LAPORAN TUGAS BESAR 2

IF3270 Pembelajaran Mesin

Convolutional Neural Network dan Recurrent Neural Network

Disusun oleh:

Shafiq Irvansyah (13522003)

Ahmad Naufal Ramadan (13522005)

Dewantoro Triatmojo (13522011)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2025

BAB I
DESKRIPSI PERSOALAN

Modul Convolutional Neural Network (CNN) dan Recurrent Neural Network (RNN) yang diimplementasikan dalam Tugas Besar II ini harus memenuhi berbagai ketentuan utama. Modul harus dapat mengakomodasi proses pelatihan dan evaluasi model menggunakan dataset yang telah ditentukan, serta mengimplementasikan forward propagation dari setiap jenis jaringan secara manual.

Pada bagian CNN, model harus mampu melakukan klasifikasi citra berdasarkan dataset CIFAR-10. Arsitektur model minimal harus terdiri dari layer Conv2D, Pooling, Flatten/Global Pooling, dan Dense. Untuk proses pelatihan, digunakan fungsi loss Sparse Categorical Cross Entropy dan optimizer Adam. Dataset CIFAR-10 yang tersedia harus dipisahkan menjadi tiga bagian, yaitu training set (40.000 data), validation set (10.000 data), dan test set (10.000 data). Modul CNN juga harus mendukung analisis terhadap pengaruh berbagai hyperparameter seperti jumlah layer konvolusi, banyaknya filter per layer, ukuran filter, dan jenis pooling layer yang digunakan. Pengujian dilakukan menggunakan metrik macro F1-score.

Selanjutnya, pada bagian RNN, model harus mampu melakukan klasifikasi teks berdasarkan dataset NusaX-Sentiment berbahasa Indonesia. Teks harus melalui tahap preprocessing yang meliputi tokenisasi menggunakan TextVectorization dan pemetaan token ke dalam vektor menggunakan embedding layer dari Keras. Arsitektur minimal RNN terdiri dari embedding layer, bidirectional/unidirectional RNN layer, dropout layer, dan dense layer. Model dilatih menggunakan fungsi loss Sparse Categorical Crossentropy dan optimizer Adam. Modul harus mendukung analisis pengaruh jumlah layer RNN, banyaknya cell RNN per layer, dan arah layer RNN.

Untuk bagian LSTM, spesifikasi dan proses yang dilakukan sama seperti RNN, tetapi dengan mengganti RNN layer menjadi LSTM layer. Analisis pengaruh hyperparameter dilakukan terhadap jumlah layer, banyaknya cell per layer, dan arah layer.

Pada seluruh arsitektur, peserta diwajibkan menyimpan bobot hasil pelatihan dengan Keras dan membuat modul forward propagation secara modular untuk setiap layer. Modul tersebut harus mampu membaca bobot hasil pelatihan dan membandingkan output forward propagation buatan sendiri dengan output dari Keras, menggunakan data test sebagai basis pengujian dan macro F1-score sebagai metrik evaluasi.

BAB II

PEMBAHASAN

2.1. Penjelasan Implementasi

2.1.1. Deskripsi Kelas beserta Deskripsi Atribut dan Methodnya

2.1.1.1. Kelas Umum

Tabel Kelas BaseModel

Atribut/Metode	Deskripsi
`layers`	List untuk menyimpan semua layer dalam model
`__init__()`	Konstruktor yang menginisialisasi list layers kosong
`add(layer)`	Menambahkan layer ke dalam list layers model
`forward(inputs)`	Metode abstrak yang harus diimplementasikan oleh subclass untuk melakukan forward propagation
`predict(inputs)`	Melakukan forward propagation dan mengembalikan indeks kelas prediksi menggunakan argmax
`load_weights_from_keras(keras_model)`	Memuat bobot dari model Keras ke model kustom, mencocokkan layer berdasarkan posisi
`evaluate(x_test, y_test, batch_size=32)`	Mengevaluasi model pada data test, mengembalikan metrik akurasi dan macro F1-score

Tabel Kelas ActivationLayer

Atribut/Metode	Deskripsi
`forward(inputs)`	Metode dasar untuk forward propagation (akan diimplementasikan oleh subclass)
`backward(grad_output)`	Metode dasar untuk backward propagation (akan diimplementasikan oleh subclass)

Tabel Kelas ReLu

Atribut/Metode	Deskripsi
`inputs`	Menyimpan nilai input untuk backward propagation
`forward(inputs)`	Menerapkan fungsi aktivasi ReLU (max(0, x)) dan menyimpan input
`backward(grad_output)`	Menghitung gradien menggunakan input yang tersimpan (gradien = 0 dimana input ≤ 0)

Tabel Kelas Sigmoid

Atribut/Metode	Deskripsi
`outputs`	Menyimpan nilai output untuk backward propagation
`forward(inputs)`	Menerapkan fungsi aktivasi sigmoid dengan clipping untuk mencegah overflow
`backward(grad_output)`	Menghitung gradien menggunakan rumus turunan sigmoid

Tabel Kelas Tanh

Atribut/Metode	Deskripsi
`outputs`	Menyimpan nilai output untuk backward propagation
`forward(inputs)`	Menerapkan fungsi aktivasi hyperbolic tangent
`backward(grad_output)`	Menghitung gradien menggunakan rumus turunan tanh

Tabel Kelas Softmax

Atribut/Metode	Deskripsi
`outputs`	Menyimpan nilai output untuk backward propagation
`forward(inputs)`	Menerapkan fungsi aktivasi softmax dengan stabilitas numerik
`backward(grad_output)`	Placeholder untuk backward propagation (TODO: perlu implementasi)

Tabel Kelas DropoutLayer

Atribut/Metode	Deskripsi
`dropout_rate`	Probabilitas untuk menghilangkan unit (default: 0.5)
`mask`	Mask biner yang menunjukkan unit mana yang disimpan/dihilangkan
`training`	Flag boolean yang menunjukkan apakah layer dalam mode training
`__init__(dropout_rate=0.5)`	Konstruktor yang mengatur dropout rate dan menginisialisasi atribut
`forward(inputs, training=True)`	Menerapkan dropout selama training atau meneruskan input tanpa perubahan saat inference
`backward(grad_output)`	Menerapkan mask yang sama pada gradien selama backpropagation
`load_weights_from_keras(keras_layer)`	Memuat dropout rate dari layer Keras

Tabel Kelas FlattenLayer

Atribut/Metode	Deskripsi

`input`	Menyimpan tensor input untuk backward pass yang potensial
`input_shape`	Menyimpan bentuk input (tidak termasuk dimensi batch)
`output`	Menyimpan tensor output yang sudah diratakan
`__init__()`	Konstruktor yang menginisialisasi semua atribut menjadi None
`load_weights_from_keras(keras_layer)`	Metode placeholder (tidak ada bobot untuk dimuat pada flatten layer)
`forward(inputs)`	Meratakan tensor input dari (batch_size, ...) menjadi (batch_size, flattened_size)

Tabel Kelas Conv2DLayer

Atribut/Metode	**Deskripsi**
`filters`	Jumlah filter/feature map output
`kernel_size`	Ukuran kernel konvolusi (diasumsikan persegi)
`activation`	Fungsi aktivasi yang diterapkan setelah konvolusi
`input_shape`	Bentuk input yang diharapkan (opsional)
`padding`	Jenis padding ('valid' atau 'same')
`weights`	Bobot kernel konvolusi dengan bentuk (filters, input_channels, kernel_size, kernel_size)
`biases`	Bias untuk setiap filter
`input`	Menyimpan input untuk backward propagation
`output`	Menyimpan output untuk backward propagation

`__init__(filters, kernel_size, activation=None, input_shape=None, padding='valid')`	Konstruktor yang menginisialisasi parameter layer
`initialize_weights(input_channels)`	Menginisialisasi bobot menggunakan inisialisasi He dan bias nol
`load_weights_from_keras(keras_layer)`	Memuat bobot dari layer Conv2D Keras dengan transposisi yang tepat
`forward(inputs)`	Melakukan operasi konvolusi 2D dengan padding dan aktivasi opsional

Tabel Kelas DenseLayer

Atribut/Metode	Deskripsi
`input_dim`	Jumlah fitur input
`output_dim`	Jumlah fitur output
`activation`	Fungsi aktivasi yang diterapkan setelah transformasi linear
`weights`	Matriks bobot dengan bentuk (input_dim, output_dim)
`biases`	Vektor bias dengan bentuk (output_dim,)
`input`	Menyimpan input untuk backward propagation
`output`	Menyimpan output untuk backward propagation
`grad_weights`	Menyimpan gradien bobot selama backward pass
`grad_biases`	Menyimpan gradien bias selama backward pass

`__init__(input_dim, output_dim, activation=None)`	Konstruktor yang menginisialisasi parameter layer dan bobot
`forward(inputs)`	Melakukan transformasi linear diikuti dengan aktivasi opsional
`load_weights_from_keras(keras_layer)`	Memuat bobot dan bias dari layer Dense Keras
`backward(grad_output)`	Menghitung gradien untuk bobot, bias, dan input

Tabel Kelas MaxPooling2DLayer

Atribut/Metode	Deskripsi
`pool_size`	Ukuran jendela pooling (tinggi, lebar)
`strides`	Nilai stride untuk operasi pooling
`input`	Menyimpan input untuk backward propagation yang potensial
`output`	Menyimpan output untuk backward propagation yang potensial
`__init__(pool_size=(2,2), strides=None)`	Konstruktor yang mengatur parameter pooling
`load_weights_from_keras(keras_layer)`	Metode placeholder (tidak ada bobot untuk layer pooling)
`forward(inputs)`	Melakukan operasi max pooling pada tensor input

Tabel Kelas AveragePooling2DLayer

Atribut/Metode	Deskripsi
`pool_size`	Ukuran jendela pooling (tinggi, lebar)
`strides`	Nilai stride untuk operasi pooling

`input`	Menyimpan input untuk backward propagation yang potensial
`output`	Menyimpan output untuk backward propagation yang potensial
`__init__(pool_size=(2,2), strides=None)`	Konstruktor yang mengatur parameter pooling
`load_weights_from_keras(keras_layer)`	Metode placeholder (tidak ada bobot untuk layer pooling)
`forward(inputs)`	Melakukan operasi average pooling pada tensor input

Tabel Kelas EmbeddingLayer

Atribut/Metode	Deskripsi
`input_dim`	Ukuran vocabulary (jumlah token unik)
`output_dim`	Dimensi vektor embedding
`weights`	Matriks embedding dengan bentuk (input_dim, output_dim)
`__init__(input_dim, output_dim)`	Konstruktor yang menginisialisasi parameter embedding dan bobot
`forward(inputs)`	Mengkonversi indeks token menjadi vektor embedding, menangani indeks di luar jangkauan
`load_weights_from_keras(keras_layer)`	Memuat bobot embedding dari layer Embedding Keras

Tabel Kelas NusaXLoader

Atribut/Metode	Deskripsi
`data_dir`	Path direktori yang berisi file dataset NusaX

`batch_size`	Ukuran batch untuk loading data
`max_sequence_length`	Panjang maksimum untuk sequence teks
`vocab_size`	Ukuran vocabulary untuk vektorisasi teks
`vectorizer`	Layer TextVectorization untuk mengkonversi teks menjadi token
`vocabulary`	List vocabulary dari vectorizer
`num_classes`	Jumlah kelas sentimen (3: negatif, netral, positif)
`__init__(data_dir=None, batch_size=32, max_sequence_length=100, vocab_size=10000)`	Konstruktor yang menginisialisasi parameter loading data
`load_data(split='train')`	Memuat data dari file CSV untuk split yang ditentukan
`_create_vectorizer(texts)`	Membuat dan melatih layer TextVectorization pada teks training
`preprocess_data(df)`	Memproses DataFrame untuk mengekstrak teks dan mengkonversi label menjadi integer
`get_dataset(split='train')`	Mengembalikan dataset TensorFlow dengan batching dan vektorisasi
`get_vectorized_data(split='train')`	Mengembalikan teks yang sudah divektorisasi dan label sebagai numpy array

`get_vocabulary()`	Mengembalikan list vocabulary dari vectorizer

2.1.1.2. Kelas CNN

Tabel Kelas CNN

Atribut/Metode	Deskripsi
`input_shape`	Bentuk input yang diharapkan untuk CNN
`num_classes`	Jumlah kelas output untuk klasifikasi
`layers`	Diwarisi dari BaseModel - list layer dalam jaringan
`__init__(input_shape, num_classes)`	Konstruktor yang menginisialisasi CNN dengan input shape dan jumlah kelas
`forward(inputs)`	Melakukan forward propagation melalui semua layer secara berurutan
`predict(inputs)`	Melakukan forward propagation dan mengembalikan indeks kelas prediksi
`load_weights_from_keras(keras_model)`	Memuat bobot dari model Keras, menangani skip input layer

2.1.1.3. Kelas RNN

Tabel Kelas RNNLayer

Atribut/Metode	Deskripsi
`input_dim`	Dimensi fitur input
`hidden_dim`	Dimensi hidden state
`bidirectional`	Flag boolean untuk pemrosesan bidirectional

`return_sequences`	Flag boolean untuk mengembalikan seluruh sequence atau hanya output terakhir
`W, U, b`	Bobot input, bobot recurrent, dan bias untuk arah forward
`backward_W, backward_U, backward_b`	Bobot input, bobot recurrent, dan bias untuk arah backward (jika bidirectional)
`_rnn_step(x_t, h_prev, W, U, b)`	Melakukan satu langkah forward propagation RNN
`forward(inputs)`	Melakukan forward propagation untuk seluruh sequence
`load_weights_from_keras(keras_layer)`	Memuat bobot dari layer Keras RNN

Tabel Kelas RNNModel

Atribut/Metode	Deskripsi
`layers`	List yang menyimpan layer-layer dalam model
`compiled`	Flag boolean yang menandakan model sudah dikompilasi
`add(layer)`	Menambahkan layer ke dalam model
`forward(inputs)`	Melakukan forward propagation melalui semua layer
`predict(inputs, batch_size)`	Memprediksi kelas untuk input yang diberikan
`predict_proba(inputs, batch_size)`	Memprediksi probabilitas kelas untuk input yang diberikan
`evaluate(x_test, y_test, batch_size)`	Mengevaluasi performa model pada data test
`load_weights_from_keras(keras_model)`	Memuat bobot dari model Keras ke model scratch

summary()	Menampilkan ringkasan arsitektur model dan jumlah parameter

2.1.1.4. Kelas LSTM

Tabel Kelas LSTMLayer

Atribut/Metode	Deskripsi
`input_dim`	Dimensi fitur input
`hidden_dim`	Dimensi hidden state
`bidirectional`	Flag boolean untuk pemrosesan bidirectional
`W_i, W_f, W_c, W_o`	Bobot input untuk input gate, forget gate, cell gate, dan output gate
`U_i, U_f, U_c, U_o`	Bobot recurrent untuk input gate, forget gate, cell gate, dan output gate
`b_i, b_f, b_c, b_o`	Vektor bias untuk input gate, forget gate, cell gate, dan output gate
`forward_*, backward_*`	Set bobot terpisah untuk LSTM bidirectional
`__init__(input_dim, hidden_dim, bidirectional=False)`	Konstruktor yang menginisialisasi parameter LSTM
`initialize_weights(direction='forward')`	Menginisialisasi bobot untuk arah yang ditentukan (forward/backward)
`forward_step(x_t, h_prev, c_prev, direction='forward')`	Melakukan satu langkah waktu LSTM dengan komputasi gate
`forward(inputs)`	Memproses seluruh sequence, mengembalikan hidden state akhir
`load_weights_from_keras(keras_layer)`	Memuat bobot dari layer LSTM atau Bidirectional LSTM Keras

Tabel Kelas LSTMModel

Atribut/Metode	Deskripsi
`layers`	Diwarisi dari BaseModel - list layer dalam jaringan
`__init__()`	Konstruktor yang menginisialisasi model LSTM
`forward(inputs)`	Melakukan forward propagation melalui semua layer secara berurutan untuk klasifikasi teks

Tabel Kelas LSTMExperiments

Atribut/Metode	Deskripsi
`batch_size`	Ukuran batch untuk training
`epochs`	Jumlah epoch training
`embedding_dim`	Dimensi vektor embedding
`data_loader`	Instance NusaXLoader untuk penanganan data
`vocab_size`	Ukuran vocabulary dari data loader
`num_classes`	Jumlah kelas dari data loader
`max_sequence_length`	Panjang sequence maksimum dari data loader
`__init__(data_loader, batch_size=32, epochs=10, embedding_dim=100)`	Konstruktor yang menginisialisasi parameter eksperimen

`_build_base_model(lstm_layers, lstm_units, bidirectional=False)`	Membangun model LSTM Keras dengan arsitektur yang ditentukan
`_train_model(model, name=None)`	Melatih model dan mengevaluasi pada test set dengan F1-score
`run_layer_count_experiment(layer_count_variants)`	Menjalankan eksperimen membandingkan jumlah layer LSTM yang berbeda
`run_cell_count_experiment(cell_count_variants)`	Menjalankan eksperimen membandingkan jumlah cell LSTM yang berbeda
`run_direction_experiment(direction_variants)`	Menjalankan eksperimen membandingkan LSTM bidirectional vs unidirectional
`_plot_comparison(histories, metric='loss', title='Model Comparison')`	Membuat plot metrik training dan validation untuk perbandingan model

2.1.2 Penjelasan Forward Propagation

2.1.2.1 CNN

Model CNN dalam project ini diimplementasikan menggunakan pendekatan modular dengan class `CNN` yang mewarisi dari `BaseModel`. Forward propagation merupakan proses dimana input data dilewatkan melalui setiap layer secara berurutan, mulai dari layer pertama hingga layer terakhir. Setiap layer akan mentransformasi input yang diterimanya dan meneruskan hasilnya ke layer berikutnya. Proses ini berlanjut hingga mencapai output final yang berupa prediksi atau klasifikasi.

```python
def forward(self, inputs):
    output = inputs

    for _, layer in enumerate(self.layers):
        output = layer.forward(output)

    return output
```

Layer konvolusi 2D adalah inti dari CNN yang bertugas melakukan ekstraksi fitur dari input image. Layer ini bekerja dengan cara menggeser filter (kernel) di atas input image dan melakukan operasi dot product antara filter dengan region input yang sedang diproses. Setiap filter akan menghasilkan satu feature map yang menunjukkan keberadaan fitur tertentu dalam input.

Proses inisialisasi bobot menggunakan metode Xavier/He initialization untuk memastikan gradien tidak vanishing atau exploding. Bobot filter diinisialisasi dengan distribusi normal yang diskalakan berdasarkan jumlah input channels dan ukuran kernel.

```python
def initialize_weights(self, input_channels):
    self.weights = np.random.randn(
        self.filters, input_channels, self.kernel_size, self.kernel_size
    ) * np.sqrt(2.0 / (self.kernel_size * self.kernel_size * input_channels))

    self.biases = np.zeros(self.filters)
```

Forward propagation pada Conv2D layer dimulai dengan penentuan apakah padding diperlukan. Padding 'valid' berarti tidak ada padding yang ditambahkan, sedangkan padding 'same' akan menambahkan padding agar ukuran output sama dengan input. Setelah itu, untuk setiap posisi dalam output, dilakukan ekstraksi region dari input sesuai ukuran kernel, kemudian dikalikan dengan filter weights dan ditambahkan bias.

```
# Perform convolution with corrected indexing
for b in range(batch_size):
    for f in range(self.filters):
        for i in range(output_height):
            for j in range(output_width):
                # Extract region for convolution
                # Region shape: (kernel_size, kernel_size, input_channels)
                region = padded_inputs[
                    b, i : i + self.kernel_size, j : j + self.kernel_size, :
                ]

                # Get filter weights for this filter
                # Filter shape: (input_channels, kernel_size, kernel_size)
                filter_weights = self.weights[f, :, :, :]

                # Transpose filter weights to match region shape
                # From (input_channels, kernel_size, kernel_size)
                # To (kernel_size, kernel_size, input_channels)
                filter_weights_transposed = np.transpose(
                    filter_weights, (1, 2, 0)
                )

                # Now both have shape (kernel_size, kernel_size, input_channels)
                # Perform element-wise multiplication and sum
                conv_result = (
                    np.sum(region * filter_weights_transposed) + self.biases[f]
                )
                output[b, i, j, f] = conv_result
```

Pooling layers bertugas untuk mengurangi dimensi spatial dari feature maps sambil mempertahankan informasi penting. Ada dua jenis pooling yang diimplementasikan: Max Pooling dan Average Pooling. Max Pooling mengambil nilai maksimum dari setiap region, sedangkan Average Pooling mengambil rata-rata nilai dalam setiap region.

Max Pooling bekerja dengan membagi input menjadi region-region non-overlapping berdasarkan pool_size dan stride. Untuk setiap region, nilai maksimum dipilih sebagai representasi dari region tersebut. Ini membantu model menjadi lebih robust terhadap small translations dalam input.

```python
def forward(self, inputs):
    self.input = inputs
    batch_size, input_height, input_width, channels = inputs.shape
    pool_height, pool_width = self.pool_size
    stride_h, stride_w = self.strides

    # Calculate output dimensions
    output_height = (input_height - pool_height) // stride_h + 1
    output_width = (input_width - pool_width) // stride_w + 1

    # Initialize output
    output = np.zeros((batch_size, output_height, output_width, channels))

    # Perform max pooling
    for b in range(batch_size):
        for c in range(channels):
            for i in range(output_height):
                for j in range(output_width):
                    # Calculate pooling region boundaries
                    start_i = i * stride_h
                    end_i = start_i + pool_height
                    start_j = j * stride_w
                    end_j = start_j + pool_width

                    # Extract pooling region and take maximum
                    region = inputs[b, start_i:end_i, start_j:end_j, c]
                    output[b, i, j, c] = np.max(region)

    self.output = output
    return output
```

Average Pooling menggunakan konsep yang sama dengan Max Pooling, namun mengambil rata-rata nilai dalam setiap region alih-alih nilai maksimum. Ini memberikan representasi yang lebih smooth dari feature maps.

```python
def forward(self, inputs):
    self.input = inputs
    batch_size, input_height, input_width, channels = inputs.shape
    pool_height, pool_width = self.pool_size
    stride_h, stride_w = self.strides

    # Calculate output dimensions
    output_height = (input_height - pool_height) // stride_h + 1
    output_width = (input_width - pool_width) // stride_w + 1

    # Initialize output
    output = np.zeros((batch_size, output_height, output_width, channels))

    # Perform average pooling
    for b in range(batch_size):
        for c in range(channels):
            for i in range(output_height):
                for j in range(output_width):
                    # Calculate pooling region boundaries
                    start_i = i * stride_h
                    end_i = start_i + pool_height
                    start_j = j * stride_w
                    end_j = start_j + pool_width

                    # Extract pooling region and take average
                    region = inputs[b, start_i:end_i, start_j:end_j, c]
                    output[b, i, j, c] = np.mean(region)

    self.output = output
    return output
```

Flatten layer bertugas mengkonversi tensor multidimensi menjadi vector 1D agar dapat diproses oleh Dense layer. Layer ini tidak melakukan transformasi matematika kompleks, hanya mengubah shape dari input. Shape asli disimpan untuk keperluan backward propagation jika diperlukan.

```python
def forward(self, inputs):
    self.input = inputs
    self.input_shape = inputs.shape[
        1:
    ]

    batch_size = inputs.shape[0]
    self.output = inputs.reshape(batch_size, -1)
    return self.output
```

Dense layer atau fully connected layer melakukan transformasi linear dengan mengalikan input dengan weight matrix dan menambahkan bias. Layer ini biasanya digunakan di bagian akhir CNN untuk klasifikasi. Jika input_dim tidak ditentukan saat inisialisasi, maka akan dihitung secara otomatis berdasarkan ukuran input yang diterima.

```
def forward(self, inputs):
    self.input = inputs

    # Initialize weights if needed
    if self.weights is None:
        self.initialize_weights_if_needed(inputs.shape[1])

    self.output = np.dot(inputs, self.weights) + self.biases

    if self.activation:
        self.output = self.activation.forward(self.output)

    return self.output
```

Fungsi aktivasi memberikan non-linearity pada neural network, memungkinkan model untuk mempelajari pola yang kompleks. ReLU (Rectified Linear Unit) adalah fungsi aktivasi yang paling umum digunakan dalam CNN karena kesederhanaannya dan kemampuannya mengatasi vanishing gradient problem.

```
class ReLU(ActivationLayer):
    def forward(self, inputs):
        self.inputs = inputs
        return np.maximum(0, inputs)

    def backward(self, grad_output):
        return grad_output * (self.inputs > 0)
```

Sigmoid menghasilkan output antara 0 dan 1, sering digunakan untuk binary classification. Clipping diterapkan untuk mencegah overflow saat menghitung eksponensial.

```python
class Sigmoid(ActivationLayer):
    def forward(self, inputs):
        self.outputs = 1.0 / (1.0 + np.exp(-np.clip(inputs, -50, 50)))
        return self.outputs

    def backward(self, grad_output):
        return grad_output * self.outputs * (1 - self.outputs)
```

Softmax digunakan untuk multi-class classification karena menghasilkan distribusi probabilitas dimana semua output berjumlah 1. Numerical stability dijaga dengan mengurangi nilai maksimum dari setiap sample sebelum menghitung eksponensial.

```python
class Softmax(ActivationLayer):
    def forward(self, inputs):
        exp_shifted = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
        self.outputs = exp_shifted / np.sum(exp_shifted, axis=1, keepdims=True)
        return self.outputs

    def backward(self, grad_output):
        # TODO
        return grad_output
```

Dropout adalah teknik regularisasi yang secara random "mematikan" beberapa neuron selama training untuk mencegah overfitting. Selama inference (testing), semua neuron aktif tetapi output diskalakan sesuai dengan dropout rate yang digunakan selama training.

```python
import numpy as np

class DropoutLayer:
    def __init__(self, dropout_rate=0.5):
        self.dropout_rate = dropout_rate
        self.mask = None
        self.training = True

    def forward(self, inputs, training=True):
        self.training = training

        if not training:
            return inputs

        # Generate dropout mask (1: keep, 0: drop)
        self.mask = np.random.binomial(1, 1 - self.dropout_rate, size=inputs.shape) / (1 - self.dropout_rate)

        # Apply mask to inputs
        return inputs * self.mask

    def backward(self, grad_output):
        if not self.training:
            return grad_output

        # Apply same mask to gradients
        return grad_output * self.mask

    def load_weights_from_keras(self, keras_layer):
        self.dropout_rate = keras_layer.rate
```

Model from scratch dapat memuat bobot dari model Keras yang telah dilatih, memungkinkan transfer knowledge dari model yang sudah trained. Proses ini melibatkan mapping antara layer-layer dalam custom model dengan layer-layer dalam Keras model, kemudian mengekstrak dan menyalin bobot sesuai format yang diharapkan.

```python
def load_weights_from_keras(self, keras_model):
    keras_layers = keras_model.layers

    # Skip input layer if it exists in Keras model
    if (
        hasattr(keras_layers[0], "input_shape")
        and len(keras_layers[0].get_weights()) == 0
    ):
        keras_layers = keras_layers[1:]

    # Load weights for each layer
    for _, (custom_layer, keras_layer) in enumerate(zip(self.layers, keras_layers)):
        try:
            if hasattr(custom_layer, "load_weights_from_keras"):
                custom_layer.load_weights_from_keras(keras_layer)
            else:
                print(f"No weights to load (layer doesn't have weights)")
        except Exception as e:
            print(f"Error loading weights: {e}")
```

Fungsi prediksi menggunakan hasil forward propagation untuk menghasilkan prediksi kelas. Jika output adalah dari softmax layer, maka argmax langsung diambil untuk mendapatkan kelas dengan probabilitas tertinggi. Jika output berupa raw logits, argmax tetap digunakan setelah memastikan dimensi output sesuai.

```python
def predict(self, inputs):
    outputs = self.forward(inputs)

    # If output is from softmax, take argmax
    if len(outputs.shape) == 2 and outputs.shape[1] == self.num_classes:
        return np.argmax(outputs, axis=1)
    else:
        # If not softmax output, assume raw logits
        return np.argmax(outputs, axis=1)
```

2.1.2.2 Simple RNN

Recurrent Neural Network (RNN) merupakan arsitektur *neuran network* yang didesain untuk memproses data sekuensial dengan memanfaatkan hidden state dari timestep sebelumnya. Perbedaan utama RNN dengan feedforward neural network terletak pada kemampuan RNN untuk mempertahankan informasi dari input sebelumnya melalui koneksi

rekursif. Hal ini memungkinkan RNN untuk menangani tugas-tugas yang melibatkan dependensi temporal seperti analisis sentimen, pengenalan ucapan, dan prediksi time series.

Operasi forward propagation RNN didasarkan pada formula matematis *h_t = tanh(W * x_t + U * h_{t-1} + b)*. Dalam formula ini, h_t merepresentasikan hidden state pada timestep t, x_t adalah input pada timestep t, W adalah matriks bobot untuk input, U adalah matriks bobot untuk hidden state sebelumnya, b adalah vektor bias, dan tanh berfungsi sebagai fungsi aktivasi non-linear. Formula ini menunjukkan bagaimana informasi dari input saat ini dikombinasikan dengan konteks dari timestep sebelumnya untuk menghasilkan representasi baru.

Implementasi time step forward propagation RNN:

```python
def _rnn_step(self, x_t, h_prev, W, U, b):
    x_t = x_t.astype(np.float32)
    h_prev = h_prev.astype(np.float32)
    W = W.astype(np.float32)
    U = U.astype(np.float32)
    b = b.astype(np.float32)

    # h_t = tanh(x_t @ W + h_prev @ U + b)
    x_proj = np.dot(x_t, W)
    h_proj = np.dot(h_prev, U)
    linear = x_proj + h_proj + b

    # tanh activation
    h_t = np.tanh(linear)

    return h_t.astype(np.float32)
```

Proses komputasi dimulai dengan konversi tipe data ke float32 untuk memastikan konsistensi. Input x_t dikalikan dengan matriks bobot W untuk mentransformasi dimensi dari input_dim ke hidden_dim. Secara paralel, hidden state sebelumnya h_prev dikalikan dengan matriks bobot rekursif U. Hasil kedua perkalian matriks ini dijumlahkan dengan bias b untuk membentuk kombinasi linear. Fungsi aktivasi tanh kemudian diterapkan pada kombinasi linear tersebut untuk menghasilkan hidden state baru h_t.

Proses forward propagation untuk sequence lengkap menggunakan iterasi melalui setiap timestep sambil memperbarui hidden state secara berkesinambungan. Implementasi fungsi forward sebagai berikut:

```python
def forward(self, inputs):

    inputs = inputs.astype(np.float32)

    # Handle input shape
    if len(inputs.shape) == 2:
        batch_size, sequence_length = inputs.shape
        inputs = inputs.reshape(batch_size, sequence_length, 1)

    batch_size, sequence_length, actual_input_dim = inputs.shape

    # Check for dimension mismatch
    if actual_input_dim != self.input_dim:
        raise ValueError(
            f"Input dimension mismatch in RNN layer: expected {self.input_dim}, got {actual_input_dim}. "
            f"Input shape: {inputs.shape}"
        )

    # Initialize hidden states with zeros
    h_forward = np.zeros((batch_size, self.hidden_dim), dtype=np.float32)

    if self.return_sequences:
        forward_outputs = np.zeros((batch_size, sequence_length, self.hidden_dim), dtype=np.float32)

    # Forward pass through time steps
    for t in range(sequence_length):
        h_forward = self._rnn_step(inputs[:, t, :], h_forward, self.W, self.U, self.b)
        if self.return_sequences:
            forward_outputs[:, t, :] = h_forward

    # Handle unidirectional case
    if not self.bidirectional:
        return forward_outputs if self.return_sequences else h_forward

    # Bidirectional case - backward pass
    h_backward = np.zeros((batch_size, self.hidden_dim), dtype=np.float32)

    if self.return_sequences:
        backward_outputs = np.zeros((batch_size, sequence_length, self.hidden_dim), dtype=np.float32)

    # Process sequence in reverse
    for t in range(sequence_length - 1, -1, -1):
        h_backward = self._rnn_step(
            inputs[:, t, :],
            h_backward,
            self.backward_W,
            self.backward_U,
            self.backward_b
        )
        if self.return_sequences:
            backward_outputs[:, t, :] = h_backward

    # Merge
    if self.return_sequences:
        return np.concatenate([forward_outputs, backward_outputs], axis=2)
    else:
        return np.concatenate([h_forward, h_backward], axis=1)
```

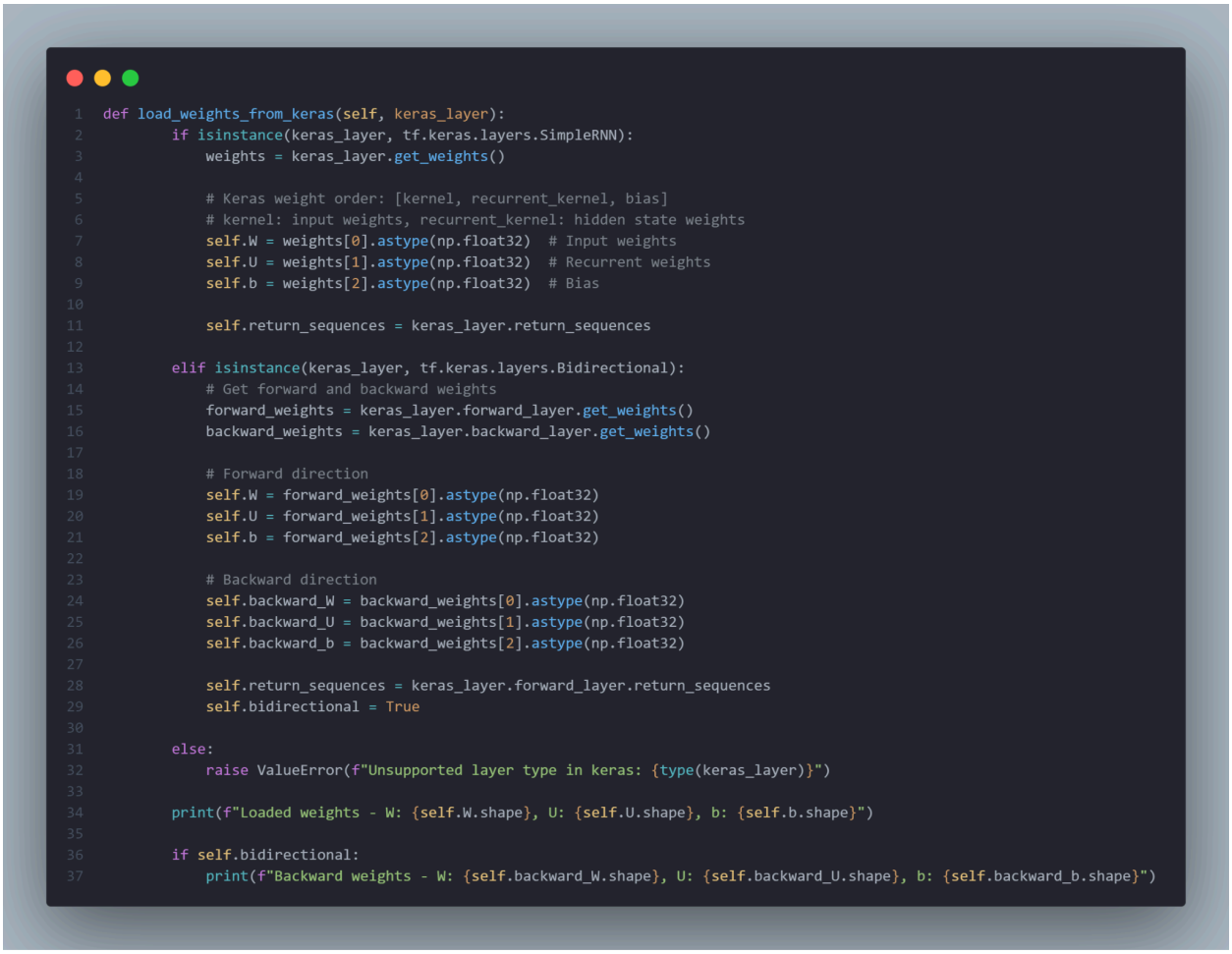
Preprocessing input dilakukan untuk menangani berbagai format input yang mungkin diterima. Validasi dimensi memastikan kesesuaian antara input yang diberikan dengan konfigurasi layer. Hidden state diinisialisasi dengan nilai nol untuk seluruh batch. Loop utama melakukan iterasi melalui setiap timestep, memanggil fungsi *_rnn_step* untuk menghitung hidden state baru berdasarkan input saat ini dan hidden state sebelumnya. Ketika parameter *return_sequences* bernilai True, output dari setiap timestep disimpan dalam array terpisah untuk dikembalikan sebagai sequence output. Sebaliknya, jika *return_sequences* bernilai False, hanya hidden state terakhir yang dikembalikan sebagai representasi dari seluruh sequence.

Bidirectional RNN memproses sequence dalam dua arah untuk menangkap konteks yang lebih komprehensif. Implementasi ini menggunakan dua set parameter terpisah untuk pemrosesan forward dan backward:

```
h_backward = np.zeros((batch_size, self.hidden_dim), dtype=np.float32)

    if self.return_sequences:
        backward_outputs = np.zeros((batch_size, sequence_length, self.hidden_dim), dtype=np.float32)

    # Process sequence in reverse
    for t in range(sequence_length - 1, -1, -1):
        h_backward = self._rnn_step(
            inputs[:, t, :],
            h_backward,
            self.backward_W,
            self.backward_U,
            self.backward_b
        )
        if self.return_sequences:
            backward_outputs[:, t, :] = h_backward

    # Merge
    if self.return_sequences:
        return np.concatenate([forward_outputs, backward_outputs], axis=2)
    else:
        return np.concatenate([h_forward, h_backward], axis=1)
```

Pemrosesan backward dilakukan dengan mengulangi sequence dalam urutan terbalik, menggunakan set parameter yang berbeda *(backward_W, backward_U, backward_b*). Hidden state untuk arah backward juga diinisialisasi secara terpisah. Setelah kedua arah pemrosesan selesai, output digabungkan melalui operasi concatenation. Untuk konfigurasi *return_sequences=True*, concatenation dilakukan pada axis feature (axis=2), sedangkan untuk *return_sequences=False*, concatenation dilakukan pada axis batch (axis=1). Hasil akhir adalah representasi yang menggabungkan informasi konteks dari kedua arah temporal.

Kompatibilitas dengan Keras memerlukan implementasi fungsi untuk memuat bobot dari model yang telah dilatih. Fungsi *load_weights_from_keras* menangani konversi format bobot:



```python
def load_weights_from_keras(self, keras_layer):
        if isinstance(keras_layer, tf.keras.layers.SimpleRNN):
            weights = keras_layer.get_weights()

            # Keras weight order: [kernel, recurrent_kernel, bias]
            # kernel: input weights, recurrent_kernel: hidden state weights
            self.W = weights[0].astype(np.float32)  # Input weights
            self.U = weights[1].astype(np.float32)  # Recurrent weights
            self.b = weights[2].astype(np.float32)  # Bias

            self.return_sequences = keras_layer.return_sequences

        elif isinstance(keras_layer, tf.keras.layers.Bidirectional):
            # Get forward and backward weights
            forward_weights = keras_layer.forward_layer.get_weights()
            backward_weights = keras_layer.backward_layer.get_weights()

            # Forward direction
            self.W = forward_weights[0].astype(np.float32)
            self.U = forward_weights[1].astype(np.float32)
            self.b = forward_weights[2].astype(np.float32)

            # Backward direction
            self.backward_W = backward_weights[0].astype(np.float32)
            self.backward_U = backward_weights[1].astype(np.float32)
            self.backward_b = backward_weights[2].astype(np.float32)

            self.return_sequences = keras_layer.forward_layer.return_sequences
            self.bidirectional = True

        else:
            raise ValueError(f"Unsupported layer type in keras: {type(keras_layer)}")

        print(f"Loaded weights - W: {self.W.shape}, U: {self.U.shape}, b: {self.b.shape}")

        if self.bidirectional:
            print(f"Backward weights - W: {self.backward_W.shape}, U: {self.backward_U.shape}, b: {self.backward_b.shape}")
```

Keras menyimpan bobot RNN dalam format tertentu, yaitu kernel untuk input weights, recurrent_kernel untuk recurrent weights, dan bias. Untuk SimpleRNN layer, ketiga komponen ini langsung dimapping ke variabel internal W, U, dan b. Untuk Bidirectional layer, proses lebih kompleks karena memerlukan ekstraksi bobot dari dua sub-layer yang terpisah. Setiap sub-layer memiliki set bobot lengkap yang harus dimuat secara independen. Konfigurasi tambahan seperti *return_sequences* juga perlu disinkronkan untuk memastikan konsistensi behavior.

Pada level model, forward propagation mengkoordinasikan multiple layers untuk menghasilkan output final. Implementasi *forward* pada class RNNModel mengelola alur data melalui arsitektur lengkap:

```python
def forward(self, inputs):
    x = inputs.astype(np.float32)

    for i, layer in enumerate(self.layers):
        try:
            x_prev_shape = x.shape

            # Special handling for dropout - now with proper import
            if isinstance(layer, DropoutLayer):
                # Always disable dropout during inference
                x = layer.forward(x, training=False)
            else:
                x = layer.forward(x)

            print(f"Layer {i} ({type(layer).__name__}): {x_prev_shape} -> {x.shape}")
        except Exception as e:
            print(f"ERROR in layer {i} ({type(layer).__name__})")
            print(f"Input shape: {x.shape}")
            print(f"Error: {e}")
            raise e

    return x
```

Fungsi ini melakukan iterasi melalui semua layer dalam model, melewatkan output dari satu layer sebagai input untuk layer berikutnya. Handling khusus diterapkan untuk DropoutLayer dimana training mode selalu diset ke False untuk memastikan konsistensi selama inference.

2.1.2.3 LSTM

Model LSTM dalam project ini diimplementasikan untuk text classification menggunakan pendekatan modular yang memungkinkan fleksibilitas dalam desain arsitektur. LSTM (Long Short-Term Memory) adalah variasi dari RNN yang dirancang khusus untuk mengatasi masalah vanishing gradient yang sering terjadi pada RNN tradisional. LSTM menggunakan gating mechanism yang terdiri dari tiga gate utama (forget gate, input gate, dan output gate) untuk mengontrol aliran informasi dalam sel memori.

Forward propagation pada LSTM model dilakukan dengan melewatkan input melalui setiap layer secara berurutan, dimulai dari embedding layer yang mengkonversi token menjadi vector representation, dilanjutkan dengan LSTM layer yang memproses informasi sequential, dan diakhiri dengan dense layer untuk klasifikasi.

```python
class LSTMModel(BaseModel):
    def __init__(self):
        super().__init__()

    def forward(self, inputs):
        """
        Perform forward propagation through all layers

        Args:
            inputs: numpy array of shape (batch_size, sequence_length)

        Returns:
            numpy array of shape (batch_size, num_classes)
        """
        x = inputs

        for layer in self.layers:
            x = layer.forward(x)

        return x
```

Embedding layer bertugas mengkonversi token integer menjadi dense vector representation yang dapat diproses oleh LSTM. Setiap token dalam vocabulary akan dipetakan ke sebuah vector dengan dimensi yang telah ditentukan. Vector embedding ini akan dipelajari selama proses training untuk merepresentasikan makna semantik dari setiap token.

Proses forward propagation pada embedding layer melibatkan lookup operation dimana untuk setiap token dalam input sequence, vector embedding yang sesuai akan diambil dari weight matrix. Jika ditemukan token yang index-nya melebihi vocabulary size, maka akan digunakan token padding (index 0) sebagai fallback.

```python
def forward(self, inputs):
    """
    Perform forward propagation for the LSTM layer

    Args:
        inputs: numpy array of shape (batch_size, sequence_length, input_dim)

    Returns:
        numpy array of shape (batch_size, hidden_dim) if bidirectional=False
        or (batch_size, hidden_dim*2) if bidirectional=True
    """
    batch_size, sequence_length, _ = inputs.shape

    # Initialize hidden and cell states
    h_forward = np.zeros((batch_size, self.hidden_dim))
    c_forward = np.zeros((batch_size, self.hidden_dim))

    # Process the sequence in forward direction
    for t in range(sequence_length):
        h_forward, c_forward = self.forward_step(
            inputs[:, t, :], h_forward, c_forward, direction="forward"
        )

    if not self.bidirectional:
        return h_forward

    # For bidirectional LSTM, process the sequence in backward direction
    h_backward = np.zeros((batch_size, self.hidden_dim))
    c_backward = np.zeros((batch_size, self.hidden_dim))

    for t in range(sequence_length - 1, -1, -1):
        h_backward, c_backward = self.forward_step(
            inputs[:, t, :], h_backward, c_backward, direction="backward"
        )

    # Concatenate forward and backward hidden states
    return np.concatenate([h_forward, h_backward], axis=1)
```

Weight loading dari Keras embedding layer sangat straightforward karena format weight matrix yang identik antara implementasi custom dan Keras.

```python
def load_weights_from_keras(self, keras_layer):
    """Load weights from a Keras LSTM layer"""
    if isinstance(keras_layer, tf.keras.layers.LSTM):
        # Keras weights order: [W_i|W_f|W_c|W_o, U_i|U_f|U_c|U_o, b_i|b_f|b_c|b_o]
        weights = keras_layer.get_weights()

        kernel = weights[0]  # Input weights (W)
        recurrent_kernel = weights[1]  # Recurrent weights (U)
        bias = weights[2]  # Biases

        # Extract weights for each gate
        hidden_dim = self.hidden_dim

        # Input weights
        self.W_i = kernel[:, :hidden_dim]
        self.W_f = kernel[:, hidden_dim : 2 * hidden_dim]
        self.W_c = kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.W_o = kernel[:, 3 * hidden_dim :]

        # Recurrent weights
        self.U_i = recurrent_kernel[:, :hidden_dim]
        self.U_f = recurrent_kernel[:, hidden_dim : 2 * hidden_dim]
        self.U_c = recurrent_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.U_o = recurrent_kernel[:, 3 * hidden_dim :]

        # Biases
        self.b_i = bias[:hidden_dim]
        self.b_f = bias[hidden_dim : 2 * hidden_dim]
        self.b_c = bias[2 * hidden_dim : 3 * hidden_dim]
        self.b_o = bias[3 * hidden_dim :]

    elif isinstance(keras_layer, tf.keras.layers.Bidirectional):
        # Extract weights from forward and backward layers
        forward_weights = keras_layer.forward_layer.get_weights()
        backward_weights = keras_layer.backward_layer.get_weights()

        hidden_dim = self.hidden_dim

        # Forward direction
        f_kernel = forward_weights[0]
        f_recurrent_kernel = forward_weights[1]
        f_bias = forward_weights[2]

        # Input weights
        self.forward_W_i = f_kernel[:, :hidden_dim]
        self.forward_W_f = f_kernel[:, hidden_dim : 2 * hidden_dim]
        self.forward_W_c = f_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.forward_W_o = f_kernel[:, 3 * hidden_dim :]

        # Recurrent weights
        self.forward_U_i = f_recurrent_kernel[:, :hidden_dim]
        self.forward_U_f = f_recurrent_kernel[:, hidden_dim : 2 * hidden_dim]
        self.forward_U_c = f_recurrent_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.forward_U_o = f_recurrent_kernel[:, 3 * hidden_dim :]

        # Biases
        self.forward_b_i = f_bias[:hidden_dim]
        self.forward_b_f = f_bias[hidden_dim : 2 * hidden_dim]
        self.forward_b_c = f_bias[2 * hidden_dim : 3 * hidden_dim]
        self.forward_b_o = f_bias[3 * hidden_dim :]

        # Backward direction
        b_kernel = backward_weights[0]
        b_recurrent_kernel = backward_weights[1]
        b_bias = backward_weights[2]

        # Input weights
        self.backward_W_i = b_kernel[:, :hidden_dim]
        self.backward_W_f = b_kernel[:, hidden_dim : 2 * hidden_dim]
        self.backward_W_c = b_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.backward_W_o = b_kernel[:, 3 * hidden_dim :]

        # Recurrent weights
        self.backward_U_i = b_recurrent_kernel[:, :hidden_dim]
        self.backward_U_f = b_recurrent_kernel[:, hidden_dim : 2 * hidden_dim]
        self.backward_U_c = b_recurrent_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.backward_U_o = b_recurrent_kernel[:, 3 * hidden_dim :]

        # Biases
        self.backward_b_i = b_bias[:hidden_dim]
        self.backward_b_f = b_bias[hidden_dim : 2 * hidden_dim]
        self.backward_b_c = b_bias[2 * hidden_dim : 3 * hidden_dim]
        self.backward_b_o = b_bias[3 * hidden_dim :]
```

LSTM layer adalah komponen utama yang menggunakan gating mechanism untuk mengontrol aliran informasi melalui time steps. Setiap LSTM cell memiliki cell state yang berfungsi sebagai memori jangka panjang dan hidden state yang berfungsi sebagai output pada setiap time step.

Inisialisasi weights dilakukan secara terpisah untuk setiap gate dan direction (jika bidirectional). Setiap gate memiliki input weights (W), recurrent weights (U), dan bias (b). Input weights menghubungkan input saat ini dengan gate, recurrent weights menghubungkan hidden state sebelumnya dengan gate, dan bias memberikan offset untuk setiap gate.

```
def initialize_weights(self, direction="forward"):
    """Initialize weights for the LSTM layer"""
    prefix = direction + "_" if self.bidirectional else ""

    # Input weights for the gates (input, forget, cell, output)
    setattr(
        self,
        f"{prefix}W_i",
        np.random.randn(self.input_dim, self.hidden_dim) * 0.01,
    )
    setattr(
        self,
        f"{prefix}W_f",
        np.random.randn(self.input_dim, self.hidden_dim) * 0.01,
    )
    setattr(
        self,
        f"{prefix}W_c",
        np.random.randn(self.input_dim, self.hidden_dim) * 0.01,
    )
    setattr(
        self,
        f"{prefix}W_o",
        np.random.randn(self.input_dim, self.hidden_dim) * 0.01,
    )

    # Recurrent weights for the gates
    setattr(
        self,
        f"{prefix}U_i",
        np.random.randn(self.hidden_dim, self.hidden_dim) * 0.01,
    )
    setattr(
        self,
        f"{prefix}U_f",
        np.random.randn(self.hidden_dim, self.hidden_dim) * 0.01,
    )
    setattr(
        self,
        f"{prefix}U_c",
        np.random.randn(self.hidden_dim, self.hidden_dim) * 0.01,
    )
    setattr(
        self,
        f"{prefix}U_o",
        np.random.randn(self.hidden_dim, self.hidden_dim) * 0.01,
    )

    # Biases for the gates
    setattr(self, f"{prefix}b_i", np.zeros(self.hidden_dim))
    setattr(self, f"{prefix}b_f", np.zeros(self.hidden_dim))
    setattr(self, f"{prefix}b_c", np.zeros(self.hidden_dim))
    setattr(self, f"{prefix}b_o", np.zeros(self.hidden_dim))
```

LSTM cell forward step merupakan inti dari komputasi LSTM dimana keempat gate dihitung dan digunakan untuk update cell state dan hidden state. Forget gate menentukan informasi mana yang akan dihapus dari cell state, input gate menentukan informasi baru mana yang akan disimpan, cell gate menghasilkan kandidat nilai baru, dan output gate menentukan bagian mana dari cell state yang akan dijadikan output.

```python
def forward_step(self, x_t, h_prev, c_prev, direction="forward"):
    """Perform one step of forward propagation for LSTM"""
    prefix = direction + "_" if self.bidirectional else ""

    # Get the weights for this direction
    W_i = getattr(self, f"{prefix}W_i")
    W_f = getattr(self, f"{prefix}W_f")
    W_c = getattr(self, f"{prefix}W_c")
    W_o = getattr(self, f"{prefix}W_o")

    U_i = getattr(self, f"{prefix}U_i")
    U_f = getattr(self, f"{prefix}U_f")
    U_c = getattr(self, f"{prefix}U_c")
    U_o = getattr(self, f"{prefix}U_o")

    b_i = getattr(self, f"{prefix}b_i")
    b_f = getattr(self, f"{prefix}b_f")
    b_c = getattr(self, f"{prefix}b_c")
    b_o = getattr(self, f"{prefix}b_o")

    # Input gate
    i_t = sigmoid(np.dot(x_t, W_i) + np.dot(h_prev, U_i) + b_i)

    # Forget gate
    f_t = sigmoid(np.dot(x_t, W_f) + np.dot(h_prev, U_f) + b_f)

    # Cell update
    c_tilde = tanh(np.dot(x_t, W_c) + np.dot(h_prev, U_c) + b_c)

    # Cell state
    c_t = f_t * c_prev + i_t * c_tilde

    # Output gate
    o_t = sigmoid(np.dot(x_t, W_o) + np.dot(h_prev, U_o) + b_o)

    # Hidden state
    h_t = o_t * tanh(c_t)

    return h_t, c_t
```

Fungsi aktivasi yang digunakan dalam LSTM adalah sigmoid dan tanh. Sigmoid digunakan untuk gate activation karena outputnya berada dalam range [0,1] yang cocok untuk mengontrol aliran informasi. Tanh digunakan untuk cell state update karena outputnya berada dalam range [-1,1] yang memungkinkan nilai negatif dan positif.

```python
def sigmoid(x):
    """Sigmoid activation function with clipping to prevent overflow"""
    return 1 / (1 + np.exp(-np.clip(x, -50, 50)))


def tanh(x):
    """Tanh activation function"""
    return np.tanh(x)
```

Forward propagation untuk unidirectional LSTM melibatkan pemrosesan sequence dari time step pertama hingga terakhir. Pada setiap time step, hidden state dan cell state diupdate menggunakan input saat ini dan state sebelumnya. Output final adalah hidden state dari time step terakhir yang merangkum informasi dari seluruh sequence.

Bidirectional LSTM memproses sequence dalam dua arah: forward (dari awal ke akhir) dan backward (dari akhir ke awal). Hal ini memungkinkan model untuk menangkap informasi kontekstual dari kedua arah, yang sering kali meningkatkan performa pada task seperti text classification. Output final adalah concatenation dari hidden state terakhir dari kedua arah.

```python
def forward(self, inputs):
    """
    Perform forward propagation for the LSTM layer

    Args:
        inputs: numpy array of shape (batch_size, sequence_length, input_dim)

    Returns:
        numpy array of shape (batch_size, hidden_dim) if bidirectional=False
        or (batch_size, hidden_dim*2) if bidirectional=True
    """
    batch_size, sequence_length, _ = inputs.shape

    # Initialize hidden and cell states
    h_forward = np.zeros((batch_size, self.hidden_dim))
    c_forward = np.zeros((batch_size, self.hidden_dim))

    # Process the sequence in forward direction
    for t in range(sequence_length):
        h_forward, c_forward = self.forward_step(
            inputs[:, t, :], h_forward, c_forward, direction="forward"
        )

    if not self.bidirectional:
        return h_forward

    # For bidirectional LSTM, process the sequence in backward direction
    h_backward = np.zeros((batch_size, self.hidden_dim))
    c_backward = np.zeros((batch_size, self.hidden_dim))

    for t in range(sequence_length - 1, -1, -1):
        h_backward, c_backward = self.forward_step(
            inputs[:, t, :], h_backward, c_backward, direction="backward"
        )

    # Concatenate forward and backward hidden states
    return np.concatenate([h_forward, h_backward], axis=1)
```

Dropout layer adalah teknik regularisasi yang digunakan untuk mencegah overfitting dengan cara secara random "mematikan" sebagian neuron selama training. Selama inference, semua neuron aktif tetapi output diskalakan sesuai dengan dropout rate untuk menjaga magnitude yang konsisten. Implementasi dropout menggunakan binomial distribution untuk menghasilkan mask yang menentukan neuron mana yang akan diaktifkan.

```python
def forward(self, inputs, training=True):
    self.training = training

    if not training:
        return inputs

    # Generate dropout mask (1: keep, 0: drop)
    self.mask = np.random.binomial(1, 1 - self.dropout_rate, size=inputs.shape) / (1 - self.dropout_rate)

    # Apply mask to inputs
    return inputs * self.mask
```

Dense layer atau fully connected layer melakukan transformasi linear dari LSTM output ke space klasifikasi. Layer ini menggunakan weight matrix dan bias vector untuk mentransformasi input dari hidden dimension ke number of classes. Aktivasi softmax biasanya diterapkan untuk menghasilkan distribusi probabilitas atas kelas-kelas yang mungkin.

```python
def forward(self, inputs):
    self.input = inputs

    # Initialize weights if needed
    if self.weights is None:
        self.initialize_weights_if_needed(inputs.shape[1])

    self.output = np.dot(inputs, self.weights) + self.biases

    if self.activation:
        self.output = self.activation.forward(self.output)

    return self.output
```

Weight loading dari Keras LSTM layer melibatkan ekstraksi dan reorganisasi weight matrices karena perbedaan format antara Keras dan implementasi custom. Keras menyimpan weights dalam format yang compact dimana semua gate weights digabungkan dalam satu matrix, sedangkan implementasi custom memisahkan weights untuk setiap gate.

Untuk unidirectional LSTM, Keras menyimpan weights dalam tiga komponen: kernel (input weights), recurrent_kernel (recurrent weights), dan bias. Setiap komponen perlu dipecah menjadi empat bagian sesuai dengan empat gate LSTM.

Untuk bidirectional LSTM, proses weight loading menjadi lebih kompleks karena perlu mengekstrak weights dari kedua arah. Keras menyimpan forward dan backward weights secara terpisah dalam forward_layer dan backward_layer attributes.

```python
def load_weights_from_keras(self, keras_layer):
    """Load weights from a Keras LSTM layer"""
    if isinstance(keras_layer, tf.keras.layers.LSTM):
        # Keras weights order: [W_i|W_f|W_c|W_o, U_i|U_f|U_c|U_o, b_i|b_f|b_c|b_o]
        weights = keras_layer.get_weights()

        kernel = weights[0]  # Input weights (W)
        recurrent_kernel = weights[1]  # Recurrent weights (U)
        bias = weights[2]  # Biases

        # Extract weights for each gate
        hidden_dim = self.hidden_dim

        # Input weights
        self.W_i = kernel[:, :hidden_dim]
        self.W_f = kernel[:, hidden_dim : 2 * hidden_dim]
        self.W_c = kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.W_o = kernel[:, 3 * hidden_dim :]

        # Recurrent weights
        self.U_i = recurrent_kernel[:, :hidden_dim]
        self.U_f = recurrent_kernel[:, hidden_dim : 2 * hidden_dim]
        self.U_c = recurrent_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.U_o = recurrent_kernel[:, 3 * hidden_dim :]

        # Biases
        self.b_i = bias[:hidden_dim]
        self.b_f = bias[hidden_dim : 2 * hidden_dim]
        self.b_c = bias[2 * hidden_dim : 3 * hidden_dim]
        self.b_o = bias[3 * hidden_dim :]

    elif isinstance(keras_layer, tf.keras.layers.Bidirectional):
        # Extract weights from forward and backward layers
        forward_weights = keras_layer.forward_layer.get_weights()
        backward_weights = keras_layer.backward_layer.get_weights()

        hidden_dim = self.hidden_dim

        # Forward direction
        f_kernel = forward_weights[0]
        f_recurrent_kernel = forward_weights[1]
        f_bias = forward_weights[2]

        # Input weights
        self.forward_W_i = f_kernel[:, :hidden_dim]
        self.forward_W_f = f_kernel[:, hidden_dim : 2 * hidden_dim]
        self.forward_W_c = f_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.forward_W_o = f_kernel[:, 3 * hidden_dim :]

        # Recurrent weights
        self.forward_U_i = f_recurrent_kernel[:, :hidden_dim]
        self.forward_U_f = f_recurrent_kernel[:, hidden_dim : 2 * hidden_dim]
        self.forward_U_c = f_recurrent_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.forward_U_o = f_recurrent_kernel[:, 3 * hidden_dim :]

        # Biases
        self.forward_b_i = f_bias[:hidden_dim]
        self.forward_b_f = f_bias[hidden_dim : 2 * hidden_dim]
        self.forward_b_c = f_bias[2 * hidden_dim : 3 * hidden_dim]
        self.forward_b_o = f_bias[3 * hidden_dim :]

        # Backward direction
        b_kernel = backward_weights[0]
        b_recurrent_kernel = backward_weights[1]
        b_bias = backward_weights[2]

        # Input weights
        self.backward_W_i = b_kernel[:, :hidden_dim]
        self.backward_W_f = b_kernel[:, hidden_dim : 2 * hidden_dim]
        self.backward_W_c = b_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.backward_W_o = b_kernel[:, 3 * hidden_dim :]

        # Recurrent weights
        self.backward_U_i = b_recurrent_kernel[:, :hidden_dim]
        self.backward_U_f = b_recurrent_kernel[:, hidden_dim : 2 * hidden_dim]
        self.backward_U_c = b_recurrent_kernel[:, 2 * hidden_dim : 3 * hidden_dim]
        self.backward_U_o = b_recurrent_kernel[:, 3 * hidden_dim :]

        # Biases
        self.backward_b_i = b_bias[:hidden_dim]
        self.backward_b_f = b_bias[hidden_dim : 2 * hidden_dim]
        self.backward_b_c = b_bias[2 * hidden_dim : 3 * hidden_dim]
        self.backward_b_o = b_bias[3 * hidden_dim :]
```

Data preprocessing untuk text classification melibatkan konversi text menjadi format numerik yang dapat diproses oleh model. NusaX data loader menyediakan fungsi untuk memuat dan memproses data sentiment analysis dalam bahasa Indonesia.

Preprocessing data dimulai dengan ekstraksi text dan label dari dataset. Label kategorikal dikonversi menjadi integer encoding untuk mempermudah proses training dan evaluasi.

```
def preprocess_data(self, df):
    """Preprocess the data and return texts and labels"""
    texts = df["text"].values
    # Map labels to integer indices if they are not already
    if df["label"].dtype == "object":
        label_map = {"negative": 0, "neutral": 1, "positive": 2}
        labels = df["label"].map(label_map).values
    else:
        labels = df["label"].values
    return texts, labels
```

Text vectorization menggunakan TextVectorization layer dari TensorFlow untuk mengkonversi text menjadi sequence of integers. Proses ini melibatkan tokenization, vocabulary creation, dan sequence padding untuk memastikan semua sequence memiliki panjang yang sama.

```
def _create_vectorizer(self, texts):
    """Create and fit the text vectorizer"""
    self.vectorizer = TextVectorization(
        max_tokens=self.vocab_size,
        output_mode="int",
        output_sequence_length=self.max_sequence_length,
    )
    self.vectorizer.adapt(texts)
    self.vocabulary = self.vectorizer.get_vocabulary()
    return self.vectorizer
```

2.2. Hasil Pengujian

2.2.1. CNN

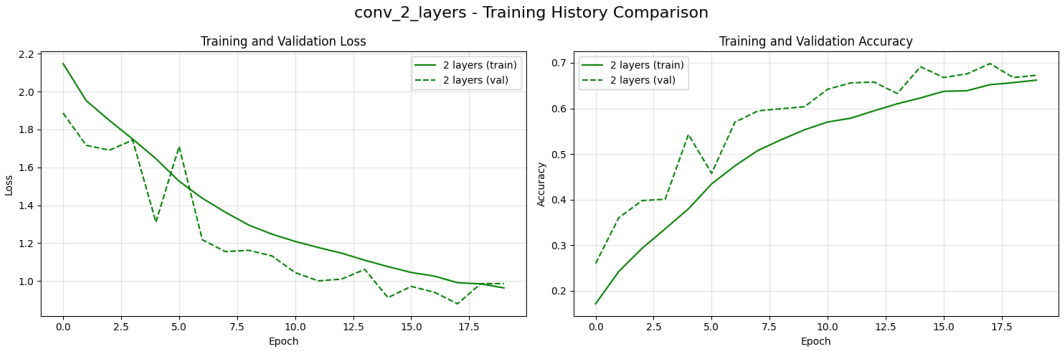
2.2.1.1. Pengaruh jumlah layer konvolusi

<table>
<tr><th>Jumlah Layer</th><th>CNN Implementasi</th></tr>
<tr><td>2</td><td>
<table>
<tr><th>Layer (type)</th><th>Output Shape</th><th>Param #</th></tr>
<tr><td>conv2d (Conv2D)</td><td>(None, 32, 32, 32)</td><td>896</td></tr>
<tr><td>batch_normalization (BatchNormalization)</td><td>(None, 32, 32, 32)</td><td>128</td></tr>
<tr><td>max_pooling2d (MaxPooling2D)</td><td>(None, 16, 16, 32)</td><td>0</td></tr>
<tr><td>dropout (Dropout)</td><td>(None, 16, 16, 32)</td><td>0</td></tr>
<tr><td>conv2d_1 (Conv2D)</td><td>(None, 16, 16, 64)</td><td>18,496</td></tr>
<tr><td>batch_normalization_1 (BatchNormalization)</td><td>(None, 16, 16, 64)</td><td>256</td></tr>
<tr><td>max_pooling2d_1 (MaxPooling2D)</td><td>(None, 8, 8, 64)</td><td>0</td></tr>
<tr><td>dropout_1 (Dropout)</td><td>(None, 8, 8, 64)</td><td>0</td></tr>
<tr><td>flatten (Flatten)</td><td>(None, 4096)</td><td>0</td></tr>
<tr><td>dense (Dense)</td><td>(None, 64)</td><td>262,208</td></tr>
<tr><td>dropout_2 (Dropout)</td><td>(None, 64)</td><td>0</td></tr>
<tr><td>dense_1 (Dense)</td><td>(None, 10)</td><td>650</td></tr>
</table>
Total params: 282,634 (1.08 MB)
Trainable params: 282,442 (1.08 MB)
Non-trainable params: 192 (768.00 B)
</td></tr>
</table>

```
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.5499 - loss: 1.2577 - val_accuracy: 0.6036 - val_loss: 1.1324
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.5674 - loss: 1.2129 - val_accuracy: 0.6421 - val_loss: 1.0445
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.5796 - loss: 1.1787 - val_accuracy: 0.6560 - val_loss: 1.0005
Epoch 13/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.5952 - loss: 1.1440 - val_accuracy: 0.6578 - val_loss: 1.0102
...
Epoch 19/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.6563 - loss: 0.9815 - val_accuracy: 0.6677 - val_loss: 0.9859
Epoch 20/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6602 - loss: 0.9628 - val_accuracy: 0.6727 - val_loss: 0.9868
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file
Test Accuracy: 0.6984
Macro F1-Score: 0.7040
Test Loss: 0.8817
```



3

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 32, 32, 32)	896
batch_normalization_2 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d_2 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_3 (Dropout)	(None, 16, 16, 32)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_4 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_4 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_5 (Dropout)	(None, 4, 4, 128)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_2 (Dense)	(None, 64)	131,136
dropout_6 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 10)	650

Total params: 225,930 (882.54 KB)

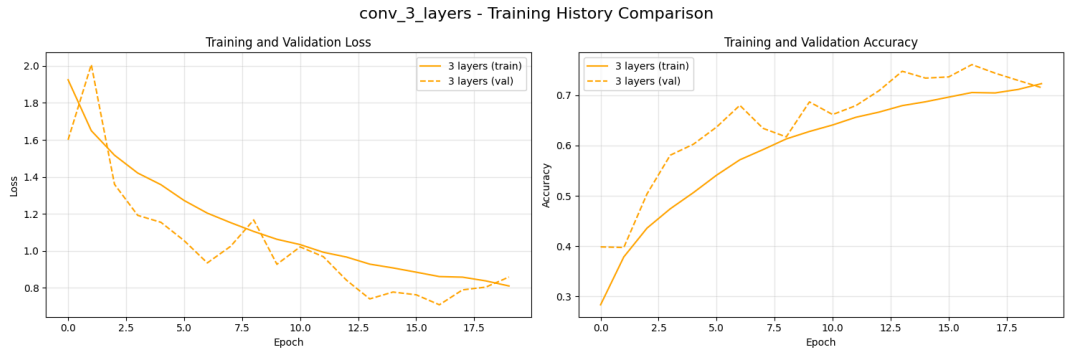
Trainable params: 225,482 (880.79 KB)

Non-trainable params: 448 (1.75 KB)

```
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.6254 - loss: 1.0690 - val_accuracy: 0.6865 - val_loss: 0.9275
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6372 - loss: 1.0354 - val_accuracy: 0.6614 - val_loss: 1.0217
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.6481 - loss: 1.0082 - val_accuracy: 0.6791 - val_loss: 0.9687
Epoch 13/20
...
Epoch 19/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.7119 - loss: 0.8353 - val_accuracy: 0.7291 - val_loss: 0.8041
Epoch 20/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.7256 - loss: 0.8013 - val_accuracy: 0.7148 - val_loss: 0.8588
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file fo
Test Accuracy: 0.7520
Macro F1-Score: 0.7513
Test Loss: 0.7252
```



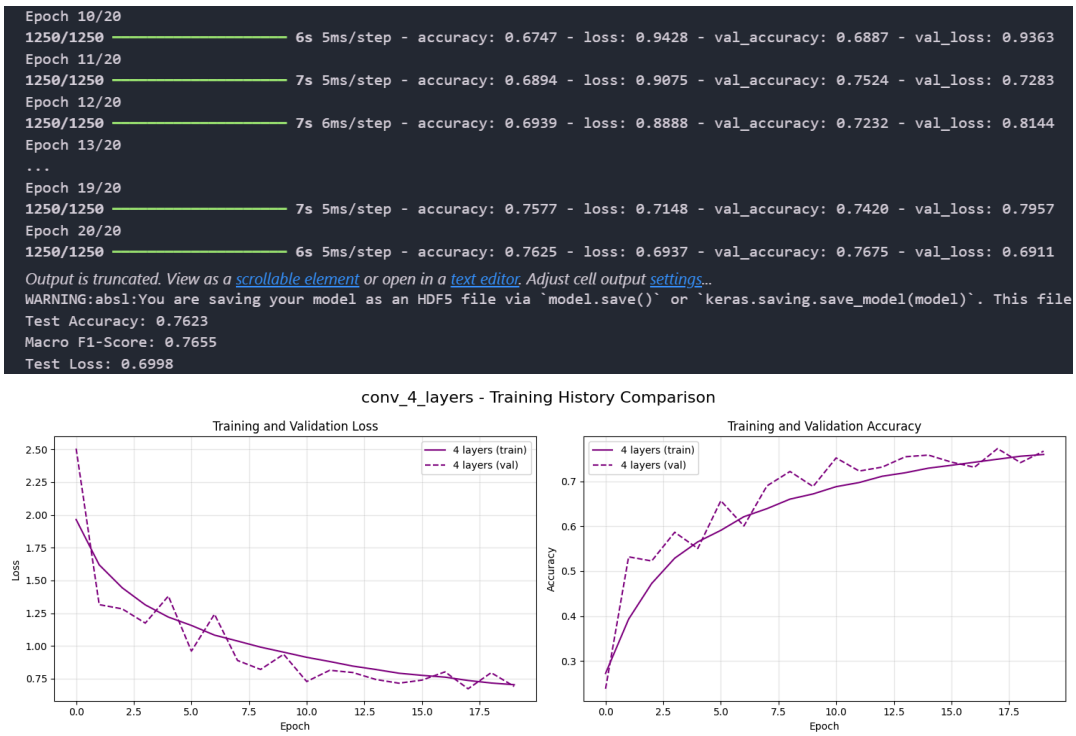
4

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 32, 32, 32)	896
batch_normalization_5 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d_5 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_7 (Dropout)	(None, 16, 16, 32)	0
conv2d_6 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_6 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_6 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_8 (Dropout)	(None, 8, 8, 64)	0
conv2d_7 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_7 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_7 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_9 (Dropout)	(None, 4, 4, 128)	0
conv2d_8 (Conv2D)	(None, 4, 4, 256)	295,168
batch_normalization_8 (BatchNormalization)	(None, 4, 4, 256)	1,024
max_pooling2d_8 (MaxPooling2D)	(None, 2, 2, 256)	0
dropout_10 (Dropout)	(None, 2, 2, 256)	0
flatten_2 (Flatten)	(None, 1024)	0
dense_4 (Dense)	(None, 64)	65,600
dropout_11 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 10)	650

Total params: 456,586 (1.74 MB)

Trainable params: 455,626 (1.74 MB)

Non-trainable params: 960 (3.75 KB)

	
Kesimpulan	Semakin banyak jumlah layer, nilai f1-score semakin bagus, total parameternya semakin banyak

2.2.1.2. Pengaruh banyak filter per layer konvolusi

Banyak Filter	CNN Implementasi

[16, 32, 64]

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 16)	448
batch_normalization (BatchNormalization)	(None, 32, 32, 16)	64
max_pooling2d (MaxPooling2D)	(None, 16, 16, 16)	0
dropout (Dropout)	(None, 16, 16, 16)	0
conv2d_1 (Conv2D)	(None, 16, 16, 32)	4,640
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 32)	128
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 32)	0
dropout_1 (Dropout)	(None, 8, 8, 32)	0
conv2d_2 (Conv2D)	(None, 8, 8, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 8, 8, 64)	256
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 64)	0
dropout_2 (Dropout)	(None, 4, 4, 64)	0
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 64)	65,600
dropout_3 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 10)	650

Total params: 90,282 (352.66 KB)

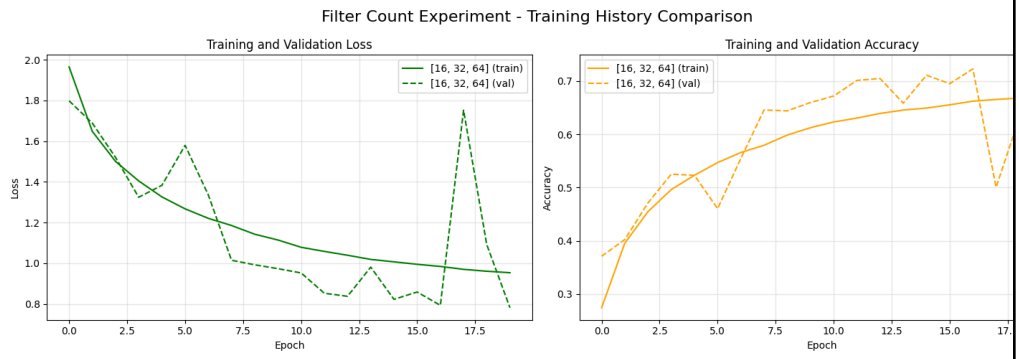
Trainable params: 90,058 (351.79 KB)

Non-trainable params: 224 (896.00 B)

```
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6142 - loss: 1.1074 - val_accuracy: 0.6596 - val_loss: 0.97
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6241 - loss: 1.0767 - val_accuracy: 0.6718 - val_loss: 0.95
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6320 - loss: 1.0544 - val_accuracy: 0.7011 - val_loss: 0.85
Epoch 13/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6376 - loss: 1.0397 - val_accuracy: 0.7048 - val_loss: 0.83
...
Epoch 19/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6700 - loss: 0.9552 - val_accuracy: 0.6307 - val_loss: 1.09
Epoch 20/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6708 - loss: 0.9491 - val_accuracy: 0.7265 - val_loss: 0.78
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This
Test Accuracy: 0.7275
Macro F1-Score: 0.7244
Test Loss: 0.8036
Total Parameters: 90,282
```



[32, 64, 128]

Layer (type)	Output Shape	Param
conv2d_3 (Conv2D)	(None, 32, 32, 32)	89
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 32)	12
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 32)	
dropout_4 (Dropout)	(None, 16, 16, 32)	
conv2d_4 (Conv2D)	(None, 16, 16, 64)	18,49
batch_normalization_4 (BatchNormalization)	(None, 16, 16, 64)	25
max_pooling2d_4 (MaxPooling2D)	(None, 8, 8, 64)	
dropout_5 (Dropout)	(None, 8, 8, 64)	
conv2d_5 (Conv2D)	(None, 8, 8, 128)	73,85
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	51
max_pooling2d_5 (MaxPooling2D)	(None, 4, 4, 128)	
dropout_6 (Dropout)	(None, 4, 4, 128)	
flatten_1 (Flatten)	(None, 2048)	
dense_2 (Dense)	(None, 64)	131,13
dropout_7 (Dropout)	(None, 64)	
dense_3 (Dense)	(None, 10)	65

Total params: 225,930 (882.54 KB)

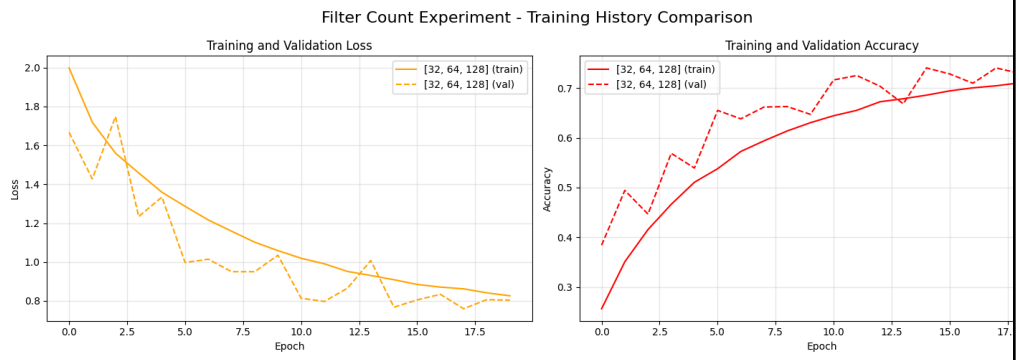
Trainable params: 225,482 (880.79 KB)

Non-trainable params: 448 (1.75 KB)

```
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.6303 - loss: 1.0539 - val_accuracy: 0.6473 - val_loss: 1
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.6468 - loss: 1.0153 - val_accuracy: 0.7165 - val_loss: 0
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.6572 - loss: 0.9865 - val_accuracy: 0.7254 - val_loss: 0
Epoch 13/20
...
Epoch 19/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.7120 - loss: 0.8363 - val_accuracy: 0.7299 - val_loss: 0
Epoch 20/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.7159 - loss: 0.8242 - val_accuracy: 0.7176 - val_loss: 0
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. Th
Test Accuracy: 0.7356
Macro F1-Score: 0.7344
Test Loss: 0.7707
Total Parameters: 225,930
```



[64, 128, 256]

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 32, 32, 64)	1,79
batch_normalization_6 (BatchNormalization)	(None, 32, 32, 64)	25
max_pooling2d_6 (MaxPooling2D)	(None, 16, 16, 64)	
dropout_8 (Dropout)	(None, 16, 16, 64)	
conv2d_7 (Conv2D)	(None, 16, 16, 128)	73,85
batch_normalization_7 (BatchNormalization)	(None, 16, 16, 128)	51
max_pooling2d_7 (MaxPooling2D)	(None, 8, 8, 128)	
dropout_9 (Dropout)	(None, 8, 8, 128)	
conv2d_8 (Conv2D)	(None, 8, 8, 256)	295,16
batch_normalization_8 (BatchNormalization)	(None, 8, 8, 256)	1,02
max_pooling2d_8 (MaxPooling2D)	(None, 4, 4, 256)	
dropout_10 (Dropout)	(None, 4, 4, 256)	
flatten_2 (Flatten)	(None, 4096)	
dense_4 (Dense)	(None, 64)	262,20
dropout_11 (Dropout)	(None, 64)	
dense_5 (Dense)	(None, 10)	65

Total params: 635,466 (2.42 MB)

Trainable params: 634,570 (2.42 MB)

Non-trainable params: 896 (3.50 KB)

	
Kesimpulan	Semakin banyak jumlah filter per layer, nilai f1-score semakin bagus, total parameternya semakin banyak

2.2.1.3. Pengaruh ukuran filter per layer konvolusi

Ukuran Filter	CNN Implementasi

[3, 3, 3]

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_1 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_2 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 64)	131,136
dropout_3 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 10)	650

Total params: 225,930 (882.54 KB)

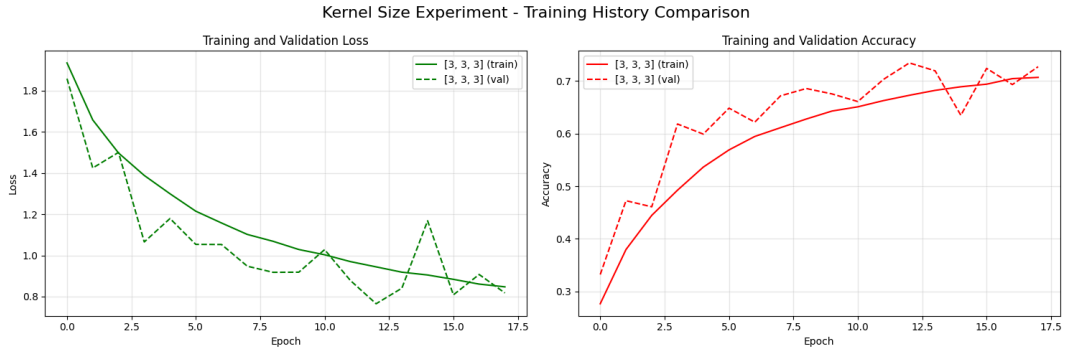
Trainable params: 225,482 (880.79 KB)

Non-trainable params: 448 (1.75 KB)

```
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.6379 - loss: 1.0329 - val_accuracy: 0.6755 - val_loss: 0.9185
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.6512 - loss: 1.0012 - val_accuracy: 0.6610 - val_loss: 1.0280
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.6662 - loss: 0.9596 - val_accuracy: 0.7033 - val_loss: 0.8785
Epoch 13/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.6736 - loss: 0.9400 - val_accuracy: 0.7345 - val_loss: 0.7654
...
Epoch 17/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.7026 - loss: 0.8661 - val_accuracy: 0.6933 - val_loss: 0.9074
Epoch 18/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.7074 - loss: 0.8451 - val_accuracy: 0.7273 - val_loss: 0.8177
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file
Test Accuracy: 0.7267
Macro F1-Score: 0.7269
Test Loss: 0.7905
Total Parameters: 225,930
Receptive Field: 15x15
```



[5, 5, 5]

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 32, 32, 32)	2,432
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_4 (Dropout)	(None, 16, 16, 32)	0
conv2d_4 (Conv2D)	(None, 16, 16, 64)	51,264
batch_normalization_4 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_4 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_5 (Dropout)	(None, 8, 8, 64)	0
conv2d_5 (Conv2D)	(None, 8, 8, 128)	204,928
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_5 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_6 (Dropout)	(None, 4, 4, 128)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_2 (Dense)	(None, 64)	131,136
dropout_7 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 10)	650

Total params: 391,306 (1.49 MB)

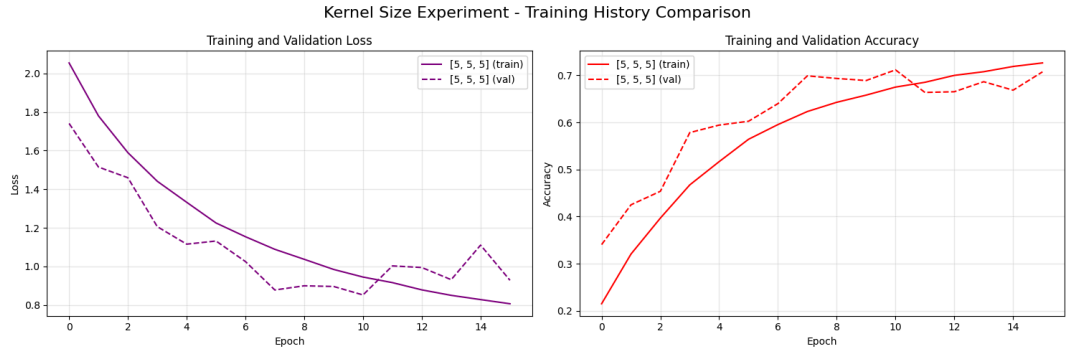
Trainable params: 390,858 (1.49 MB)

Non-trainable params: 448 (1.75 KB)

```
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.6565 - loss: 0.9812 - val_accuracy: 0.6888 - val_loss: 0.8957
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - accuracy: 0.6750 - loss: 0.9412 - val_accuracy: 0.7114 - val_loss: 0.8518
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - accuracy: 0.6833 - loss: 0.9184 - val_accuracy: 0.6635 - val_loss: 1.0024
Epoch 13/20
...
Epoch 15/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.7146 - loss: 0.8266 - val_accuracy: 0.6683 - val_loss: 1.1102
Epoch 16/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.7301 - loss: 0.7931 - val_accuracy: 0.7072 - val_loss: 0.9278
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file
Test Accuracy: 0.7033
Macro F1-Score: 0.7121
Test Loss: 0.8732
Total Parameters: 391,306
Receptive Field: 29x29
```



[7, 7, 7]

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 32, 32, 32)	4,736
batch_normalization_6 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d_6 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_8 (Dropout)	(None, 16, 16, 32)	0
conv2d_7 (Conv2D)	(None, 16, 16, 64)	100,416
batch_normalization_7 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_7 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_9 (Dropout)	(None, 8, 8, 64)	0
conv2d_8 (Conv2D)	(None, 8, 8, 128)	401,536
batch_normalization_8 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_8 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_10 (Dropout)	(None, 4, 4, 128)	0
flatten_2 (Flatten)	(None, 2048)	0
dense_4 (Dense)	(None, 64)	131,136
dropout_11 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 10)	650

Total params: 639,370 (2.44 MB)

Trainable params: 638,922 (2.44 MB)

Non-trainable params: 448 (1.75 KB)

	
Kesimpulan	Semakin besar ukuran filter per layer konvolusi, total parameter semakin banyak, f1-score fluktuatif

2.2.1.4. Pengaruh jenis pooling layer

Pooling Layer	CNN Implementasi

Max Pooling	

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_1 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_2 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 64)	131,136
dropout_3 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 10)	650

Total params: 225,930 (882.54 KB)

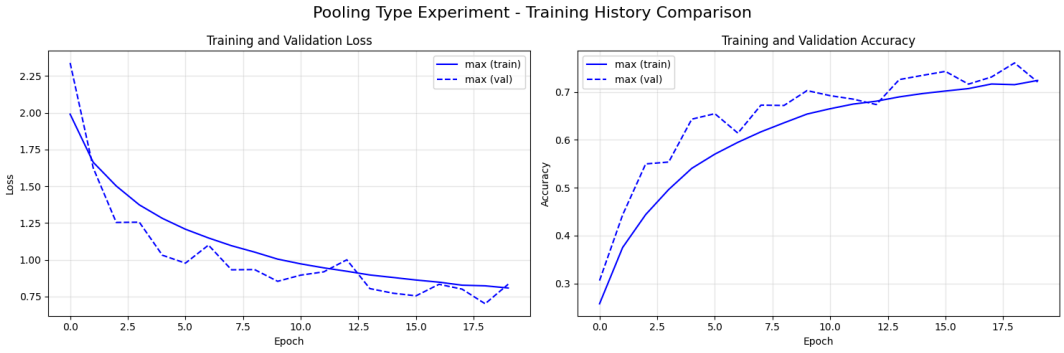
Trainable params: 225,482 (880.79 KB)

Non-trainable params: 448 (1.75 KB)

```
Epoch 10/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6573 - loss: 1.0057 - val_accuracy: 0.7030 - val_loss: 0.8530
Epoch 11/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.6622 - loss: 0.9758 - val_accuracy: 0.6924 - val_loss: 0.8950
Epoch 12/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6729 - loss: 0.9531 - val_accuracy: 0.6851 - val_loss: 0.9174
Epoch 13/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.6793 - loss: 0.9219 - val_accuracy: 0.6738 - val_loss: 0.9995
...
Epoch 19/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.7145 - loss: 0.8247 - val_accuracy: 0.7608 - val_loss: 0.7003
Epoch 20/20
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.7228 - loss: 0.8077 - val_accuracy: 0.7209 - val_loss: 0.8332
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file fo
Test Accuracy: 0.7615
Macro F1-Score: 0.7603
Test Loss: 0.7098
Total Parameters: 225,930
```



Average Pooling

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 32, 32, 32)	896
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 32)	128
average_pooling2d (AveragePooling2D)	(None, 16, 16, 32)	0
dropout_4 (Dropout)	(None, 16, 16, 32)	0
conv2d_4 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_4 (BatchNormalization)	(None, 16, 16, 64)	256
average_pooling2d_1 (AveragePooling2D)	(None, 8, 8, 64)	0
dropout_5 (Dropout)	(None, 8, 8, 64)	0
conv2d_5 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	512
average_pooling2d_2 (AveragePooling2D)	(None, 4, 4, 128)	0
dropout_6 (Dropout)	(None, 4, 4, 128)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_2 (Dense)	(None, 64)	131,136
dropout_7 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 10)	650

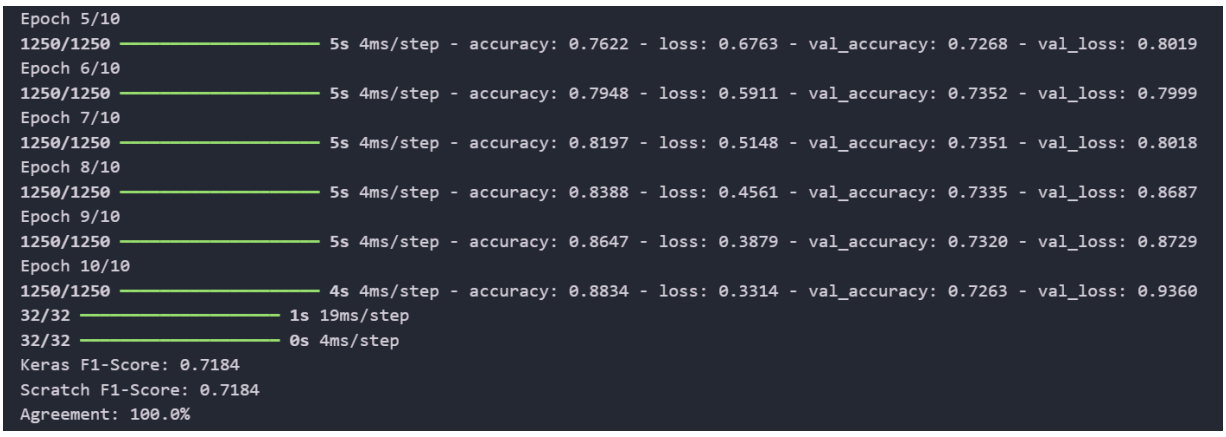
Total params: 225,930 (882.54 KB)

Trainable params: 225,482 (880.79 KB)

Non-trainable params: 448 (1.75 KB)

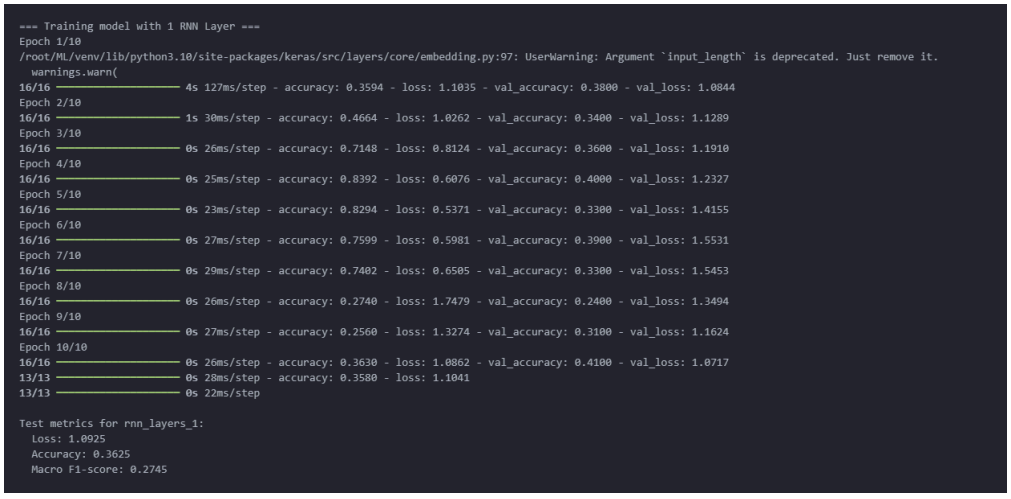
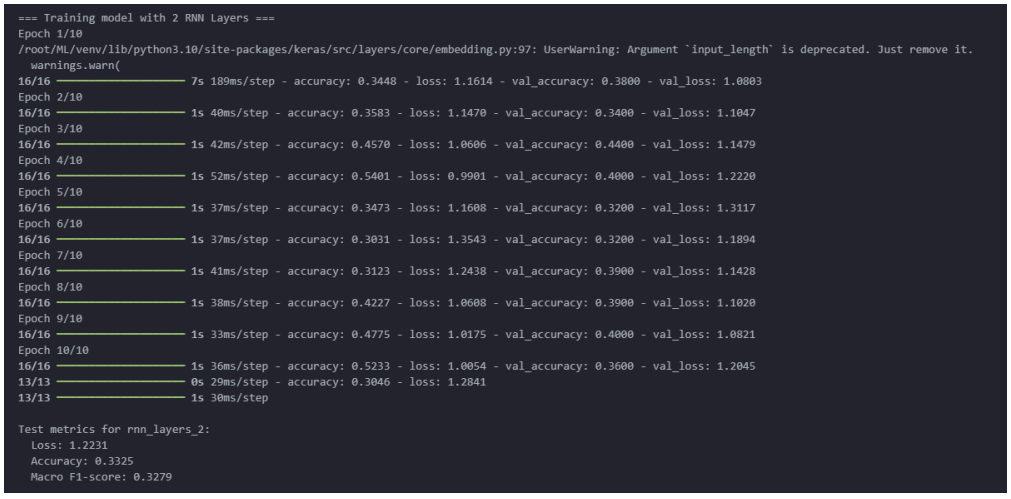
	
Kesimpulan	Total parameter sama, nilai f1-score average pooling lebih bagus dibandingkan max pooling

2.2.1.5. Perbandingan model Keras dan model from scratch

Keras vs Manual

Hasil percobaan perbandingan model Keras dengan Manual menghasilkan nilai f1-score yang sama besar

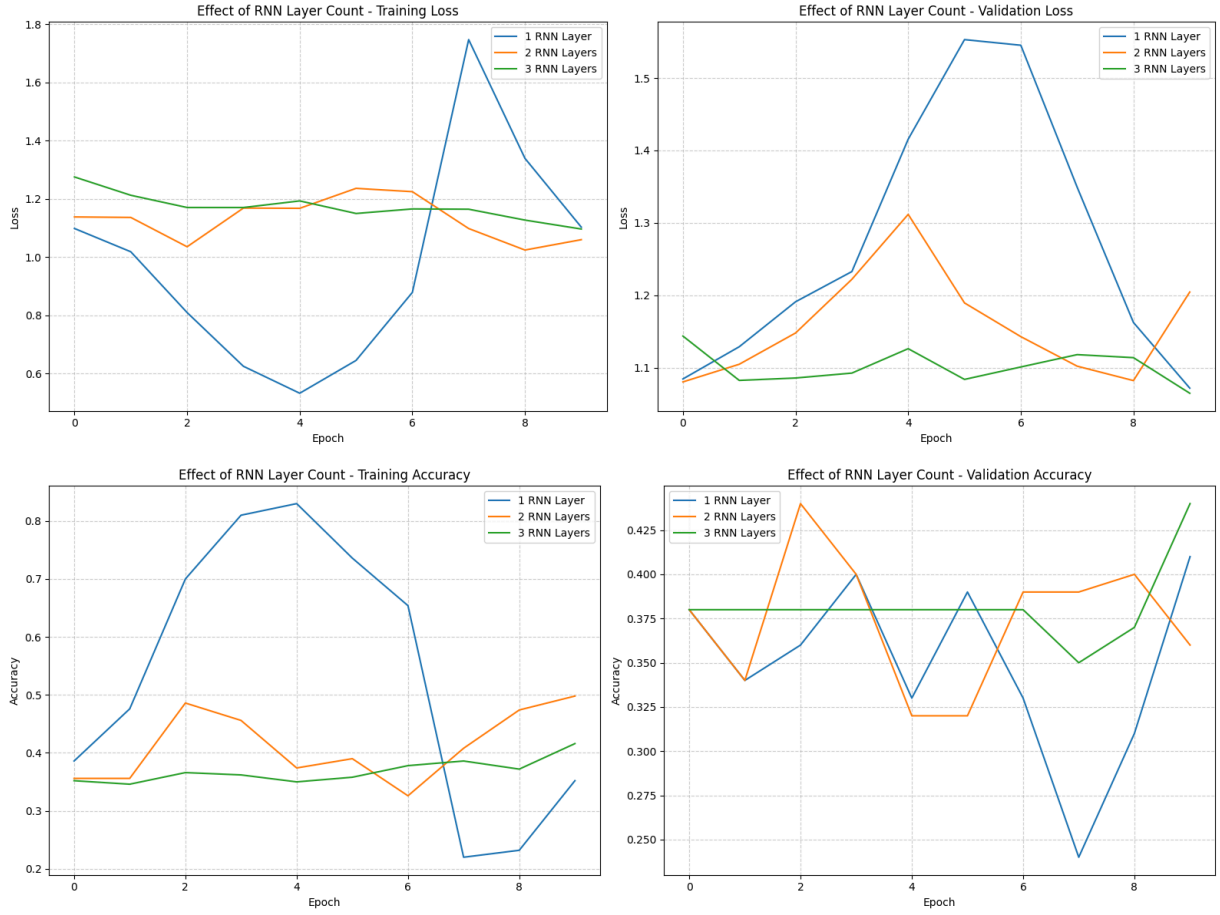
2.2.2. Simple RNN

2.2.2.1. Pengaruh jumlah layer RNN

Jumlah Layer	Hasil Eksekusi
1	
2	

3

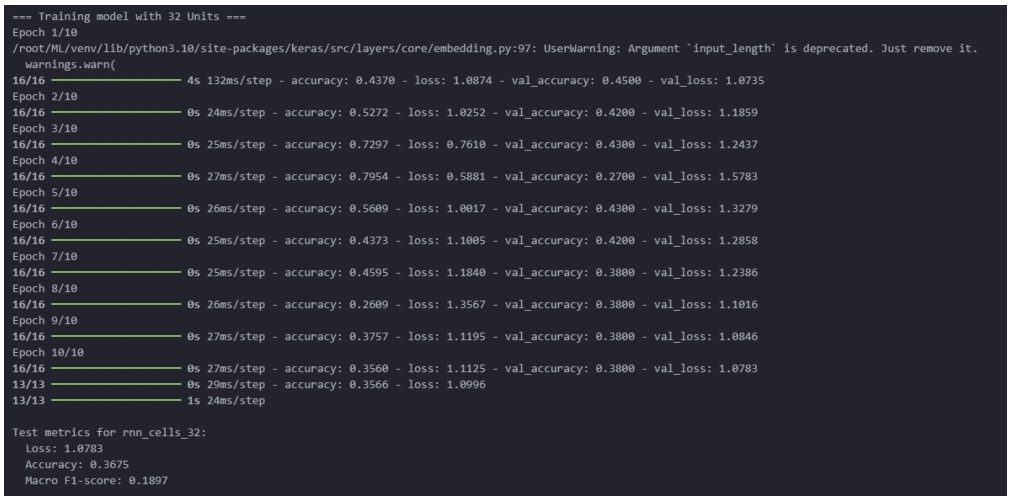
```
=== Training model with 3 RNN Layers ===
Epoch 1/10
/root/ML/venv/lib/python3.10/site-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
16/16 ━━━━━━━━━━━━━━━━━━━━ 11s 261ms/step - accuracy: 0.3482 - loss: 1.3259 - val_accuracy: 0.3800 - val_loss: 1.1436
Epoch 2/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 55ms/step - accuracy: 0.3085 - loss: 1.2896 - val_accuracy: 0.3800 - val_loss: 1.0824
Epoch 3/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - accuracy: 0.3425 - loss: 1.1848 - val_accuracy: 0.3800 - val_loss: 1.0857
Epoch 4/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 46ms/step - accuracy: 0.3525 - loss: 1.1909 - val_accuracy: 0.3800 - val_loss: 1.0925
Epoch 5/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.3683 - loss: 1.2079 - val_accuracy: 0.3800 - val_loss: 1.1262
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 44ms/step - accuracy: 0.3655 - loss: 1.1498 - val_accuracy: 0.3800 - val_loss: 1.0837
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - accuracy: 0.3888 - loss: 1.1262 - val_accuracy: 0.3800 - val_loss: 1.1009
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - accuracy: 0.4102 - loss: 1.1889 - val_accuracy: 0.3500 - val_loss: 1.1180
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.3661 - loss: 1.1348 - val_accuracy: 0.3700 - val_loss: 1.1138
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - accuracy: 0.3980 - loss: 1.1166 - val_accuracy: 0.4400 - val_loss: 1.0646
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - accuracy: 0.3447 - loss: 1.1327
13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 42ms/step

Test metrics for rnn_layers_3:
  Loss: 1.0970
  Accuracy: 0.3775
  Macro F1-score: 0.3444
```



Jumlah lapisan RNN cukup berpengaruh disini. Dimana dengan 1 layer RNN, cenderung overfit karena saat training loss turun validation loss nya malah naik. Sedangkan model dengan 2 dan 3 layer RNN cenderung stabil dan tidak fluktuatif. Namun secara keseluruhan RNN dengan 3 layer memiliki performa validation loss yang terbaik dan Training Loss yang stabil. Hal ini mungkiin dikarenakan layer yang banyak mampu menangkap pattern yang lebih

general sehingga tidak overfit dan underfit.

2.2.2.2. Pengaruh banyak cell RNN per layer

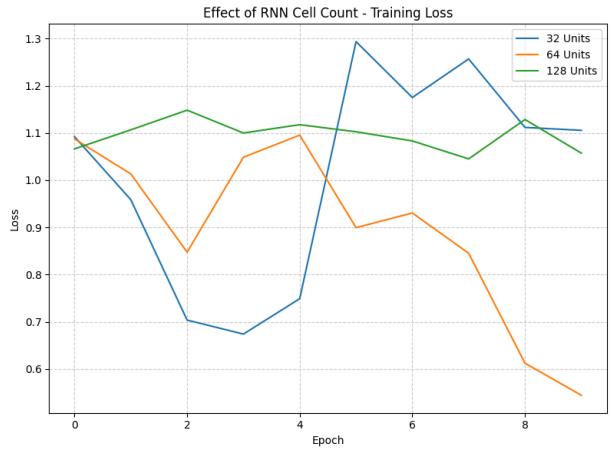
Jumlah *Cell*	Hasil Eksekusi
1	

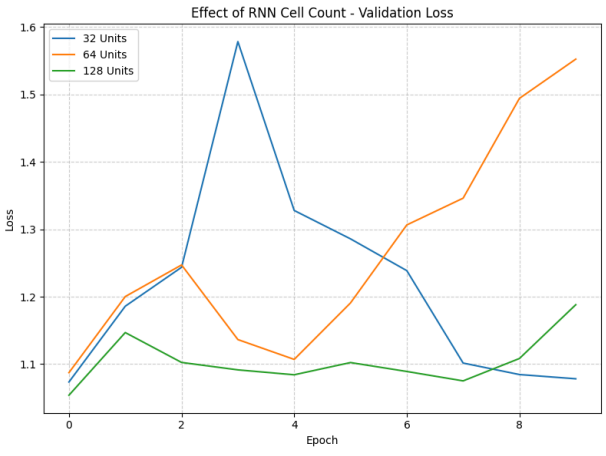
2

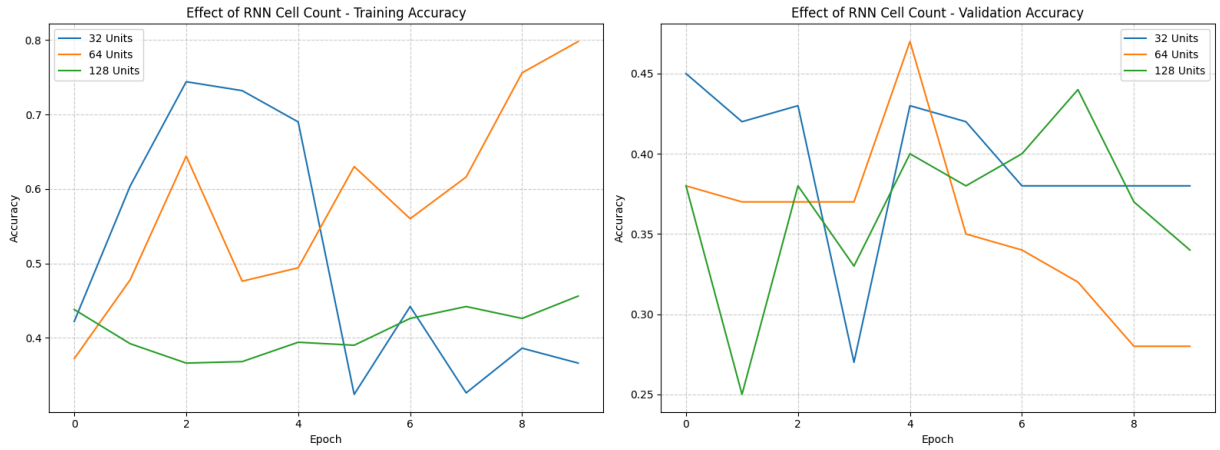
```
=== Training model with 64 Units ===
Epoch 1/10
/root/ML/venv/lib/python3.10/site-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
16/16 ━━━━━━━━━━━━━━━━━━━━ 4s 148ms/step - accuracy: 0.3511 - loss: 1.0973 - val_accuracy: 0.3800 - val_loss: 1.0873
Epoch 2/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.3964 - loss: 1.0612 - val_accuracy: 0.3700 - val_loss: 1.2002
Epoch 3/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.5729 - loss: 0.9211 - val_accuracy: 0.3700 - val_loss: 1.2471
Epoch 4/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.5382 - loss: 0.9473 - val_accuracy: 0.3700 - val_loss: 1.1364
Epoch 5/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.4633 - loss: 1.1117 - val_accuracy: 0.4700 - val_loss: 1.1070
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.5742 - loss: 0.9798 - val_accuracy: 0.3500 - val_loss: 1.1908
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6106 - loss: 0.8666 - val_accuracy: 0.3400 - val_loss: 1.3065
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.5807 - loss: 0.8775 - val_accuracy: 0.3200 - val_loss: 1.3461
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.7183 - loss: 0.6592 - val_accuracy: 0.2800 - val_loss: 1.4942
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.7643 - loss: 0.5830 - val_accuracy: 0.2800 - val_loss: 1.5523
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.2790 - loss: 1.6918
13/13 ━━━━━━━━━━━━━━━━━━━━ 4s 322ms/step

Test metrics for rnn_cells_64:
  Loss: 1.6094
  Accuracy: 0.2975
  Macro F1-score: 0.2702
```

3

```
=== Training model with 128 Units ===
Epoch 1/10
/root/ML/venv/lib/python3.10/site-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
16/16 ━━━━━━━━━━━━━━━━━━━━ 4s 135ms/step - accuracy: 0.4323 - loss: 1.0701 - val_accuracy: 0.3800 - val_loss: 1.0539
Epoch 2/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.4146 - loss: 1.0870 - val_accuracy: 0.2500 - val_loss: 1.1468
Epoch 3/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.3198 - loss: 1.1881 - val_accuracy: 0.3800 - val_loss: 1.1025
Epoch 4/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.3616 - loss: 1.1101 - val_accuracy: 0.3300 - val_loss: 1.0915
Epoch 5/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3817 - loss: 1.1421 - val_accuracy: 0.4000 - val_loss: 1.0842
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.3763 - loss: 1.1084 - val_accuracy: 0.3800 - val_loss: 1.1024
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3834 - loss: 1.1085 - val_accuracy: 0.4000 - val_loss: 1.0891
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.4460 - loss: 1.0516 - val_accuracy: 0.4400 - val_loss: 1.0752
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.5204 - loss: 1.0315 - val_accuracy: 0.3700 - val_loss: 1.1084
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.4679 - loss: 1.0457 - val_accuracy: 0.3400 - val_loss: 1.1881
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.3659 - loss: 1.1589
13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step

Test metrics for rnn_cells_128:
  Loss: 1.1352
  Accuracy: 0.3750
  Macro F1-score: 0.2840
```







Jumlah Cell/neuron di RNN terlihat memiliki efek yang cukup terlihat dimana model dengan cell terbanyak berperforma buruk di training , namun berpeforma baik di validation. Tidak seperti yang cellnya 32 dan 64 dimana sering terjadi fluktuatiif. Hal ini mungkin karena Cell yang banyak mungkin memang bisa overfit, namun karena dibantu dropout layer menyebabkan model bisa menangkap pola yang lebih general.

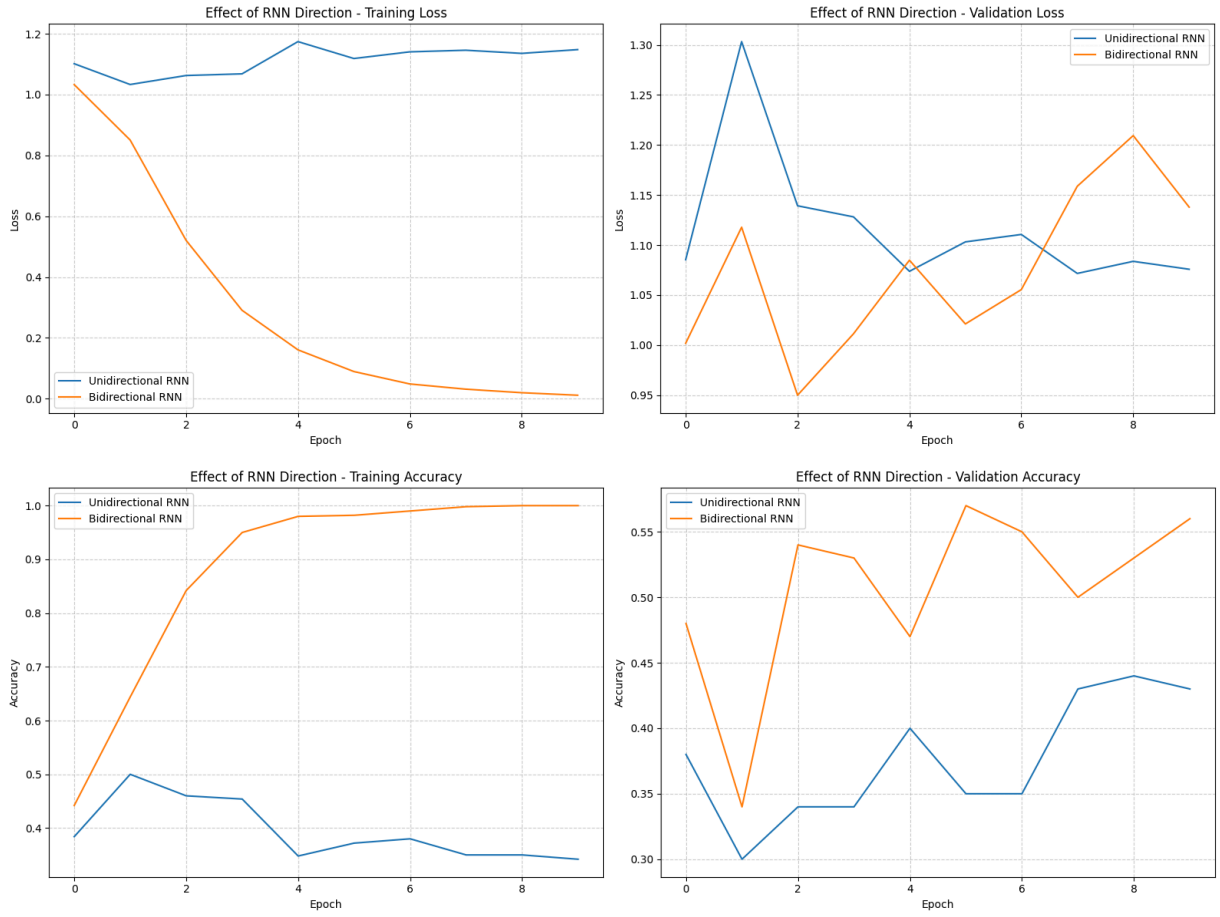
2.2.2.3. Pengaruh jenis layer RNN berdasarkan arah

Jenis Arah	Hasil Eksekusi

Unidirectional	`=== Training model with Unidirectional RNN RNN ===`
	`Epoch 1/10`
	`/root/ML/venv/lib/python3.10/site-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is deprecated. Just remove it.`
	`  warnings.warn(`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 129ms/step - accuracy: 0.3567 - loss: 1.1107 - val_accuracy: 0.3800 - val_loss: 1.0853`
	`Epoch 2/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.4926 - loss: 1.0353 - val_accuracy: 0.3000 - val_loss: 1.3034`
	`Epoch 3/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.4358 - loss: 1.0651 - val_accuracy: 0.3400 - val_loss: 1.1392`
	`Epoch 4/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.4445 - loss: 1.0729 - val_accuracy: 0.3400 - val_loss: 1.1281`
	`Epoch 5/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3423 - loss: 1.1800 - val_accuracy: 0.4000 - val_loss: 1.0737`
	`Epoch 6/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.4003 - loss: 1.1027 - val_accuracy: 0.3500 - val_loss: 1.1031`
	`Epoch 7/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3955 - loss: 1.1247 - val_accuracy: 0.3500 - val_loss: 1.1105`
	`Epoch 8/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3545 - loss: 1.1545 - val_accuracy: 0.4300 - val_loss: 1.0716`
	`Epoch 9/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3605 - loss: 1.1481 - val_accuracy: 0.4400 - val_loss: 1.0837`
	`Epoch 10/10`
	`16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3386 - loss: 1.1538 - val_accuracy: 0.4300 - val_loss: 1.0758`
	`13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3734 - loss: 1.1347`
	`13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step`
	
	`Test metrics for rnn_unidirectional:`
	`  Loss: 1.0944`
	`  Accuracy: 0.3850`
	`  Macro F1-score: 0.3022`

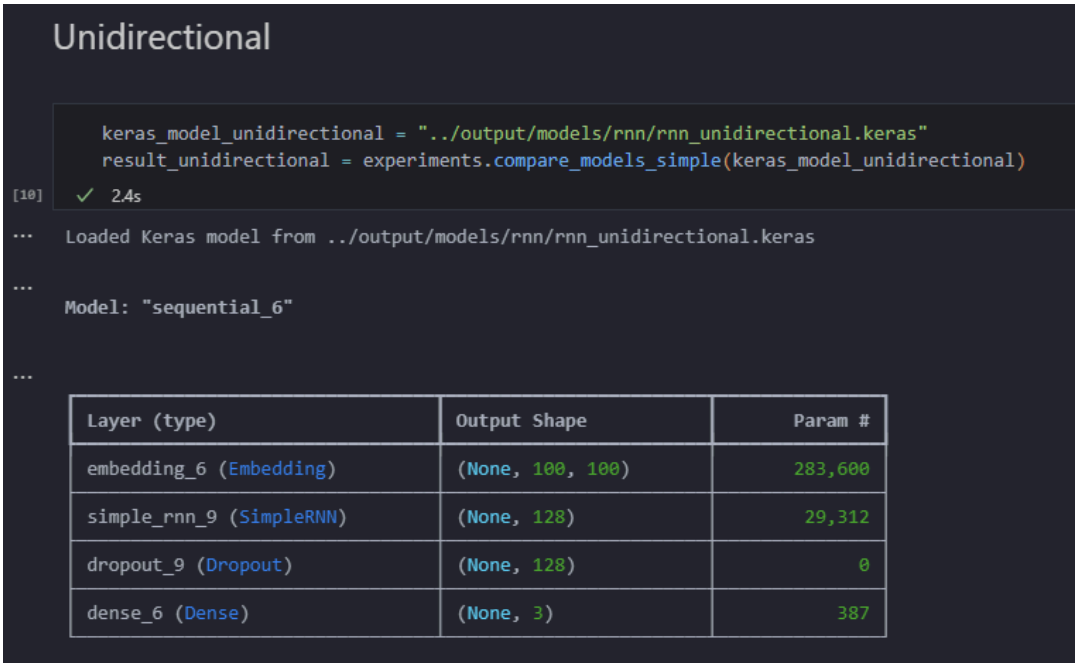
Bidirectional

```
=== Training model with Bidirectional RNN RNN ===
Epoch 1/10
/root/ML/venv/lib/python3.10/site-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
16/16 ━━━━━━━━━━━━━━━━━━━━ 6s 184ms/step - accuracy: 0.4116 - loss: 1.0628 - val_accuracy: 0.4800 - val_loss: 1.0018
Epoch 2/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 42ms/step - accuracy: 0.6479 - loss: 0.8403 - val_accuracy: 0.3400 - val_loss: 1.1178
Epoch 3/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 41ms/step - accuracy: 0.8392 - loss: 0.5516 - val_accuracy: 0.5400 - val_loss: 0.9498
Epoch 4/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 41ms/step - accuracy: 0.9547 - loss: 0.3059 - val_accuracy: 0.5300 - val_loss: 1.0113
Epoch 5/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 42ms/step - accuracy: 0.9820 - loss: 0.1550 - val_accuracy: 0.4700 - val_loss: 1.0848
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 39ms/step - accuracy: 0.9778 - loss: 0.0969 - val_accuracy: 0.5700 - val_loss: 1.0211
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 40ms/step - accuracy: 0.9905 - loss: 0.0524 - val_accuracy: 0.5500 - val_loss: 1.0554
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 45ms/step - accuracy: 0.9994 - loss: 0.0336 - val_accuracy: 0.5000 - val_loss: 1.1587
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 4s 280ms/step - accuracy: 1.0000 - loss: 0.0208 - val_accuracy: 0.5300 - val_loss: 1.2094
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 43ms/step - accuracy: 1.0000 - loss: 0.0125 - val_accuracy: 0.5600 - val_loss: 1.1379
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - accuracy: 0.5507 - loss: 1.3040
13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 36ms/step

Test metrics for rnn_bidirectional:
  Loss: 1.2761
  Accuracy: 0.5350
  Macro F1-score: 0.5351
```



Bidirectional memiliki kualitas yang lebih bagus secara keseluruhan. Hal ini mungkin disebabkan dataset NusaX ini memerlukan konteks dua arah untuk menghasilkan prediksi yang baik. Namun untuk dataset yang lain belum tentu berlaku seperti itu

2.2.2.4. Perbandingan model Keras dan model from scratch

Unidirectional

Secara keseluruhan, backpropagation antara from scratch dan keras menghasilkan prediksi yang sama persis.

Bidirectional

Bidirectional

```
keras_model_bidirectional = "../output/models/rnn/rnn_cells_32.keras"
result_bidirectional = experiments.compare_models_simple(keras_model_bidirectional)
```

[12] ✓ 1.5s

```
Loaded Keras model from ../output/models/rnn/rnn_cells_32.keras

Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
embedding_3 (Embedding)	(None, 100, 100)	283,600
simple_rnn_6 (SimpleRNN)	(None, 32)	4,256
dropout_6 (Dropout)	(None, 32)	0
dense_3 (Dense)	(None, 3)	99

```
print("=== Model Comparison Bidirectional Result ===")
print(f"Keras Accuracy       : {result_bidirectional['keras_metrics']['accuracy']:.4f}")
print(f"Keras Macro F1       : {result_bidirectional['keras_metrics']['macro_f1']:.4f}")
print(f"Scratch Accuracy     : {result_bidirectional['scratch_metrics']['accuracy']:.4f}")
print(f"Scratch Macro F1     : {result_bidirectional['scratch_metrics']['macro_f1']:.4f}")
print(f"Model Agreement      : {result_bidirectional['model_agreement']:.4f}")
```

[15] ✓ 0.0s

```
=== Model Comparison Bidirectional Result ===
Keras Accuracy       : 0.4050
Keras Macro F1       : 0.3666
Scratch Accuracy     : 0.4050
Scratch Macro F1     : 0.3666
Model Agreement      : 1.0000
```

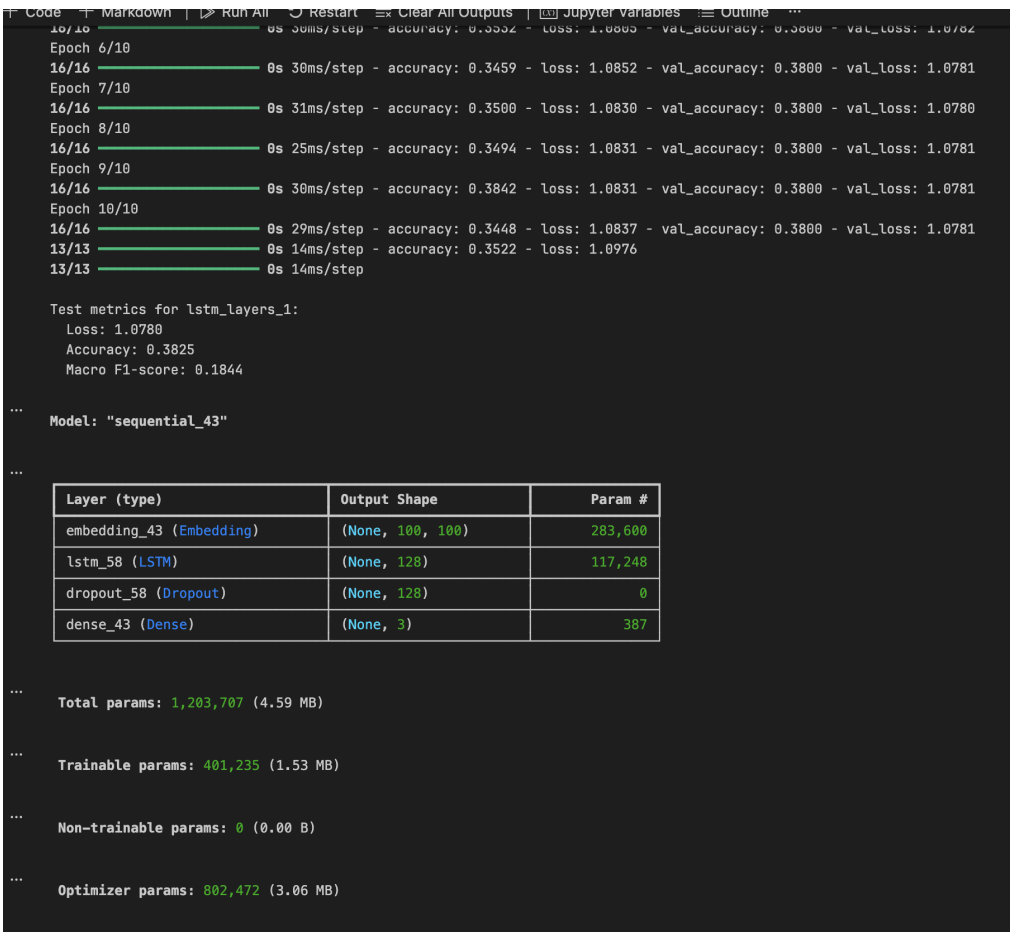
Secara keseluruhan, backpropagation antara from scratch dan keras menghasilkan prediksi yang sama persis.

2.2.3. LSTM

2.2.3.1. Pengaruh jumlah layer LSTM

Jumlah Layer	Hasil Eksekusi

1


Code
Markdown
Run All
Restart
Clear All Outputs
Jupyter Variables
Outline
16/16 0s 30ms/step - accuracy: 0.3532 - loss: 1.0865 - val_accuracy: 0.3800 - val_loss: 1.0782
Epoch 6/10
16/16 0s 30ms/step - accuracy: 0.3459 - loss: 1.0852 - val_accuracy: 0.3800 - val_loss: 1.0781
Epoch 7/10
16/16 0s 31ms/step - accuracy: 0.3500 - loss: 1.0830 - val_accuracy: 0.3800 - val_loss: 1.0780
Epoch 8/10
16/16 0s 25ms/step - accuracy: 0.3494 - loss: 1.0831 - val_accuracy: 0.3800 - val_loss: 1.0781
Epoch 9/10
16/16 0s 30ms/step - accuracy: 0.3842 - loss: 1.0831 - val_accuracy: 0.3800 - val_loss: 1.0781
Epoch 10/10
16/16 0s 29ms/step - accuracy: 0.3448 - loss: 1.0837 - val_accuracy: 0.3800 - val_loss: 1.0781
13/13 0s 14ms/step - accuracy: 0.3522 - loss: 1.0976
13/13 0s 14ms/step
Test metrics for lstm_layers_1:
Loss: 1.0780
Accuracy: 0.3825
Macro F1-score: 0.1844
Model: "sequential_43"
Layer (type) Output Shape Param #
embedding_43 (Embedding) (None, 100, 100) 283,600
lstm_58 (LSTM) (None, 128) 117,248
dropout_58 (Dropout) (None, 128) 0
dense_43 (Dense) (None, 3) 387
Total params: 1,203,707 (4.59 MB)
Trainable params: 401,235 (1.53 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 802,472 (3.06 MB)

2

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.3675 - loss: 1.0814 - val_accuracy: 0.3800 - val_loss: 1.0780
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.3470 - loss: 1.0810 - val_accuracy: 0.3800 - val_loss: 1.0781
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - accuracy: 0.3277 - loss: 1.0815 - val_accuracy: 0.3800 - val_loss: 1.0781
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.3526 - loss: 1.0811 - val_accuracy: 0.3800 - val_loss: 1.0780
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.3463 - loss: 1.0810 - val_accuracy: 0.3800 - val_loss: 1.0780
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.3522 - loss: 1.0975
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step

Test metrics for lstm_layers_2:
  Loss: 1.0779
  Accuracy: 0.3825
  Macro F1-score: 0.1844

Model: "sequential_44"
```

Layer (type)	Output Shape	Param #
embedding_44 (Embedding)	(None, 100, 100)	283,600
lstm_59 (LSTM)	(None, 100, 128)	117,248
dropout_59 (Dropout)	(None, 100, 128)	0
lstm_60 (LSTM)	(None, 64)	49,408
dropout_60 (Dropout)	(None, 64)	0
dense_44 (Dense)	(None, 3)	195

```
Total params: 1,351,355 (5.16 MB)

Trainable params: 450,451 (1.72 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 900,904 (3.44 MB)
```

3

```
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 41ms/step - accuracy: 0.3349 - loss: 1.0861 - val_accuracy: 0.3800 - val_loss: 1.0780
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 38ms/step - accuracy: 0.3829 - loss: 1.0826 - val_accuracy: 0.3800 - val_loss: 1.0780
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 37ms/step - accuracy: 0.3887 - loss: 1.0809 - val_accuracy: 0.3800 - val_loss: 1.0783
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.3522 - loss: 1.0990
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step

Test metrics for lstm_layers_3:
  Loss: 1.0781
  Accuracy: 0.3825
  Macro F1-score: 0.1844
```

Model: "sequential_45"

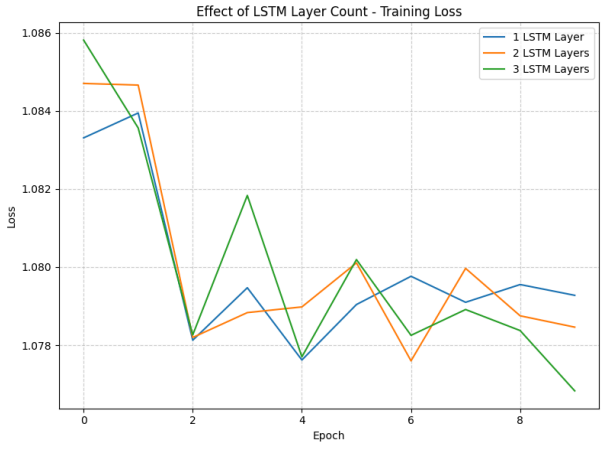
Layer (type)	Output Shape	Param #
embedding_45 (Embedding)	(None, 100, 100)	283,600
lstm_61 (LSTM)	(None, 100, 128)	117,248
dropout_61 (Dropout)	(None, 100, 128)	0
lstm_62 (LSTM)	(None, 100, 64)	49,408
dropout_62 (Dropout)	(None, 100, 64)	0
lstm_63 (LSTM)	(None, 32)	12,416
dropout_63 (Dropout)	(None, 32)	0
dense_45 (Dense)	(None, 3)	99

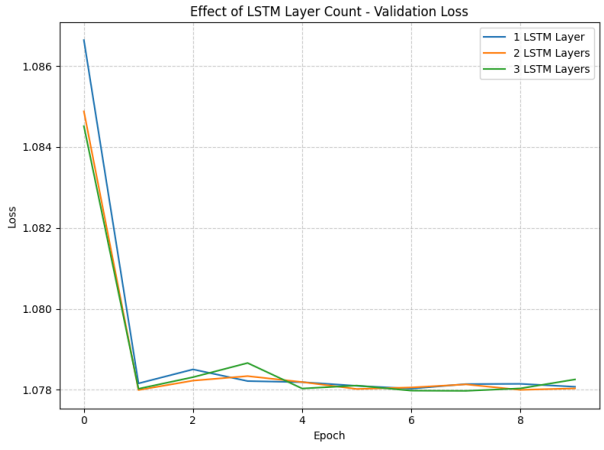
Total params: 1,388,315 (5.30 MB)

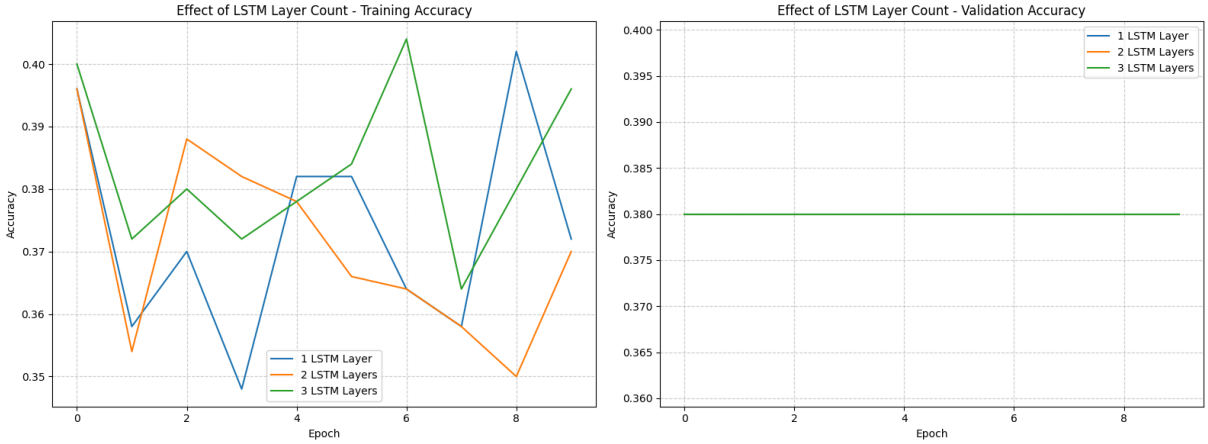
Trainable params: 462,771 (1.77 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 925,544 (3.53 MB)







Kesimpulan	Jumlah layer pada LSTM unidirectional tidak berpengaruh pada analisis sentimen teks, karena model akan selalu memilih salah satu label.

2.2.3.2. Pengaruh banyak cell LSTM per layer

Jumlah Cell	Hasil Eksekusi

1

```
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3747 - loss: 1.0808 - val_accuracy: 0.3800 - val_loss: 1.0781
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3609 - loss: 1.0803 - val_accuracy: 0.3800 - val_loss: 1.0782
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3079 - loss: 1.0822 - val_accuracy: 0.3800 - val_loss: 1.0780
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.3405 - loss: 1.0814 - val_accuracy: 0.3800 - val_loss: 1.0780
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.3875 - loss: 1.0804 - val_accuracy: 0.3800 - val_loss: 1.0783
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.3522 - loss: 1.0990
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step

Test metrics for lstm_cells_64:
  Loss: 1.0782
  Accuracy: 0.3825
  Macro F1-score: 0.1844

Model: "sequential_46"
```

Layer (type)	Output Shape	Param #
embedding_46 (Embedding)	(None, 100, 100)	283,600
lstm_64 (LSTM)	(None, 64)	42,240
dropout_64 (Dropout)	(None, 64)	0
dense_46 (Dense)	(None, 3)	195

```
Total params: 978,107 (3.73 MB)

Trainable params: 326,035 (1.24 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 652,072 (2.49 MB)
```

2

```
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.3474 - loss: 1.0807 - val_accuracy: 0.3800 - val_loss: 1.0780
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - accuracy: 0.3487 - loss: 1.0795 - val_accuracy: 0.3800 - val_loss: 1.0783
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3593 - loss: 1.0825 - val_accuracy: 0.3800 - val_loss: 1.0782
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.3508 - loss: 1.0828 - val_accuracy: 0.3800 - val_loss: 1.0781
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3717 - loss: 1.0818 - val_accuracy: 0.3800 - val_loss: 1.0780
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.3522 - loss: 1.0971
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step

Test metrics for lstm_cells_128:
  Loss: 1.0779
  Accuracy: 0.3825
  Macro F1-score: 0.1844

Model: "sequential_47"
```

Layer (type)	Output Shape	Param #
embedding_47 (Embedding)	(None, 100, 100)	283,600
lstm_65 (LSTM)	(None, 128)	117,248
dropout_65 (Dropout)	(None, 128)	0
dense_47 (Dense)	(None, 3)	387

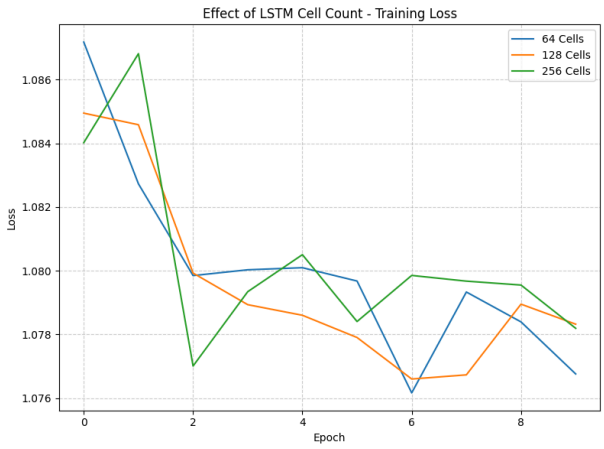
```
Total params: 1,203,707 (4.59 MB)

Trainable params: 401,235 (1.53 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 802,472 (3.06 MB)
```

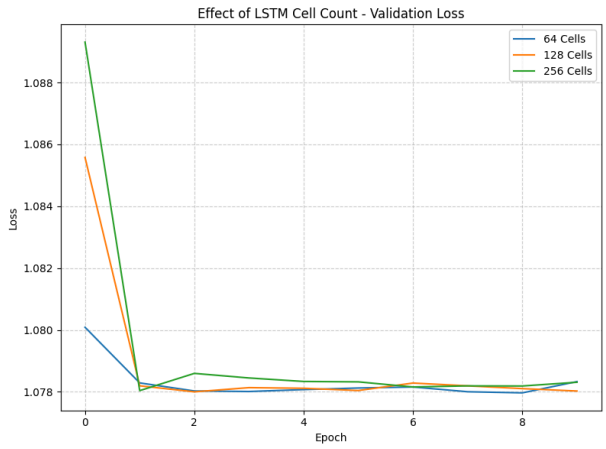
3

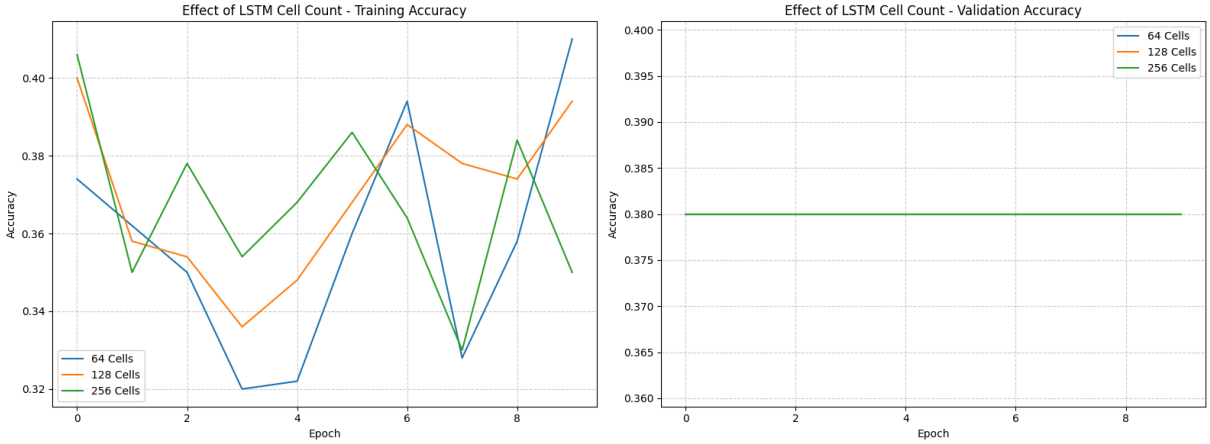
```
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.3398 - loss: 1.0830 - val_accuracy: 0.3800 - val_loss: 1.0783
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3588 - loss: 1.0817 - val_accuracy: 0.3800 - val_loss: 1.0783
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.3437 - loss: 1.0849 - val_accuracy: 0.3800 - val_loss: 1.0782
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.3278 - loss: 1.0833 - val_accuracy: 0.3800 - val_loss: 1.0782
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3632 - loss: 1.0829 - val_accuracy: 0.3800 - val_loss: 1.0782
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.3348 - loss: 1.0813 - val_accuracy: 0.3800 - val_loss: 1.0783
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.3522 - loss: 1.0983
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step

Test metrics for lstm_cells_256:
  Loss: 1.0781
  Accuracy: 0.3825
  Macro F1-score: 0.1844

Model: "sequential_48"
```

Layer (type)	Output Shape	Param #
embedding_48 (Embedding)	(None, 100, 100)	283,600
lstm_66 (LSTM)	(None, 256)	365,568
dropout_66 (Dropout)	(None, 256)	0
dense_48 (Dense)	(None, 3)	771

```
Total params: 1,949,819 (7.44 MB)

Trainable params: 649,939 (2.48 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 1,299,880 (4.96 MB)
```



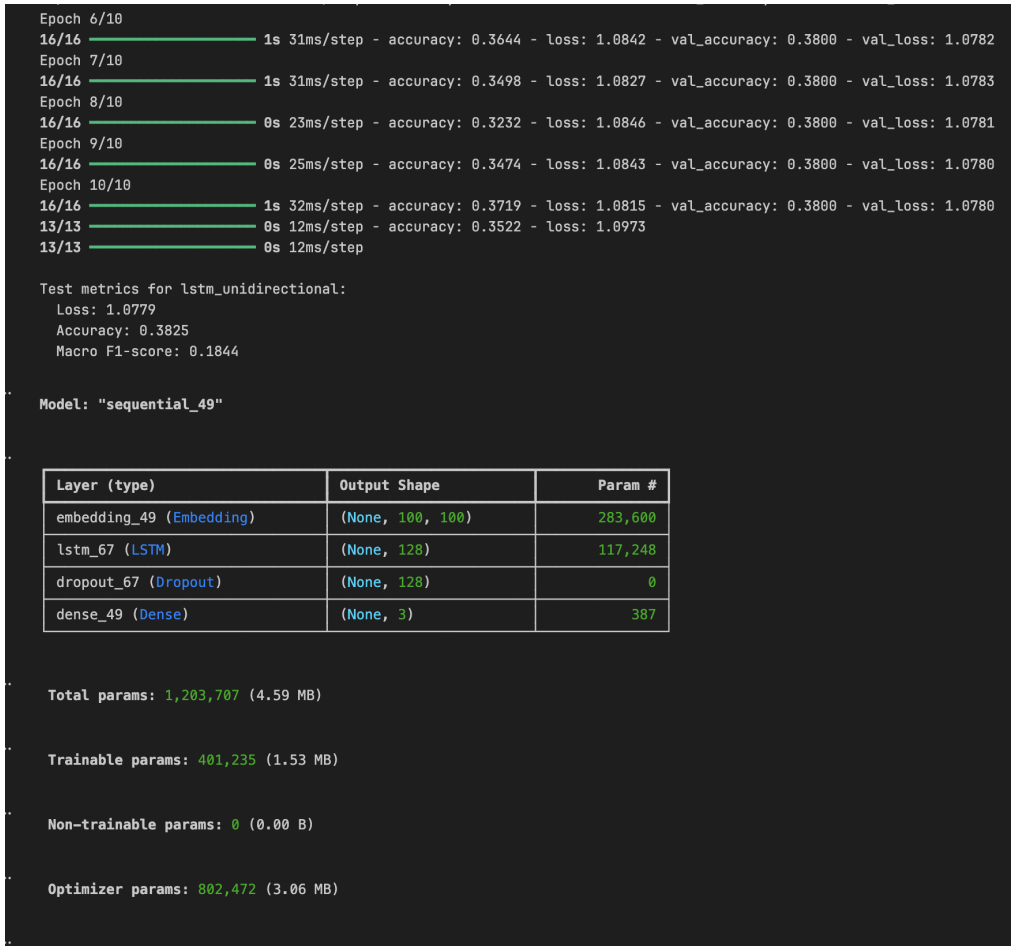




Kesimpulan	Jumlah banyak cell pada LSTM unidirectional tidak berpengaruh pada analisis sentimen teks, karena model akan selalu memilih salah satu label.

2.2.3.3. Pengaruh jenis layer LSTM berdasarkan arah

Arah	Hasil Eksekusi

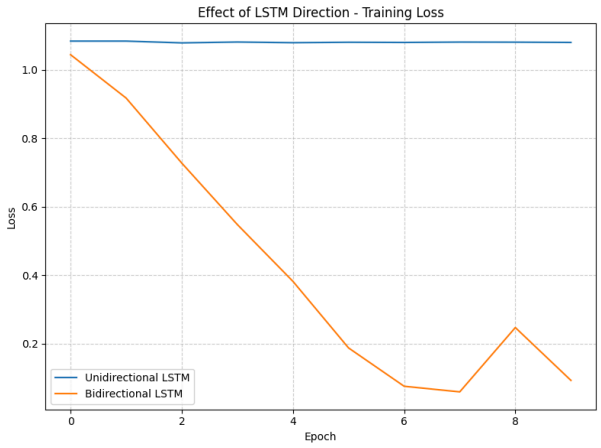
Unidirectional	

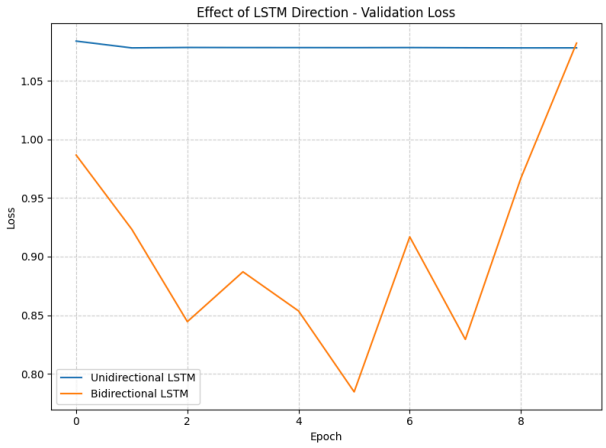
Bidirectional

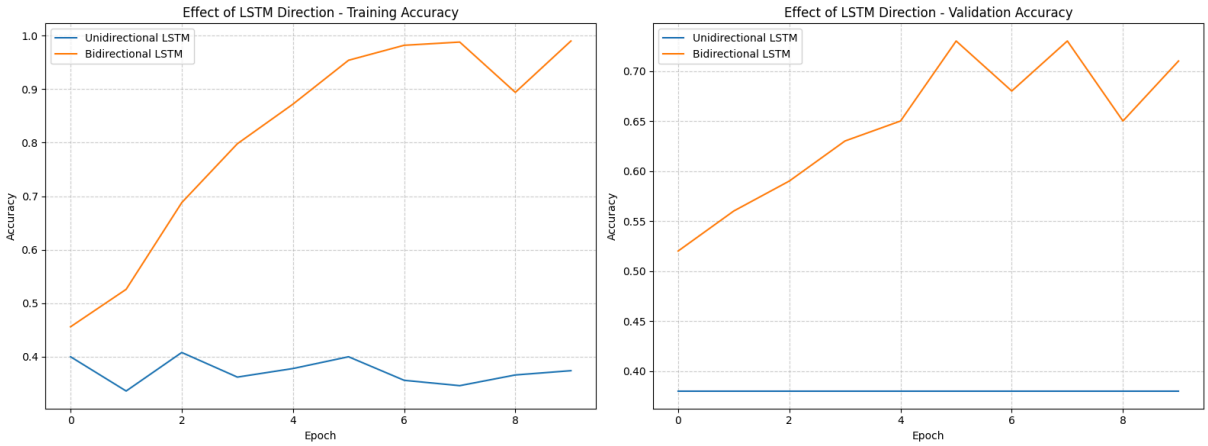
```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 47ms/step - accuracy: 0.7533 - loss: 0.5552 - val_accuracy: 0.6300 - val_loss: 0.8870
Epoch 5/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - accuracy: 0.8432 - loss: 0.3933 - val_accuracy: 0.6500 - val_loss: 0.8537
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - accuracy: 0.9601 - loss: 0.1981 - val_accuracy: 0.7300 - val_loss: 0.7847
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - accuracy: 0.9831 - loss: 0.0778 - val_accuracy: 0.6800 - val_loss: 0.9168
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - accuracy: 0.9935 - loss: 0.0525 - val_accuracy: 0.7300 - val_loss: 0.8294
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - accuracy: 0.9204 - loss: 0.1842 - val_accuracy: 0.6500 - val_loss: 0.9670
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 44ms/step - accuracy: 0.9908 - loss: 0.1015 - val_accuracy: 0.7100 - val_loss: 1.0821
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.6767 - loss: 1.1851
13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step

Test metrics for lstm_bidirectional:
  Loss: 1.0497
  Accuracy: 0.6975
  Macro F1-score: 0.6714

Model: "sequential_50"
```

Layer (type)	Output Shape	Param #
embedding_50 (Embedding)	(None, 100, 100)	283,600
bidirectional_5 (Bidirectional)	(None, 256)	234,496
dropout_68 (Dropout)	(None, 256)	0
dense_50 (Dense)	(None, 3)	771

```
Total params: 1,556,603 (5.94 MB)

Trainable params: 518,867 (1.98 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 1,037,736 (3.96 MB)
```







Kesimpulan	Variasi LSTM unidirectional dan bidirectional sangat mempengaruhi kinerja model dalam analisis sentimen teks. Pada LSTM unidirectional, model akan selalu memprediksi satu kelas saja sehingga akurasinya hanya sebatas persentase satu kelas tersebut. Sementara pada LSTM bidirectional, model bekerja dengan baik hingga mencapai akurasi sekitar 70%.

2.2.3.4. Perbandingan model Keras dan model from scratch

```
# Get test data
x_test, y_test = data_loader.get_vectorized_data('test')

# Compare predictions
comparison = compare_keras_vs_scratch(keras_model, scratch_model, x_test, y_test, batch_size=32)

print("\nKeras Model Metrics:")
print(f"Accuracy: {comparison['keras_metrics']['accuracy']:.4f}")
print(f"Macro F1-Score: {comparison['keras_metrics']['macro_f1']:.4f}")

print("\nFrom-Scratch Model Metrics:")
print(f"Accuracy: {comparison['scratch_metrics']['accuracy']:.4f}")
print(f"Macro F1-Score: {comparison['scratch_metrics']['macro_f1']:.4f}")

print(f"\nModel Agreement: {comparison['model_agreement']:.4f}")
```

✓ 1.0s Python

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step

Keras Model Metrics:
Accuracy: 0.6975
Macro F1-Score: 0.6714

From-Scratch Model Metrics:
Accuracy: 0.7025
Macro F1-Score: 0.6724

Model Agreement: 0.9850
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step

Text: dasar tak [UNK] diri kau belajar [UNK] bukan untuk di [UNK] tapi untuk melindungi diri sendiri [UNK]...
True sentiment: Negative
Keras prediction: Negative ✓
Scratch prediction: Negative ✓

Text: hai [UNK] untuk [UNK] tiket [UNK] jadwal diharuskan [UNK] pembayaran dulu ya
True sentiment: Neutral
Keras prediction: Neutral ✓
Scratch prediction: Neutral ✓

Text: restoran dengan [UNK] penyajian berbeda piring tempat makan diganti dengan [UNK] daun pisang menu ma...
True sentiment: Positive
Keras prediction: Positive ✓
Scratch prediction: Positive ✓

Text: kafe ini menyediakan makanan tradisional [UNK] kafe anak muda selain tempat yang nyaman harga terbil...
True sentiment: Positive
Keras prediction: Positive ✓
Scratch prediction: Positive ✓

Text: ini salah satu [UNK] generasi yang tidak berguna bukan malah [UNK] untuk [UNK] tapi malah jadi [UNK]...
True sentiment: Negative
Keras prediction: Negative ✓
Scratch prediction: Negative ✓
```

BAB III

KESIMPULAN DAN SARAN

3.1. Kesimpulan

Berdasarkan hasil pengujian yang telah dilakukan, implementasi Convolutional Neural Network (CNN) menunjukkan bahwa penambahan jumlah layer konvolusi dan filter per layer secara konsisten meningkatkan performa F1-score, meskipun dengan trade-off peningkatan jumlah parameter yang signifikan. Dari segi hyperparameter, ukuran filter tidak menunjukkan pola yang konsisten terhadap performa, namun average pooling terbukti lebih efektif dibandingkan max pooling pada dataset CIFAR-10. Validasi implementasi menunjukkan bahwa model from scratch berhasil menghasilkan performa yang identik dengan model Keras, membuktikan kebenaran implementasi forward propagation manual.

Pada implementasi Recurrent Neural Network (RNN), hasil eksperimen menunjukkan bahwa model dengan 3 layer RNN memberikan performa terbaik dengan stabilitas training yang baik, sementara model dengan 1 layer cenderung mengalami overfitting. Perbandingan antara bidirectional dan unidirectional RNN menunjukkan bahwa bidirectional RNN secara konsisten mengungguli unidirectional RNN pada task analisis sentimen, menunjukkan pentingnya konteks dua arah dalam pemahaman teks. Forward propagation from scratch menghasilkan prediksi yang identik dengan Keras untuk kedua variant, memvalidasi kebenaran implementasi.

Hasil pengujian Long Short-Term Memory (LSTM) mengungkapkan temuan yang menarik dimana LSTM unidirectional mengalami masalah serius dengan hanya memprediksi satu kelas, mengindikasikan kegagalan dalam pembelajaran pola yang beragam. Sebaliknya, LSTM bidirectional berhasil mencapai akurasi sekitar 70% dan mampu memprediksi multiple kelas dengan baik. Implementasi manual LSTM juga berhasil divalidasi dengan model Keras, menunjukkan konsistensi dalam forward propagation.

3.2. Saran

Untuk pengembangan selanjutnya, disarankan melakukan optimasi hyperparameter yang lebih sistematis melalui grid search atau random search untuk menemukan konfigurasi optimal bagi setiap model. Implementasi teknik regularisasi seperti dropout dan gradient clipping perlu diterapkan untuk meningkatkan stabilitas training dan mencegah overfitting. Investigasi lebih lanjut terhadap masalah LSTM unidirectional yang gagal mempelajari multiple kelas juga diperlukan, kemungkinan melalui perbaikan inisialisasi bobot atau strategi pembelajaran yang berbeda.

BAB IV

REFERENSI

- https://d2l.ai/chapter_recurrent-modern/lstm.html
- https://d2l.ai/chapter_recurrent-modern/deep-rnn.html
- https://d2l.ai/chapter_recurrent-modern/bi-rnn.html
- https://d2l.ai/chapter_recurrent-neural-networks/index.html
- https://d2l.ai/chapter_convolutional-neural-networks/index.html
  https://numpy.org/doc/2.1/reference/generated/numpy.einsum.html

LAMPIRAN

NIM	Pembagian Tugas
13522003	Implementasi RNN from scratch, testing parameter RNN, laporan bagian RNN, Deskripsi Kelas RNN, Penjelasan Forward Propagation RNN.
13522005	Implementasi CNN from scratch, testing parameter CNN, laporan bagian deskripsi persoalan dan hasil pengujian CNN
13522011	Implementasi LSTM from scratch, testing parameter LSTM, laporan bagian LSTM, Deskripsi Kelas LSTM, Penjelasan Forward Propagation LSTM.