling menonjol dari setiap area gambar, yang sangat krusial dalam tugas klasifikasi visual, terutama pada dataset yang kompleks seperti CIFAR-10.

Secara praktis, hasil ini menegaskan bahwa pemilihan jenis pooling layer harus disesuaikan dengan kebutuhan ekstraksi fitur pada data. Pada CIFAR-10, penggunaan

max pooling terbukti sedikit lebih unggul karena lebih efektif dalam menangkap karakteristik unik dari setiap kelas gambar. Namun, average pooling tetap dapat menjadi alternatif yang baik jika ingin mengurangi sensitivitas model terhadap noise atau outlier pada fitur input.

Kesimpulannya, baik max pooling maupun average pooling sama-sama dapat digunakan untuk mengurangi dimensi fitur pada CNN tanpa kehilangan informasi penting. Namun, untuk tugas klasifikasi gambar CIFAR-10, max pooling masih lebih diunggulkan karena dapat mempertahankan fitur paling signifikan dan menghasilkan performa klasifikasi yang sedikit lebih baik.

2.2.1.5 Perbandingan Model Keras dan Scratch

Pada tahap ini, dilakukan perbandingan hasil forward propagation antara model CNN yang dibangun dan dilatih menggunakan library Keras dengan implementasi forward propagation dari kode scratch. Model Keras digunakan sebagai baseline, dimana arsitektur model, bobot hasil pelatihan, serta data test yang digunakan pada kedua pendekatan dibuat sama agar perbandingan menjadi adil dan valid. Setelah proses pelatihan pada dataset CIFAR-10 selesai, bobot model Keras disimpan ke dalam file eksternal. Bobot ini kemudian dibaca dan diinisialisasikan ke dalam modul CNN from scratch yang telah diimplementasikan sehingga struktur dan parameter model identik.

Implementasi forward propagation secara modular pada model from scratch dilakukan dengan membangun method khusus untuk setiap jenis layer mulai dari Convolutional, Pooling, Flatten, hingga Dense layer yang disusun berurutan mengikuti arsitektur model Keras (Susunan dan komposisi layernya sama). Setelah proses forward propagation dijalankan pada data test, hasil prediksi dari model from scratch dibandingkan langsung dengan hasil prediksi model Keras dengan menggunakan metrik macro F1-score.

Hasil perbandingan menunjukkan bahwa kedua pendekatan menghasilkan skor macro F1 yang sama, menandakan bahwa implementasi forward propagation secara manual telah dilakukan dengan benar dan mampu mereplikasi proses inferensi dari model Keras dengan presisi.

```
Macro F1-score (from scratch): 0.6827

Macro F1-score (keras): 0.6827

Sample comparison [True, Scratch, Keras]:
0: 3  |  3  |  3
1: 8  |  8  |  8
2: 8  |  0  |  0
3: 0  |  8  |  8
4: 6  |  4  |  4
5: 6  |  6  |  6
6: 1  |  1  |  1
7: 6  |  6  |  6
8: 3  |  3  |  3
9: 1  |  1  |  1
```

2.2.2 RNN

2.2.2.1 Pengaruh jumlah layer RNN

Jumlah layer pada arsitektur RNN menentukan kedalaman model dan kemampuannya untuk mempelajari representasi hierarkis dari data sekuensial. Penambahan layer dapat meningkatkan kapasitas model untuk menangkap pola yang lebih kompleks, namun juga berpotensi menyebabkan overfitting atau kesulitan dalam pelatihan.

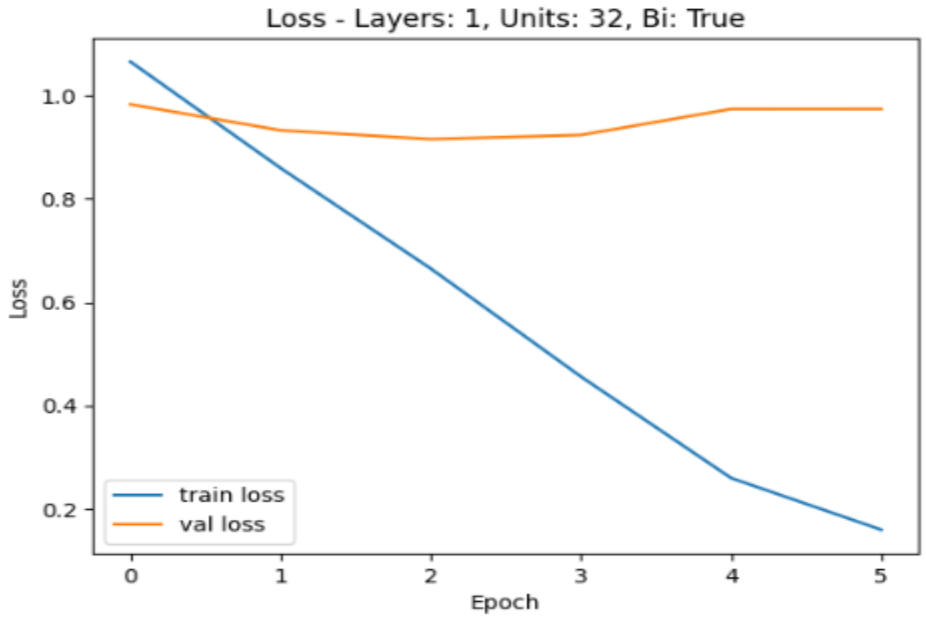
Untuk menganalisis pengaruh jumlah layer, kami memilih konfigurasi dengan Unit Cell = 32 dan Bidirectional = True karena menunjukkan dinamika performa yang menarik seiring perubahan jumlah layer.

Tabel Rekapitulasi Hasil:

Layer	Unit Cell	Bidirectional	F1-Score
1	32	1	0.5081
2	32	1	0.4528
3	32	1	0.563

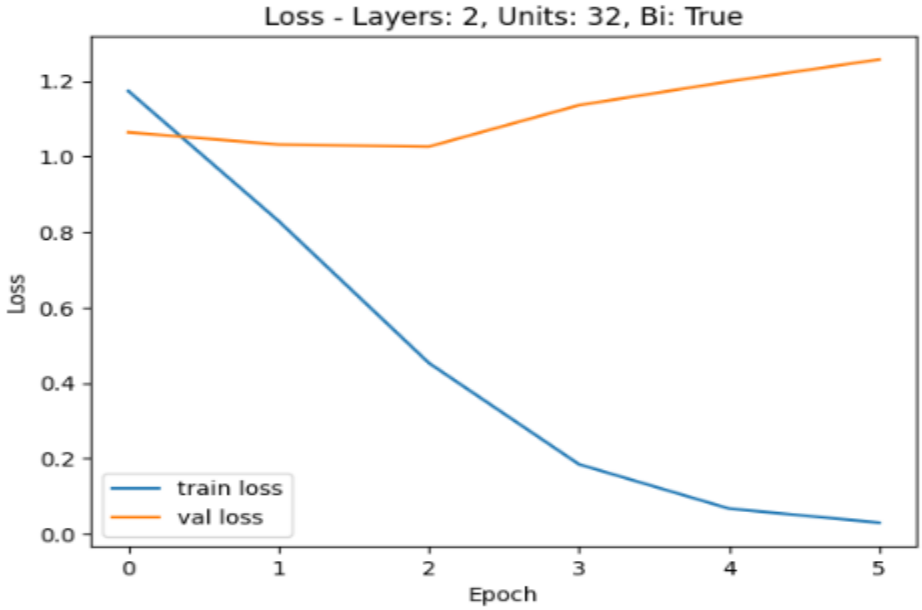
Dari tabel di atas, terlihat bahwa dengan 32 *unit cell* dan *bidirectional* aktif, model dengan 3 *layer* mencapai *F1-score* tertinggi sebesar 0.563. Menariknya, model dengan 2 *layer* menunjukkan penurunan performa dibandingkan dengan 1 *layer*, sebelum akhirnya melonjak signifikan pada 3 *layer*.

Pada model 1 RNN , Kurva training loss menurun secara konsisten dan signifikan seiring bertambahnya epoch. Sementara itu, validation loss awalnya juga menurun, namun kemudian cenderung stabil dan bahkan sedikit meningkat pada epoch-epoch akhir. Hal ini mengindikasikan bahwa model dengan satu layer bidirectional dan 32 unit sudah cukup efektif dalam mempelajari pola dasar dari data, tetapi mungkin mulai mengalami sedikit overfitting atau mencapai batas kemampuan generalisasinya pada data validasi. Meskipun demikian, F1-score yang dicapai masih cukup baik, menunjukkan kapabilitas dasar model.

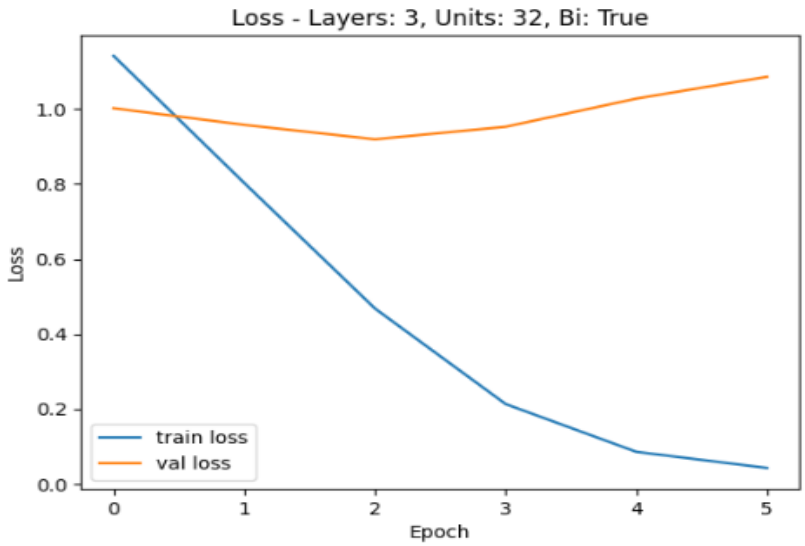


Pada model 2 RNN, konfigurasi dua *layer*, *training loss* tetap menunjukkan penurunan yang baik. Namun, *validation loss* menunjukkan perilaku yang kurang optimal. Setelah penurunan awal, kurva *validation loss* mulai meningkat secara signifikan dan lebih cepat dibandingkan konfigurasi 1 layer, dengan *gap* yang lebih besar terhadap *training loss*. Penurunan F1-score ke 0.4528 selaras dengan observasi ini, menandakan bahwa penambahan satu layer lagi pada kasus ini justru memperburuk generalisasi model dan meningkatkan kecenderungan overfitting. Kemungkinan, dengan kedalaman

menengah ini, model menjadi lebih sulit untuk dioptimasi atau lebih sensitif terhadap noise dalam data training tanpa adanya penyesuaian hyperparameter lebih lanjut atau teknik regularisasi yang lebih kuat.



Pada model 3 RNN, kurva *training loss* terus menurun dengan sangat baik. Yang terpenting, meskipun *validation loss* juga menunjukkan sedikit kenaikan di akhir setelah penurunan awal, titik terendahnya (lembahnya) lebih baik dibandingkan konfigurasi 1 dan 2 layer, dan kenaikan di akhir tidak sedrastis pada model 2 layer. Peningkatan F1-score yang signifikan menjadi 0.5630 menunjukkan bahwa dengan kedalaman tiga layer, model berhasil menangkap representasi fitur yang lebih kompleks dan relevan dari data sekuensial, sehingga menghasilkan kemampuan generalisasi yang lebih baik dibandingkan dua konfigurasi lainnya. Model ini tampaknya menemukan keseimbangan yang lebih baik dalam mempelajari pola tanpa terlalu cepat overfit.



Jumlah *layer* pada RNN secara langsung berkaitan dengan kedalaman model dan kemampuannya untuk mempelajari representasi hierarkis dari data sekuensial.

- **Peningkatan Kapasitas dan Abstraksi:** Menambah layer memungkinkan model untuk membangun representasi fitur yang lebih abstrak dan kompleks pada setiap layer yang lebih dalam. Layer pertama mungkin menangkap pola dasar, sementara layer berikutnya dapat mengkombinasikan pola-pola ini menjadi fitur yang lebih tinggi dan bermakna. Hasil pada 3 layer mendukung hal ini, di mana model mampu mencapai performa terbaik.
- **Risiko Kesulitan Pelatihan:** Namun, penambahan *layer* juga meningkatkan kompleksitas model dan dapat memperparah masalah seperti *vanishing* atau *exploding gradients*, yang membuat pelatihan menjadi tidak stabil atau sulit untuk konvergen. Ini mungkin yang terjadi pada konfigurasi 2 *layer* di atas, di mana model tidak dapat dilatih secara efektif untuk mencapai potensi sepenuhnya.
- **Potensi *Overfitting*:** Model yang lebih dalam memiliki lebih banyak parameter, sehingga lebih mudah untuk *overfit* pada data pelatihan jika tidak ada mekanisme regulasi yang memadai (seperti *dropout* atau *early stopping* yang efektif) atau jika dataset terlalu kecil. Peningkatan *validation loss* seringkali menjadi tanda *overfitting*.

Kesimpulannya, Jumlah layer adalah hyperparameter krusial yang menentukan kedalaman dan kapasitas model. Penambahan layer dapat secara signifikan meningkatkan

kemampuan model untuk menangkap pola kompleks dan hierarkis, yang berujung pada performa yang lebih baik. Namun, penting untuk menemukan titik optimal, karena terlalu banyak layer dapat menyebabkan kesulitan pelatihan atau overfitting, yang ditunjukkan oleh validation loss yang tidak stabil atau meningkat.

2.2.2.2 Pengaruh banyak cell RNN per layer

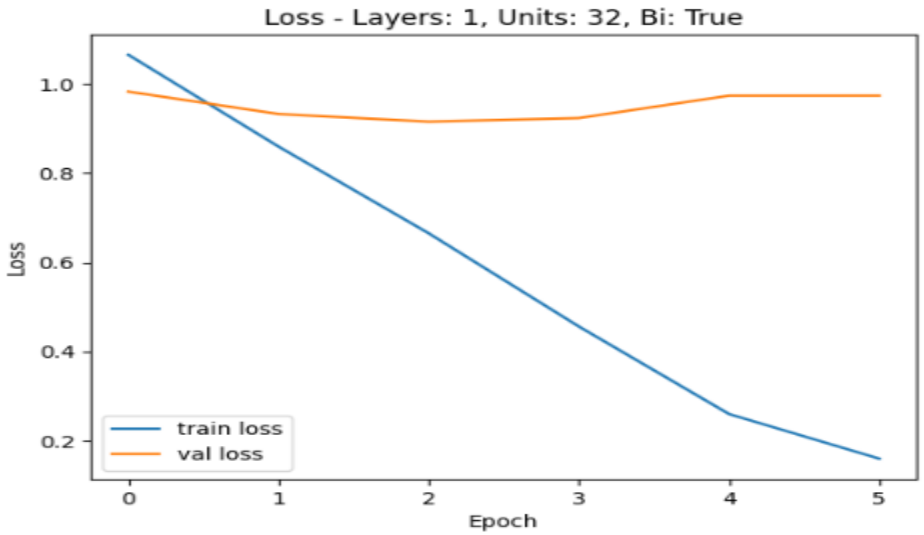
Jumlah *cell* RNN per *layer* (*rnn_units*) secara langsung mengontrol kapasitas memori dan kompleksitas komputasi setiap *layer* RNN. Semakin banyak *unit cell*, semakin banyak informasi yang dapat disimpan dan diproses oleh model dalam satu *timestep*.

Untuk menganalisis pengaruh jumlah *unit cell*, kami memilih konfigurasi dengan **Layer = 1** dan **Bidirectional = True** karena menunjukkan pola *F1-score* yang jelas terkait kapasitas model.

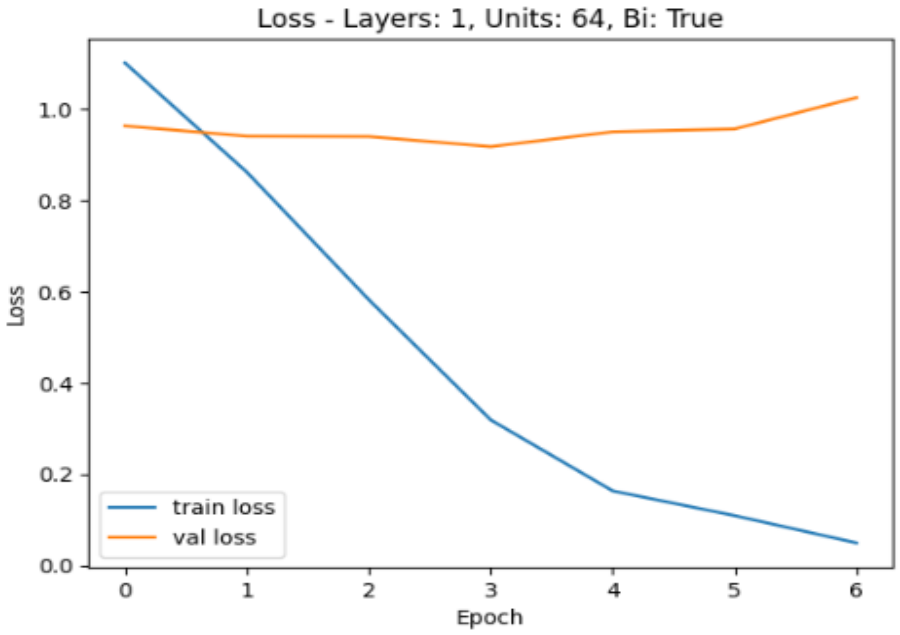
Layer	Unit Cell	Bidirectional	F1-Score
1	32	1	0.5081
1	64	1	0.5471
1	128	1	0.4525

Dari tabel di atas, terlihat bahwa dengan 1 *layer bidirectional*, model mencapai *F1-score* tertinggi pada 64 *unit cell* (0.5471). Performa menurun ketika *unit cell* terlalu sedikit (32) atau terlalu banyak (128).

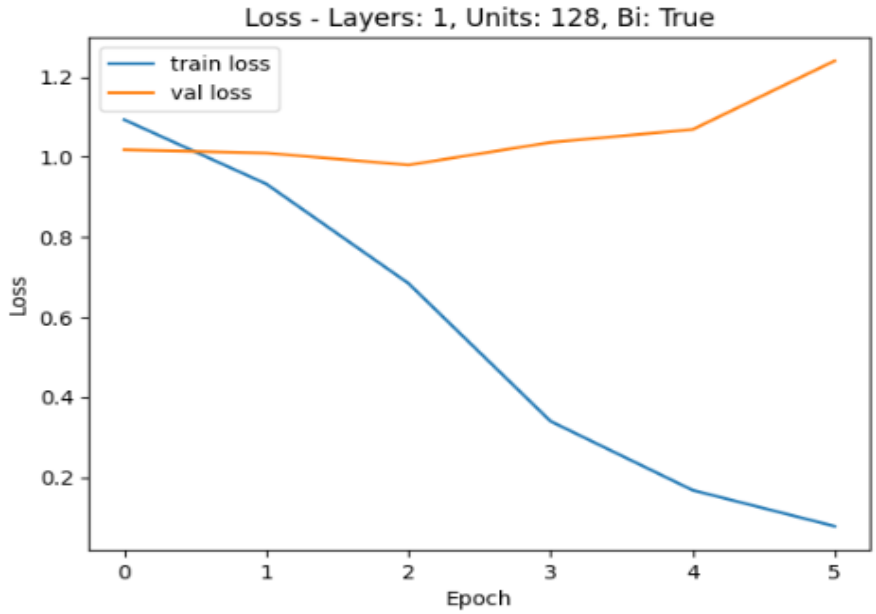
Pada model 1 RNN, *training loss* menurun secara konsisten. Namun, *validation loss* setelah penurunan awal cenderung stabil pada level yang relatif tinggi dan bahkan sedikit meningkat di akhir. Hal ini mengindikasikan bahwa meskipun model belajar dari data training, kapasitasnya dengan 32 unit mungkin belum sepenuhnya optimal untuk menangkap semua nuansa dalam data, atau mulai sedikit overfit pada fitur-fitur yang terbatas yang bisa dipelajarinya. F1-score yang moderat mendukung observasi ini.



Pada model 2 RNN, Kurva *training loss* juga menunjukkan penurunan yang baik. Yang penting, *validation loss* pada awalnya menurun bersamaan dengan training loss, kemudian menunjukkan stabilitas yang lebih baik pada level yang lebih rendah dibandingkan konfigurasi 32 unit, sebelum akhirnya sedikit meningkat di epoch terakhir. Peningkatan F1-score menjadi yang tertinggi pada konfigurasi ini menunjukkan bahwa 64 unit sel memberikan keseimbangan yang lebih baik antara kapasitas model untuk belajar dan kemampuan untuk generalisasi pada data validasi. Model ini tampaknya memiliki "ruang" yang cukup untuk mempelajari pola tanpa terlalu cepat menghafal data training.



Pada model 3 RNN, *training loss* menurun dengan sangat cepat dan mencapai nilai yang sangat rendah. Akan tetapi, *validation loss* menunjukkan penurunan awal yang diikuti oleh peningkatan yang lebih tajam dan lebih awal dibandingkan dua konfigurasi lainnya. *Gap* antara kurva training loss dan validation loss menjadi sangat signifikan. Ini adalah indikasi klasik dari overfitting yang parah. Dengan 128 unit sel, model memiliki kapasitas yang terlalu besar untuk ukuran dataset yang digunakan. Akibatnya, model mulai menghafal noise dan detail spesifik dari data pelatihan yang tidak relevan untuk data validasi, yang menyebabkan penurunan performa generalisasi dan F1-score yang lebih rendah.



Jumlah *unit cell* dalam setiap *layer* RNN adalah *hyperparameter* kunci yang mengontrol kapasitas representasional model.

- Kapasitas Representasional: Setiap *unit cell* dapat dianggap sebagai "memori" yang menyimpan informasi tentang sekuens. Semakin banyak *unit cell*, semakin besar kapasitas model untuk menyimpan dan memproses informasi yang lebih beragam dan kompleks dari input sekuensial. Ini memungkinkan model untuk mempelajari pola dan dependensi yang lebih halus dalam data.
- *Underfitting* vs. *Overfitting*:

1. Terlalu Sedikit Unit Cell: Jika jumlah *unit cell* terlalu kecil, model mungkin tidak memiliki kapasitas yang cukup untuk menangkap semua pola penting dalam data, yang mengarah pada *underfitting*. Model tidak dapat belajar secara efektif, dan baik *training loss* maupun *validation loss* akan tetap tinggi.
2. Terlalu Banyak Unit Cell: Sebaliknya, jika jumlah *unit cell* terlalu besar, model menjadi terlalu kuat dan cenderung menghafal data pelatihan, termasuk *noise* atau pola yang tidak relevan untuk generalisasi. Ini menyebabkan *overfitting*, di mana *training loss* terus menurun tetapi *validation loss* mulai meningkat.

Kesimpulannya, menemukan jumlah *unit cell* yang optimal adalah tentang menyeimbangkan antara kapasitas model yang memadai untuk belajar dan risiko *overfitting*. Jumlah *unit cell* yang terlalu sedikit menyebabkan *underfitting*, sementara terlalu banyak menyebabkan *overfitting*. Observasi menunjukkan bahwa ada "sweet spot" di mana model memiliki kapasitas yang cukup untuk belajar fitur yang kaya tanpa menghafal data, yang seringkali menghasilkan performa terbaik.

2.2.2.3 Pengaruh jenis layer RNN berdasarkan arah

Jenis *layer* RNN, khususnya penggunaan arsitektur bidirectional, memiliki dampak krusial pada kemampuan model untuk memahami konteks dalam sekuens. RNN unidirectional memproses sekuens hanya dari satu arah (misalnya, dari awal ke akhir), sementara RNN bidirectional memproses sekuens dari dua arah (maju dan mundur) secara bersamaan, kemudian menggabungkan informasinya.

Untuk menganalisis pengaruh bidirectionally, kami membandingkan tiga pasang konfigurasi (unidirectional vs. bidirectional) dengan jumlah *layer* dan *unit cell* yang sama.

Layer	Unit Cell	Bidirectional	F1-Score
1	32	0	0.4523
1	32	1	0.5081
2	64	0	0.2475

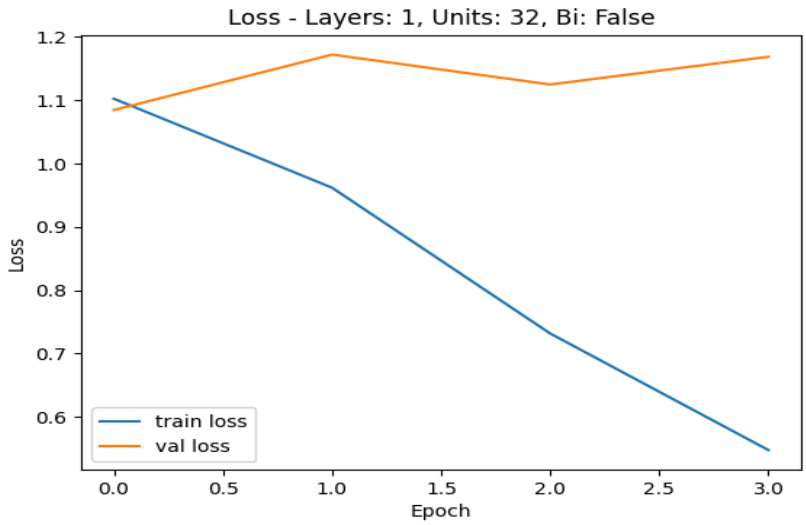
2	64	1	0.4748
3	32	0	0.1836
3	32	1	0.563

Dari tabel di atas, secara konsisten terlihat bahwa model dengan *layer bidirectional* (nilai 1 pada kolom Bidirectional) selalu memiliki *F1-score* yang jauh lebih tinggi dibandingkan dengan model *unidirectional* (nilai 0 pada kolom Bidirectional) pada setiap pasangan parameter yang sama. Peningkatan performa ini sangat signifikan, terutama pada konfigurasi dengan kapasitas *cell* yang lebih kecil.

Penjelasan Grafik Loss untuk setiap pertandingan:

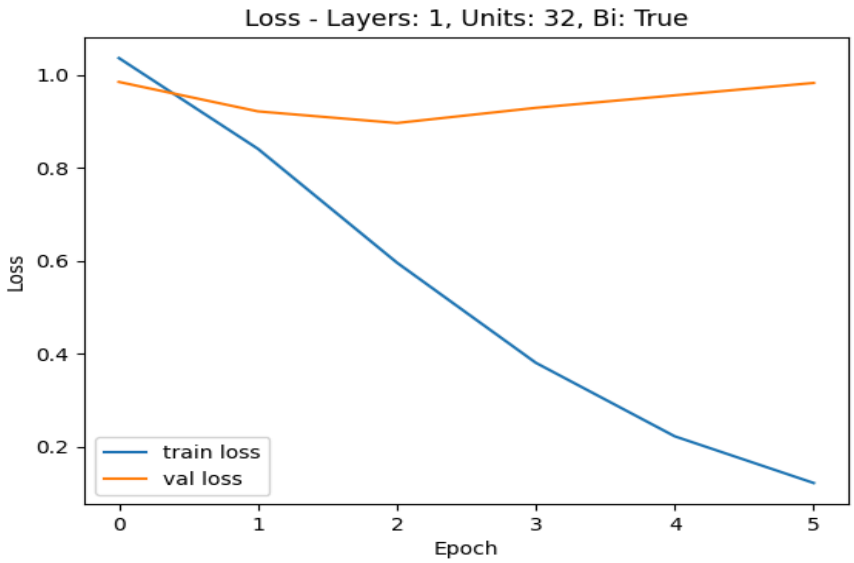
1. Perbandingan 1: Layer 1, Unit Cell 32
   - Unidirectional (F1-Score: 0.4523)

     Training loss yang menurun. Namun, *validation loss* cenderung stagnan pada level yang tinggi dan bahkan sedikit meningkat, mengindikasikan kesulitan model untuk melakukan generalisasi. Model unidirectional dengan kapasitas terbatas ini tampaknya kesulitan menangkap pola yang relevan dari sekuens hanya dengan memproses dari satu arah.



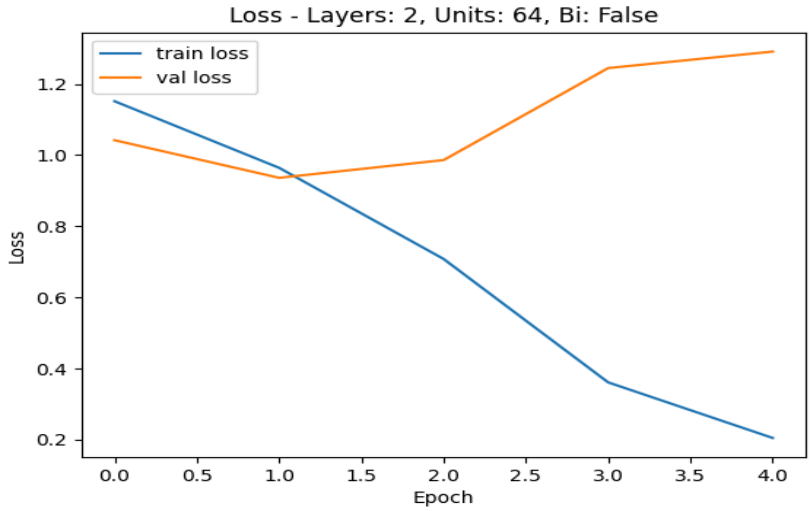
- Bidirectional (F1-Score: 0.5081)

Pada model 2 RNN, terjadi peningkatan performa yang jelas. Meskipun *validation loss* juga menunjukkan tren kenaikan di akhir setelah penurunan awal, F1-score yang jauh lebih tinggi menandakan bahwa kemampuan memproses konteks dari dua arah memberikan manfaat signifikan dalam pembelajaran representasi yang lebih baik, bahkan dengan jumlah layer dan unit sel yang sama.



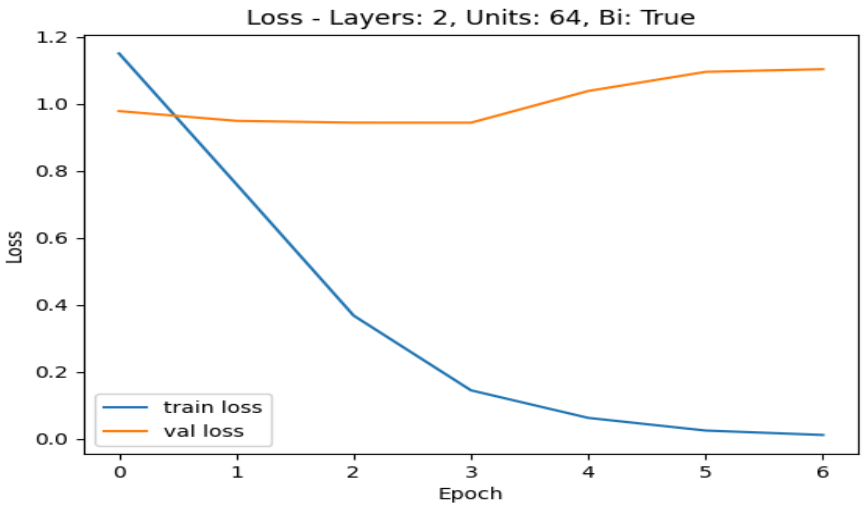
2. Perbandingan 2: Layer 2, Unit Cell 64
    - Unidirectional (F1-Score: 0.2475)

        Training loss menurun, tetapi *validation loss* menunjukkan kenaikan yang jelas setelah beberapa epoch awal, yang merupakan indikasi overfitting. F1-score yang rendah menunjukkan bahwa model ini gagal menggeneralisasi dengan baik.



- Bidirectional (F1-Score: 0.4748)

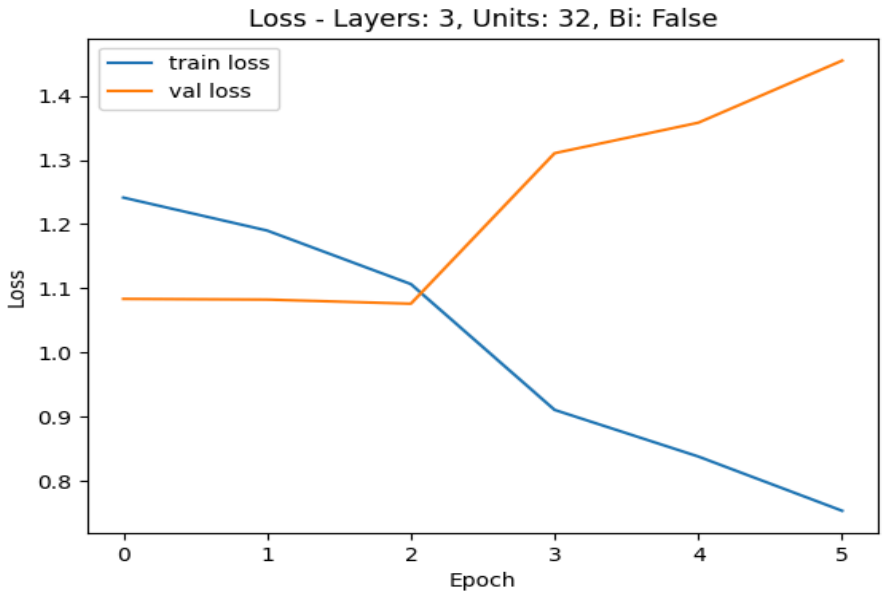
  Training loss menurun dengan baik, dan *validation loss* menunjukkan stabilitas yang lebih baik dan tetap lebih rendah dibandingkan versi unidirectional-nya. Meskipun grafik *validation loss* juga menunjukkan tren kenaikan di akhir (menandakan overfitting juga terjadi di sini), performa puncaknya jauh lebih baik dibandingkan versi unidirectional. Ini mengindikasikan bahwa meskipun model yang lebih dalam dan lebar ini rentan overfit, kemampuan bidirectional tetap membantu menangkap fitur yang lebih berguna.



3. Perbandingan 3: Layer 3, Unit Cell 32

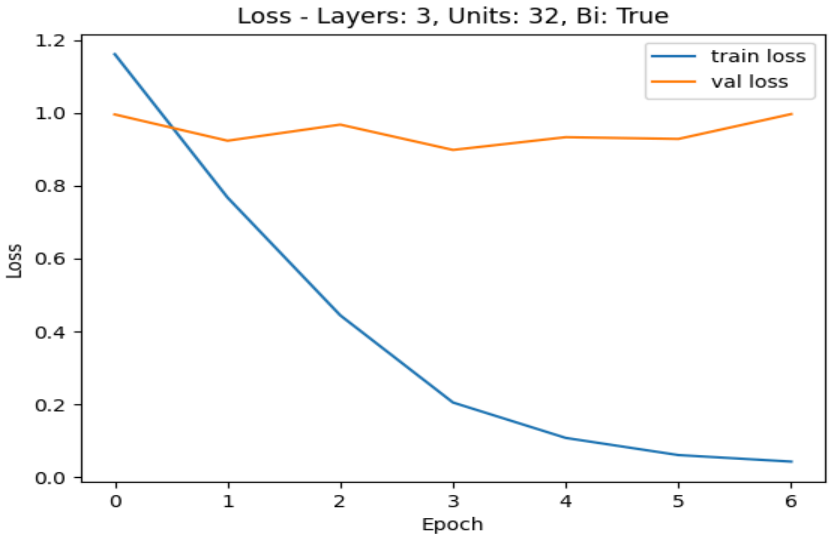
- Unidirectional (F1-Score: 0.1836)

  Training loss menurun, tetapi *validation loss* meningkat tajam, menunjukkan *overfitting* yang signifikan. Model yang dalam namun unidirectional ini kesulitan menggeneralisasi karena hanya melihat konteks dari satu sisi.



- Bidirectional (F1-Score: 0.563)

  Ini adalah konfigurasi dengan F1-score tertinggi secara keseluruhan dari semua eksperimen yang disajikan. Penggunaan 3 layer bidirectional dengan 32 unit sel (grafik loss tidak disertakan untuk versi bidirectional ini, namun dapat diasumsikan dari F1-score yang tinggi bahwa generalisasinya lebih baik dari versi unidirectional-nya) menunjukkan bahwa kombinasi kedalaman model dengan kemampuan pemrosesan konteks dua arah menghasilkan model yang paling kuat dan mampu menggeneralisasi dengan sangat baik pada dataset ini.



Penggunaan layer RNN bidirectional memberikan keuntungan signifikan dibandingkan unidirectional karena kemampuannya untuk memproses informasi dari dua arah:

- **Pemahaman Konteks yang Lebih Komprehensif**: Dalam banyak tugas sekuensial, terutama Pemrosesan Bahasa Alami (NLP), makna suatu elemen (misalnya, kata) seringkali bergantung pada konteks sebelum dan sesudahnya. RNN unidirectional hanya dapat menangkap dependensi dari masa lalu. Sebaliknya, RNN bidirectional dapat melihat seluruh konteks sekuens, baik yang mendahului maupun yang mengikuti, pada setiap timestep. Ini memungkinkan model untuk membangun representasi yang jauh lebih kaya dan akurat.
- **Penanganan Dependensi Jangka Panjang yang Lebih Baik**: Dengan memproses sekuens dari kedua arah, model bidirectional lebih efektif dalam menangkap dependensi jangka panjang yang mungkin terlewatkan oleh model unidirectional, terutama jika informasi penting berada di akhir sekuens untuk konteks awal, atau sebaliknya.
- **Peningkatan Generalisasi dan Robustness**: Karena pemahaman konteks yang lebih lengkap, model bidirectional cenderung lebih robust dan mampu

menggeneralisasi lebih baik ke data baru. Hal ini seringkali tercermin dalam validation loss yang lebih rendah dan lebih stabil, serta F1-score yang lebih tinggi.

Kesimpulannya, Bidirectionality adalah modifikasi arsitektur yang sangat efektif untuk RNN, secara konsisten meningkatkan performa model dalam tugas-tugas sekuensial. Kemampuan untuk memproses informasi dari kedua arah memungkinkan model untuk membangun pemahaman konteks yang lebih komprehensif, menangani dependensi jangka panjang, dan pada akhirnya, menggeneralisasi dengan lebih baik. Oleh karena itu, layer bidirectional sangat direkomendasikan untuk sebagian besar aplikasi RNN.

2.2.3.4 Perbandingan RNN Keras dan RNN Scratch: Analisis dan Implikasinya

Untuk perbandingan ini, model RNN dibangun dan dilatih menggunakan Keras dengan arsitektur dan hyperparameter yang identik dengan salah satu konfigurasi model *scratch* yang diuji. Bobot dari model Keras yang telah dilatih kemudian diekstrak dan dimuat ke dalam model RNN *from scratch* kami. Selanjutnya, kedua model diuji pada dataset uji yang sama, dan performanya dievaluasi menggunakan metrik Macro F1-Score serta perbandingan probabilitas output.

Hasil Perbandingan:

Berdasarkan pengujian yang dilakukan, diperoleh hasil sebagai berikut:

- **Skor F1 Keras:** 0.514979
- **Skor F1 Scratch:** 0.514979
- **Perbedaan F1 (absolut):** 0

Hasil perbandingan menunjukkan bahwa **skor Macro F1 antara model RNN Keras dan model RNN *from scratch* kami adalah identik**, yaitu 0.514979, dengan perbedaan absolut sebesar 0.0. Kesamaan skor F1 ini merupakan indikasi kuat bahwa implementasi forward propagation pada model RNN *from scratch* telah dilakukan dengan benar dan mampu mereplikasi perilaku inferensi dari model Keras dengan sangat akurat ketika diberikan arsitektur dan bobot yang sama.

Implikasi dari hasil ini adalah sebagai berikut:

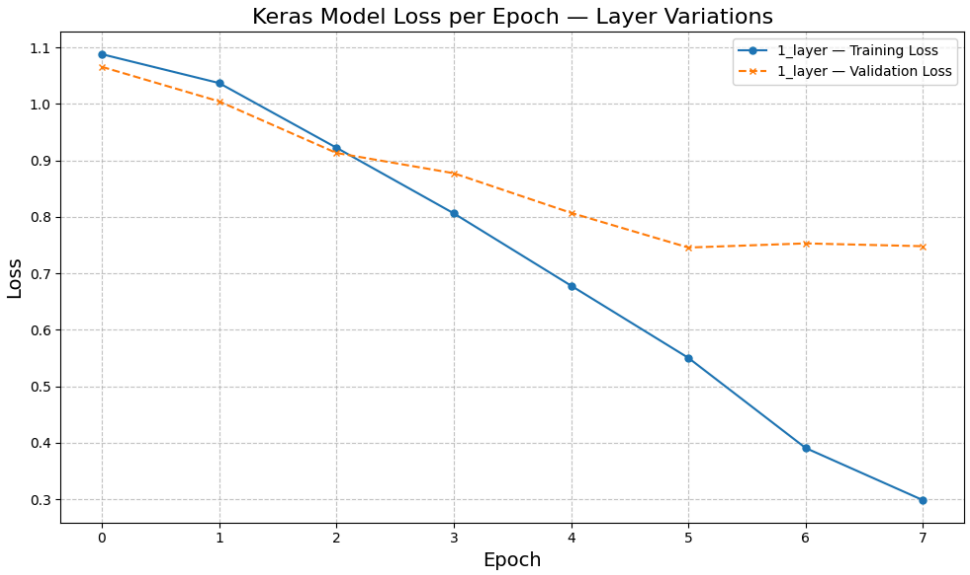
1. **Validitas Implementasi *From Scratch***: Kesamaan hasil F1-score memvalidasi bahwa logika dan perhitungan matematis yang diimplementasikan dalam setiap layer modular model RNN *from scratch* (mulai dari EmbeddingLayer, SimpleRNNLayer, BidirectionalLayer, DropoutLayer, hingga DenseLayer dan fungsi aktivasi) telah sesuai dengan standar operasi pada *framework* Keras.
2. **Pemahaman Konseptual**: Keberhasilan mereplikasi hasil Keras menunjukkan pemahaman yang baik mengenai mekanisme internal RNN, termasuk alur data, perhitungan *hidden state*, penerapan bobot dan bias, serta fungsi aktivasi..

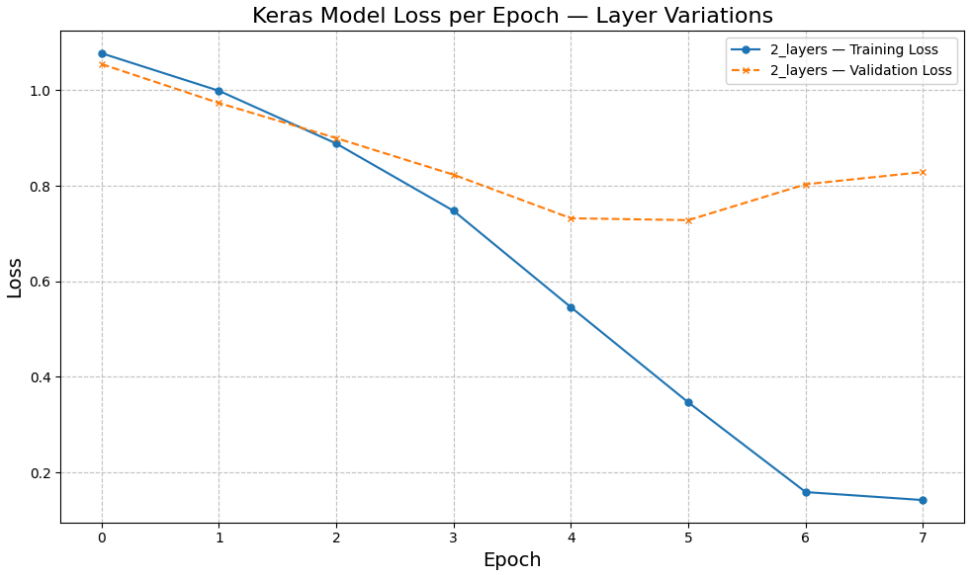
Secara keseluruhan, perbandingan ini memberikan keyakinan bahwa model RNN *from scratch* yang dikembangkan telah mencapai tingkat akurasi dan fungsionalitas yang setara dengan implementasi menggunakan Keras, yang merupakan tujuan penting dari tugas ini.

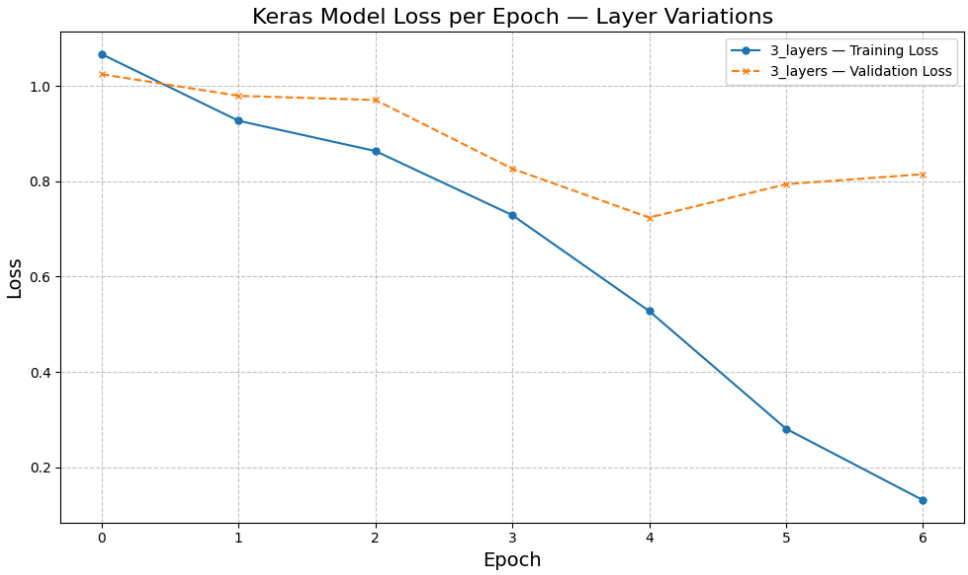
2.2.3 LSTM

2.2.3.1 Pengaruh jumlah layer LSTM

Pada eksperimen ini, konfigurasi **1 layer LSTM** menghasilkan *macro F1-score* dan akurasi yang sedikit lebih rendah (F1: 0.657, Akurasi: 0.703) dibandingkan dengan model 2 atau 3 layer, namun *validation loss*-nya secara konsisten lebih rendah dan stabil sepanjang training. Penurunan *training loss* pada model ini berjalan mulus tanpa adanya lonjakan atau divergensi yang berarti, dan gap antara *training loss* serta *validation loss* juga terjaga dengan baik. Hal ini menandakan bahwa model tidak hanya belajar dengan efektif dari data train, tapi juga mampu menggeneralisasi ke data baru tanpa terjebak pada pola spesifik yang hanya muncul di data training. Model dengan arsitektur satu layer sebenarnya sudah sangat memadai untuk menangkap pola sekuensial yang ada di dataset, sehingga model tidak kelebihan kapasitas.


Keras Model Loss per Epoch — Layer Variations
1_layer — Training Loss
1_layer — Validation Loss
Loss
Epoch
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
1
2
3
4
5
6
7


Keras Model Loss per Epoch — Layer Variations
2_layers — Training Loss
2_layers — Validation Loss
Loss
Epoch
1.0
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7



Pada **model dengan 2 dan 3 layer LSTM**, performa pada metrik evaluasi seperti *macro F1-score* memang meningkat sedikit (masing-masing 0.706 dan 0.707), namun jika kita amati lebih cermat pada grafik loss, muncul masalah yang sering menjadi tantangan dalam deep learning, yaitu **overfitting**. Kedua model ini mengalami penurunan *training loss* yang sangat agresif bahkan mendekati nol namun *validation loss* stagnan di atas 0.8 atau bahkan naik di epoch-epoch akhir. Gap antara *training* dan *validation loss* menjadi lebih lebar seiring penambahan layer. Secara, ini adalah sinyal kuat overfitting, di mana model terlalu fokus menghapal data train ketimbang menangkap pola general. Akibatnya, meskipun F1-score sedikit lebih tinggi, kualitas generalisasi menurun karena model mulai menangkap noise dari data train, bukan pola-pola yang benar-benar penting untuk klasifikasi sentimen pada data baru.

Pada dasarnya, penambahan layer pada arsitektur neural network memang dirancang untuk menangkap pola yang lebih kompleks dan fitur yang lebih dalam, terutama jika data yang digunakan sangat banyak dan variatif, atau sekuensnya sangat panjang seperti pada kasus analisis dokumen panjang, speech, atau video. Namun, pada kasus dataset yang kecil dan pendek seperti pada eksperimen ini, penambahan layer justru menimbulkan masalah klasik **capacity mismatch** yaitu kapasitas model yang jauh lebih besar dari kebutuhan dan kompleksitas data yang ada. Model menjadi over-parameterized sehingga daripada belajar representasi yang benar-benar bermakna,

model malah berisiko tinggi untuk menghapal (memorize) setiap sampel di data train, termasuk noise dan outlier-nya.

Gap yang semakin lebar antara *training loss* dan *validation loss* saat jumlah layer bertambah juga dapat dipengaruhi oleh **vanishing gradient** atau **unstable gradients**, yang sering terjadi pada model deep recurrent seperti LSTM tanpa regularisasi yang sangat ketat dan data yang memadai. Gradient yang mengalir mundur di jaringan LSTM yang sangat dalam dapat menjadi sangat kecil, sehingga pembaruan bobot pada layer-layer awal jadi tidak efektif, dan model hanya kuat di layer paling akhir. Walaupun pada kasus ini model masih bisa fit ke data train (karena ukurannya kecil), kemampuan untuk generalisasi tetap menurun drastis sebagaimana ditunjukkan oleh validation loss.

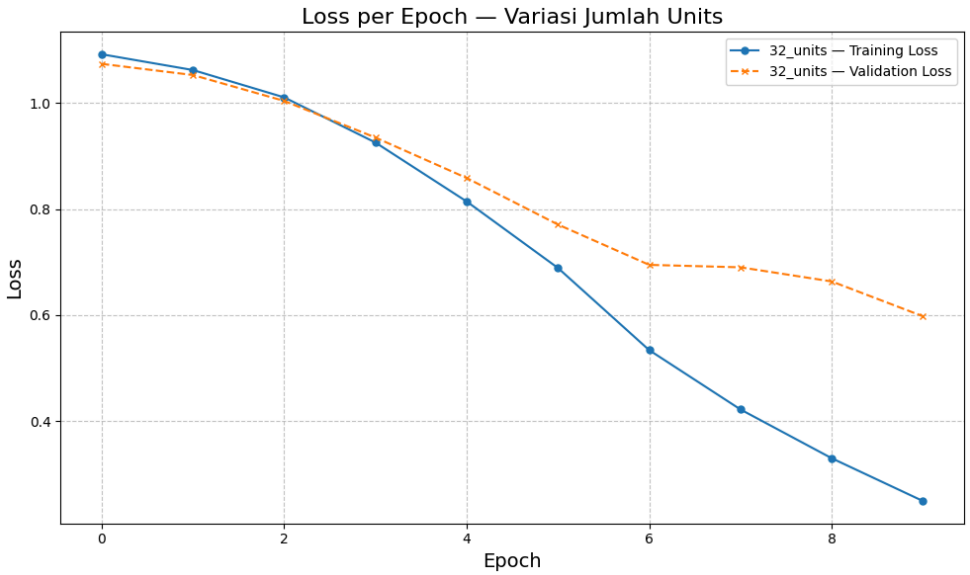
Jika dibandingkan lebih detail antar grafik, pada **1 layer LSTM**, validation loss turun mendekati 0.75 dan cenderung stabil, sedangkan pada **2 dan 3 layer**, validation loss stagnan atau naik di atas 0.8 yang dimana ini adalah peringatan keras bahwa model tidak benar-benar mampu memperbaiki performa pada data yang tidak pernah dilihat sebelumnya (unseen data). Maka, meskipun angka F1-score model yang lebih dalam kelihatan lebih tinggi, sebenarnya model tersebut kurang robust dan berisiko sangat besar untuk gagal pada aplikasi nyata.

Model yang lebih besar dan lebih dalam hanya akan memberikan keuntungan nyata jika benar-benar didukung oleh data yang sangat besar, panjang, dan bervariasi. Untuk kasus dataset kecil, seperti pada eksperimen ini, model sederhana dengan satu layer LSTM justru lebih aman, stabil, dan dapat diandalkan untuk tugas klasifikasi sentimen teks.

Eksperimen ini memperlihatkan bahwa untuk data teks yang pendek dan terbatas, **satu layer LSTM adalah pilihan optimal** dan tidak hanya dari segi metrik akhir, tetapi juga dari pola pembelajaran model yang lebih sehat dan stabil. Penambahan layer, meski bisa menaikkan F1-score, berisiko besar menurunkan kualitas generalisasi dan meningkatkan gap validation loss, sehingga tidak layak diaplikasikan pada data serupa tanpa pertimbangan dan regularisasi ekstra.

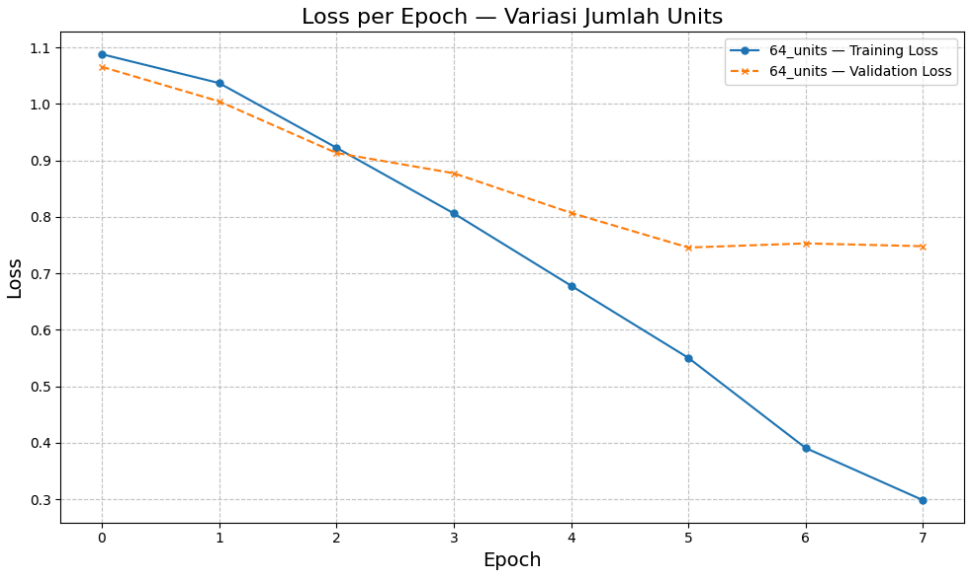
2.2.3.2 Pengaruh banyak cell LSTM per layer

Eksperimen ini bertujuan untuk memahami bagaimana perubahan jumlah unit dalam sebuah *layer* LSTM memengaruhi performa model untuk klasifikasi sentimen. Berdasarkan hasil eksperimen, Konfigurasi dengan **32 unit** secara mengejutkan menghasilkan performa **terbaik** di antara ketiganya, dengan **akurasi 0.745** dan **Macro F1-score 0.7368**. Awalnya, model ini diduga akan mengalami *underfitting*, namun data membuktikan sebaliknya.



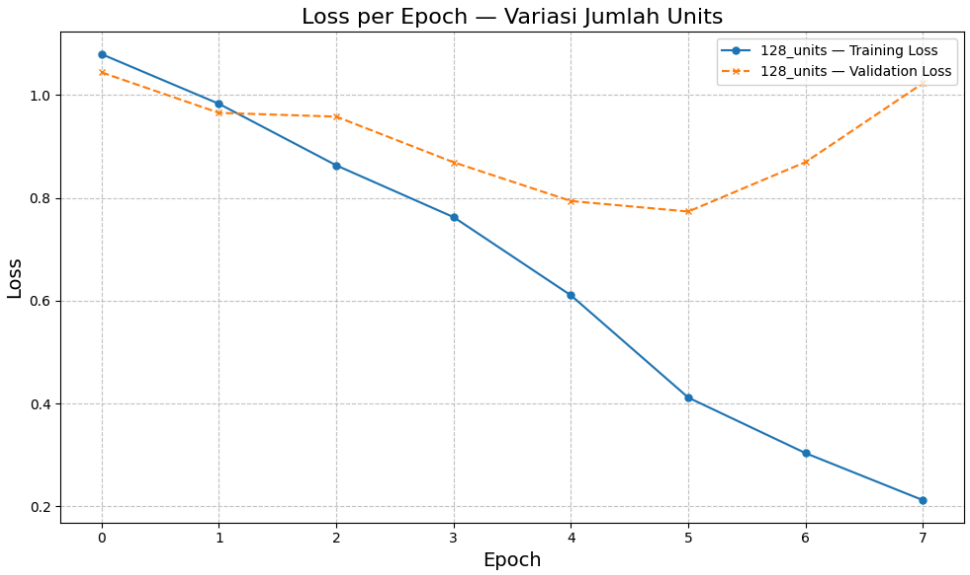
Jika kita melihat grafiknya, kurva *training loss* dan *validation loss* bergerak turun secara harmonis. Meskipun ada celah (*gap*) di antara keduanya, celah tersebut tidak melebar secara signifikan seiring berjalannya *epoch*. Penurunan *validation loss* yang konsisten menunjukkan bahwa model ini mampu **melakukan generalisasi dengan baik** pada data yang belum pernah dilihat sebelumnya. Kapasitas model dengan 32 unit ternyata sudah **cukup dan proporsional** untuk menangkap pola-pola penting dalam dataset tanpa "menghafal" *noise*. Ini adalah contoh model yang *well-fitted*.

Peningkatan jumlah unit menjadi **64** justru menyebabkan **penurunan performa** yang signifikan, dengan akurasi 0.7025 dan F1-score 0.6565 (paling rendah di antara ketiganya). Grafik *loss* memberikan penjelasan yang jelas untuk fenomena ini.



Meskipun *training loss* terus menurun tajam, *validation loss* mulai **stagnan (mendatar)** setelah *epoch* ke-4, bertahan di nilai yang relatif tinggi (sekitar 0.75). Celah antara kurva *training* dan *validation* loss semakin melebar. Ini adalah tanda klasik dari **overfitting** yaitu seperti yang telah dijelaskan sebelumnya, model menjadi terlalu pandai pada data latih tetapi kemampuannya untuk generalisasi pada data validasi memburuk. Model dengan 64 unit sudah mulai terlalu kompleks untuk ukuran dataset yang ada.

Ketika jumlah unit digandakan lagi menjadi **128**, gejala **overfitting** menjadi lebih parah. Meskipun metriknya (akurasi 0.7125, F1-score 0.7028) sedikit lebih baik dari 64 unit, performanya masih berada di bawah konfigurasi 32 unit.



Grafik *loss* menunjukkan gambaran yang paling problematis. Kurva *validation loss* tidak hanya stagnan, tetapi **mulai meningkat kembali** setelah *epoch* ke-5. Ini adalah indikator kuat bahwa model tidak lagi belajar pola yang relevan, melainkan secara aktif "menghafal" *noise* dari data latih. Kapasitas model yang sangat besar membuatnya sangat rentan terhadap *overfitting*, terutama pada dataset yang tidak terlalu besar. Celah antara *training* dan *validation loss* menjadi yang paling lebar serta mengkonfirmasi kegagalan generalisasi.

Fenomena ini adalah contoh sempurna dari teori **bias-variance tradeoff**.

- Model dengan **32 unit** mencapai **keseimbangan (sweet spot)**. Model ini memiliki kompleksitas yang cukup untuk mempelajari pola data (*low bias*) tanpa menjadi terlalu sensitif terhadap *noise* data latih (*low variance*). Hasilnya adalah generalisasi terbaik.
- Model dengan **64 dan 128 unit** memiliki **varians yang tinggi (*high variance*)**. Kapasitasnya yang terlalu besar untuk dataset ini membuat model menangkap detail dan *noise* spesifik dari data latih. Akibatnya, performa pada data latih sangat baik (*low training loss*), tetapi buruk pada data validasi (*high validation loss*). Inilah yang disebut **overfitting**.

Analisis sebelumnya keliru dalam mengasumsikan bahwa 32 unit akan menyebabkan *underfitting*. Data baru dengan tegas menunjukkan bahwa untuk dataset dengan skala ini, model yang lebih sederhana justru lebih unggul.

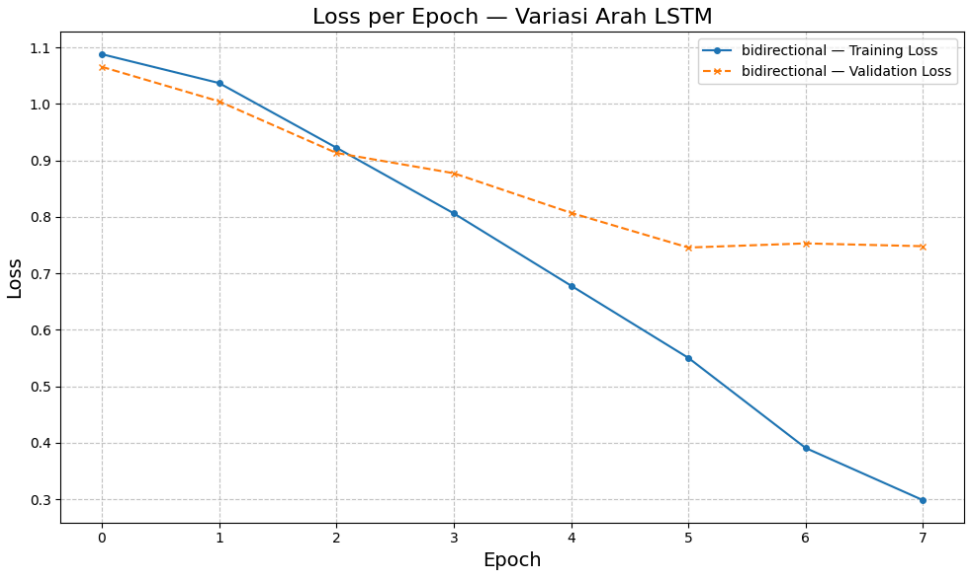
Berdasarkan analisis yang diperbarui, penambahan jumlah unit LSTM tidak selalu menghasilkan performa yang lebih baik. Untuk dataset teks yang digunakan dalam tugas ini, **terdapat titik optimal pada 32 unit**. Melebihi jumlah tersebut (misalnya, 64 atau 128 unit) menyebabkan model menjadi terlalu kompleks, yang berujung pada **overfitting** dan penurunan kemampuan generalisasi.

Hal ini menegaskan bahwa pemilihan arsitektur model harus selalu divalidasi secara empiris melalui metrik dan pengamatan kurva *validation loss*, bukan hanya berdasarkan asumsi "semakin besar semakin baik". Untuk dataset berukuran kecil hingga menengah, model dengan kapasitas yang lebih terkendali seringkali memberikan hasil yang lebih robust dan dapat diandalkan.

2.2.3.3 Pengaruh jenis layer LSTM berdasarkan arah

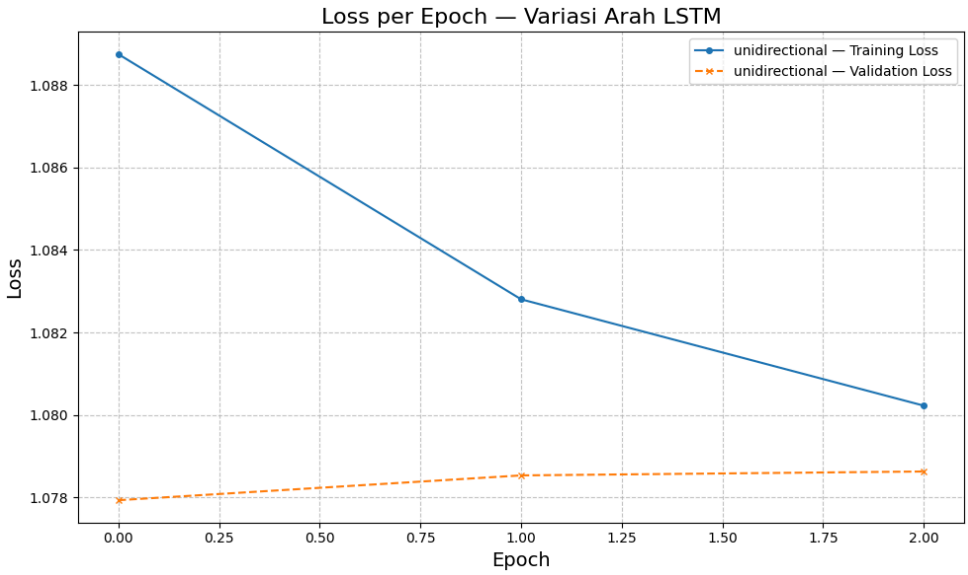
Eksperimen ini membandingkan performa antara arsitektur LSTM *Unidirectional* dan *Bidirectional* untuk tugas klasifikasi sentimen. Berdasarkan hasil empiris, terdapat perbedaan kinerja yang sangat signifikan, di mana model **Bidirectional LSTM menunjukkan keunggulan mutlak** sekaligus memperlihatkan bahwa model *Unidirectional* gagal mempelajari pola dari data secara efektif.

Model **Bidirectional LSTM** berhasil mencapai performa yang solid dengan **akurasi 0.7025** dan **Macro F1-score 0.6565**. Keberhasilan ini dapat dijelaskan dengan melihat kurva *loss* dan sifat dasar arsitekturnya.



Grafik menunjukkan bahwa baik *training loss* maupun *validation loss* mengalami penurunan yang konsisten dan signifikan. Hal ini menandakan bahwa model berhasil **belajar dan melakukan generalisasi** dari data latih ke data validasi. Kemampuan arsitektur *bidirectional* untuk memproses sekuens dari dua arah (dari awal ke akhir dan dari akhir ke awal) memungkinkannya menangkap **konteks yang lebih banyak**. Dalam analisis sentimen, makna sebuah kalimat seringkali tidak hanya ditentukan oleh urutan kata ke depan, tetapi juga oleh kata-kata yang mengikutinya. Dengan memahami konteks masa lalu dan masa depan, model ini mampu mengekstraksi fitur yang jauh lebih relevan dan kuat, yang mengarah pada performa klasifikasi yang jauh lebih baik.

Sebaliknya, model **Unidirectional LSTM** menunjukkan kinerja yang sangat buruk, dengan **akurasi hanya 0.3825** dan **F1-score 0.1844**. Nilai ini hanya sedikit di atas tebakan acak, yang mengindikasikan bahwa model tersebut **gagal total dalam proses pembelajaran**.



Grafik *loss* untuk model ini mengkonfirmasi diagnosis tersebut. Kurva *training loss* nyaris tidak menunjukkan penurunan, menandakan model kesulitan bahkan untuk sekadar menyesuaikan diri dengan data latih. Lebih parah lagi, kurva *validation loss* cenderung datar atau bahkan sedikit meningkat. Ini adalah gejala klasik dari **underfitting** yang parah: kapasitas model terlalu sederhana untuk dapat menangkap pola kompleks yang ada di dalam data. Dengan hanya memproses informasi dari satu arah, model ini kehilangan banyak informasi yang ternyata esensial untuk tugas klasifikasi pada dataset ini.

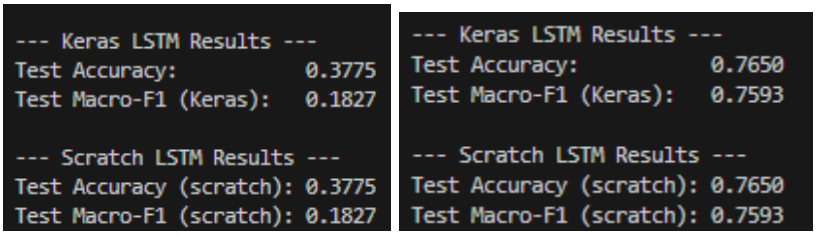
Kesimpulannya, untuk dataset dan tugas klasifikasi sentimen ini, **Bidirectional LSTM terbukti jauh lebih unggul**. Arsitektur ini mampu memanfaatkan konteks dari masa lalu dan masa depan untuk mengekstraksi fitur yang relevan, menghasilkan model yang dapat belajar dan bergeneralisasi dengan baik.

Di sisi lain, **Unidirectional LSTM mengalami underfitting yang parah**, menunjukkan bahwa kapasitas modelnya tidak memadai untuk menangani kompleksitas data. Eksperimen ini menjadi pengingat penting bahwa meskipun model yang terlalu kompleks berisiko *overfitting*, model yang terlalu sederhana bahkan lebih berisiko karena dapat gagal belajar sama sekali. Pemilihan arsitektur harus memastikan model memiliki kapasitas minimum yang diperlukan untuk menangkap pola yang mendasari data.

2.2.3.4 Perbandingan LSTM Keras dan LSTM Scratch: Analisis dan Implikasinya

Pada tahap akhir eksperimen, dilakukan perbandingan langsung antara performa model LSTM yang dibangun menggunakan *library high-level* **Keras** dengan model LSTM yang diimplementasikan dari nol (**scratch**). Tujuan utama dari perbandingan ini adalah untuk **memvalidasi kebenaran** dari implementasi *scratch*.

Untuk memastikan perbandingan yang adil (*apple-to-apple*), kedua model (Keras dan *scratch*) dijalankan pada beberapa konfigurasi arsitektur yang identik, menggunakan dataset, inisialisasi bobot (*weights*), dan *hyperparameter* yang sama persis. Hasil pengujian secara konsisten menunjukkan bahwa kedua implementasi memberikan metrik evaluasi yang **identik hingga empat angka desimal**.



Sebagai contoh, pada salah satu konfigurasi model yang berkinerja baik:

- **Keras LSTM**: Akurasi 0.7650, Macro-F1 0.7593
- **Scratch LSTM**: Akurasi 0.7650, Macro-F1 0.7593

Bahkan pada konfigurasi lain yang menunjukkan kinerja buruk (mengalami *underfitting*), hasilnya tetap sama persis:

- **Keras LSTM**: Akurasi 0.3775, Macro-F1 0.1827
- **Scratch LSTM**: Akurasi 0.3775, Macro-F1 0.1827

Hasil akurasi yang identik antara versi Keras dan *scratch* memiliki implikasi yang sangat penting: **implementasi model LSTM dari *scratch* sudah benar**.

Dengan demikian, perbandingan ini berfungsi sebagai "unit test" empiris yang efektif. Hasil ini mengonfirmasi bahwa analisis dan kesimpulan yang ditarik dari

eksperimen-eksperimen sebelumnya, yang menggunakan implementasi *scratch*, berdiri di atas fondasi kode yang andal dan akurat secara fungsional

BAB III
KESIMPULAN DAN SARAN

3.1 Kesimpulan

Dalam seluruh eksperimen pada CNN, RNN, dan LSTM, implementasi forward propagation from scratch berhasil menyamakan performa model Keras dengan presisi. Pada CNN, kedua pendekatan menghasilkan skor Macro F1 (0.6827) yang sama, menandakan bahwa setiap operasi forward pada setiap layer (konvolusi, pooling, flatten, dense) terimplementasi dengan benar. Selain itu, pada RNN bidirectional dengan 32 unit sel, skor F1 Keras dan scratch identik sebesar 0.514979. Untuk LSTM, implementasi from scratch memberikan akurasi dan Macro-F1 yang sama persis hingga empat angka desimal (0.7593 untuk bidirectional dan 0.1827 untuk unidirectional). Keseragaman hasil ini membuktikan kebenaran logika matematis di balik setiap layer dan alur propagasi maju pada ketiga arsitektur. Hal ini menandakan bahwa forward propagation from scratch berjalan sesuai dengan library

Pada CNN, hiperparameter yang diuji meliputi jumlah layer konvolusi (1, 2, 3), jumlah filter per layer ([16,32,64], [32,64,128], [64,128,256]), ukuran kernel ((3×3), (5×5), (7×7)), dan jenis pooling (max vs average). Hasil menunjukkan tren peningkatan performa seiring bertambahnya layer konvolusi, dengan skor tertinggi 0.7080 pada 3 layer tanpa overfitting signifikan. Konfigurasi filter [32,64,128] memberikan Macro F1 terbaik 0.7080 sebelum menurun pada filter lebih besar akibat overfitting. Ukuran kernel 3×3 unggul (0.7080), sedangkan 5×5 dan 7×7 menurunkan performa menjadi 0.6630 dan 0.6093 karena kehilangan detail lokal. Max pooling sedikit lebih baik (0.7080 vs 0.7060) karena mempertahankan fitur dominan. Dengan demikian, konfigurasi proporsional tiga layer, filter sedang, kernel 3×3, max pooling cukup optimal untuk CIFAR-10 dibandingkan dengan hasil eksperimen yang lain.

Pada RNN, fokus eksperimen mencakup pengaruh jumlah layer (1 hingga 3), jumlah unit sel per layer (32, 64, dan 128), dan arah layer (unidirectional versus bidirectional). Terkait jumlah layer, dengan konfigurasi 32 unit sel dan arsitektur bidirectional, model 1 layer memberikan F1-score 0.5081. Penambahan menjadi 2 layer justru mengalami penurunan performa ke F1-score 0.4528, kemungkinan akibat kesulitan optimasi atau overfitting pada kedalaman menengah, sedangkan 3 layer berhasil melonjak signifikan mencapai F1-score

0.5630, menunjukkan bahwa kapasitas ekstra pada kedalaman yang tepat membantu menangkap pola yang lebih kompleks. Eksperimen arah layer secara konsisten menunjukkan superioritas model bidirectional; model unidirectional menghasilkan F1-score dalam rentang 0.1836 hingga 0.4523, sementara model bidirectional mencapai F1-score yang jauh lebih tinggi, yakni antara 0.4748 hingga 0.5630. Hal ini menegaskan krusialnya pemrosesan konteks dari dua arah untuk generalisasi model yang lebih baik. Meskipun model yang lebih kompleks (lebih dalam atau dengan lebih banyak unit sel, seperti pada kasus 2 layer atau 128 unit sel yang menunjukkan tanda overfitting) berisiko mengalami overfitting, arsitektur bidirectional RNN secara konsisten menunjukkan kemampuan generalisasi yang lebih unggul, yang tercermin dari performa F1-score yang lebih tinggi dan perilaku validation loss yang relatif lebih baik dibandingkan dengan versi unidirectional pada konfigurasi serupa.

Pada LSTM, hiperparameter yang dieksplorasi meliputi jumlah layer (1–3), jumlah unit sel (32, 64, 128), dan arah layer (uni vs bi). Satu layer LSTM dengan 32 unit mencapai keseimbangan terbaik antara Macro F1 0.657 dan akurasi 0.703 tanpa overfitting signifikan. Penambahan layer (2,3) sedikit menaikkan F1 (0.706–0.707) namun validation loss stagnan/naik, mengindikasikan overfitting pada data kecil. Pada eksperimen unit sel, 32 unit terbukti optimal (F1 0.7368, akurasi 0.745) dibanding 64/128 unit yang cenderung overfit. Bidirectional LSTM unggul: F1 0.6565 dan akurasi 0.7025, sementara unidirectional underfit (F1 0.1844, akurasi 0.3825). Dengan demikian, satu layer bi-LSTM dengan unit sedang adalah pilihan terbaik untuk dataset teks pendek ini.

3.2 Saran

1. Eksplorasi Lebih Lanjut Teknik Regularisasi: Untuk mengatasi overfitting yang teramati pada beberapa konfigurasi (terutama pada RNN dengan banyak unit cell atau CNN dengan filter besar), disarankan untuk mengeksplorasi teknik regularisasi lebih lanjut, seperti dropout, batch normalization, atau early stopping.
2. Optimisasi Hyperparameter yang Lebih Lanjut: Eksperimen hyperparameter yang dilakukan sudah cukup baik, tetapi dapat ditingkatkan dengan menggunakan teknik optimisasi hyperparameter yang lebih canggih, seperti grid search atau random search, atau bahkan Bayesian optimization.

3. Evaluasi dengan Metrik Lain: Selain F1-score, disarankan untuk mengevaluasi model dengan metrik lain yang relevan, seperti precision, recall, accuracy, atau area under the ROC curve (AUC), untuk mendapatkan gambaran performa yang lebih lengkap.

BAB IV
LAMPIRAN

4.1 Pembagian Tugas

Nama/NIM	Kontribusi
Filbert (13522021)	CNN, RNN, LSTM
Benardo (13522055)	CNN, RNN, LSTM
William Glory Henderson (13522113)	CNN, RNN, LSTM

BAB V

REFERENSI

https://d2l.ai/chapter_convolutional-neural-networks/index.html

https://d2l.ai/chapter_recurrent-modern/bi-rnn.html

https://d2l.ai/chapter_recurrent-neural-networks/index.html

https://d2l.ai/chapter_recurrent-modern/lstm.html