

IF3270 Pembelajaran Mesin

Convolutional Neural Network dan Recurrent Neural Network

Laporan Tugas Besar 2

Disusun untuk memenuhi tugas mata kuliah Pembelajaran Mesin untuk
Semester 6 tahun ajaran 2024 / 2025



Oleh :

Aland Mulia Pratama	13522124
Hafizh Hananta Akbari	13522132
Christian Justin Hendrawan	13522135

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

DAFTAR ISI

DAFTAR ISI	1
BAB I	
DESKRIPSI PERSOALAN	3
1.1. Convolutional Neural Network	3
1.1.1. Pelatihan dan Analisis Model CNN dengan Keras	3
1.1.2. Implementasi Forward Propagation From Scratch	4
1.2. Simple Recurrent Neural Network	5
1.2.1. Pra-pemrosesan Data Teks	5
1.2.2. Pelatihan dan Analisis Model RNN dengan Keras	5
1.2.3. Implementasi Forward Propagation From Scratch	6
1.3. Long-Short Term Memory Network	7
1.3.1. Pra-pemrosesan Data Teks	7
1.3.2. Pelatihan dan Analisis Model LSTM dengan Keras	7
1.3.3. Implementasi Forward Propagation From Scratch	8
BAB II	
Penjelasan implementasi	9
2.1. Struktur Direktori dan File Proyek	9
2.2. Implementasi From Scratch: Deskripsi Kelas dan Forward Propagation	10
2.2.1. CNN	10
2.2.1.1. Konfigurasi (cnn/config.py)	10
2.2.1.2. Model (cnn/model.py)	11
2.2.1.3. Pelatihan (cnn/train.py)	15
2.2.1.4. Implementasi Forward From Scratch (cnn/from_scratch.py)	23
2.2.2. Simple Recurrent Neural Network (Simple RNN)	48
2.2.2.1. Konfigurasi (simplernn/config.py)	48
2.2.2.2. Pra-pemrosesan Data (simplernn/data_preprocessing.py)	50
2.2.2.3. Model (simplernn/model.py)	53
2.2.2.4. Pelatihan (simplernn/train.py)	56
2.2.2.5. Implementasi Forward Propagation From Scratch (simplernn/from_scratch.py)	63
2.2.3. Long-Short Term Memory Network (LSTM)	74
2.2.3.1. Konfigurasi (lstm/config.py)	74
2.2.3.2. Pra-pemrosesan Data (lstm/data_preprocessing.py)	77
2.2.3.3. Model (lstm/model.py)	79
2.2.3.4. Pelatihan (lstm/train.py)	82
2.2.3.5. Implementasi Forward Propagation From Scratch (lstm/from_scratch_lstm.py)	91
BAB III	
Hasil Pengujian	104
3.1. Convolutional Neural Network (CNN)	104

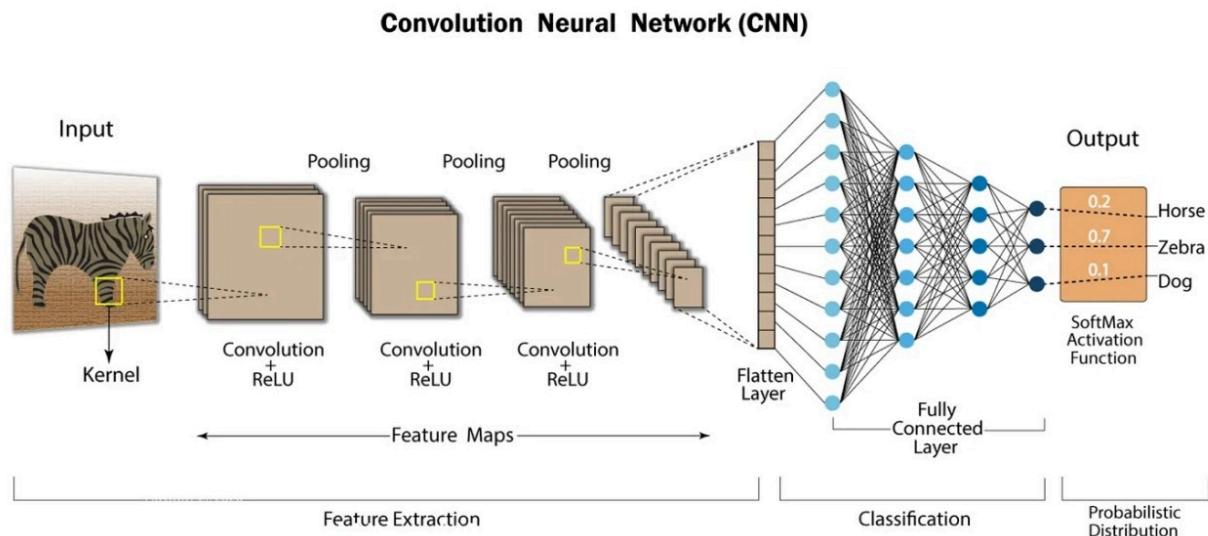
3.1.1. Pengaruh jumlah layer konvolusi	104
3.1.2. Pengaruh banyak filter per layer konvolusi	107
3.1.3. Pengaruh ukuran filter per layer konvolusi	110
3.1.4. Pengaruh jenis pooling layer	113
3.2. Simple Recurrent Neural Network (Simple RNN)	116
3.2.1. Pengaruh Jumlah Layer RNN	116
3.2.2. Pengaruh Banyak Cell RNN per Layer	120
3.2.3. Pengaruh Jenis Layer RNN Berdasarkan Arah	123
3.3. Long-Short Term Memory Network (LSTM)	126
3.3.1. Pengaruh Jumlah Layer LSTM	126
3.3.2. Pengaruh Banyak Cell LSTM per Layer	129
3.3.3. Pengaruh Jenis Layer LSTM Berdasarkan Arah	132
BAB IV	136
Kesimpulan dan Saran	136
4.1. Kesimpulan	136
4.2. Saran	137
BAB V	138
Referensi	138
Lampiran	139

BAB I

DESKRIPSI PERSOALAN

Tugas Besar II mata kuliah IF3270 Pembelajaran Mesin ini bertujuan untuk memberikan pemahaman praktis mengenai implementasi Convolutional Neural Network (CNN) dan Recurrent Neural Network (RNN). Mahasiswa ditugaskan untuk mengimplementasikan fungsi forward propagation secara manual (from scratch) untuk arsitektur CNN, Simple RNN, dan Long Short-Term Memory (LSTM).

1.1. Convolutional Neural Network



Gambar 1.1.1 Arsitektur Convolution Neural Network

1.1.1. Pelatihan dan Analisis Model CNN dengan Keras

Mahasiswa diminta untuk melatih sebuah model CNN pada dataset CIFAR-10 dengan spesifikasi sebagai berikut:

- **Arsitektur Model:** Model harus mengandung *layer Conv2D*, *Pooling layers* (Max atau Average), *Flatten/Global Pooling layer*, dan *Dense layer*. Urutan dan jumlah spesifik dari *layer-layer* ini dapat disesuaikan.
- **Fungsi Kerugian (Loss Function):** Menggunakan *Sparse Categorical Crossentropy*, yang sesuai untuk klasifikasi multikelas dengan label integer.
- **Optimizer:** Menggunakan optimizer Adam.
- **Pembagian Dataset:** Dataset CIFAR-10 yang awalnya terdiri dari data latih dan data uji, perlu dibagi ulang menjadi tiga bagian: data latih baru (40.000 sampel), data validasi (10.000 sampel, diambil dari data latih awal), dan data uji (10.000 sampel), dengan rasio pembagian data latih awal menjadi data latih baru dan data validasi sebesar 4:1.

Selanjutnya, akan dilakukan serangkaian eksperimen untuk menganalisis pengaruh beberapa *hyperparameter* kunci dalam CNN:

- **Pengaruh Jumlah Layer Konvolusi:** Melakukan pelatihan dengan 3 variasi jumlah *layer* konvolusi.
- **Pengaruh Banyak Filter per Layer Konvolusi:** Melakukan pelatihan dengan 3 variasi kombinasi jumlah *filter* pada *layer-layer* konvolusi.
- **Pengaruh Ukuran Filter per Layer Konvolusi:** Melakukan pelatihan dengan 3 variasi kombinasi ukuran *filter* pada *layer-layer* konvolusi.
- **Pengaruh Jenis Pooling Layer:** Melakukan pelatihan dengan 2 variasi jenis *pooling layer* (Max Pooling dan Average Pooling).

Untuk setiap variasi *hyperparameter* di atas, evaluasi mencakup:

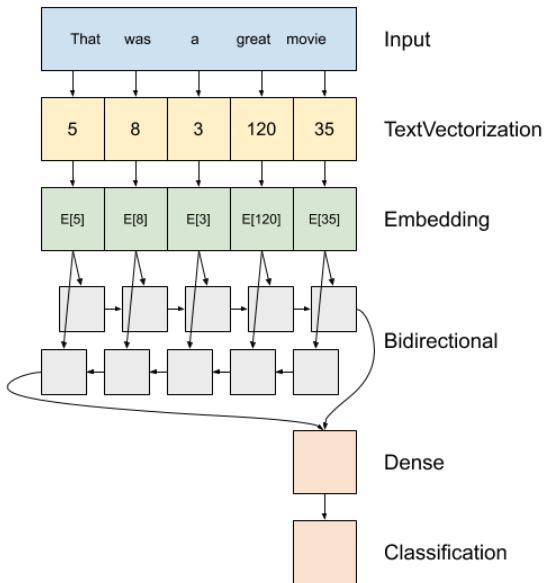
- Perbandingan hasil akhir prediksi menggunakan metrik *macro f1-score*.
- Perbandingan grafik *training loss* dan *validation loss* untuk setiap epoch.
- Penyusunan kesimpulan mengenai bagaimana variasi *hyperparameter* tersebut mempengaruhi kinerja model.
- Bobot dari model hasil pelatihan terbaik (atau salah satu model representatif) harus disimpan.

1.1.2. Implementasi *Forward Propagation From Scratch*

Berdasarkan model CNN yang telah dilatih menggunakan Keras (dan bobot yang telah disimpan), mahasiswa diminta untuk membuat modul *forward propagation from scratch*. Ketentuan implementasi adalah sebagai berikut:

- Modul harus dapat membaca dan menggunakan bobot dari model yang telah dilatih dengan Keras.
- Implementasi *forward propagation* direkomendasikan bersifat modular, artinya terdapat fungsi atau metode *forward propagation* terpisah untuk setiap jenis *layer* (misalnya, *forward_conv2d*, *forward_pooling*, *forward_dense*, dll.).
- Khusus untuk *Dense layer*, diperbolehkan menggunakan kembali implementasi *forward propagation* dari modul *Feedforward Neural Network* (FFNN) yang telah dibuat pada Tugas Besar I.
- Pengujian implementasi *forward propagation from scratch* dilakukan dengan membandingkan hasilnya terhadap *output* dari fungsi *predict* Keras pada data uji. Metrik yang digunakan untuk perbandingan adalah *macro f1-score*.

1.2. Simple Recurrent Neural Network



Gambar 1.2.1 Arsitektur Simple Recurrent Neural Network

Bagian tugas ini difokuskan pada pemahaman dan implementasi *Recurrent Neural Network* (RNN) untuk tugas klasifikasi teks. Prosesnya melibatkan beberapa tahapan utama: pra-pemrosesan data teks, pelatihan model RNN menggunakan *library* Keras dengan dataset NusaX-Sentiment (Bahasa Indonesia) termasuk analisis pengaruh *hyperparameter*, dan implementasi fungsi *forward propagation* RNN secara manual (*from scratch*).

1.2.1. Pra-pemrosesan Data Teks

Sebelum data dapat diproses oleh model RNN, diperlukan konversi teks menjadi representasi numerik melalui langkah-langkah berikut:

- **Tokenization:** Tahap ini mengubah data teks mentah menjadi urutan (*sequence*) token integer. Setiap token merepresentasikan satu unit kata atau sub-kata. Untuk keperluan tugas ini, proses tokenisasi akan memanfaatkan *layer TextVectorization* dari Keras.
- **Embedding Function:** Setelah tokenisasi, setiap token integer akan dipetakan ke dalam ruang vektor berdimensi-n menggunakan fungsi *embedding*. Representasi vektor ini memungkinkan operasi matematis antar token. Implementasi fungsi *embedding* akan menggunakan *Embedding layer* yang disediakan oleh Keras.

1.2.2. Pelatihan dan Analisis Model RNN dengan Keras

Mahasiswa diminta untuk melatih model RNN untuk klasifikasi sentimen pada dataset teks berbahasa Indonesia, NusaX-Sentiment, dengan ketentuan:

- **Arsitektur Model:** Model harus mencakup jenis *layer* berikut: *Embedding layer*, *Bidirectional RNN layer* dan/atau *Unidirectional RNN layer*, *Dropout layer*, dan *Dense layer*. Urutan serta jumlah *layer* dapat disesuaikan oleh mahasiswa.
- **Fungsi Kerugian (Loss Function):** Menggunakan *Sparse Categorical Crossentropy*.
- **Optimizer:** Menggunakan optimizer Adam.
- **Pembagian Dataset:** Dataset NusaX-Sentiment perlu dibagi menjadi data latih, data validasi, dan data uji untuk evaluasi model yang komprehensif (rasio pembagian disesuaikan dengan praktik umum atau jika ada ketentuan pada dataset).

Selanjutnya, akan dilakukan serangkaian eksperimen untuk menganalisis pengaruh *hyperparameter* pada kinerja model RNN:

- **Pengaruh Jumlah Layer RNN:** Melakukan pelatihan dengan 3 variasi jumlah *layer* RNN.
- **Pengaruh Banyak Cell RNN per Layer:** Melakukan pelatihan dengan 3 variasi kombinasi jumlah *cell* RNN pada setiap *layer* RNN.
- **Pengaruh Jenis Layer RNN berdasarkan Arah Pemrosesan:** Melakukan pelatihan dengan 2 variasi jenis *layer* RNN (menggunakan *unidirectional RNN* atau *bidirectional RNN*).

Untuk setiap variasi *hyperparameter* di atas, evaluasi meliputi:

- Perbandingan hasil prediksi akhir dengan metrik *macro f1-score*.
- Perbandingan grafik *training loss* dan *validation loss* per epoch.
- Penyusunan kesimpulan mengenai dampak perubahan *hyperparameter* tersebut terhadap performa model.
- Bobot (*weights*) dari model hasil pelatihan dengan Keras harus disimpan.

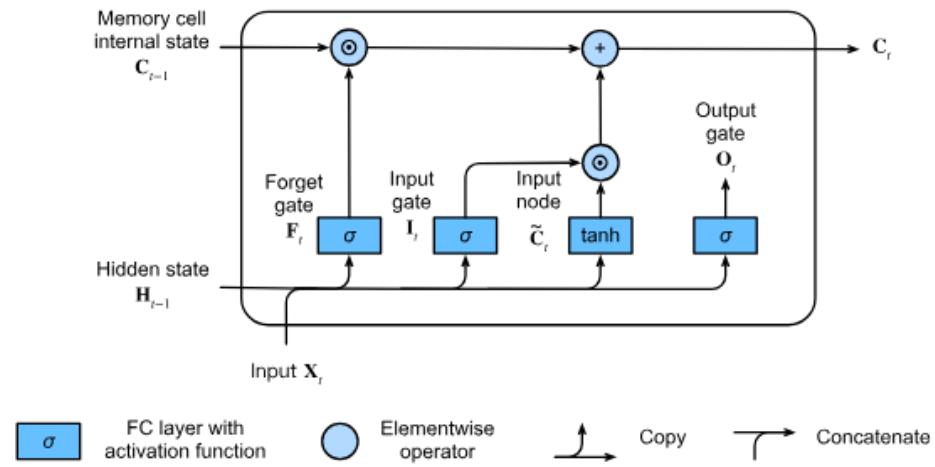
1.2.3. Implementasi *Forward Propagation From Scratch*

Tahap terakhir adalah membuat modul *forward propagation from scratch* untuk model RNN yang telah dilatih dan disimpan bobotnya. Ketentuan implementasinya adalah:

- Modul harus mampu memuat dan menggunakan bobot (*weights*) dari model yang sebelumnya telah dilatih dengan Keras.
- Sangat direkomendasikan untuk mengimplementasikan *forward propagation* secara modular, dengan membuat metode atau fungsi *forward propagation* untuk setiap jenis *layer* yang digunakan (misalnya, *forward_embedding*, *forward_rnn_cell*, *forward_dropout*, *forward_dense*).
- Implementasi *forward propagation from scratch* akan diuji dengan membandingkan hasilnya terhadap *output* dari fungsi *predict* pada model Keras, menggunakan data uji. Metrik yang digunakan untuk validasi adalah *macro f1-score*.

- Catatan: Untuk *Dense layer*, mahasiswa diperbolehkan menggunakan kembali implementasi *forward propagation* dari modul *Feedforward Neural Network* (FFNN) yang telah dikembangkan pada Tugas Besar I.

1.3. Long-Short Term Memory Network



Gambar 1.3.1 Arsitektur Long-Short Term Memory Network

Bagian tugas ini berfokus pada implementasi dan pemahaman *Long Short-Term Memory* (LSTM), salah satu varian dari RNN, untuk tugas klasifikasi teks. Prosesnya mencakup pra-pemrosesan data teks, pelatihan model LSTM menggunakan *library* Keras dengan dataset NusaX-Sentiment (Bahasa Indonesia), analisis pengaruh *hyperparameter*, dan implementasi fungsi *forward propagation* LSTM secara manual (*from scratch*).

1.3.1. Pra-pemrosesan Data Teks

Langkah awal adalah mengubah data teks menjadi format numerik yang dapat diproses oleh model LSTM. Tahapan ini identik dengan pra-pemrosesan untuk RNN biasa dan meliputi:

- **Tokenization:** Menggunakan *layer TextVectorization* dari Keras untuk mengubah teks menjadi urutan (*sequence*) token integer.
- **Embedding Function:** Menggunakan *Embedding layer* dari Keras untuk memetakan setiap token integer ke dalam representasi vektor berdimensi-n.

1.3.2. Pelatihan dan Analisis Model LSTM dengan Keras

Mahasiswa diminta untuk melatih model LSTM untuk klasifikasi sentimen pada dataset NusaX-Sentiment dengan ketentuan sebagai berikut:

- **Arsitektur Model:** Model harus menyertakan *layer* berikut: *Embedding layer*, *Bidirectional LSTM layer* dan/atau *Unidirectional LSTM layer*, *Dropout layer*, dan *Dense layer*. Urutan dan jumlah spesifik dari *layer-layer* ini dapat disesuaikan.
- **Fungsi Kerugian (Loss Function):** Menggunakan *Sparse Categorical Crossentropy*.

- **Optimizer:** Menggunakan optimizer Adam.
- **Pembagian Dataset:** Dataset NusaX-Sentiment perlu dibagi menjadi data latih, data validasi, dan data uji untuk evaluasi model yang komprehensif.

Serangkaian eksperimen akan dilakukan untuk menganalisis pengaruh *hyperparameter* tertentu terhadap kinerja model LSTM:

- **Pengaruh Jumlah Layer LSTM:** Melakukan pelatihan dengan 3 variasi jumlah *layer* LSTM.
- **Pengaruh Banyak Cell LSTM per Layer:** Melakukan pelatihan dengan 3 variasi kombinasi jumlah *cell* LSTM pada setiap *layer* LSTM.
- **Pengaruh Jenis Layer LSTM berdasarkan Arah Pemrosesan:** Melakukan pelatihan dengan 2 variasi jenis *layer* LSTM (menggunakan *unidirectional* LSTM atau *bidirectional* LSTM).

Untuk setiap variasi *hyperparameter* di atas, evaluasi akan mencakup:

- Perbandingan hasil prediksi akhir menggunakan metrik *macro f1-score*.
- Perbandingan grafik *training loss* dan *validation loss* per epoch.
- Penyusunan kesimpulan mengenai dampak perubahan *hyperparameter* tersebut terhadap performa model.
- Bobot (*weights*) dari model hasil pelatihan dengan Keras harus disimpan.

1.3.3. Implementasi *Forward Propagation From Scratch*

Tahap berikutnya adalah membuat modul *forward propagation from scratch* untuk model LSTM yang telah dilatih dan bobotnya disimpan. Ketentuan implementasi adalah:

- Modul harus dapat memuat dan menggunakan bobot (*weights*) dari model yang sebelumnya telah dilatih dengan Keras.
- Implementasi *forward propagation* sangat disarankan bersifat modular, dengan membuat metode atau fungsi *forward propagation* untuk setiap jenis *layer* yang digunakan (misalnya, *forward_embedding*, *forward_lstm_cell*, *forward_dropout*, *forward_dense*).
- Implementasi *forward propagation from scratch* akan diuji dengan membandingkan hasilnya terhadap *output* dari fungsi *predict* pada model Keras, menggunakan data uji. Metrik yang digunakan untuk validasi adalah *macro f1-score*.
- Catatan: Untuk *Dense layer*, mahasiswa diperbolehkan menggunakan kembali implementasi *forward propagation* dari modul *Feedforward Neural Network* (FFNN) yang telah dikembangkan pada Tugas Besar I.

BAB II

Penjelasan implementasi

2.1. Struktur Direktori dan File Proyek

Untuk menjaga kerapian, modularitas, dan kemudahan navigasi dalam pengembangan ketiga arsitektur neural network (CNN, Simple RNN, dan LSTM) pada tugas besar ini, proyek diorganisir ke dalam struktur direktori dan file yang sistematis. Pengorganisasian ini memisahkan komponen-komponen utama seperti kode sumber, data, konfigurasi, notebook eksperimen, dan implementasi from scratch untuk setiap arsitektur. Berikut adalah visualisasi dan penjelasan mengenai struktur direktori proyek yang digunakan.

```
src
  ├── cnn
  │   ├── config.py
  │   ├── data_preprocessing.py
  │   ├── from_scratch.py
  │   ├── main_cnn.ipynb
  │   ├── model.py
  │   └── train.py
  ├── lstm
  │   ├── config.py
  │   ├── data
  │   │   ├── test.csv
  │   │   ├── train.csv
  │   │   └── valid.csv
  │   ├── data_preprocessing.py
  │   ├── from_scratch_lstm.py
  │   ├── main_lstm.ipynb
  │   ├── model.py
  │   └── train.py
  └── simplernn
      ├── config.py
      ├── data
      │   ├── test.csv
      │   ├── train.csv
      │   └── valid.csv
      ├── data_preprocessing.py
      ├── from_scratch.py
      ├── main_rnn.ipynb
      ├── model.py
      └── train.py
```

2.2. Implementasi From Scratch: Deskripsi Kelas dan Forward Propagation

Bagian ini bertujuan untuk memberikan pemahaman komprehensif mengenai seluruh aspek implementasi yang dilakukan untuk ketiga arsitektur jaringan saraf tiruan: Convolutional Neural Network (CNN), Simple Recurrent Neural Network (Simple RNN), dan Long Short-Term Memory (LSTM). Untuk setiap arsitektur, pembahasan akan diorganisir ke dalam beberapa sub-bagian utama yang mencerminkan alur kerja pengembangan model, mulai dari persiapan data hingga pembuatan implementasi forward propagation from scratch.

2.2.1. CNN

2.2.1.1. Konfigurasi (cnn/config.py)

Pengembangan model Convolutional Neural Network (CNN) untuk tugas klasifikasi gambar pada dataset CIFAR-10 melibatkan penetapan parameter-parameter kunci yang mengontrol aspek optimisasi dan strategi penghentian pelatihan. Parameter-parameter ini disimpan secara terpusat dalam sebuah file config.py untuk memudahkan pengelolaan dan produktivitas.

Berikut adalah rincian parameter yang didefinisikan beserta kegunaannya:

1. Parameter Optimisasi (terkait Penjadwalan Learning Rate):

- LR_FACTOR: Faktor pengurangan learning rate. Ketika performa model pada data validasi tidak menunjukkan perbaikan setelah sejumlah epoch tertentu (ditentukan oleh LR_PATIENCE), learning rate saat ini akan dikalikan dengan faktor ini (misalnya, jika LR_FACTOR = 0.5, learning rate akan dikurangi setengahnya).
- LR_PATIENCE: Jumlah epoch kesabaran untuk penjadwalan learning rate. Ini adalah jumlah epoch di mana model akan menunggu perbaikan pada metrik validasi sebelum learning rate dikurangi. Jika tidak ada perbaikan setelah epoch sebanyak LR_PATIENCE, learning rate akan disesuaikan.
- MIN_LR: Batas minimum learning rate. Penjadwal learning rate tidak akan mengurangi learning rate lebih rendah dari nilai ini, memastikan bahwa proses optimisasi tidak terhenti karena learning rate yang terlalu kecil.

2. Parameter Early Stopping (untuk menghentikan pelatihan jika tidak ada perbaikan):

- ES_PATIENCE: Jumlah epoch kesabaran untuk early stopping. Ini adalah jumlah epoch di mana model akan menunggu perbaikan pada metrik validasi yang dipantau (misalnya, val_loss atau val_accuracy). Jika tidak ada perbaikan yang signifikan pada metrik tersebut setelah epoch sebanyak ES_PATIENCE, pelatihan akan dihentikan lebih awal untuk mencegah overfitting dan menghemat waktu komputasi.

```
config.py
```

```
ES_PATIENCE = 5          # Early stopping patience
LR_FACTOR = 0.5          # Learning rate reduction factor
LR_PATIENCE = 3          # Learning rate reduction patience
MIN_LR = 1e-7            # Minimum learning rate
```

2.2.1.2. Model (cnn/model.py)

Definisi arsitektur model Convolutional Neural Network (CNN) diimplementasikan dalam sebuah skrip Python, misalnya model_cnn.py. Skrip ini berisi fungsi create_cnn_model yang bertanggung jawab untuk membangun dan mengompilasi model CNN menggunakan pustaka Keras dari TensorFlow. Fungsi ini dirancang untuk fleksibel, memungkinkan pengguna menyesuaikan berbagai hyperparameter kunci seperti jumlah layer konvolusi, jumlah filter, ukuran kernel, dan jenis pooling layer, sehingga memfasilitasi eksperimentasi dengan berbagai konfigurasi arsitektur.

Berikut adalah tahapan dan komponen utama dalam pembentukan model melalui fungsi create_cnn_model:

1. Inisialisasi Model Sequential

Model dibangun sebagai tumpukan layer Keras menggunakan models.Sequential(). Pendekatan sekuensial ini cocok untuk model di mana data mengalir secara linear dari satu layer ke layer berikutnya.

2. Layer Konvolusi (Convolutional Layers)

Inti dari model CNN adalah serangkaian layer konvolusi dan pooling yang ditambahkan secara dinamis berdasarkan parameter yang diberikan:

- Loop Pembangunan Layer: Fungsi ini melakukan iterasi sebanyak conv_layers untuk menambahkan blok konvolusi-pooling.
- Layer Conv2D:
 - Untuk layer konvolusi pertama ($i == 0$), input_shape=(32, 32, 3) secara eksplisit didefinisikan, yang mengindikasikan bahwa model ini dirancang untuk menerima gambar input berukuran 32x32 piksel dengan 3 channel warna (misalnya, dataset CIFAR-10).
 - Untuk setiap layer Conv2D (dinamai conv2d_1, conv2d_2, dst.):
 - Jumlah filter ditentukan oleh filters_per_layer[i].
 - Ukuran kernel (filter) ditentukan oleh (kernel_sizes[i], kernel_sizes[i]).
 - Fungsi aktivasi yang digunakan adalah ReLU (activation='relu').

- padding='valid' digunakan, yang berarti tidak ada padding yang diterapkan pada input, sehingga dimensi output dari layer konvolusi mungkin akan berkurang.
- Layer Pooling:
 - Setelah setiap layer Conv2D, sebuah layer pooling ditambahkan.
 - Jenis pooling ditentukan oleh parameter pooling_type:
 - Jika 'max', maka layers.MaxPooling2D((2, 2)) (dengan nama max_pooling2d_1, dst.) digunakan, yang mengambil nilai maksimum dalam setiap window pooling.
 - Jika 'average', maka layers.AveragePooling2D((2, 2)) (dengan nama average_pooling2d_1, dst.) digunakan, yang mengambil nilai rata-rata dalam setiap window pooling.
 - Ukuran window pooling adalah (2, 2), yang akan mengurangi dimensi spasial feature map sebanyak setengahnya.

3. Layer Flatten

Setelah blok konvolusi dan pooling, layers.Flatten(name='flatten') ditambahkan. Layer ini berfungsi untuk meratakan (mengubah) output feature map 2D (atau 3D jika termasuk channel) dari layer pooling terakhir menjadi vektor 1D. Ini diperlukan sebagai transisi sebelum masuk ke layer Dense (fully connected).

4. Layer Dense (Fully Connected Layers)

Setelah proses flattening, satu atau lebih layer Dense ditambahkan untuk melakukan klasifikasi berdasarkan fitur yang telah diekstraksi:

- Hidden Dense Layer: Sebuah layers.Dense(64, activation='relu', name='dense_1') ditambahkan. Layer ini memiliki 64 unit dan menggunakan fungsi aktivasi ReLU.
- Output Dense Layer: Layer terakhir adalah layers.Dense(10, activation='softmax', name='output_dense').
 - Layer ini memiliki 10 unit, yang sesuai dengan jumlah kelas pada dataset seperti CIFAR-10.
 - Fungsi aktivasi softmax digunakan untuk menghasilkan distribusi probabilitas atas 10 kelas output, yang cocok untuk tugas klasifikasi multikelas.

5. Kompilasi Model

Setelah semua layer didefinisikan, model dikompilasi menggunakan metode compile():

- Optimizer: optimizer='adam' dipilih sebagai algoritma optimasi. Adam adalah optimizer yang populer dan seringkali memberikan hasil yang baik dengan konfigurasi default.
- Loss Function: loss='sparse_categorical_crossentropy' digunakan. Loss function ini cocok untuk tugas klasifikasi multikelas di mana label target berupa integer (misalnya, 0, 1, 2, ..., 9) dan bukan one-hot encoded.
- Metrics: metrics=['accuracy'] ditentukan untuk memantau akurasi klasifikasi selama proses pelatihan dan evaluasi model.

```
model.py
```

```
from tensorflow.keras import layers, models

def create_cnn_model(conv_layers=2,
                     filters_per_layer=[32, 64],
                     kernel_sizes=[3, 3],
                     pooling_type='max'):
    """
    Membangun model CNN dengan arsitektur yang dapat
    disesuaikan

    Parameter:
    -----
    conv_layers : int
        Total jumlah lapisan konvolusi
    filters_per_layer : list
        Jumlah filter yang digunakan pada setiap lapisan
        konvolusi
    kernel_sizes : list
        Ukuran kernel yang diterapkan pada masing-masing
        lapisan
    pooling_type : str
        Jenis pooling yang digunakan, bisa 'max' atau
        'average'

    Hasil:
    -----
    tensorflow.keras.Model
        Model CNN yang telah dikompilasi dan siap
        digunakan
    """
    model = models.Sequential()
```

```

# Add convolutional layers
for i in range(conv_layers):
    if i == 0:
        # First layer needs input shape
        model.add(layers.Conv2D(
            filters_per_layer[i],
            (kernel_sizes[i], kernel_sizes[i]),
            activation='relu',
            input_shape=(32, 32, 3),
            padding='valid',
            name=f'conv2d_{i+1}')
        ))
        print(f"After Conv Layer {i + 1}, output
shape: {model.output_shape}")
    else:
        model.add(layers.Conv2D(
            filters_per_layer[i],
            (kernel_sizes[i], kernel_sizes[i]),
            activation='relu',
            padding='valid',
            name=f'conv2d_{i+1}')
        ))
        print(f"After Conv Layer {i + 1}, output
shape: {model.output_shape}")

    # Add pooling layer after each conv layer
    if pooling_type == 'max':
        model.add(layers.MaxPooling2D((2, 2),
name=f'max_pooling2d_{i+1}'))
    else:
        model.add(layers.AveragePooling2D((2, 2),
name=f'average_pooling2d_{i+1}'))

    # Flatten layer
    model.add(layers.Flatten(name='flatten'))

    # Dense layers
    model.add(layers.Dense(64, activation='relu',
name='dense_1'))
    model.add(layers.Dense(10, activation='softmax',
name='output_dense'))

    # Compile model
    model.compile(optimizer='adam',

```

```
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

return model
```

2.2.1.3. Pelatihan (cnn/train.py)

Proses pelatihan model Convolutional Neural Network (CNN) dan evaluasinya diatur dalam file train.py. File ini berisi fungsi utama train_and_evaluate_model yang mengelola siklus pelatihan dan evaluasi, serta beberapa fungsi pendukung untuk visualisasi dan penyimpanan hasil. Langkah-langkah kerja train_and_evaluate_model dapat diuraikan sebagai berikut:

1. Persiapan Awal

Fungsi train_and_evaluate_model dimulai dengan menerima beberapa argumen penting:

- model: Objek model CNN Keras yang sudah dibuat sebelumnya (misalnya, menggunakan fungsi create_cnn_model dari modul lain).
- x_train, y_train, x_val, y_val, x_test, y_test: Data training, validasi, dan test yang sudah diproses sebelumnya (misalnya, sudah dinormalisasi dan dibagi menjadi split yang sesuai).
- model_name: String nama model, digunakan untuk penamaan file checkpoint dan dalam visualisasi.
- epochs: Jumlah epoch untuk pelatihan (default 10, namun idealnya diambil dari config.EPOCHS).

Pada tahap ini, fungsi juga memastikan direktori checkpoints/ ada untuk menyimpan bobot model.

2. Pengaturan Callbacks

Serangkaian callbacks dari Keras disiapkan untuk memantau dan mengontrol proses pelatihan secara cermat. Callbacks ini menggunakan parameter dari file config.py (seperti config.ES_PATIENCE, config.LR_FACTOR, config.LR_PATIENCE, config.MIN_LR):

- ModelCheckpoint: Callback ini bertugas menyimpan bobot model (save_weights_only=True) ke file di direktori checkpoints/ dengan nama {model_name}.weights.h5. Penyimpanan hanya dilakukan jika val_loss pada epoch saat ini lebih baik (lebih rendah, karena mode="min") dari epoch-epoch sebelumnya (save_best_only=True).
- EarlyStopping: Callback ini akan menghentikan proses pelatihan jika val_loss tidak menunjukkan perbaikan (mode="min") setelah sejumlah epoch kesabaran yang ditentukan oleh config.ES_PATIENCE. Opsi restore_best_weights=True memastikan bahwa bobot model

dikembalikan ke kondisi terbaik yang tercatat selama pelatihan sebelum dihentikan.

- ReduceLROnPlateau: Callback ini berfungsi untuk mengurangi learning rate secara dinamis jika val_loss tidak membaik (mode="min"). Learning rate akan dikalikan dengan config.LR_FACTOR setelah periode kesabaran config.LR_PATIENCE, hingga mencapai batas minimum config.MIN_LR.

3. Pelatihan Model

Proses pelatihan model CNN dijalankan menggunakan metode model.fit().

Metode ini menerima:

- Data training (x_{train} , y_{train}).
- batch_size (dalam kode ini diatur ke 32, namun idealnya diambil dari config.BATCH_SIZE).
- Jumlah epochs yang telah ditentukan.
- Data validasi (x_{val} , y_{val}) untuk pemantauan performa model pada data yang tidak terlihat selama training per epoch.
- Daftar callbacks yang telah disiapkan sebelumnya. Objek history yang berisi catatan metrik (loss, accuracy, val_loss, val_accuracy) selama setiap epoch pelatihan akan disimpan.

4. Pemuatan Bobot Terbaik

Setelah pelatihan selesai (baik karena mencapai jumlah epoch maksimum atau dihentikan oleh EarlyStopping), fungsi ini secara eksplisit mencoba memuat bobot terbaik yang disimpan oleh ModelCheckpoint dari file {checkpoint_path}. Langkah ini bertujuan untuk memastikan model yang dievaluasi adalah model dengan performa terbaik pada set validasi, melengkapi mekanisme restore_best_weights dari EarlyStopping.

5. Evaluasi Model

Model dengan bobot terbaiknya kemudian dievaluasi pada dataset test (x_{test} , y_{test}):

- model.predict(x_{test}) digunakan untuk mendapatkan probabilitas prediksi untuk setiap kelas pada data test.
- Fungsi compute_macro_f1_score() (diimpor dari data_preprocessing.py) digunakan untuk menghitung skor Macro F1 berdasarkan label sebenarnya (y_{test}) dan prediksi model. Skor Macro F1 ini kemudian dicetak.

6. Evaluasi Model

ModelCheckpoint menangani penyimpanan bobot model terbaik (.weights.h5) selama pelatihan. Bobot ini dapat dimuat kembali ke arsitektur model yang sama untuk penggunaan di masa depan atau inferensi.

7. Nilai Kembali (Return Value)

Fungsi train_and_evaluate_model mengembalikan tiga objek utama:

- model: Objek model Keras yang telah dilatih dan bobot terbaiknya telah dimuat.
- history: Objek History dari Keras yang berisi catatan metrik pelatihan dan validasi per epoch.
- test_f1: Nilai Macro F1-score hasil evaluasi model pada dataset test.

```
train.py

import numpy as np
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
from data_preprocessing import compute_macro_f1_score
from tensorflow.keras.callbacks import ModelCheckpoint,
EarlyStopping, ReduceLROnPlateau
import os
import config
import pandas as pd
import seaborn as sns

# Fungsi untuk melatih dan mengevaluasi model
def train_and_evaluate_model(model, x_train, y_train, x_val,
y_val, x_test, y_test,
model_name, epochs=10):
    """
    Train and evaluate a CNN model

    Returns:
    -----
    tuple: (model, history, test_f1_score)
    """
    print(f"\nTraining {model_name}...")

    # Create checkpoints directory if it doesn't exist
    os.makedirs("checkpoints", exist_ok=True)

    # Siapkan callback untuk training
```

```

checkpoint_path = f"checkpoints/{model_name}.weights.h5"
callbacks = [
    ModelCheckpoint(
        checkpoint_path,
        monitor="val_loss",  # Pantau val_loss untuk simpan
model terbaik
        mode="min",
        save_best_only=True,
        save_weights_only=True,
        verbose=1,
    ),
    EarlyStopping(
        monitor="val_loss",
        mode="min",
        patience=config.ES_PATIENCE,  # Berhenti kalau
tidak ada perbaikan
        verbose=1,
        restore_best_weights=True,
    ),
    ReduceLROnPlateau(
        monitor="val_loss",
        mode="min",
        factor=config.LR_FACTOR,  # Faktor pengurangan
learning rate
        patience=config.LR_PATIENCE,
        min_lr=config.MIN_LR,
        verbose=1,
    ),
]
# Train model with callbacks
history = model.fit(
    x_train, y_train,
    batch_size=32,
    epochs=epochs,
    validation_data=(x_val, y_val),
    callbacks=callbacks,
    verbose=1
)

# Load best weights if checkpoint exists
if os.path.exists(checkpoint_path):
    print(f"Loading best weights from {checkpoint_path}")
    model.load_weights(checkpoint_path)

```

```

# Evaluate on test set
test_predictions = model.predict(x_test)
test_f1 = compute_macro_f1_score(y_test, test_predictions)

print(f'{model_name} - Test Macro F1-Score: {test_f1:.4f}')

return model, history, test_f1

def plot_individual_training_history(history, model_name):
    """Plot training and validation curves for individual
model"""
    plt.figure(figsize=(12, 4))

    # Plot loss
    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Training Loss',
color='blue')
    plt.plot(history.history['val_loss'], label='Validation
Loss', color='red', linestyle='--')
    plt.title(f'{model_name} - Loss Curves')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Plot accuracy
    plt.subplot(1, 2, 2)
    plt.plot(history.history['accuracy'], label='Training
Accuracy', color='blue')
    plt.plot(history.history['val_accuracy'], label='Validation
Accuracy', color='red', linestyle='--')
    plt.title(f'{model_name} - Accuracy Curves')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

def plot_confusion_matrix(y_true, y_pred, class_names,
model_name):
    """Plot confusion matrix"""

```

```

cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names,
            yticklabels=class_names)
plt.title(f'Confusion Matrix - {model_name}')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()

def plot_combined_comparison(histories, model_names, f1_scores,
                             train_times):
    """Plot combined comparison charts"""
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # 1. Loss comparison
    axes[0, 0].set_title('Training Loss Comparison')
    for history, name in zip(histories, model_names):
        axes[0, 0].plot(history.history['loss'],
                        label=f'{name}')
    axes[0, 0].set_xlabel('Epoch')
    axes[0, 0].set_ylabel('Training Loss')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # 2. Validation loss comparison
    axes[0, 1].set_title('Validation Loss Comparison')
    for history, name in zip(histories, model_names):
        axes[0, 1].plot(history.history['val_loss'],
                        label=f'{name}', linestyle='--')
    axes[0, 1].set_xlabel('Epoch')
    axes[0, 1].set_ylabel('Validation Loss')
    axes[0, 1].legend()
    axes[0, 1].grid(True, alpha=0.3)

    # 3. F1-Score comparison
    axes[1, 0].bar(model_names, f1_scores, color=['skyblue',
                                                'lightgreen', 'salmon'])
    axes[1, 0].set_title('F1-Score Comparison')
    axes[1, 0].set_ylabel('F1-Score')

```

```

        axes[1, 0].set_ylim(0, 1)
        for i, score in enumerate(f1_scores):
            axes[1, 0].text(i, score + 0.01, f'{score:.4f}', ha='center')

    # 4. Training time comparison
    axes[1, 1].bar(model_names, train_times, color=['orange', 'purple', 'brown'])
    axes[1, 1].set_title('Training Time Comparison')
    axes[1, 1].set_ylabel('Training Time (seconds)')
    for i, time_val in enumerate(train_times):
        axes[1, 1].text(i, time_val + max(train_times) * 0.01, f'{time_val:.1f}s', ha='center')

    plt.tight_layout()
    plt.show()

def save_results_to_csv_conv(model_names, accuracies,
                             f1_scores, losses, train_times, save_path):
    """Save experiment results to CSV"""
    num_layers = [int(name.split('_')[0]) for name in model_names]

    results_df = pd.DataFrame({
        'num_layers': num_layers,
        'accuracy': accuracies,
        'f1_score': f1_scores,
        'loss': losses,
        'train_time': train_times
    })

    results_df.to_csv(save_path, index=False)
    print(f"Results saved to {save_path}")

    return results_df

def save_results_to_csv_filters(model_names, accuracies,
                               f1_scores, losses, train_times, save_path):
    """Save filter experiment results to CSV"""
    filter_configs = []
    for name in model_names:
        parts = name.split('_')
        config = f"{parts[1]}-{parts[2]}-{parts[3]}"
        filter_configs.append(config)

```

```

results_df = pd.DataFrame({
    'filter_config': filter_configs,
    'accuracy': accuracies,
    'f1_score': f1_scores,
    'loss': losses,
    'train_time': train_times
})

results_df.to_csv(save_path, index=False)
print(f"Results saved to {save_path}")

return results_df

def save_results_to_csv_kernels(model_names, accuracies,
f1_scores, losses, train_times, filename):
    """Save kernel sizes experiment results to CSV"""
    import pandas as pd

    kernel_configs = []
    for name in model_names:
        kernels_part = name.split('kernels_')[1]
        kernel_configs.append(kernels_part)

    results_dict = {
        'Model_Name': model_names,
        'Kernel_Configuration': kernel_configs,
        'Test_Accuracy': [f"{acc:.4f}" for acc in accuracies],
        'F1_Score': [f"{f1:.4f}" for f1 in f1_scores],
        'Test_Loss': [f"{loss:.4f}" for loss in losses],
        'Training_Time_seconds': [f"{time:.2f}" for time in
train_times]
    }

    df = pd.DataFrame(results_dict)
    df.to_csv(filename, index=False)
    print(f"\n💾 Results saved to: {filename}")

    return df

def save_results_to_csv_pooling(model_names, accuracies,
f1_scores, losses, train_times, filename):
    """Save pooling types experiment results to CSV"""
    import pandas as pd

```

```

pooling_types = []
for name in model_names:
    # "max_pooling" atau "average_pooling"
    pooling_type = name.split('_pooling')[0]
    pooling_types.append(pooling_type.capitalize())

results_dict = {
    'Model_Name': model_names,
    'Pooling_Type': pooling_types,
    'Test_Accuracy': [f"{acc:.4f}" for acc in accuracies],
    'F1_Score': [f"{f1:.4f}" for f1 in f1_scores],
    'Test_Loss': [f"{loss:.4f}" for loss in losses],
    'Training_Time_seconds': [f"{time:.2f}" for time in
train_times]
}

df = pd.DataFrame(results_dict)
df.to_csv(filename, index=False)
print(f"\n💾 Results saved to: {filename}")

return df

```

2.2.1.4. Implementasi Forward From Scratch (cnn/from_scratch.py)

Implementasi propagasi maju (forward propagation) dari awal (from scratch) untuk model Convolutional Neural Network (CNN) dikembangkan dalam file from_scratch.py. Tujuan utama dari implementasi ini adalah untuk mereplikasi fungsionalitas prediksi dari model CNN yang telah dilatih menggunakan Keras, namun dengan memanfaatkan operasi-operasi dasar dari pustaka NumPy untuk perhitungan matematisnya. Pendekatan ini memungkinkan pemahaman yang lebih mendalam tentang cara kerja internal CNN. Implementasi ini dikemas dalam kelas CNNFromScratch dan dilengkapi dengan fungsi run_from_scratch_comparison untuk validasi dan analisis perbandingan.

Kelas CNNFromScratch

Kelas ini merupakan inti dari implementasi CNN from scratch. Kelas ini dirancang untuk menangani pemutuan model Keras yang sudah ada, ekstraksi bobot dan konfigurasi layer, serta eksekusi proses propagasi maju secara manual.

1. Inisialisasi (`__init__`)

Pada tahap inisialisasi, objek CNNFromScratch menerima path ke model Keras yang telah disimpan (keras_model_path).

- Konstruktor memuat model Keras menggunakan `tf.keras.models.load_model()`. Model Keras ini digunakan sebagai referensi untuk arsitektur dan bobot.
- Atribut `self.weights` (sebuah dictionary) diinisialisasi untuk menyimpan bobot-bobot yang diekstrak dari setiap layer.
- Atribut `self.layer_info` (sebuah dictionary) diinisialisasi untuk menyimpan informasi konfigurasi dari setiap layer Keras (misalnya, jumlah filter, ukuran kernel, strides, padding, jenis aktivasi, dll.).
- Proses utama dalam inisialisasi adalah pemanggilan metode `self.extract_weights_from_keras_model()` untuk mengisi atribut-atribut tersebut.

2. Ekstraksi Bobot dan Konfigurasi Model Keras (`extract_weights_from_keras_model`)

Metode ini bertanggung jawab untuk mengiterasi melalui setiap layer pada model Keras yang telah dimuat.

- Untuk setiap layer, metode ini mengidentifikasi jenisnya (misalnya, Conv2D, MaxPooling2D, AveragePooling2D, Flatten, Dense).
- Jika layer memiliki bobot (seperti Conv2D dan Dense), bobot-bobot tersebut (kernel dan bias) diekstrak dan disimpan dalam dictionary `self.weights` dengan kunci yang deskriptif (misalnya, `{layer_name}_kernel`, `{layer_name}_bias`).
- Informasi konfigurasi penting dari setiap layer (seperti filters, kernel_size, strides, padding, activation untuk Conv2D; pool_size, strides, padding untuk layer pooling; units, activation untuk Dense) disimpan dalam dictionary `self.layer_info`. Informasi ini krusial untuk memandu proses propagasi maju from scratch agar sesuai dengan arsitektur model Keras aslinya.

3. Helper Function: Konversi Gambar ke Kolom (`im2col`)

Metode `im2col` adalah fungsi pembantu kunci yang digunakan untuk optimasi operasi konvolusi.

- Metode ini mengubah patch-patch input dari gambar menjadi kolom-kolom dalam sebuah matriks. Transformasi ini memungkinkan operasi konvolusi 2D dihitung sebagai perkalian matriks tunggal (General Matrix Multiply - GEMM), yang jauh lebih efisien dalam NumPy dibandingkan dengan implementasi menggunakan loop bersarang.
- Fungsi ini menangani perhitungan padding ('valid' atau 'same') untuk memastikan dimensi output sesuai dengan yang diharapkan.

4. Forward Propagation Layer Conv2D (conv2d_forward)

Metode ini mengimplementasikan propagasi maju untuk sebuah layer Conv2D.

- Menerima input (feature map dari layer sebelumnya), bobot kernel, bias, serta parameter stride, padding, dan fungsi aktivasi.
- Menggunakan metode `self.im2col()` untuk mengubah input menjadi format kolom.
- Mengubah bobot kernel menjadi format matriks yang sesuai.
- Melakukan operasi konvolusi utama melalui perkalian matriks antara matriks kolom input dan matriks kernel, kemudian menambahkan bias.
- Output matriks kemudian diubah kembali ke bentuk tensor (feature map) dengan dimensi yang sesuai.
- Fungsi aktivasi (ReLU atau Softmax, sesuai konfigurasi layer) diterapkan pada hasil konvolusi.

5. Forward Propagation Layer Pooling (pooling_forward)

Metode ini mengimplementasikan propagasi maju untuk layer pooling (MaxPooling2D atau AveragePooling2D).

- Menerima input (feature map), ukuran window pooling, stride, dan jenis pooling ('max' atau 'average').
- Menghitung dimensi output berdasarkan parameter yang diberikan.
- Secara iteratif (namun dioptimalkan untuk memproses semua channel sekaligus per window), metode ini mengekstrak patch dari input dan menerapkan operasi pooling yang sesuai (np.max untuk Max Pooling atau np.mean untuk Average Pooling) pada setiap patch.

6. Forward Propagation Layer Flatten (flatten_forward)

Metode ini mengimplementasikan propagasi maju untuk layer Flatten.

- Menerima input tensor (biasanya output dari layer konvolusi/pooling terakhir).
- Mengubah tensor input multidimensi menjadi vektor 1D per sampel dalam batch, menggunakan `inputs.reshape(batch_size, -1)`.

7. Forward Propagation Layer Dense (dense_forward)

Metode ini mengimplementasikan propagasi maju untuk layer Dense (fully-connected).

- Menerima input (vektor dari layer Flatten atau layer Dense sebelumnya), bobot kernel, dan bias, serta nama fungsi aktivasi.
- Melakukan operasi linear utama melalui perkalian matriks antara input dan bobot kernel, kemudian menambahkan bias:

$$\text{Output} = \text{Input} \cdot W + b$$

- Fungsi aktivasi (ReLU atau Softmax dengan stabilisasi numerik, sesuai konfigurasi layer) diterapkan pada hasil operasi linear.

8. Forward Propagation Keseluruhan Model (forward)

Metode ini mengorkestrasi keseluruhan proses propagasi maju dari input gambar hingga output probabilitas prediksi.

- Menerima tensor input gambar.
- Mengiterasi melalui daftar layer dari model Keras asli (`self.keras_model.layers`) untuk menjaga urutan operasi yang benar.
- Untuk setiap layer:
 - Informasi tipe dan konfigurasi layer diambil dari `self.layer_info`.
 - Bobot yang sesuai diambil dari `self.weights`.
 - Metode forward propagation yang relevan untuk tipe layer tersebut (`conv2d_forward`, `pooling_forward`, `flatten_forward`, atau `dense_forward`) dipanggil dengan input dari layer sebelumnya dan parameter yang sesuai.
 - Output dari satu layer menjadi input untuk layer berikutnya.
- Output akhir dari layer Dense terakhir (setelah aktivasi Softmax) adalah probabilitas prediksi model from scratch.

9. Prediksi Gambar (predict)

Metode ini adalah *interface* sederhana untuk melakukan prediksi pada data input baru.

- Menerima array NumPy berisi gambar-gambar input.
- Memanggil metode `self.forward()` untuk mendapatkan prediksi probabilitas.

10. Perbandingan dengan Keras (compare_with_keras)

Metode ini dirancang untuk memvalidasi akurasi dan performa implementasi from scratch terhadap model Keras asli.

- Menerima data gambar uji (`test_data`) dan labelnya (`test_labels`).
- Melakukan prediksi pada `test_data` menggunakan implementasi from scratch (`self.predict()`) dan juga menggunakan model Keras asli (`self.keras_model.predict()`).
- Mengukur dan mencetak waktu eksekusi untuk kedua implementasi.
- Menghitung akurasi kecocokan implementasi: persentase kesamaan kelas prediksi antara from scratch dan Keras.
- Menghitung Mean Absolute Error (MAE) antara vektor probabilitas prediksi dari kedua implementasi.

- Menghitung dan mencetak Macro F1-score untuk kedua implementasi pada `test_labels`.
- Mengembalikan prediksi dari kedua implementasi dan akurasi kecocokan.

11. Perhitungan Macro F1-Score (`compute_macro_f1_score`)

Metode ini menghitung skor F1 rata-rata makro untuk evaluasi klasifikasi multikelas.

- Menerima label sebenarnya (`y_true`) dan probabilitas prediksi (`y_pred`).
- Mengonversi probabilitas prediksi menjadi label kelas.
- Menghitung F1-score untuk setiap kelas secara individual, kemudian mengambil rata-ratanya tanpa pembobotan berdasarkan jumlah sampel per kelas.

Kelas CNNFromScratch

Fungsi pendukung ini berada di luar kelas `CNNFromScratch` dan bertugas untuk mengelola keseluruhan proses perbandingan dan pelaporan secara komprehensif:

1. Memuat Data: Memuat dataset CIFAR-10 (data latih, validasi, dan uji beserta nama kelas) menggunakan fungsi `load_cifar10_data()` dari modul `data_preprocessing.py`.
2. Inisialisasi Objek: Menginisialisasi objek `CNNFromScratch` dengan menyediakan path ke model Keras yang telah dilatih (`model_path`).
3. Perbandingan Implementasi: Memanggil metode `scratch_cnn.compare_with_keras()` menggunakan data uji untuk mendapatkan prediksi dari kedua implementasi serta metrik perbandingan awal (akurasi kecocokan, MAE, waktu prediksi, dan F1-score individual).
4. Perhitungan Metrik Akhir: Menghitung metrik evaluasi performa tambahan seperti akurasi keseluruhan pada data uji untuk kedua implementasi.
5. Pelaporan Detail: Mencetak laporan terstruktur yang mencakup:
 - a. Path model Keras yang digunakan.
 - b. Akurasi kecocokan implementasi dan MAE.
 - c. Perbandingan waktu prediksi.
 - d. Metrik performa pada data uji (Akurasi dan Macro F1-score) untuk from scratch dan Keras.
 - e. Laporan klasifikasi (`classification_report` dari `scikit-learn`) untuk kedua implementasi, yang merinci presisi, recall, dan F1-score per kelas.
 - f. Matriks konfusi (confusion matrix) untuk kedua implementasi.
6. Visualisasi (jika prediksi from scratch valid): Menyediakan serangkaian visualisasi untuk analisis lebih mendalam:

- a. Plot perbandingan confusion matrix antara implementasi from scratch dan Keras.
- b. Plot perbandingan distribusi probabilitas prediksi maksimum.
- c. Plot perbandingan rata-rata perbedaan absolut probabilitas untuk sampel acak.
- d. Menampilkan beberapa contoh gambar uji beserta label sebenarnya dan label yang diprediksi oleh kedua implementasi.

Fungsi ini memastikan bahwa implementasi from scratch tidak hanya berjalan tetapi juga menghasilkan output yang sebanding (atau identik dalam batas toleransi numerik) dengan model Keras aslinya, serta memberikan wawasan tentang potensi perbedaan performa atau perilaku.

```
from_scratch.py

import tensorflow as tf
import numpy as np
import time
import matplotlib.pyplot as plt
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    f1_score as sklearn_f1_score,
)
from data_preprocessing import (
    load_cifar10_data,
    compute_macro_f1_score
)

class CNNFromScratch:
    """
        Implementasi teroptimasi dari model CNN yang dibuat dari nol menggunakan operasi vektorisasi dengan NumPy.
        Kelas ini memuat bobot dari model Keras yang telah dilatih, lalu melakukan propagasi maju dengan peningkatan performa yang signifikan.
    """

    def __init__(self, keras_model_path):
        """
            Inisialisasi CNN dari nol dengan memuat bobot dari model Keras yang sudah dilatih sebelumnya.
        """


```

```

Parameter:
-----
keras_model_path : str
    Lokasi file model Keras yang sudah disimpan.
"""
print(f"Loading Keras model from: {keras_model_path}")
self.keras_model =
tf.keras.models.load_model(keras_model_path)
self.keras_model.summary()

# Extract weights from each layer of the Keras model
self.extract_weights_from_keras_model()
print("Model loaded and weights extracted
successfully")

def extract_weights_from_keras_model(self):
    """Mengambil bobot dari model Keras yang telah
dimuat"""
    self.weights = {}
    self.layer_info = {}

    for layer in self.keras_model.layers:
        layer_name = layer.name
        layer_type = type(layer).__name__

        print(f"Processing layer: {layer_name}
({layer_type})")

        if hasattr(layer, 'get_weights') and
len(layer.get_weights()) > 0:
            layer_weights = layer.get_weights()

            if layer_type == 'Conv2D':
                # Conv2D layer: [kernel, bias]
                self.weights[f"{layer_name}_kernel"] =
layer_weights[0]
                self.weights[f"{layer_name}_bias"] =
layer_weights[1]

            # Store layer configuration
            self.layer_info[layer_name] = {
                'type': 'Conv2D',
                'filters': layer.filters,
                'kernel_size': layer.kernel_size,

```

```

        'strides': layer.strides,
        'padding': layer.padding,
        'activation': layer.activation.__name__
    }

    print(f"  Kernel shape:
{layer_weights[0].shape}")
    print(f"  Bias shape:
{layer_weights[1].shape}")

    elif layer_type == 'Dense':
        # Dense layer: [kernel, bias]
        self.weights[f"{layer_name}_kernel"] =
layer_weights[0]
        self.weights[f"{layer_name}_bias"] =
layer_weights[1]

        self.layer_info[layer_name] = {
            'type': 'Dense',
            'units': layer.units,
            'activation': layer.activation.__name__
        }

        print(f"  Kernel shape:
{layer_weights[0].shape}")
        print(f"  Bias shape:
{layer_weights[1].shape}")

    elif layer_type in ['MaxPooling2D',
'AveragePooling2D']:
        # Pooling layers don't have weights but we need
their config
        self.layer_info[layer_name] = {
            'type': layer_type,
            'pool_size': layer.pool_size,
            'strides': layer.strides,
            'padding': layer.padding
        }

    elif layer_type == 'Flatten':
        self.layer_info[layer_name] = {'type':
'Flatten'}

def im2col(self, inputs, kernel_size, strides, padding):

```

```

"""
Mengubah gambar menjadi matriks kolom agar proses
konvolusi bisa dilakukan lebih efisien lewat perkalian matriks.
Teknik ini jadi kunci optimasi karena mengubah operasi
konvolusi jadi operasi GEMM (General Matrix Multiplication).

Parameter:
-----
inputs : numpy.ndarray
    Data input dengan format (batch_size, tinggi,
lebar, jumlah_channel)
kernel_size : tuple
    Ukuran kernel, yaitu (tinggi_kernel, lebar_kernel)
strides : tuple
    Langkah pergeseran (stride), dalam bentuk
(vertikal, horizontal)
padding : str
    Jenis padding, bisa 'valid' atau 'same'

Return:
-----
numpy.ndarray
    Matriks kolom dengan ukuran (batch_size *
tinggi_output * lebar_output, tinggi_kernel * lebar_kernel *
jumlah_channel_input)
tuple
    (tinggi_output, lebar_output) - ukuran output
setelah konvolusi
"""

batch_size, input_height, input_width, input_channels =
inputs.shape
kernel_height, kernel_width = kernel_size
stride_h, stride_w = strides

# Calculate output dimensions
if padding == 'valid':
    output_height = (input_height - kernel_height) //
stride_h + 1
    output_width = (input_width - kernel_width) //
stride_w + 1
    pad_h = pad_w = 0
else: # padding == 'same'
    output_height = input_height // stride_h
    output_width = input_width // stride_w

```

```

        pad_h = max(0, (output_height - 1) * stride_h +
kernel_height - input_height)
        pad_w = max(0, (output_width - 1) * stride_w +
kernel_width - input_width)

        # Apply padding if needed
        if pad_h > 0 or pad_w > 0:
            inputs = np.pad(inputs,
                            ((0, 0), (pad_h//2, pad_h -
pad_h//2),
                            (pad_w//2, pad_w - pad_w//2), (0,
0)),
                            mode='constant')

        # Pre-allocate output matrix
        col_matrix = np.zeros((batch_size * output_height * output_width,
                                kernel_height * kernel_width * input_channels))

        # Vectorized im2col conversion
        idx = 0
        for b in range(batch_size):
            for h in range(output_height):
                for w in range(output_width):
                    h_start = h * stride_h
                    h_end = h_start + kernel_height
                    w_start = w * stride_w
                    w_end = w_start + kernel_width

                    # Extract and flatten the patch
                    patch = inputs[b, h_start:h_end,
w_start:w_end, :]
                    col_matrix[idx] = patch.flatten()
                    idx += 1

        return col_matrix, (output_height, output_width)

    def conv2d_forward(self, inputs, kernel, bias, strides=(1,
1), padding='valid', activation='relu'):
        """
        Proses forward pass konvolusi 2D yang dioptimasi menggunakan im2col dan perkalian matriks.

```

```

Parameter:
-----
inputs : numpy.ndarray
    Tensor input dengan bentuk (batch_size, tinggi,
lebar, jumlah_channel)
kernel : numpy.ndarray
    Kernel konvolusi dengan bentuk (tinggi_kernel,
lebar_kernel, jumlah_channel_input, jumlah_channel_output)
bias : numpy.ndarray
    Vektor bias dengan bentuk (jumlah_channel_output,)
strides : tuple
    Langkah pergeseran (stride) konvolusi
padding : str
    Jenis padding ('valid' atau 'same')
activation : str
    Nama fungsi aktivasi

Return:
-----
numpy.ndarray
    Tensor output hasil konvolusi
"""
batch_size = inputs.shape[0]
kernel_height, kernel_width, input_channels,
output_channels = kernel.shape

# Convert image to column matrix
col_matrix, (output_height, output_width) =
self.im2col(
    inputs, (kernel_height, kernel_width), strides,
padding
)

# Reshape kernel for matrix multiplication
kernel_matrix = kernel.reshape(-1, output_channels)

# Perform convolution as matrix multiplication (GEMM)
# This is much faster than nested loops
output_matrix = np.dot(col_matrix, kernel_matrix) +
bias

# Reshape output back to tensor format
output = output_matrix.reshape(batch_size,
output_height, output_width, output_channels)

```

```

# Apply activation function vectorized
if activation == 'relu':
    output = np.maximum(0, output)
elif activation == 'softmax':
    # Apply softmax along the last dimension
    exp_output = np.exp(output - np.max(output,
axis=-1, keepdims=True))
    output = exp_output / np.sum(exp_output, axis=-1,
keepdims=True)

return output

def pooling_forward(self, inputs, pool_size=(2, 2),
strides=(2, 2), pooling_type='max'):
    """
        Proses forward pass pada layer flatten (sudah
dioptimasi dengan reshape).

    Parameter:
    -----
    inputs : numpy.ndarray
        Tensor input

    Return:
    -----
    numpy.ndarray
        Tensor hasil flatten dengan bentuk (batch_size,
ukuran_flatten)
    """
    batch_size, input_height, input_width, channels =
inputs.shape
    pool_h, pool_w = pool_size
    stride_h, stride_w = strides

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

    # Pre-allocate output
    output = np.zeros((batch_size, output_height,
output_width, channels))

    # Vectorized pooling - process all channels at once

```

```

        for h in range(output_height):
            for w in range(output_width):
                h_start = h * stride_h
                h_end = h_start + pool_h
                w_start = w * stride_w
                w_end = w_start + pool_w

                    # Extract patches for all batches and channels
                    simultaneously
                    patches = inputs[:, h_start:h_end,
w_start:w_end, :]

                        # Apply pooling operation vectorized across
batch and channel dimensions
                        if pooling_type == 'max':
                            output[:, h, w, :] = np.max(patches,
axis=(1, 2))
                        else: # average pooling
                            output[:, h, w, :] = np.mean(patches,
axis=(1, 2))

                    return output

    def flatten_forward(self, inputs):
        """
        Flatten layer forward pass (already optimized with
        reshape)

        Parameters:
        -----
        inputs : numpy.ndarray
            Input tensor

        Returns:
        -----
        numpy.ndarray
            Flattened tensor of shape (batch_size,
            flattened_size)
        """
        batch_size = inputs.shape[0]
        return inputs.reshape(batch_size, -1)

    def dense_forward(self, inputs, kernel, bias,
activation='relu'):

```

```

"""
Proses forward pass pada dense layer yang sudah dioptimasi menggunakan operasi vektorisasi.

Parameter:
-----
inputs : numpy.ndarray
    Tensor input
kernel : numpy.ndarray
    Matriks bobot
bias : numpy.ndarray
    Vektor bias
activation : str
    Nama fungsi aktivasi

Return:
-----
numpy.ndarray
    Tensor output setelah melalui dense layer
"""

# Linear transformation using optimized BLAS operations
output = np.dot(inputs, kernel) + bias

# Apply activation function vectorized
if activation == 'relu':
    output = np.maximum(0, output)
elif activation == 'softmax':
    # Numerically stable softmax
    exp_output = np.exp(output - np.max(output, axis=1,
keepdims=True))
    output = exp_output / np.sum(exp_output, axis=1,
keepdims=True)

return output

def forward(self, inputs):
"""
Proses forward pass lengkap yang sudah dioptimasi.

Parameter:
-----
inputs : numpy.ndarray
    Tensor input dengan bentuk (batch_size, tinggi,
lebar, jumlah_channel)

```

```

Return:
-----
numpy.ndarray
    Probabilitas output
"""
x = inputs.copy()

# Process each layer in order
for layer in self.keras_model.layers:
    layer_name = layer.name
    layer_info = self.layer_info.get(layer_name, {})
    layer_type = layer_info.get('type', '')

    if layer_type == 'Conv2D':
        kernel = self.weights[f'{layer_name}_kernel']
        bias = self.weights[f'{layer_name}_bias']
        activation = layer_info['activation']
        strides = layer_info['strides']
        padding = layer_info['padding']

        x = self.conv2d_forward(x, kernel, bias,
strides, padding, activation)
        print(f"After {layer_name}: {x.shape}")

    elif layer_type in ['MaxPooling2D',
'AveragePooling2D']:
        pool_size = layer_info['pool_size']
        strides = layer_info['strides']
        pooling_type = 'max' if layer_type ==
'MaxPooling2D' else 'average'

        x = self.pooling_forward(x, pool_size, strides,
pooling_type)
        print(f"After {layer_name}: {x.shape}")

    elif layer_type == 'Flatten':
        x = self.flatten_forward(x)
        print(f"After {layer_name}: {x.shape}")

    elif layer_type == 'Dense':
        kernel = self.weights[f'{layer_name}_kernel']
        bias = self.weights[f'{layer_name}_bias']
        activation = layer_info['activation']

```

```

        x = self.dense_forward(x, kernel, bias,
activation)
        print(f"After {layer_name}: {x.shape}")

    return x

def predict(self, inputs):
    """
    Melakukan prediksi terhadap data input.

    Parameter:
    -----
    inputs : numpy.ndarray
        Gambar input

    Return:
    -----
    numpy.ndarray
        Probabilitas prediksi untuk tiap kelas
    """
    return self.forward(inputs)

def compare_with_keras(self, test_data, test_labels):
    """
    Membandingkan hasil prediksi dari implementasi manual
dengan model Keras.

    Parameter:
    -----
    test_data : numpy.ndarray
        Gambar uji
    test_labels : numpy.ndarray
        Label uji

    Return:
    -----
    tuple
        (prediksi_manual, prediksi_keras, akurasi)
    """
    # Get predictions from our scratch implementation
    print("Getting predictions from scratch
implementation...")
    start_time_scratch = time.time()

```

```

scratch_preds = self.predict(test_data)
scratch_time = time.time() - start_time_scratch

# Get predictions from Keras
print("Getting predictions from Keras model...")
start_time_keras = time.time()
keras_preds = self.keras_model.predict(test_data)
keras_time = time.time() - start_time_keras

# Calculate accuracy between the two implementations
scratch_classes = np.argmax(scratch_preds, axis=1)
keras_classes = np.argmax(keras_preds, axis=1)
implementation_accuracy = np.mean(scratch_classes == keras_classes)

# Calculate F1 scores
scratch_f1 = self.compute_macro_f1_score(test_labels,
scratch_preds)
keras_f1 = self.compute_macro_f1_score(test_labels,
keras_preds)

print(f"\nPrediction time comparison:")
print(f"From scratch: {scratch_time:.4f} seconds")
print(f"Keras: {keras_time:.4f} seconds")
print(f"Time ratio (scratch/keras): "
{scratch_time/keras_time:.2f}x")

print(f"\nImplementation match accuracy:
{implementation_accuracy:.4f}")
print(f"Mean absolute error between predictions:
{np.mean(np.abs(scratch_preds - keras_preds)):.6f}")

print(f"\nTest set metrics:")
print(f"From scratch - Macro F1-Score:
{scratch_f1:.4f}")
print(f"Keras - Macro F1-Score: {keras_f1:.4f}")

return scratch_preds, keras_preds,
implementation_accuracy

def compute_macro_f1_score(self, y_true, y_pred):
"""
Menghitung skor F1 makro untuk klasifikasi multi-kelas.

```

```

Parameter:
-----
y_true : numpy.ndarray
    Label asli (bisa dalam bentuk one-hot atau label
sebagai angka)
y_pred : numpy.ndarray
    Probabilitas hasil prediksi

Return:
-----
float
    Nilai F1-score makro
"""

# Convert predictions to class labels
if y_pred.ndim > 1:
    y_pred_classes = np.argmax(y_pred, axis=1)
else:
    y_pred_classes = y_pred

# Convert true labels to class labels if one-hot
encoded
if y_true.ndim > 1:
    y_true_classes = np.argmax(y_true, axis=1)
else:
    y_true_classes = y_true

# Get unique classes
classes = np.unique(np.concatenate([y_true_classes,
y_pred_classes]))
f1_scores = []

for cls in classes:
    # Calculate precision and recall for each class
    tp = np.sum((y_true_classes == cls) &
(y_pred_classes == cls))
    fp = np.sum((y_true_classes != cls) &
(y_pred_classes == cls))
    fn = np.sum((y_true_classes == cls) &
(y_pred_classes != cls))

    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) if (tp + fn) > 0 else 0

    f1 = 2 * precision * recall / (precision + recall)

    f1_scores.append(f1)

return np.mean(f1_scores)

```

```

if (precision + recall) > 0 else 0
    f1_scores.append(f1)

return np.mean(f1_scores)

def run_from_scratch_comparison(model_path):
    """
    Run comparison between CNN from scratch implementation and
    Keras model

    Parameters:
    -----
    model_path : str
        Path to the saved Keras CNN model
    """

    print("\n" + "=" * 50)
    print("CNN FROM SCRATCH IMPLEMENTATION")
    print("=" * 50)

    # Load data - sesuaikan dengan fungsi load_cifar10_data()
    try:
        (x_train, y_train), (x_val, y_val), (x_test, y_test),
        class_names = load_cifar10_data()

        # Extract variables for compatibility
        test_images = x_test
        test_labels = y_test
        label_mapping = {name: i for i, name in
        enumerate(class_names)} # Create mapping dict
        num_classes = len(class_names)

        print(f"Loaded test data: {len(test_images)} samples")
        print(f"Image shape: {test_images.shape[1:]}")
        print(f"Number of classes: {num_classes}")
        print(f"Class names: {class_names}")
    except Exception as e:
        print(f"Error loading data: {e}")
        return None

    # Initialize from-scratch implementation
    try:
        scratch_cnn = CNNFromScratch(model_path)
    except Exception as e:

```

```

        print(f"Error initializing CNN from scratch: {e}")
        return None

    # Compare implementations on test data
    print("\nComparing implementations on test data...")

    try:
        scratch_preds, keras_preds, implementation_accuracy = (
            scratch_cnn.compare_with_keras(test_images,
test_labels)
        )
    except Exception as e:
        print(f"Error during comparison: {e}")
        return None

    # Get predicted classes
    scratch_classes = None
    mae = 0.0

    if scratch_preds is not None and not (
        np.any(np.isnan(scratch_preds)) or
np.any(np.isinf(scratch_preds))
    ):
        scratch_classes = np.argmax(scratch_preds, axis=1)
        mae = np.mean(np.abs(scratch_preds - keras_preds))
    else:
        print("WARNING: NaN or Inf found in scratch
predictions!")
        mae = float("inf")

    keras_classes = np.argmax(keras_preds, axis=1)

    # Calculate metrics for both implementations
    scratch_accuracy = 0.0
    scratch_f1 = 0.0
    if scratch_classes is not None:
        scratch_accuracy = np.mean(scratch_classes ==
test_labels)
        scratch_f1 = compute_macro_f1_score(test_labels,
scratch_preds)
    else:
        print("Warning: Scratch metrics cannot be calculated
due to invalid predictions (NaN/Inf).")

```

```

keras_accuracy = np.mean(keras_classes == test_labels)
keras_f1 = compute_macro_f1_score(test_labels, keras_preds)

# --- Start Detailed Report ---
report_lines = []
report_lines.append("CNN FROM SCRATCH EVALUATION")
report_lines.append("=" * 50 + "\n")
report_lines.append(f"Keras Model Path: {model_path}\n")

report_lines.append(
    f"Implementation match accuracy (classes): "
{implementation_accuracy:.6f}"
)
report_lines.append(
    f"Mean absolute error (MAE) between prediction
probabilities: {mae:.8f}\n"
)
report_lines.append("Test set metrics:")
report_lines.append(
    f" From scratch - Accuracy: {scratch_accuracy:.4f}, F1
Score (Macro): {scratch_f1:.4f}"
)
report_lines.append(
    f" Keras - Accuracy: {keras_accuracy:.4f}, F1
Score (Macro): {keras_f1:.4f}\n"
)

# Use class_names directly from load_cifar10_data()
if scratch_classes is not None:
    report_lines.append("From scratch - Classification
Report:")
    report_lines.append(
        classification_report(
            test_labels,
            scratch_classes,
            target_names=class_names,
            zero_division=0,
        )
    )
    report_lines.append("\n")
else:
    report_lines.append(
        "From scratch - Classification Report: Not

```

```

available due to invalid predictions.\n"
    )

report_lines.append("Keras - Classification Report:")
report_lines.append(
    classification_report(
        test_labels, keras_classes,
target_names=class_names, zero_division=0
    )
)
report_lines.append("\n")

if scratch_classes is not None:
    cm_scratch_obj = confusion_matrix(test_labels,
scratch_classes)
    report_lines.append("From scratch - Confusion Matrix:")
    report_lines.append(np.array2string(cm_scratch_obj,
separator=", "))
    report_lines.append("\n")
else:
    report_lines.append(
        "From scratch - Confusion Matrix: Not available due
to invalid predictions.\n"
    )

cm_keras_obj = confusion_matrix(test_labels, keras_classes)
report_lines.append("Keras - Confusion Matrix:")
report_lines.append(np.array2string(cm_keras_obj,
separator=", "))
report_lines.append("\n")

# Print all detailed reports to console
print("\n--- Detailed From Scratch Implementation Report
---")
for line in report_lines:
    print(line)
print("--- End of Detailed Report ---")

# Visualization only if scratch predictions are valid
if scratch_classes is not None and scratch_preds is not
None:
    # Create and display confusion matrices
    plt.figure(figsize=(13, 5.5))

```

```

plt.subplot(1, 2, 1)
plt.imshow(cm_scratch_obj, interpolation="nearest",
cmap=plt.cm.Blues)
plt.title("From Scratch Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names, rotation=45,
ha="right")
plt.yticks(tick_marks, class_names)
for i in range(cm_scratch_obj.shape[0]):
    for j in range(cm_scratch_obj.shape[1]):
        plt.text(
            j,
            i,
            format(cm_scratch_obj[i, j], "d"),
            horizontalalignment="center",
            color=(
                "white"
                if cm_scratch_obj[i, j] >
cm_scratch_obj.max() / 2.0
                else "black"
            ),
        )
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.tight_layout()

plt.subplot(1, 2, 2)
plt.imshow(cm_keras_obj, interpolation="nearest",
cmap=plt.cm.Blues)
plt.title("Keras Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names, rotation=45,
ha="right")
plt.yticks(tick_marks, class_names)
for i in range(cm_keras_obj.shape[0]):
    for j in range(cm_keras_obj.shape[1]):
        plt.text(
            j,
            i,
            format(cm_keras_obj[i, j], "d"),
            horizontalalignment="center",
            color=

```

```

        "white"
        if cm_keras_obj[i, j] >
cm_keras_obj.max() / 2.0
            else "black"
        ) ,
    )
plt.ylabel("True label")
plt.xlabel("Predicted label")

plt.suptitle("Confusion Matrix Comparison",
fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
plt.close()

# Compare prediction distributions
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.hist(np.max(scratch_preds, axis=1), bins=20,
alpha=0.7, label="Scratch")
plt.title("Scratch: Max Probability Distribution")
plt.xlabel("Max Probability")
plt.ylabel("Count")
plt.legend()

plt.subplot(1, 2, 2)
plt.hist(
    np.max(keras_preds, axis=1),
    bins=20,
    alpha=0.7,
    label="Keras",
    color="orange",
)
plt.title("Keras: Max Probability Distribution")
plt.xlabel("Max Probability")
plt.ylabel("Count")
plt.legend()

plt.suptitle("Maximum Probability Distribution
Comparison", fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
plt.close()

```

```

# Compare individual predictions
plt.figure(figsize=(10, 6))
sample_size = min(20, len(test_images))
if len(test_images) > 0:
    sample_indices = np.random.choice(
        len(test_images), sample_size, replace=False
    )
    if scratch_preds is not None and len(scratch_preds) == len(keras_preds):
        scratch_sample_preds =
scratch_preds[sample_indices]
        keras_sample_preds =
keras_preds[sample_indices]
        abs_diffs = np.abs(scratch_sample_preds -
keras_sample_preds)
        mean_diffs_per_sample = np.mean(abs_diffs,
axis=1)

        plt.bar(range(sample_size),
mean_diffs_per_sample)
        plt.title("Average Absolute Probability
Difference per Sample")
        plt.xlabel("Random Sample Index")
        plt.ylabel("MAE Probability")
        plt.xticks(
            range(sample_size),
            sample_indices.astype(str),
            rotation=45,
            ha="right",
        )
        plt.grid(axis="y", linestyle="--", alpha=0.7)
        plt.tight_layout()
        plt.show()
    else:
        print(
            "Cannot plot prediction differences because
scratch_preds is invalid or length mismatch."
        )
else:
    print("No test data available to plot individual
prediction differences.")
    plt.close()

# Additional visualization: Show some sample images

```

```

with predictions
    plt.figure(figsize=(15, 8))
    sample_size = min(8, len(test_images))
    sample_indices = np.random.choice(len(test_images),
sample_size, replace=False)

    for i, idx in enumerate(sample_indices):
        plt.subplot(2, 4, i + 1)

        # Display CIFAR-10 image (32x32x3 RGB format)
        img = test_images[idx]
        plt.imshow(img) # CIFAR-10 images are already in
RGB format

        true_label = class_names[test_labels[idx]]
        scratch_pred = class_names[scratch_classes[idx]]
        keras_pred = class_names[keras_classes[idx]]

        title = f"True: {true_label}\nScratch:
{scratch_pred}\nKeras: {keras_pred}"
        plt.title(title, fontsize=8)
        plt.axis('off')

        plt.suptitle("Sample Predictions Comparison",
fontsize=14)
        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.show()
        plt.close()

    else:
        print("Plotting skipped due to invalid scratch
predictions.")

return scratch_cnn

```

2.2.2. Simple Recurrent Neural Network (Simple RNN)

2.2.2.1. Konfigurasi (simplernn/config.py)

Pengembangan model Simple Recurrent Neural Network (Simple RNN) untuk tugas klasifikasi sentimen pada dataset NusaX-Sentiment diawali dengan penetapan berbagai parameter konfigurasi. Parameter-parameter ini disimpan secara terpusat dalam sebuah file config.py untuk memudahkan pengelolaan, modifikasi, dan reproduktifitas eksperimen. Penggunaan file konfigurasi ini memastikan bahwa semua

bagian dari kode, mulai dari pra-pemrosesan data, pembangunan model, hingga proses pelatihan, menggunakan pengaturan yang konsisten, kecuali saat parameter tersebut secara eksplisit diubah untuk keperluan analisis hyperparameter.

Berikut adalah rincian singkat parameter yang didefinisikan beserta kegunaannya:

1. Parameter Pemrosesan Data:

- MAX_TOKENS: Jumlah maksimum token unik dalam vocabulary model.
- OUTPUT_SEQ_LEN: Panjang standar untuk semua sekuens input (dipotong/di-padding).

2. Parameter Pelatihan:

- BATCH_SIZE: Jumlah sampel per iterasi pelatihan.
- EPOCHS: Jumlah total siklus pelatihan pada keseluruhan dataset training.
- RANDOM_SEED: Nilai seed untuk inisialisasi acak demi reproduktifitas.

3. Arsitektur Model Default:

- EMBEDDING_DIM: Dimensi vektor yang merepresentasikan setiap token.
- RNN_UNITS: Jumlah unit (sel) dalam setiap layer Simple RNN.
- NUM_RNN_LAYERS: Jumlah layer Simple RNN yang disusun bertumpuk.
- BIDIRECTIONAL: Menentukan apakah RNN memproses sekuens dua arah atau satu arah.

4. Parameter Regularisasi (untuk mencegah overfitting):

- DROPOUT_RATE: Proporsi unit yang di-dropout setelah layer RNN.
- EMBEDDING_DROPOUT: Proporsi unit yang di-dropout (SpatialDropout1D) setelah layer embedding.
- RECURRENT_DROPOUT: Proporsi unit yang di-dropout pada koneksi rekuren di dalam sel RNN.
- L2_REG: Faktor regularisasi L2 untuk bobot layer.

5. Parameter Optimisasi (untuk Adam optimizer dan penjadwalan learning rate):

- LEARNING_RATE: Tingkat pembelajaran awal untuk optimizer Adam.
- LR_FACTOR: Faktor pengurangan learning rate jika performa validasi stagnan.
- LR_PATIENCE: Jumlah epoch menunggu sebelum learning rate dikurangi.
- MIN_LR: Batas minimum learning rate saat pengurangan.

6. Parameter Early Stopping (untuk menghentikan pelatihan jika tidak ada perbaikan):

- ES_PATIENCE: Jumlah epoch menunggu sebelum pelatihan dihentikan jika metrik validasi tidak membaik.

```
config.py

# Parameter pemrosesan data
MAX_TOKENS = 8000
OUTPUT_SEQ_LEN = 80

# Parameter training
BATCH_SIZE = 24
EPOCHS = 30
RANDOM_SEED = 42

# Arsitektur model default
EMBEDDING_DIM = 96
RNN_UNITS = 48
NUM_RNN_LAYERS = 1
BIDIRECTIONAL = True

# Parameter regularisasi - untuk mencegah overfitting
DROPOUT_RATE = 0.3
EMBEDDING_DROPOUT = 0.2
RECURRENT_DROPOUT = 0.1
L2_REG = 0.01

# Parameter optimisasi
LEARNING_RATE = 0.001
LR_FACTOR = 0.7
LR_PATIENCE = 3
MIN_LR = 0.00001

# Parameter early stopping
ES_PATIENCE = 6
```

2.2.2.2. Pra-pemrosesan Data (simplernn/data_preprocessing.py)

Tahap pra-pemrosesan data merupakan langkah krusial sebelum data teks dapat digunakan untuk melatih model Simple Recurrent Neural Network (Simple RNN). Tujuannya adalah untuk mengubah data teks mentah dari dataset NusaX-Sentiment menjadi format numerik yang dapat dipahami dan diproses oleh jaringan saraf. Semua fungsi terkait pra-pemrosesan data ini diorganisir dalam file data_preprocessing.py.

Proses ini melibatkan dua fungsi utama:

1. `load_data()`

Fungsi ini bertanggung jawab untuk membaca dataset dari file CSV yang telah dipisahkan menjadi data latih (train.csv), data validasi (valid.csv), dan data uji (test.csv). Selanjutnya, fungsi ini melakukan pemetaan label sentimen yang berupa teks (misalnya, "negative", "neutral", "positive") ke representasi numerik integer (0, 1, 2). Konversi ini diperlukan karena model neural network umumnya bekerja dengan input dan output numerik, dan loss function SparseCategoricalCrossentropy mengharapkan label target dalam bentuk integer. Output dari fungsi ini adalah tuple yang berisi teks dan label yang sudah dikonversi untuk masing-masing split data (train, valid, test), beserta informasi pemetaan label dan jumlah kelas.

2. `def create_text_vectorizer(train_texts, max_tokens, output_sequence_length):`

Fungsi ini bertugas untuk mengubah sekuens teks menjadi sekuens integer (tokenisasi). Hal ini dicapai dengan memanfaatkan layer TextVectorization dari Keras. Layer TextVectorization pertama-tama diinisialisasi dengan parameter konfigurasi seperti `max_tokens` dan `output_sequence_length`.

Kemudian, metode `adapt()` pada objek TextVectorization dipanggil menggunakan data teks dari training set. Proses adapt ini bertujuan agar vectorizer mempelajari vocabulary (kosakata) dari data training, termasuk frekuensi kemunculan setiap kata untuk menentukan token mana saja yang akan dimasukkan ke dalam vocabulary hingga batas `max_tokens`. Fungsi ini mengembalikan objek TextVectorization yang telah "dilatih", vocabulary yang telah dipelajari, dan ukuran vocabulary tersebut. Vectorizer ini kemudian digunakan untuk mengonversi semua data teks (train, valid, dan test) menjadi sekuens integer sebelum dimasukkan ke layer embedding model.

3. `compute_f1_score(y_true, y_pred)`

Selain dua fungsi utama tersebut, file `data_preprocessing.py` juga menyertakan fungsi utilitas yakni `compute_f1_score` yang digunakan untuk menghitung macro F1-score, yang merupakan metrik evaluasi utama untuk membandingkan performa model dalam tugas ini.

```
data_preprocessing.py

import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import TextVectorization
from sklearn.metrics import f1_score, classification_report
```

```

import config

def load_data():
    train_df = pd.read_csv("data/train.csv")
    valid_df = pd.read_csv("data/valid.csv")
    test_df = pd.read_csv("data/test.csv")

    label_mapping = {"negative": 0, "neutral": 1, "positive": 2}

    train_labels = np.array([label_mapping[label] for label in train_df["label"]])
    valid_labels = np.array([label_mapping[label] for label in valid_df["label"]])
    test_labels = np.array([label_mapping[label] for label in test_df["label"]])

    return (
        (train_df["text"].values, train_labels),
        (valid_df["text"].values, valid_labels),
        (test_df["text"].values, test_labels),
        label_mapping,
        len(label_mapping),
    )

def create_text_vectorizer(
    train_texts,
    max_tokens=config.MAX_TOKENS,
    output_sequence_length=config.OUTPUT_SEQ_LEN,
):
    vectorizer = TextVectorization(
        max_tokens=max_tokens,
        output_mode="int",
        output_sequence_length=output_sequence_length,
        name="text_vectorizer",
    )

    vectorizer.adapt(train_texts)

    vocab = vectorizer.get_vocabulary()
    vocab_size = len(vocab)

```

```

    return vectorizer, vocab, vocab_size

def compute_f1_score(y_true, y_pred):
    y_pred_classes = np.argmax(y_pred, axis=1)
    return f1_score(y_true, y_pred_classes, average="macro")

```

2.2.2.3. Model (simplernn/model.py)

Definisi arsitektur model Simple Recurrent Neural Network (Simple RNN) diimplementasikan dalam file model.py. File ini berisi fungsi create_rnn_model yang bertanggung jawab untuk membangun dan mengompilasi model RNN menggunakan pustaka Keras, sesuai dengan parameter yang diberikan atau nilai default dari config.py. Fleksibilitas fungsi ini memungkinkan eksperimentasi dengan berbagai konfigurasi hyperparameter.

Berikut adalah tahapan dan komponen utama dalam pembentukan model melalui fungsi create_rnn_model (vocab_size, embedding_dim, rnn_units, num_rnn_layers, bidirectional, dropout_rate, l2_reg, num_classes=3, learning_rate):

1. Inisialisasi Model Sequential

Model dibangun sebagai tumpukan layer Keras Sequential dan diberi nama "Simple_RNN_Classifier".

2. Layer Embedding

Layer pertama adalah layers.Embedding. Layer ini bertugas mengubah sekuens token integer (hasil dari TextVectorization) menjadi sekuens vektor padat (dense vectors) dengan dimensi yang ditentukan oleh embedding_dim. input_dim untuk layer ini adalah vocab_size (ukuran vocabulary yang didapat dari TextVectorization), dan output_dim adalah embedding_dim. Regularisasi L2 (embeddings_regularizer=l2(l2_reg)) diterapkan pada bobot embedding untuk membantu mencegah overfitting. Layer ini diberi nama "embedding_layer".

3. Layer SpatialDropout1D

Layer SpatialDropout1D ditambahkan setelah embedding. Layer ini berfungsi sebagai regularisasi dengan men-dropout seluruh dimensi embedding untuk sebuah token secara acak.

4. Layer SimpleRNN (Unidirectional atau Bidirectional)

Satu atau lebih layer SimpleRNN (dengan rnn_units per layer) ditambahkan. Pengaturan return_sequences disesuaikan untuk memungkinkan stacking layer.

Setiap layer SimpleRNN dapat secara opsional dibungkus dengan layer Bidirectional untuk memproses sekuens dari kedua arah. Regularisasi L2 diterapkan pada bobot kernel, rekuren, dan bias, serta recurrent_dropout digunakan untuk koneksi rekuren internal. Layer Dropout standar juga ditambahkan setelah setiap layer RNN (atau Bidirectional RNN).

5. Layer Output (Dense)

Sebuah layer Dense dengan fungsi aktivasi softmax digunakan sebagai layer output untuk klasifikasi multikelas (sebanyak num_classes). Regularisasi L2 juga diterapkan pada layer ini.

6. Kompilasi Model

Model dikompilasi menggunakan optimizer Adam (dengan learning_rate yang dapat diatur dan clipnorm=1.0 untuk gradient clipping), loss function sparse_categorical_crossentropy, dan metrik evaluasi accuracy.

```
model.py
```

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
import config

def create_rnn_model(
    vocab_size,
    embedding_dim=config.EMBEDDING_DIM,
    rnn_units=config.RNN_UNITS,
    num_rnn_layers=config.NUM_RNN_LAYERS,
    bidirectional=config.BIDIRECTIONAL,
    dropout_rate=config.DROPOUT_RATE,
    l2_reg=config.L2_REG,
    num_classes=3,
    learning_rate=config.LEARNING_RATE,
):
    model = models.Sequential(name="Simple_RNN_Classifier")

    model.add(
        layers.Embedding(
            input_dim=vocab_size,
            output_dim=embedding_dim,
```

```

        embeddings_regularizer=l2(l2_reg) ,
        name="embedding_layer",
    )
)

model.add(
    layers.SpatialDropout1D(config.EMBEDDING_DROPOUT,
name="embedding_dropout")
)

for i in range(num_rnn_layers):
    return_sequences = i < num_rnn_layers - 1

    rnn_layer = layers.SimpleRNN(
        rnn_units,
        return_sequences=return_sequences,
        kernel_regularizer=l2(l2_reg),
        recurrent_regularizer=l2(l2_reg),
        bias_regularizer=l2(l2_reg),
        recurrent_dropout=config.RECURRENT_DROPOUT,
        name=f"simplernn_layer_{i+1}",
    )

    if bidirectional:
        model.add(layers.Bidirectional(rnn_layer,
name=f"bidirectional_rnn_{i+1}"))
    else:
        model.add(rnn_layer)

    model.add(layers.Dropout(dropout_rate,
name=f"dropout_rnn_{i+1}"))

model.add(
    layers.Dense(
        num_classes,
        activation="softmax",
        kernel_regularizer=l2(l2_reg),
        bias_regularizer=l2(l2_reg),
        name="output_dense_layer",
    )
)

optimizer = Adam(learning_rate=learning_rate, clipnorm=1.0)
model.compile(

```

```
optimizer=optimizer,  
        loss="sparse_categorical_crossentropy", # Untuk label  
kategorikal dalam bentuk integer  
        metrics=["accuracy"],  
)  
  
return model
```

2.2.2.4. Pelatihan (simplernn/train.py)

Proses pelatihan model Simple Recurrent Neural Network (Simple RNN) dan evaluasinya diatur dalam file train.py. Fungsi utama yang menangani siklus ini adalah train_and_evaluate_model, yang langkah-langkah kerjanya dapat diuraikan sebagai berikut:

1. Persiapan Awal

Fungsi dimulai dengan menerima berbagai parameter konfigurasi sebagai argumen, seperti jumlah layer RNN (num_rnn_layers), jumlah unit per layer (rnn_units), status bidirectional (bidirectional), serta parameter regularisasi dan laju pembelajaran. Selain itu, pada tahap ini, dataset (latih, validasi, dan uji) beserta pemetaan label dimuat menggunakan fungsi load_data() dari modul data_preprocessing.py. Opsi untuk menggunakan pembobotan kelas (use_class_weights) juga dievaluasi di sini; jika aktif, bobot akan dihitung berdasarkan distribusi kelas pada data training untuk menangani potensi ketidakseimbangan data.

2. Vektorisasi Teks

Setelah data dimuat, langkah selanjutnya adalah mempersiapkan teks untuk model. Objek TextVectorization dari Keras dibuat menggunakan fungsi create_text_vectorizer() dari data_preprocessing.py. Vectorizer ini kemudian diadaptasi (fit) ke data teks training untuk membangun vocabulary berdasarkan config.MAX_TOKENS dan config.OUTPUT_SEQ_LEN. Setelah vocabulary terbentuk, semua data teks (training, validasi, dan testing) dikonversi menjadi sekuens integer.

3. Pembuatan tf.data.Dataset

Untuk optimasi dan efisiensi pipeline data selama pelatihan, data sekuens dan label yang sudah berbentuk numerik dikonversi menjadi objek tf.data.Dataset. Untuk dataset training, dilakukan proses pengacakan (shuffle), pembagian menjadi batch (batch sesuai config.BATCH_SIZE), dan pengambilan data di latar

belakang (prefetch). Dataset validasi dan test juga diubah menjadi tf.data.Dataset dengan proses batching dan prefetching.

4. Pembuatan Model

Dengan data yang telah siap, model RNN dibangun menggunakan fungsi `create_rnn_model()` dari modul `model.py`. Fungsi ini menerima parameter konfigurasi yang relevan (seperti ukuran vocabulary, `embedding_dim`, jumlah layer RNN, unit RNN, status bidirectional, `dropout_rate`, `l2_reg`, jumlah kelas, dan `learning_rate`) untuk membentuk arsitektur model sesuai dengan iterasi eksperimen yang sedang berjalan. Ringkasan arsitektur model (`model.summary()`) kemudian ditampilkan.

5. Pengaturan Callbacks

Serangkaian callbacks dari Keras disiapkan untuk memantau dan mengontrol proses pelatihan. Ini termasuk `ModelCheckpoint` yang bertugas menyimpan bobot dari model dengan performa `val_loss` terbaik selama pelatihan, `EarlyStopping` untuk menghentikan proses pelatihan jika `val_loss` tidak menunjukkan perbaikan setelah sejumlah epoch (`config.ES_PATIENCE`) dan mengembalikan bobot terbaik, serta `ReduceLROnPlateau` untuk mengurangi `learning_rate` secara dinamis jika `val_loss` stagnan.

6. Pelatihan Model

Proses pelatihan model dijalankan menggunakan metode `model.fit()`. Metode ini menerima dataset training, dataset validasi (untuk pemantauan performa), jumlah epoch yang ditentukan dalam `config.EPOCHS`, serta daftar callbacks yang telah disiapkan. Jika pembobotan kelas diaktifkan, `class_weight` juga akan disertakan.

7. Evaluasi Model

Setelah pelatihan selesai (baik karena mencapai jumlah epoch maksimum atau dihentikan oleh `EarlyStopping`), model dengan bobot terbaiknya dievaluasi pada dataset test. Metode `model.evaluate()` digunakan untuk mendapatkan nilai `test_loss` dan `test_accuracy`. Kemudian, `model.predict()` digunakan untuk mendapatkan probabilitas prediksi pada data test, yang selanjutnya digunakan untuk menghitung macro F1-score melalui fungsi `compute_f1_score()`. Laporan klasifikasi yang merinci presisi, recall, dan F1-score per kelas juga dicetak.

8. Penyimpanan Model dan Vectorizer

Untuk reproduktifitas dan penggunaan di masa depan, model Keras yang telah dilatih secara penuh (termasuk arsitektur dan bobotnya) disimpan dalam format

.keras ke direktori models/. Selain itu, objek TextVectorization yang telah diadaptasi (yang berisi vocabulary) juga disimpan. Penyimpanan vectorizer ini krusial untuk memastikan bahwa data baru atau data untuk implementasi from scratch diproses dengan cara yang sama persis.

9. Nilai Kembali

Terakhir, fungsi train_and_evaluate_model mengembalikan beberapa artefak penting: objek model Keras yang telah dilatih, objek history yang berisi catatan metrik selama pelatihan, probabilitas prediksi pada data test, label sebenarnya dari data test, objek vectorizer yang digunakan, dan sebuah dictionary yang berisi metrik evaluasi akhir pada data test (test_loss, test_accuracy, test_f1).

Selain train_and_evaluate_model, file train.py juga berisi fungsi plot_training_history yang digunakan untuk memvisualisasikan kurva loss dan akurasi training dan validasi terhadap epoch. Ini membantu dalam menganalisis proses pembelajaran model.

```
train.py
```

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint,
EarlyStopping, ReduceLROnPlateau
from data_preprocessing import load_data,
create_text_vectorizer, compute_f1_score
from model import create_rnn_model
import config
import os

def train_and_evaluate_model(
    num_rnn_layers=config.NUM_RNN_LAYERS,
    rnn_units=config.RNN_UNITS,
    bidirectional=config.BIDIRECTIONAL,
    dropout_rate=config.DROPOUT_RATE,
    l2_reg=config.L2_REG,
    learning_rate=config.LEARNING_RATE,
    model_name="simple_rnn_model",
    use_class_weights=True,
):
    print(f"\n--- Training Model: {model_name} ---")
    print(
```

```

        f"Config: Layers={num_rnn_layers}, Units={rnn_units}, "
        f"Bidirectional={bidirectional},
Dropout={dropout_rate}, L2={l2_reg}"
    )

# Muat dataset
(
    (train_texts, train_labels),
    (valid_texts, valid_labels),
    (test_texts, test_labels),
    label_mapping,
    num_classes,
) = load_data()

print(
    f"Ukuran dataset: Train={len(train_texts)}, "
    f"Valid={len(valid_texts)}, Test={len(test_texts)}"
)
print(f"Distribusi kelas di data train:
{np.bincount(train_labels)}")

# Hitung bobot kelas untuk mengatasi ketidakseimbangan data
if use_class_weights:
    class_counts = np.bincount(train_labels)
    total_samples = len(train_labels)
    class_weights = {
        i: total_samples / (len(class_counts) * count)
        for i, count in enumerate(class_counts)
    }
    print(f"Menggunakan bobot kelas: {class_weights}")
else:
    class_weights = None

# Buat dan siapkan text vectorizer
vectorizer, vocab, vocab_size = create_text_vectorizer(
    train_texts,
    max_tokens=config.MAX_TOKENS,
    output_sequence_length=config.OUTPUT_SEQ_LEN,
)
print(f"Ukuran vocabulary: {vocab_size}")

# Ubah teks jadi sequence angka
train_sequences = vectorizer(train_texts)
valid_sequences = vectorizer(valid_texts)

```

```

test_sequences = vectorizer(test_texts)

# Buat dataset TF untuk efisiensi training
train_dataset =
tf.data.Dataset.from_tensor_slices((train_sequences,
train_labels))
train_dataset = (
    train_dataset.shuffle(len(train_texts) * 4,
reshuffle_each_iteration=True)
    .batch(config.BATCH_SIZE)
    .prefetch(tf.data.AUTOTUNE)
)

valid_dataset =
tf.data.Dataset.from_tensor_slices((valid_sequences,
valid_labels))
valid_dataset =
valid_dataset.batch(config.BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

test_dataset =
tf.data.Dataset.from_tensor_slices((test_sequences,
test_labels))
test_dataset =
test_dataset.batch(config.BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

model = create_rnn_model(
    vocab_size=vocab_size,
    embedding_dim=config.EMBEDDING_DIM,
    rnn_units=rnn_units,
    num_rnn_layers=num_rnn_layers,
    bidirectional=bidirectional,
    dropout_rate=dropout_rate,
    l2_reg=l2_reg,
    num_classes=num_classes,
    learning_rate=learning_rate,
)
if config.OUTPUT_SEQ_LEN is not None:
    model.build(input_shape=(None, config.OUTPUT_SEQ_LEN))
else:
    print("Peringatan: config.OUTPUT_SEQ_LEN tidak disetel,
model.build() dilewati. Summary mungkin unbuilt.")

```

```

model.summary()

checkpoint_path = f"checkpoints/{model_name}.weights.h5"
callbacks = [
    ModelCheckpoint(
        checkpoint_path,
        monitor="val_loss", # Pantau val_loss untuk simpan
model terbaik
        mode="min",
        save_best_only=True,
        save_weights_only=True,
        verbose=1,
    ),
    EarlyStopping(
        monitor="val_loss",
        mode="min",
        patience=config.ES_PATIENCE, # Berhenti kalau
tidak ada perbaikan
        verbose=1,
        restore_best_weights=True,
    ),
    ReduceLROnPlateau(
        monitor="val_loss",
        mode="min",
        factor=config.LR_FACTOR, # Faktor pengurangan
learning rate
        patience=config.LR_PATIENCE,
        min_lr=config.MIN_LR,
        verbose=1,
    ),
]

print("Mulai pelatihan model...")
history = model.fit(
    train_dataset,
    validation_data=valid_dataset,
    epochs=config.EPOCHS,
    callbacks=callbacks,
    verbose=1,
    class_weight=class_weights,
)

print("Evaluasi di test set dengan bobot terbaik...")

```

```

    test_loss, test_acc = model.evaluate(test_dataset,
verbose=1)

    test_pred_probs = model.predict(test_dataset)
    test_f1 = compute_f1_score(test_labels, test_pred_probs)

    print(f"Test Loss: {test_loss:.4f}")
    print(f"Test Accuracy: {test_acc:.4f}")
    print(f"Test Macro F1: {test_f1:.4f}")

    y_pred = np.argmax(test_pred_probs, axis=1)
    from sklearn.metrics import classification_report,
confusion_matrix

    class_names = list(label_mapping.keys())
    print("\nLaporan Klasifikasi:")
    print(
        classification_report(
            test_labels, y_pred, target_names=class_names,
zero_division=0
        )
    )

full_model_path = f"models/{model_name}_full_model.keras"
model.save(full_model_path)
print(f"Model disimpan di: {full_model_path}")

vectorizer_path = f"models/{model_name}_vectorizer.keras"
tf.keras.models.save_model(
    tf.keras.Sequential([vectorizer]),
    name="text_vectorization_model",
    vectorizer_path,
)
print(f"Vectorizer disimpan di: {vectorizer_path}")

return (
    model,
    history,
    test_pred_probs,
    test_labels,
    vectorizer,
    {"test_loss": test_loss, "test_accuracy": test_acc,
"test_f1": test_f1},
)

```

```

def plot_training_history(history, model_name):
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(history.history["loss"], label="Loss Training")
    plt.plot(history.history["val_loss"], label="Loss Validasi")
    plt.title(f"Kurva Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history["accuracy"], label="Akurasi Training")
    plt.plot(history.history["val_accuracy"], label="Akurasi Validasi")
    plt.title(f"Kurva Akurasi")
    plt.xlabel("Epoch")
    plt.ylabel("Akurasi")
    plt.legend()

    plt.tight_layout()
    plt.show()

```

**2.2.2.5. Implementasi Forward Propagation From Scratch
(simplernn/from_scratch.py)**

Implementasi forward propagation from scratch untuk model Simple Recurrent Neural Network (Simple RNN) dikembangkan dalam file `from_scratch.py`. Tujuan utama dari implementasi ini adalah untuk mereplikasi fungsionalitas prediksi dari model Simple RNN yang telah dilatih menggunakan Keras, namun dengan menggunakan operasi-operasi dasar dari pustaka NumPy untuk perhitungan matematisnya. Implementasi ini dikemas dalam kelas `SimpleRNNFromScratch` dan dilengkapi dengan fungsi `run_from_scratch_comparison` untuk validasi.

Kelas SimpleRNNFromScratch

Kelas ini merupakan inti dari implementasi from scratch dan dirancang untuk menangani pemutuan model, ekstraksi bobot, dan eksekusi forward propagation.

1. Inisialisasi (`__init__`)

Pada tahap inisialisasi, objek `SimpleRNNFromScratch` menerima path ke model Keras yang telah disimpan (`keras_model_path`) dan path ke vectorizer yang sesuai (`vectorizer_path`). Konstruktor kemudian memuat model Keras (tanpa mengompilasinya ulang) dan memuat objek `TextVectorization` dari Keras. Atribut `self.weights` (sebuah dictionary) dan `self.rnn_layer_configs` (sebuah list) diinisialisasi untuk menyimpan bobot dan konfigurasi layer RNN yang akan diekstrak. Proses utama dalam inisialisasi adalah pemanggilan metode `self.extract_weights_from_keras_model()` untuk mengisi atribut-atribut tersebut.

2. Ekstraksi Bobot Model Keras (`extract_weights_from_keras_model`)

Metode ini bertanggung jawab untuk mengiterasi melalui setiap layer pada model Keras yang telah dimuat. Untuk setiap layer, metode ini mengidentifikasi jenisnya (Embedding, Bidirectional SimpleRNN, SimpleRNN, atau Dense) dan mengekstrak bobot-bobot yang relevan (misalnya, bobot embedding, kernel input, kernel rekuren, dan bias untuk layer RNN, serta kernel dan bias untuk layer Dense). Bobot-bobot ini disimpan dalam dictionary `self.weights` dengan kunci yang deskriptif. Untuk layer RNN, metadata tambahan seperti tipe (unidirectional/bidirectional) dan nama layer disimpan dalam list `self.rnn_layer_configs` untuk memandu proses forward propagation nantinya, terutama jika terdapat beberapa layer RNN yang bertumpuk.

3. Forward Propagation Layer Embedding (`embedding_forward`)

Metode ini mengimplementasikan forward pass untuk layer Embedding. Berdasarkan sekuens token integer input (indices), metode ini melakukan lookup pada matriks bobot embedding (`self.weights["embedding"]`) yang telah diekstrak untuk menghasilkan representasi vektor untuk setiap token dalam sekuens.

4. Forward Propagation Satu Pass Simple RNN (_simple_rnn_pass)

Metode ini merupakan implementasi inti dari satu pass (maju atau mundur) untuk sebuah layer Simple RNN. Metode ini menerima input sekuens (hasil dari layer sebelumnya), bobot kernel input (kernel), bobot kernel rekuren (recurrent_kernel), dan bias (bias). Secara iteratif untuk setiap timestep dalam sekuens, metode ini menghitung *hidden state* baru h_t menggunakan rumus :

$$h_t = \tanh(x_t \cdot W_{ih} + h_{t-1} \cdot W_{hh} + b)$$

di mana x_t adalah input pada timestep t , h_{t-1} adalah *hidden state* dari timestep sebelumnya, W_{ih} adalah bobot kernel input, dan W_{hh} adalah bobot kernel rekuren. Metode ini juga menangani parameter `return_sequences` (apakah mengembalikan semua *hidden state* atau hanya yang terakhir) dan `go_backwards` (apakah memproses sekuens dari belakang ke depan, yang penting untuk layer backward dalam Bidirectional RNN).

5. Forward Propagation Layer Dense (dense_forward)

Metode ini mengimplementasikan forward pass untuk layer Dense output. Ia menerima input dari layer RNN terakhir, kemudian melakukan perkalian matriks dengan bobot kernel output (kernel) dan menambahkan bias (bias). Hasilnya (logit) kemudian dilewatkan melalui fungsi aktivasi softmax (yang juga diimplementasikan di sini dengan stabilisasi numerik) untuk menghasilkan distribusi probabilitas atas kelas-kelas sentimen.

6. Forward Propagation Keseluruhan Model (forward)

Metode ini mengorkestrasi keseluruhan proses forward propagation dari awal hingga akhir. Dimulai dengan melewatkannya sekuens input melalui `embedding_forward`. Kemudian, output embedding diproses secara berurutan oleh setiap layer RNN yang telah dikonfigurasi (disimpan dalam `self.rnn_layer_configs`). Untuk setiap layer RNN, metode ini menentukan apakah layer tersebut unidirectional atau bidirectional, mengambil bobot yang sesuai, dan memanggil `_simple_rnn_pass` (satu kali untuk unidirectional, atau dua kali – sekali maju dan sekali mundur – untuk bidirectional). Jika layer RNN adalah bidirectional, output dari pass maju dan mundur akan digabungkan (`concatenate`). Output dari layer RNN terakhir kemudian dimasukkan ke `dense_forward` untuk menghasilkan probabilitas prediksi akhir.

7. Prediksi Teks (predict)

Metode ini menerima daftar teks mentah sebagai input. Pertama, teks tersebut diubah menjadi sekuens token integer menggunakan self.vectorizer yang telah dimuat. Sekuens integer ini kemudian dilewatkan ke metode self.forward untuk mendapatkan prediksi probabilitas.

8. Perbandingan dengan Keras (compare_with_keras)

Metode ini dirancang untuk memvalidasi implementasi from scratch. Ia mengambil teks input, melakukan prediksi menggunakan metode self.predict (from scratch) dan juga menggunakan self.keras_model.predict() (dari Keras). Kemudian, metode ini membandingkan waktu eksekusi, akurasi kecocokan kelas prediksi, dan Mean Absolute Error (MAE) antara probabilitas prediksi dari kedua implementasi.

Fungsi run_from_scratch_comparison

Fungsi pendukung ini berada di luar kelas SimpleRNNFromScratch dan bertugas untuk:

1. Memuat data test.
2. Menginisialisasi objek SimpleRNNFromScratch dengan path model dan vectorizer Keras yang ditentukan.
3. Memanggil metode compare_with_keras pada objek tersebut untuk mendapatkan prediksi dari kedua implementasi dan metrik perbandingan awal.
4. Menghitung metrik evaluasi akhir (seperti akurasi dan macro F1-score) pada data test untuk prediksi dari implementasi from scratch dan Keras.
5. Menampilkan laporan detail yang mencakup metrik kecocokan, metrik performa pada data test (termasuk classification report dan confusion matrix), dan perbandingan waktu prediksi.
6. Menyediakan opsi untuk memvisualisasikan confusion matrix, distribusi probabilitas prediksi, dan perbedaan prediksi individual antara implementasi from scratch dan Keras untuk analisis lebih mendalam.

```
from_scratch.py

import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    f1_score as sklearn_f1_score,
)
```

```

import time
import os

# Load modul lokal
from data_preprocessing import (
    load_data,
    compute_f1_score,
)

class SimpleRNNFromScratch:
    def __init__(self, keras_model_path, vectorizer_path):
        print(f"Loading Keras model from: {keras_model_path}")
        self.keras_model =
tf.keras.models.load_model(keras_model_path, compile=False)

        # Muat vectorizer untuk pemrosesan teks
        print(f"Loading vectorizer from: {vectorizer_path}")
        self.vectorizer_model = tf.keras.models.load_model(
            vectorizer_path, compile=False
        )
        self.vectorizer = self.vectorizer_model.layers[0]

        # Ekstrak bobot dari setiap layer model Keras
        self.weights = {}
        self.rnn_layer_configs = []
        self.extract_weights_from_keras_model()

        print("Model loaded and weights extracted
successfully.")

    def extract_weights_from_keras_model(self):
        print("\nExtracting weights from Keras model...")
        rnn_layer_counter = 0
        for layer in self.keras_model.layers:
            layer_name = layer.name
            print(f"Processing layer: {layer_name} (Type:
{type(layer).__name__})")

            if isinstance(layer, tf.keras.layers.Embedding):
                if layer_name == "embedding_layer":
                    self.weights["embedding"] =
layer.get_weights()[0]
                    print(

```

```

        f" Extracted embedding weights, shape:
{self.weights['embedding'].shape}"
    )

        elif isinstance(layer,
tf.keras.layers.Bidirectional):
            if isinstance(layer.forward_layer,
tf.keras.layers.SimpleRNN):
                rnn_layer_counter += 1
                config = {
                    "type": "bidirectional",
                    "name": layer_name,
                    "keras_layer_name": layer_name, # Nama
layer Bidirectional di Keras
                    "weights_prefix":
f"bidir_rnn_{rnn_layer_counter}_",
                }

                forward_rnn = layer.forward_layer
                backward_rnn = layer.backward_layer

                prefix = config["weights_prefix"]
                self.weights[prefix + "fwd_kernel"] =
forward_rnn.get_weights()[0]
                self.weights[prefix +
"fwd_recurrent_kernel"] = (
                    forward_rnn.get_weights()[1]
                )
                self.weights[prefix + "fwd_bias"] =
forward_rnn.get_weights()[2]

                self.weights[prefix + "bwd_kernel"] =
backward_rnn.get_weights()[0]
                self.weights[prefix +
"bwd_recurrent_kernel"] = (
                    backward_rnn.get_weights()[1]
                )
                self.weights[prefix + "bwd_bias"] =
backward_rnn.get_weights()[2]

                self.rnn_layer_configs.append(config)
                print(
                    f" Extracted Bidirectional SimpleRNN
weights for '{layer_name}' with prefix '{prefix}'"

```

```

        )
        print(
            f"      Forward kernel shape:
{self.weights[prefix + 'fwd_kernel'].shape}"
        )
        print(
            f"      Backward kernel shape:
{self.weights[prefix + 'bwd_kernel'].shape}"
        )

        elif isinstance(layer, tf.keras.layers.SimpleRNN)
and not any(
            layer_name in cfg["keras_layer_name"]
            for cfg in self.rnn_layer_configs
            if "keras_layer_name" in cfg and layer_name in
cfg["keras_layer_name"]
        ):
            is_part_of_bidirectional = False
            for keras_layer_outer in
self.keras_model.layers:
                if isinstance(
                    keras_layer_outer,
tf.keras.layers.Bidirectional
                ) and (
                    layer ==
keras_layer_outer.forward_layer
                    or layer ==
keras_layer_outer.backward_layer
                ):
                    is_part_of_bidirectional = True
                    break
            if is_part_of_bidirectional:
                continue

            rnn_layer_counter += 1
            config = {
                "type": "unidirectional",
                "name": layer_name, # Nama layer SimpleRNN
di Keras
                "keras_layer_name": layer_name,
                "weights_prefix":
f"unidir_rnn_{rnn_layer_counter}_",
            }

```

```

        prefix = config["weights_prefix"]
        self.weights[prefix + "kernel"] =
layer.get_weights() [0]
        self.weights[prefix + "recurrent_kernel"] =
layer.get_weights() [1]
        self.weights[prefix + "bias"] =
layer.get_weights() [2]

        self.rnn_layer_configs.append(config)
        print(
            f" Extracted Unidirectional SimpleRNN
weights for '{layer_name}' with prefix '{prefix}'"
        )
        print(f" Kernel shape: {self.weights[prefix
+ 'kernel'].shape}")

    elif isinstance(layer, tf.keras.layers.Dense):
        if layer_name == "output_dense_layer":
            self.weights["output_kernel"] =
layer.get_weights() [0]
            self.weights["output_bias"] =
layer.get_weights() [1]
            print(
                f" Extracted output dense layer
weights. Kernel: {self.weights['output_kernel'].shape}, Bias:
{self.weights['output_bias'].shape}"
            )

        print(
            f"Finished weight extraction. Found
{len(self.rnn_layer_configs)} RNN layer configurations."
        )

def embedding_forward(self, indices):
    return self.weights["embedding"] [indices]

def _simple_rnn_pass(
    self,
    inputs,
    kernel,
    recurrent_kernel,
    bias,
    return_sequences=True,
    go_backwards=False,

```

```

) :
    batch_size, seq_length, _ = inputs.shape
    units = bias.shape[0]

    h_t = np.zeros((batch_size, units))

    if return_sequences:
        outputs = np.zeros((batch_size, seq_length, units))

    time_steps = range(seq_length)
    if go_backwards:
        time_steps = range(seq_length - 1, -1, -1)

    for t in time_steps:
        x_t = inputs[:, t, :]
        h_t = np.tanh(np.dot(x_t, kernel) + np.dot(h_t,
recurrent_kernel) + bias)
        if return_sequences:
            if go_backwards:
                outputs[:, t, :] = h_t
            else:
                outputs[:, t, :] = h_t

    return outputs if return_sequences else h_t

def dense_forward(self, inputs, kernel, bias):
    logits = np.dot(inputs, kernel) + bias
    exp_logits = np.exp(logits - np.max(logits, axis=1,
keepdims=True))
    probabilities = exp_logits / np.sum(exp_logits, axis=1,
keepdims=True)
    return probabilities

def forward(self, text_sequences):
    # 1. Embedding Layer
    current_tensor = self.embedding_forward(text_sequences)

    # 2. RNN Layers (Bidirectional or Unidirectional)
    num_rnn_layers = len(self.rnn_layer_configs)
    for i, rnn_config in enumerate(self.rnn_layer_configs):
        is_last_rnn_in_stack = i == num_rnn_layers - 1
        return_sequences_for_this_layer = not
is_last_rnn_in_stack

```

```

prefix = rnn_config["weights_prefix"]

if rnn_config["type"] == "bidirectional":
    fwd_kernel = self.weights[prefix +
"fwd_kernel"]
    fwd_recurrent_kernel = self.weights[prefix +
"fwd_recurrent_kernel"]
    fwd_bias = self.weights[prefix + "fwd_bias"]

    bwd_kernel = self.weights[prefix +
"bwd_kernel"]
    bwd_recurrent_kernel = self.weights[prefix +
"bwd_recurrent_kernel"]
    bwd_bias = self.weights[prefix + "bwd_bias"]

# Forward pass
h_fwd = self._simple_rnn_pass(
    current_tensor,
    fwd_kernel,
    fwd_recurrent_kernel,
    fwd_bias,
    return_sequences=True,
    go_backwards=False,
)

# Backward pass
h_bwd = self._simple_rnn_pass(
    current_tensor,
    bwd_kernel,
    bwd_recurrent_kernel,
    bwd_bias,
    return_sequences=True,
    go_backwards=True,
)

if return_sequences_for_this_layer:
    current_tensor = np.concatenate([h_fwd,
h_bwd], axis=-1)
else:
    current_tensor = np.concatenate(
        [h_fwd[:, -1, :], h_bwd[:, 0, :]],
axis=-1
)

```

```

        elif rnn_config["type"] == "unidirectional":
            kernel = self.weights[prefix + "kernel"]
            recurrent_kernel = self.weights[prefix +
"recurrent_kernel"]
            bias = self.weights[prefix + "bias"]

            current_tensor = self._simple_rnn_pass(
                current_tensor,
                kernel,
                recurrent_kernel,
                bias,

return_sequences=return_sequences_for_this_layer,
                go_backwards=False,
            )

rnn_output = current_tensor

# 3. Output Dense Layer
output = self.dense_forward(
    rnn_output, self.weights["output_kernel"],
self.weights["output_bias"]
)
return output

def predict(self, texts):
    sequences = self.vectorizer(texts).numpy()
    return self.forward(sequences)

def compare_with_keras(self, texts):
    start_time_scratch = time.time()
    scratch_preds = self.predict(texts)
    scratch_time = time.time() - start_time_scratch

    start_time_keras = time.time()
    sequences_tf = self.vectorizer(texts)
    keras_preds = self.keras_model.predict(sequences_tf)
    keras_time = time.time() - start_time_keras

    scratch_classes = np.argmax(scratch_preds, axis=1)
    keras_classes = np.argmax(keras_preds, axis=1)

    # Periksa apakah ada NaN atau Inf dalam prediksi
scratch

```

```

        if np.any(np.isnan(scratch_preds)) or
np.any(np.isinf(scratch_preds)):
            print("WARNING: NaN or Inf found in scratch
predictions!")
            implementation_accuracy = 0.0
            mae = float("inf")
        else:
            implementation_accuracy = np.mean(scratch_classes
== keras_classes)
            mae = np.mean(np.abs(scratch_preds - keras_preds))

        print(f"\nPerbandingan waktu prediksi:")
        print(f"From scratch: {scratch_time:.4f} detik")
        print(f"Keras: {keras_time:.4f} detik")
        if keras_time > 0:
            print(f"Ratio waktu (scratch/keras):
{scratch_time/keras_time:.2f}x")
        else:
            print(f"Ratio waktu (scratch/keras): N/A (Keras
time is zero)")

        print(
            f"\nAkurasi kecocokan implementasi (kelas):
{implementation_accuracy:.6f}"
        )
        print(f"Mean absolute error antar probabilitas
prediksi: {mae:.8f}")

    return scratch_preds, keras_preds,
implementation_accuracy, mae

```

2.2.3. Long-Short Term Memory Network (LSTM)

2.2.3.1. Konfigurasi (lstm/config.py)

Pengembangan model Long Short-Term Memory (LSTM) untuk tugas klasifikasi sentimen pada dataset NusaX-Sentiment diawali dengan penetapan berbagai parameter konfigurasi yang esensial. Parameter-parameter ini disimpan secara terpusat dalam sebuah file config.py yang didedikasikan untuk model LSTM. Tujuan dari sentralisasi konfigurasi ini adalah untuk menjamin konsistensi, mempermudah modifikasi parameter, dan mendukung reproduktifitas hasil dalam setiap tahapan pengembangan, mulai dari pra-pemrosesan data hingga proses pelatihan model.

Berikut adalah rincian singkat parameter yang didefinisikan dalam config.py untuk model LSTM beserta kegunaannya:

1. Parameter Pemrosesan Data:

- MAX_TOKENS: Jumlah maksimum token unik dalam vocabulary model.
- OUTPUT_SEQ_LEN: Panjang standar untuk semua sekuens input (dipotong/di-padding).

2. Parameter Pelatihan:

- BATCH_SIZE: Jumlah sampel per iterasi pelatihan.
- EPOCHS: Jumlah total siklus pelatihan pada keseluruhan dataset training.
- RANDOM_SEED: Nilai seed untuk inisialisasi acak demi reproduktifitas.

3. Arsitektur Model Default:

- EMBEDDING_DIM: Dimensi dari vektor embedding yang akan merepresentasikan setiap token dalam sekuens input.
- LSTM_UNITS: Jumlah unit atau sel memori yang terdapat dalam setiap layer LSTM.
- NUM_LSTM_LAYERS: Jumlah default layer LSTM yang akan disusun secara bertumpuk (stacked) dalam model.
- BIDIRECTIONAL_LSTM: Menentukan apakah LSTM memproses sekuens dua arah atau satu arah.

4. Parameter Regularisasi (untuk mencegah overfitting):

- DROPOUT_RATE: Proporsi unit yang di-dropout setelah layer LSTM.
- EMBEDDING_DROPOUT: Proporsi unit yang di-nonaktifkan pada layer SpatialDropout1D yang diterapkan setelah layer embedding.
- RECURRENT_DROPOUT_LSTM: Proporsi unit yang di-nonaktifkan pada koneksi rekuren di dalam sel LSTM itu sendiri, sebagai bentuk regularisasi internal.
- L2_REG: Faktor regularisasi L2 (lambda) yang diterapkan pada bobot-bobot (kernel, rekuren, dan bias) dari layer-layer utama untuk membatasi kompleksitas model dan mencegah bobot menjadi terlalu besar.

5. Parameter Optimisasi (untuk Adam optimizer dan penjadwalan learning rate):

- LEARNING_RATE: Nilai tingkat pembelajaran awal untuk optimizer Adam.
- LR_FACTOR: Faktor pengurangan learning rate jika performa validasi stagnan.
- LR_PATIENCE: Jumlah epoch yang akan ditunggu sebelum learning rate dikurangi jika tidak ada perbaikan performa yang signifikan.

- MIN_LR: Batas terendah yang diizinkan untuk learning rate selama proses pengurangan dinamis.

6. Parameter Early Stopping (untuk menghentikan pelatihan jika tidak ada perbaikan):

- ES_PATIENCE: Jumlah epoch yang akan ditunggu sebelum proses pelatihan dihentikan apabila tidak ada perbaikan pada metrik validasi yang dipantau

```
config.py

# Parameter pemrosesan data
MAX_TOKENS = 8000
OUTPUT_SEQ_LEN = 80

# Parameter training
BATCH_SIZE = 32
EPOCHS = 24
RANDOM_SEED = 42

# Arsitektur model LSTM default
EMBEDDING_DIM = 128
LSTM_UNITS = 64
NUM_LSTM_LAYERS = 2
BIDIRECTIONAL_LSTM = True

# Parameter regularisasi
DROPOUT_RATE = 0.2
EMBEDDING_DROPOUT = 0.1
RECURRENT_DROPOUT_LSTM = 0.05 # Dropout internal di LSTM
L2_REG = 0.0001

# Parameter optimisasi
LEARNING_RATE = 0.001
LR_FACTOR = 0.7
LR_PATIENCE = 3
MIN_LR = 0.00001

# Parameter early stopping
ES_PATIENCE = 6
```

2.2.3.2. Pra-pemrosesan Data (lstm/data_preprocessing.py)

Tahap pra-pemrosesan data merupakan langkah krusial sebelum data teks dapat digunakan untuk melatih model Long Short-Term Memory (LSTM). Tujuannya adalah untuk mengubah data teks mentah dari dataset NusaX-Sentiment menjadi format numerik yang dapat dipahami dan diproses oleh jaringan saraf. Semua fungsi terkait pra-pemrosesan data ini diorganisir dalam file data_preprocessing.py.

Proses ini melibatkan dua fungsi utama:

1. load_data()

Fungsi ini bertanggung jawab untuk membaca dataset dari file CSV yang telah dipisahkan menjadi data latih (train.csv), data validasi (valid.csv), dan data uji (test.csv). Selanjutnya, fungsi ini melakukan pemetaan label sentimen yang berupa teks (misalnya, "negative", "neutral", "positive") ke representasi numerik integer (0, 1, 2). Konversi ini diperlukan karena model neural network umumnya bekerja dengan input dan output numerik, dan loss function SparseCategoricalCrossentropy mengharapkan label target dalam bentuk integer. Output dari fungsi ini adalah tuple yang berisi teks dan label yang sudah dikonversi untuk masing-masing split data (train, valid, test), beserta informasi pemetaan label dan jumlah kelas.

2. def create_text_vectorizer(train_texts, max_tokens, output_sequence_length):

Fungsi ini bertugas untuk mengubah sekuens teks menjadi sekuens integer (tokenisasi). Hal ini dicapai dengan memanfaatkan layer TextVectorization dari Keras. Layer TextVectorization pertama-tama diinisialisasi dengan parameter konfigurasi seperti max_tokens dan output_sequence_length.

Kemudian, metode adapt() pada objek TextVectorization dipanggil menggunakan data teks dari training set. Proses adapt ini bertujuan agar vectorizer mempelajari vocabulary (kosakata) dari data training, termasuk frekuensi kemunculan setiap kata untuk menentukan token mana saja yang akan dimasukkan ke dalam vocabulary hingga batas max_tokens. Fungsi ini mengembalikan objek TextVectorization yang telah "dilatih", vocabulary yang telah dipelajari, dan ukuran vocabulary tersebut. Vectorizer ini kemudian digunakan untuk mengonversi semua data teks (train, valid, dan test) menjadi sekuens integer sebelum dimasukkan ke layer embedding model.

3. compute_f1_score(y_true, y_pred)

Selain dua fungsi utama tersebut, file data_preprocessing.py juga menyertakan fungsi utilitas yakni compute_f1_score yang digunakan untuk menghitung macro

F1-score, yang merupakan metrik evaluasi utama untuk membandingkan performa model dalam tugas ini.

```
data_preprocessing.py

import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import TextVectorization
from sklearn.metrics import f1_score
import config

def load_data(base_path="data") :
    train_df = pd.read_csv(f"{base_path}/train.csv")
    valid_df = pd.read_csv(f"{base_path}/valid.csv")
    test_df = pd.read_csv(f"{base_path}/test.csv")

    label_mapping = {"negative": 0, "neutral": 1, "positive": 2}

    train_labels = np.array([label_mapping[label] for label in train_df["label"]])
    valid_labels = np.array([label_mapping[label] for label in valid_df["label"]])
    test_labels = np.array([label_mapping[label] for label in test_df["label"]])

    return (
        (train_df["text"].values, train_labels),
        (valid_df["text"].values, valid_labels),
        (test_df["text"].values, test_labels),
        label_mapping,
        len(label_mapping),
    )

def create_text_vectorizer(
    train_texts,
    max_tokens=config.MAX_TOKENS,
    output_sequence_length=config.OUTPUT_SEQ_LEN,
) :
    vectorizer = TextVectorization(
        max_tokens=max_tokens,
```

```

        output_mode="int",
        output_sequence_length=output_sequence_length,
        name="text_vectorizer_lstm",
    )
vectorizer.adapt(train_texts)
vocab = vectorizer.get_vocabulary()
vocab_size = len(vocab)
return vectorizer, vocab, vocab_size

def compute_f1_score(y_true, y_pred_probs):
    y_pred_classes = np.argmax(y_pred_probs, axis=1)
    return f1_score(y_true, y_pred_classes, average="macro",
zero_division=0)

```

2.2.3.3. Model (lstm/model.py)

Definisi arsitektur model Long Short-Term Memory (LSTM) untuk tugas klasifikasi sentimen dirancang dalam file model.py. File ini berisi fungsi utama create_lstm_model yang bertanggung jawab untuk membangun dan mengompilasi model LSTM menggunakan Keras. Fungsi ini menerima berbagai parameter konfigurasi, sebagian besar diambil dari file config.py (seperti dimensi embedding, jumlah unit LSTM, jumlah layer, status bidirectional, dan parameter regularisasi), sehingga memungkinkan pembuatan model dengan arsitektur yang fleksibel untuk keperluan eksperimentasi.

Berikut adalah tahapan dan komponen utama dalam pembentukan model LSTM melalui fungsi create_lstm_model:

1. Inisialisasi Model Sequential

Model LSTM dibangun sebagai sebuah tumpukan layer models.Sequential dari Keras, yang diberi nama "LSTM_Classifier". Struktur sekuensial ini memungkinkan penambahan layer-layer secara berurutan.

2. Layer Embedding dan Spatial Dropout

Layer pertama, layers.Embedding, mengubah sekuens token integer input menjadi sekuens vektor padat dengan dimensi embedding_dim. Layer ini dapat diberikan regularisasi L2 pada bobotnya. Untuk regularisasi tambahan pada representasi embedding, sebuah layer layers.SpatialDropout1D dapat ditambahkan setelahnya, yang bekerja dengan men-dropout seluruh dimensi fitur untuk setiap token.

3. Layer LSTM (Unidirectional atau Bidirectional) dan Dropout Antar Layer

Satu atau lebih layer inti layers.LSTM ditambahkan, masing-masing dengan lstm_units sel memori. Pengaturan return_sequences dikelola secara otomatis untuk mendukung penumpukan (stacking) layer LSTM. Setiap layer LSTM dapat dibungkus dengan layers.Bidirectional untuk memungkinkan pemrosesan sekuens dari kedua arah (maju dan mundur), sehingga menangkap konteks yang lebih kaya. Untuk mengontrol overfitting, regularisasi L2 dapat diterapkan pada berbagai bobot LSTM, recurrent_dropout digunakan untuk koneksi internal sel LSTM, dan layer layers.Dropout standar dapat ditambahkan setelah setiap layer LSTM atau Bidirectional LSTM.

4. Layer Output (Dense)

Sebagai tahap akhir untuk klasifikasi, sebuah layer layers.Dense dengan jumlah unit sesuai num_classes (jumlah kelas sentimen) ditambahkan. Layer ini menggunakan fungsi aktivasi "softmax" untuk menghasilkan distribusi probabilitas atas kelas-kelas target. Regularisasi L2 juga dapat diterapkan pada bobot layer Dense ini.

5. Kompilasi Model

Model LSTM yang telah tersusun kemudian dikompilasi dengan menetapkan optimizer Adam (yang dikonfigurasi dengan learning_rate dari config.py dan clipnorm=1.0 untuk gradient clipping), loss function "sparse_categorical_crossentropy" yang cocok untuk klasifikasi multikelas dengan label integer, serta metrik "accuracy" untuk dipantau selama pelatihan dan evaluasi.

Setelah dikompilasi, fungsi create_lstm_model mengembalikan objek model Keras yang siap untuk dilatih menggunakan data teks yang telah melalui tahap pra-pemrosesan.

```
model.py
```

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
import config

def create_lstm_model(
    vocab_size,
    embedding_dim=config.EMBEDDING_DIM,
    lstm_units=config.LSTM_UNITS,
```

```

        num_lstm_layers=config.NUM_LSTM_LAYERS,
        bidirectional=config.BIDIRECTIONAL_LSTM,
        dropout_rate=config.DROPOUT_RATE,
        embedding_dropout_rate=config.EMBEDDING_DROPOUT,
        recurrent_dropout_rate=config.RECURRENT_DROPOUT_LSTM,
        l2_reg_strength=config.L2_REG,
        num_classes=3,
        learning_rate=config.LEARNING_RATE,
    ) :
    model = models.Sequential(name="LSTM_Classifier")

    def l2_regularizer(strength):
        return l2(strength) if strength > 0 else None

    # 1. Embedding Layer
    model.add(
        layers.Embedding(
            input_dim=vocab_size,
            output_dim=embedding_dim,
            embeddings_regularizer=l2_regularizer(
                l2_reg_strength
            ),
            name="embedding_layer",
        )
    )
    if embedding_dropout_rate > 0:
        model.add(
            layers.SpatialDropout1D(embedding_dropout_rate,
name="embedding_dropout")
        )

    # 2. LSTM Layers
    for i in range(num_lstm_layers):
        return_sequences = i < num_lstm_layers - 1

        lstm_layer = layers.LSTM(
            lstm_units,
            return_sequences=return_sequences,
            kernel_regularizer=l2_regularizer(l2_reg_strength),
            recurrent_regularizer=l2_regularizer(l2_reg_strength),
            bias_regularizer=l2_regularizer(l2_reg_strength),
            recurrent_dropout=recurrent_dropout_rate,
            name=f"lstm_layer_{i+1}",

```

```

        )

    if bidirectional:
        model.add(
            layers.Bidirectional(lstm_layer,
name=f"bidirectional_lstm_{i+1}")
        )
    else:
        model.add(lstm_layer)

    # Dropout setelah setiap layer LSTM/Bidirectional LSTM
    if dropout_rate > 0:
        model.add(layers.Dropout(dropout_rate,
name=f"dropout_lstm_{i+1}"))

# 3. Dense Output Layer
model.add(
    layers.Dense(
        num_classes,
        activation="softmax",
        kernel_regularizer=l2_regularizer(l2_reg_strength),
        bias_regularizer=l2_regularizer(l2_reg_strength),
        name="output_dense_layer",
    )
)
)

# Kompilasi model
optimizer = Adam(learning_rate=learning_rate, clipnorm=1.0)
model.compile(
    optimizer=optimizer,
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
return model

```

2.2.3.4. Pelatihan (lstm/train.py)

Proses pelatihan dan evaluasi untuk model Long Short-Term Memory (LSTM) diatur dalam file train.py yang spesifik untuk LSTM. File ini mencakup fungsi utama train_and_evaluate_lstm_model yang mengelola seluruh siklus pelatihan, evaluasi, dan penyimpanan artefak model, serta fungsi plot_training_history untuk visualisasi kurva pembelajaran. Berikut adalah langkah-langkah kerja utama dari fungsi

`train_and_evaluate_lstm_model:`

1. Persiapan Awal dan Konfigurasi

Fungsi ini diawali dengan menerima berbagai parameter untuk konfigurasi model LSTM, seperti jumlah layer LSTM (`num_lstm_layers`), jumlah unit per layer (`lstm_units`), status bidirectional (`bidirectional`), serta berbagai tingkat dropout (`dropout_rate`, `embedding_dropout_rate`, `recurrent_dropout_rate`), kekuatan regularisasi L2 (`l2_reg`), dan laju pembelajaran (`learning_rate`). Nama model (`model_name`) juga diterima untuk penamaan file output. Setelah itu, dataset (`latih`, `validasi`, dan `uji`) beserta pemetaan label dimuat menggunakan fungsi `load_data()` dari `data_preprocessing.py`. Jika parameter `use_class_weights` diatur `True`, fungsi akan menghitung bobot untuk setiap kelas berdasarkan distribusinya di data training, yang bertujuan untuk mengatasi potensi ketidakseimbangan kelas selama proses pelatihan.

2. Vektorisasi Teks

Langkah berikutnya adalah mempersiapkan data teks. Objek `TextVectorization` dari Keras dibuat dan diadaptasi ke data teks training menggunakan fungsi `create_text_vectorizer()` dari `data_preprocessing.py`. Proses adaptasi ini membangun `vocabulary` berdasarkan `config.MAX_TOKENS` dan mengatur `config.OUTPUT_SEQ_LEN` sebagai panjang sekuens output. Setelah `vocabulary` terbentuk, semua data teks (training, validasi, dan testing) dikonversi menjadi sekuens integer.

3. Pembuatan tf.data.Dataset

Untuk efisiensi dalam pipeline input data selama pelatihan, sekuens integer dan label yang sesuai dikonversi menjadi objek `tf.data.Dataset`. Dataset training secara khusus di-shuffle (dengan `config.RANDOM_SEED` untuk reproduktifitas), dibagi menjadi batch sesuai `config.BATCH_SIZE`, dan di-prefetch untuk mengoptimalkan pemuatian data. Dataset validasi dan test juga diubah menjadi `tf.data.Dataset` dengan proses batching dan prefetching.

4. Pembuatan Model LSTM Keras

Dengan data yang telah siap, model LSTM dibangun dengan memanggil fungsi `create_lstm_model()` dari modul `model.py` (yang spesifik untuk LSTM). Fungsi ini menerima parameter konfigurasi yang relevan (seperti ukuran `vocabulary`, `embedding_dim`, `lstm_units`, `num_lstm_layers`, `bidirectional`, berbagai `dropout_rate`, `l2_reg_strength`, jumlah kelas, dan `learning_rate`) untuk membentuk arsitektur model LSTM sesuai dengan iterasi eksperimen yang sedang

dijalankan. Ringkasan arsitektur model (model.summary()) kemudian ditampilkan untuk verifikasi.

5. Pengaturan Callbacks Keras

Serangkaian callbacks dari Keras disiapkan untuk memantau dan mengendalikan proses pelatihan secara dinamis. Ini termasuk:

- ModelCheckpoint: Untuk menyimpan bobot dari model dengan performa val_loss terbaik selama pelatihan ke path checkpoints/{model_name}.weights.h5. Hanya bobot terbaik yang disimpan (save_best_only=True, save_weights_only=True).
- EarlyStopping: Untuk menghentikan proses pelatihan jika val_loss tidak menunjukkan perbaikan setelah sejumlah epoch yang ditentukan oleh config.ES_PATIENCE. Callback ini juga akan mengembalikan bobot terbaik yang telah disimpan.
- ReduceLROnPlateau: Untuk mengurangi learning_rate secara otomatis (dengan faktor config.LR_FACTOR) jika val_loss stagnan selama periode config.LR_PATIENCE epoch, hingga batas minimum config.MIN_LR.

6. Pelatihan Model LSTM

Proses pelatihan model LSTM dijalankan menggunakan metode model.fit(). Metode ini menerima dataset training, dataset validasi (untuk pemantauan performa selama pelatihan), jumlah epoch yang ditentukan dalam config.EPOCHS, daftar callbacks yang telah disiapkan, dan class_weight jika diaktifkan. Durasi pelatihan juga dicatat.

7. Evaluasi Model pada Data Test

Setelah pelatihan selesai, model dengan bobot terbaiknya dievaluasi pada dataset test. Metode model.evaluate() digunakan untuk mendapatkan nilai test_loss dan test_accuracy. Selanjutnya, model.predict() digunakan untuk mendapatkan probabilitas prediksi pada data test, yang kemudian digunakan untuk menghitung macro F1-score melalui fungsi compute_f1_score() dari data_preprocessing.py. Hasil metrik evaluasi dan laporan klasifikasi yang merinci presisi, recall, dan F1-score per kelas juga ditampilkan.

8. Penyimpanan Model dan Vectorizer

Model Keras LSTM yang telah dilatih secara penuh (termasuk arsitektur dan bobotnya) disimpan dalam format .keras ke direktori models/ dengan nama file {model_name}_full_model.keras. Selain itu, objek TextVectorization yang telah diadaptasi juga disimpan sebagai model Keras terpisah. Penyimpanan kedua

artefak ini penting untuk memastikan konsistensi dan kemudahan penggunaan kembali di masa mendatang, termasuk untuk implementasi from scratch.

9. Pengembalian Hasil

Terakhir, fungsi `train_and_evaluate_lstm_model` mengembalikan beberapa output penting: objek model LSTM Keras yang telah dilatih, objek history yang berisi catatan metrik selama pelatihan (loss dan akurasi untuk training dan validasi per epoch), probabilitas prediksi pada data test, label sebenarnya dari data test, objek vectorizer yang digunakan, dan sebuah dictionary yang berisi metrik evaluasi akhir pada data test (`test_loss`, `test_accuracy`, `test_f1`, dan `training_time`).

Selain fungsi utama tersebut, file `train.py` juga menyertakan fungsi `plot_training_history` yang digunakan untuk memvisualisasikan kurva loss dan akurasi dari proses pelatihan, yang sangat membantu dalam menganalisis bagaimana model LSTM belajar dan mengidentifikasi potensi overfitting.

```
train.py
```

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint,
EarlyStopping, ReduceLROnPlateau
import os
import time
import config
from data_preprocessing import load_data,
create_text_vectorizer, compute_f1_score
from model import create_lstm_model


def train_and_evaluate_lstm_model(
    num_lstm_layers=config.NUM_LSTM_LAYERS,
    lstm_units=config.LSTM_UNITS,
    bidirectional=config.BIDIRECTIONAL_LSTM,
    dropout_rate=config.DROPOUT_RATE,
    embedding_dropout_rate=config.EMBEDDING_DROPOUT,
    recurrent_dropout_rate=config.RECURRENT_DROPOUT_LSTM,
    l2_reg=config.L2_REG,
    learning_rate=config.LEARNING_RATE,
    model_name="lstm_model",
    use_class_weights=True,
```

```

) :
    print(f"\n--- Training LSTM Model: {model_name} ---")
    print(
        f"Config: Layers={num_lstm_layers}, Units={lstm_units},
"
        f"Bidirectional={bidirectional},
Dropout={dropout_rate}, EmbDropout={embedding_dropout_rate},
RecDropout={recurrent_dropout_rate}, L2={l2_reg}"
    )

# Muat dataset
(
    (train_texts, train_labels),
    (valid_texts, valid_labels),
    (test_texts, test_labels),
    label_mapping,
    num_classes,
) = load_data()

print(
    f"Ukuran dataset: Train={len(train_texts)}, "
    f"Valid={len(valid_texts)}, Test={len(test_texts)}"
)
print(f"Distribusi kelas di data train:
{np.bincount(train_labels)}")

# Hitung bobot kelas untuk mengatasi ketidakseimbangan data
if use_class_weights:
    class_counts = np.bincount(train_labels)
    total_samples = len(train_labels)
    class_weights = {
        i: total_samples / (len(class_counts) * count) if
count > 0 else 0
        for i, count in enumerate(class_counts)
    }
    print(f"Menggunakan bobot kelas: {class_weights}")
else:
    class_weights = None

# Buat dan siapkan text vectorizer
vectorizer, vocab, vocab_size = create_text_vectorizer(
    train_texts,
    max_tokens=config.MAX_TOKENS,
    output_sequence_length=config.OUTPUT_SEQ_LEN,

```

```

    )
print(f"Ukuran vocabulary: {vocab_size}")

# Ubah teks jadi sequence angka
train_sequences = vectorizer(train_texts)
valid_sequences = vectorizer(valid_texts)
test_sequences = vectorizer(test_texts)

# Buat dataset TF untuk efisiensi training
train_dataset =
tf.data.Dataset.from_tensor_slices((train_sequences,
train_labels))
train_dataset = (
    train_dataset.shuffle(
        len(train_texts) * 4,
reshuffle_each_iteration=True, seed=config.RANDOM_SEED
    )
    .batch(config.BATCH_SIZE)
    .prefetch(tf.data.AUTOTUNE)
)

valid_dataset =
tf.data.Dataset.from_tensor_slices((valid_sequences,
valid_labels))
valid_dataset =
valid_dataset.batch(config.BATCH_SIZE).prefetch(tf.data.AUTOTUN
E)

test_dataset =
tf.data.Dataset.from_tensor_slices((test_sequences,
test_labels))
test_dataset =
test_dataset.batch(config.BATCH_SIZE).prefetch(tf.data.AUTOTUNE
)

# Buat model LSTM sesuai konfigurasi
model = create_lstm_model(
    vocab_size=vocab_size,
    embedding_dim=config.EMBEDDING_DIM,
    lstm_units=lstm_units,
    num_lstm_layers=num_lstm_layers,
    bidirectional=bidirectional,
    dropout_rate=dropout_rate,
    embedding_dropout_rate=embedding_dropout_rate,
)

```

```

        recurrent_dropout_rate=recurrent_dropout_rate,
        l2_reg_strength=l2_reg,
        num_classes=num_classes,
        learning_rate=learning_rate,
    )

    if config.OUTPUT_SEQ_LEN is not None:
        model.build(input_shape=(None, config.OUTPUT_SEQ_LEN))
    else:
        print("Peringatan: config.OUTPUT_SEQ_LEN tidak disetel,
model.build() dilewati. Summary mungkin unbuilt.")

    # Tampilkan ringkasan model
    model.summary()

    os.makedirs("checkpoints", exist_ok=True)
    checkpoint_path = f"checkpoints/{model_name}.weights.h5"

    callbacks = [
        ModelCheckpoint(
            checkpoint_path,
            monitor="val_loss",
            mode="min",
            save_best_only=True,
            save_weights_only=True,
            verbose=1,
        ),
        EarlyStopping(
            monitor="val_loss",
            mode="min",
            patience=config.ES_PATIENCE,
            verbose=1,
            restore_best_weights=True,
        ),
        ReduceLROnPlateau(
            monitor="val_loss",
            mode="min",
            factor=config.LR_FACTOR,
            patience=config.LR_PATIENCE,
            min_lr=config.MIN_LR,
            verbose=1,
        ),
    ]

```

```

# Latih model
print("Mulai pelatihan model LSTM...")
start_train_time = time.time()
history = model.fit(
    train_dataset,
    validation_data=valid_dataset,
    epochs=config.EPOCHS,
    callbacks=callbacks,
    verbose=1,
    class_weight=class_weights,
)
training_duration = time.time() - start_train_time
print(f"Pelatihan selesai dalam {training_duration:.2f} detik.")

# Evaluasi di test set
print("Evaluasi di test set dengan bobot terbaik...")
test_loss, test_acc = model.evaluate(test_dataset,
verbose=1)

# Ambil prediksi untuk menghitung F1
test_pred_probs = model.predict(test_dataset)
test_f1 = compute_f1_score(
    test_labels, test_pred_probs
)

print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
print(f"Test Macro F1: {test_f1:.4f}")

# Hitung dan tampilkan metrik per kelas
y_pred_classes = np.argmax(test_pred_probs, axis=1)
from sklearn.metrics import (
    classification_report,
)

class_names = list(label_mapping.keys())
print("\nLaporan Klasifikasi:")
print(
    classification_report(
        test_labels, y_pred_classes,
        target_names=class_names, zero_division=0
    )
)

```

```

os.makedirs("models", exist_ok=True)
full_model_path = f"models/{model_name}_full_model.keras"
model.save(full_model_path)
print(f"Model lengkap disimpan di: {full_model_path}")

# Simpan vectorizer
vectorizer_path = f"models/{model_name}_vectorizer.keras"
# Buat model Sequential sementara hanya untuk menyimpan
TextVectorization layer
vectorizer_export_model = tf.keras.Sequential(
    [vectorizer], name="text_vectorization_model_lstm"
)
vectorizer_export_model.save(vectorizer_path,
save_format="keras")
print(f"Vectorizer disimpan di: {vectorizer_path}")

return (
    model,
    history,
    test_pred_probs,
    test_labels,
    vectorizer,
    {
        "test_loss": test_loss,
        "test_accuracy": test_acc,
        "test_f1": test_f1,
        "training_time": training_duration,
    },
)
)

def plot_training_history(history, model_name):
    """Plot dan simpan kurva history training"""
    plt.figure(figsize=(12, 5))

    # Plot Loss
    plt.subplot(1, 2, 1)
    plt.plot(history.history["loss"], label="Loss Training")
    plt.plot(history.history["val_loss"], label="Loss Validasi")
    plt.title(f"Kurva Loss - {model_name}")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")

```

```

plt.legend()
plt.grid(True)

# Plot Accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history["accuracy"], label="Akurasi Training")
plt.plot(history.history["val_accuracy"], label="Akurasi Validasi")
plt.title(f"Kurva Akurasi - {model_name}")
plt.xlabel("Epoch")
plt.ylabel("Akurasi")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

```

2.2.3.5. Implementasi Forward Propagation From Scratch ([lstm/from_scratch_lstm.py](#))

Implementasi forward propagation from scratch untuk model Long Short-Term Memory (LSTM) dikembangkan dalam file `from_scratch_lstm.py`. Tujuannya adalah untuk mereplikasi fungsionalitas prediksi dari model LSTM yang telah dilatih menggunakan Keras, dengan mengandalkan operasi-operasi dasar dari pustaka NumPy untuk semua perhitungan matematis inti. Fungsionalitas ini dikemas dalam kelas `LSTMFromScratch` dan didukung oleh fungsi `run_lstm_from_scratch_comparison` untuk proses validasi. File ini juga mendefinisikan fungsi aktivasi sigmoid dan tanh kustom menggunakan NumPy.

Kelas `LSTMFromScratch`

Kelas ini menjadi inti dari implementasi from scratch untuk LSTM, menangani pemuat model, ekstraksi bobot, dan eksekusi forward propagation secara modular.

1. Inisialisasi (`__init__`)

Konstruktor kelas `LSTMFromScratch` menerima path ke file model Keras LSTM yang telah disimpan (`keras_model_path`) dan path ke file vectorizer Keras (`vectorizer_path`). Saat inisialisasi, model Keras dan vectorizer dimuat. Atribut-atribut penting seperti `self.weights` (sebuah dictionary untuk menyimpan semua bobot model) dan `self.lstm_layer_configs` (sebuah list untuk menyimpan konfigurasi setiap layer LSTM dalam model) diinisialisasi. Proses utama yang

dipanggil selanjutnya adalah `self.extract_weights_from_keras_model()` untuk mengisi atribut-atribut tersebut dengan informasi dari model Keras yang dimuat.

2. Ekstraksi Bobot Model Keras (`extract_weights_from_keras_model`)

Metode ini bertugas mengiterasi setiap layer dalam model Keras LSTM yang telah dimuat. Ia mengidentifikasi jenis layer seperti Embedding, Bidirectional (yang membungkus layer LSTM), LSTM (standalone/unidirectional), dan Dense. Untuk setiap layer yang relevan, bobot-bobotnya (misalnya, matriks embedding, kernel input, kernel rekuren, dan bias untuk layer LSTM, serta kernel dan bias untuk layer Dense) diekstrak dan disimpan dalam dictionary `self.weights` dengan kunci yang unik dan deskriptif. Untuk layer LSTM, metadata seperti tipe (unidirectional/bidirectional), nama layer Keras, indeks tumpukan (`stack_idx` untuk menangani urutan layer LSTM yang bertumpuk), awalan nama bobot, dan status `return_sequences` dari layer Keras asli disimpan dalam list `self.lstm_layer_configs`. Daftar konfigurasi ini kemudian diurutkan berdasarkan `stack_idx` untuk memastikan pemrosesan layer LSTM dalam urutan yang benar selama forward pass.

3. Forward Propagation Layer Embedding (`embedding_forward`)

Metode ini mengimplementasikan operasi forward pass untuk layer Embedding. Berdasarkan sekuens token integer input (indices), metode ini melakukan lookup pada matriks bobot embedding (`self.weights["embedding"]`) untuk menghasilkan representasi vektor untuk setiap token dalam sekuens input.

4. Forward Propagation Satu Sel LSTM (`_lstm_cell_forward`)

Ini adalah metode inti yang mengimplementasikan kalkulasi untuk satu sel LSTM pada satu timestep. Metode ini menerima input pada timestep saat ini (x_t), hidden state dari timestep sebelumnya (h_{prev}), cell state dari timestep sebelumnya (c_{prev}), serta matriks bobot gabungan untuk input (W_{all}), matriks bobot gabungan untuk hidden state rekuren (U_{all}), dan vektor bias gabungan (b_{all}). Di dalamnya, dilakukan perhitungan untuk keempat gate LSTM (input gate i_t , forget gate f_t , output gate o_t) dan *candidate cell state* \tilde{c}_t . Matriks bobot dan bias gabungan akan di-slice untuk mendapatkan bobot dan bias spesifik untuk masing-masing gate. Setelah itu, cell state baru dihitung dengan rumus :

$$c_t = f_t \odot c_{prev} + i_t \odot \tilde{c}_t$$

Dan hidden state baru dihitung dengan rumus :

$$h_t = o_t \odot \tanh(c_t)$$

Fungsi aktivasi sigmoid dan tanh digunakan sesuai dengan mekanisme standar LSTM.

5. Forward Propagation Satu Pass Layer LSTM (`_lstm_pass`)

Metode ini mengelola proses forward pass untuk satu layer LSTM penuh (baik untuk pass maju maupun mundur) melalui seluruh sekuens input. Ia mengiterasi setiap timestep dalam sekuens, memanggil `_lstm_cell_forward` pada setiap langkah untuk menghitung hidden state dan cell state baru. Metode ini juga menangani parameter `return_sequences` (apakah mengembalikan semua hidden state dari setiap timestep atau hanya hidden state terakhir) dan parameter `go_backwards` (apakah memproses sekuens dari arah belakang ke depan, yang digunakan untuk pass mundur pada Bidirectional LSTM). Output hidden state disimpan dalam urutan waktu alami meskipun diproses secara terbalik jika `go_backwards` aktif.

6. Forward Propagation Layer Dense (`dense_forward`)

Metode ini menjalankan forward pass untuk layer Dense output. Prosesnya meliputi perkalian matriks antara input (dari layer LSTM terakhir) dengan bobot kernel output, penambahan bias, dan aplikasi fungsi aktivasi softmax (dengan stabilisasi numerik) untuk menghasilkan distribusi probabilitas atas kelas-kelas target.

7. Forward Propagation Keseluruhan Model (`forward`)

Metode ini mengorkestrasi keseluruhan alur forward propagation dari input sekuens token hingga output probabilitas prediksi. Dimulai dengan melewatkannya input melalui `embedding_forward`. Output `embedding` kemudian diproses secara sekuensial oleh setiap layer LSTM yang terdaftar dalam `self.lstm_layer_configs`. Untuk setiap konfigurasi layer LSTM, metode ini menentukan apakah layer tersebut unidirectional atau bidirectional, mengambil bobot yang sesuai dari `self.weights`, dan memanggil `_lstm_pass` (satu kali untuk unidirectional, atau dua kali – sekali maju dan sekali mundur – untuk bidirectional). Jika layer LSTM adalah bidirectional, output dari pass maju dan mundur akan digabungkan (`concatenate`). Cara penggabungan (seluruh sekuens atau hanya state terakhir) bergantung pada konfigurasi `return_sequences` dari layer Keras asli yang disimpan dalam `config_entry`. Output dari tumpukan layer LSTM terakhir kemudian dimasukkan ke `dense_forward` untuk menghasilkan prediksi akhir.

8. Prediksi Teks (`predict`)

Metode ini menerima daftar teks mentah sebagai input. Pertama, teks tersebut diubah menjadi sekuens token integer menggunakan objek self.vectorizer yang telah dimuat. Sekuens integer ini kemudian dilewatkan ke metode self.forward untuk mendapatkan prediksi probabilitas dari model LSTM from scratch.

9. Perbandingan dengan Keras (compare_with_keras)

Metode ini bertujuan untuk memvalidasi kebenaran implementasi LSTM from scratch. Ia mengambil teks input, melakukan prediksi menggunakan metode self.predict (implementasi from scratch) dan juga menggunakan self.keras_model.predict() (prediksi dari model Keras asli). Kemudian, metode ini membandingkan waktu eksekusi, akurasi kecocokan kelas prediksi (berdasarkan argmax dari probabilitas), dan Mean Absolute Error (MAE) antara probabilitas prediksi dari kedua implementasi. Pengecekan terhadap nilai NaN atau Inf pada prediksi from scratch juga dilakukan.

Fungsi run_lstm_from_scratch_comparison:

Fungsi pendukung ini berada di luar kelas LSTMFromScratch dan bertugas untuk:

1. Memuat data test menggunakan load_data().
2. Menginisialisasi objek LSTMFromScratch dengan memberikan path ke model Keras LSTM dan vectorizer yang relevan.
3. Memanggil metode compare_with_keras pada objek LSTMFromScratch untuk mendapatkan prediksi dari kedua implementasi (Keras dan from scratch) serta metrik perbandingan awal seperti MAE dan akurasi kecocokan implementasi.
4. Menghitung metrik evaluasi akhir pada data test, seperti akurasi keseluruhan dan macro F1-score, untuk prediksi dari kedua implementasi.
5. Menampilkan laporan detail yang mencakup metrik kecocokan implementasi, perbandingan metrik performa pada data test (termasuk classification report dan confusion matrix untuk kedua implementasi), dan perbandingan waktu prediksi.
6. Menyediakan fungsionalitas untuk memvisualisasikan confusion matrix, distribusi probabilitas prediksi, dan perbedaan prediksi individual antara implementasi from scratch dan Keras, untuk analisis yang lebih mendalam.

```
from_scratch_lstm.py

import numpy as np
import tensorflow as tf
import time
import os
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report,
```

```

confusion_matrix

# Impor dari modul lokal di direktori yang sama
import config
from data_preprocessing import (
    load_data,
    compute_f1_score,
)

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

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

class LSTMFromScratch:
    def __init__(self, keras_model_path, vectorizer_path):
        print(f"Loading Keras LSTM model from: {keras_model_path}")
        self.keras_model =
            tf.keras.models.load_model(keras_model_path, compile=False)

        print(f"Loading vectorizer from: {vectorizer_path}")
        self.vectorizer_model = tf.keras.models.load_model(
            vectorizer_path, compile=False
        )
        self.vectorizer = self.vectorizer_model.layers[0]

        self.weights = {}
        self.lstm_layer_configs = []
        self.extract_weights_from_keras_model()
        print("LSTM Model loaded and weights extracted successfully.")

    def extract_weights_from_keras_model(self):
        print("\nExtracting weights from Keras LSTM model...")
        lstm_stack_idx = 0

        for layer in self.keras_model.layers:
            layer_name = layer.name

```

```

        print(f"Processing layer: {layer_name} (Type:
{type(layer).__name__})")

        if isinstance(layer, tf.keras.layers.Embedding):
            if layer.name == "embedding_layer":
                self.weights["embedding"] =
layer.get_weights()[0]
                print(
                    f" Extracted embedding weights, shape:
{self.weights['embedding'].shape}"
                )

            elif isinstance(layer,
tf.keras.layers.Bidirectional) and isinstance(
                layer.forward_layer, tf.keras.layers.LSTM
            ):
                lstm_stack_idx += 1
                fwd_lstm_layer = layer.forward_layer
                bwd_lstm_layer = layer.backward_layer

                config_entry = {
                    "type": "bidirectional",
                    "keras_layer_name": layer_name,
                    "stack_idx": lstm_stack_idx,
                    "fwd_prefix":
f"bidir_fwd_lstm_{lstm_stack_idx}_",
                    "bwd_prefix":
f"bidir_bwd_lstm_{lstm_stack_idx}_",
                    "return_sequences": layer.return_sequences,
                }
                self.lstm_layer_configs.append(config_entry)

                # Bobot Forward LSTM
                f_kernel, f_recurrent_kernel, f_bias =
fwd_lstm_layer.get_weights()
                self.weights[config_entry["fwd_prefix"]] +
"kernel" = f_kernel
                self.weights[config_entry["fwd_prefix"]] +
"recurrent_kernel" = (
                    f_recurrent_kernel
                )
                self.weights[config_entry["fwd_prefix"]] +
"bias" = f_bias
                print(

```

```

        f" Extracted Bidirectional Forward LSTM
({fwd_lstm_layer.name}) weights with prefix
'{config_entry['fwd_prefix']}'"
    )

        # Bobot Backward LSTM
        b_kernel, b_recurrent_kernel, b_bias =
bwd_lstm_layer.get_weights()
        self.weights[config_entry["bwd_prefix"]] +
"kernel"] = b_kernel
        self.weights[config_entry["bwd_prefix"]] +
"recurrent_kernel"] = (
            b_recurrent_kernel
        )
        self.weights[config_entry["bwd_prefix"]] +
"bias"] = b_bias
        print(
            f" Extracted Bidirectional Backward LSTM
({bwd_lstm_layer.name}) weights with prefix
'{config_entry['bwd_prefix']}'"
        )

    elif isinstance(layer, tf.keras.layers.LSTM) and
not any(
        layer.name
        in (
            self.keras_model.get_layer(
                cfg["keras_layer_name"]
            ).forward_layer.name,
            self.keras_model.get_layer(
                cfg["keras_layer_name"]
            ).backward_layer.name,
        )
        for cfg in self.lstm_layer_configs
        if cfg["type"] == "bidirectional"
    ):

        lstm_stack_idx += 1
        config_entry = {
            "type": "unidirectional",
            "keras_layer_name": layer_name,
            "stack_idx": lstm_stack_idx,
            "prefix": f"uni_lstm_{lstm_stack_idx}_",
            "return_sequences": layer.return_sequences,

```

```

        }
        self.lstm_layer_configs.append(config_entry)

        kernel, recurrent_kernel, bias =
layer.get_weights()
        self.weights[config_entry["prefix"] + "kernel"] =
kernel
        self.weights[config_entry["prefix"] +
"recurrent_kernel"] = (
            recurrent_kernel
        )
        self.weights[config_entry["prefix"] + "bias"] =
bias
        print(
            f" Extracted Unidirectional LSTM
({layer_name}) weights with prefix '{config_entry['prefix']}' "
        )

        elif isinstance(layer, tf.keras.layers.Dense):
            if layer.name == "output_dense_layer":
                self.weights["output_kernel"] =
layer.get_weights() [0]
                self.weights["output_bias"] =
layer.get_weights() [1]
                print(
                    f" Extracted output dense layer
weights. Kernel: {self.weights['output_kernel'].shape}, Bias:
{self.weights['output_bias'].shape}"
                )

        # Urutkan konfigurasi layer LSTM berdasarkan stack_idx
untuk memastikan urutan pemrosesan yang benar
        self.lstm_layer_configs.sort(key=lambda x:
x["stack_idx"])
        print(
            f"Finished weight extraction. Found
{len(self.lstm_layer_configs)} LSTM stack configurations."
        )

    def embedding_forward(self, indices):
        return self.weights["embedding"] [indices]

    def _lstm_cell_forward(self, x_t, h_prev, c_prev, W_all,
U_all, b_all):

```

```

units = U_all.shape[0]

# Proyeksi gabungan
z = np.dot(x_t, W_all) + np.dot(h_prev, U_all) + b_all

# Input gate
i = sigmoid(z[:, :units])
# Forget gate
f = sigmoid(z[:, units : 2 * units])
# Candidate cell state (g atau c_tilde)
c_candidate = tanh(z[:, 2 * units : 3 * units])
# Output gate
o = sigmoid(z[:, 3 * units :])

# Update cell state
c_t = f * c_prev + i * c_candidate
# Update hidden state
h_t = o * tanh(c_t)

return h_t, c_t

def _lstm_pass(
    self, inputs, W_all, U_all, b_all,
return_sequences=True, go_backwards=False
):
    batch_size, seq_length, _ = inputs.shape
    units = U_all.shape[0] # Hidden units

    h_t = np.zeros((batch_size, units))
    c_t = np.zeros((batch_size, units))

    if return_sequences:
        outputs_h = np.zeros((batch_size, seq_length,
units))
        # outputs_c = np.zeros((batch_size, seq_length,
units))

    time_steps = range(seq_length)
    if go_backwards:
        time_steps = range(seq_length - 1, -1, -1)

    for t_idx, t_val in enumerate(time_steps):
        x_step = inputs[:, t_val, :]
        h_t, c_t = self._lstm_cell_forward(x_step, h_t,

```

```

c_t, W_all, U_all, b_all)

        if return_sequences:
            # Simpan dalam urutan alami meskipun diproses
            terbalik
            storage_idx = t_val
            outputs_h[:, storage_idx, :] = h_t
            # outputs_c[:, storage_idx, :] = c_t

        return (
            outputs_h if return_sequences else h_t
        )

    def dense_forward(self, inputs, kernel, bias):
        logits = np.dot(inputs, kernel) + bias
        exp_logits = np.exp(logits - np.max(logits, axis=1,
keepdims=True))
        probabilities = exp_logits / np.sum(exp_logits, axis=1,
keepdims=True)
        return probabilities

    def forward(self, text_sequences):
        # 1. Embedding Layer
        current_tensor = self.embedding_forward(text_sequences)
        # print(f"After Embedding:
shape={current_tensor.shape}")

        # 2. LSTM Layers
        for config_entry in self.lstm_layer_configs:
            # print(f"Processing LSTM stack
{config_entry['stack_idx']}": Type={config_entry['type']},
KerasReturnSeq={config_entry['return_sequences']}")

                # return_sequences untuk pass LSTM ini ditentukan
                oleh konfigurasi Keras layer
                return_sequences_for_this_pass =
config_entry["return_sequences"]

                if config_entry["type"] == "bidirectional":
                    fwd_W = self.weights[config_entry["fwd_prefix"]
+ "kernel"]
                    fwd_U = self.weights[config_entry["fwd_prefix"]
+ "recurrent_kernel"]
                    fwd_b = self.weights[config_entry["fwd_prefix"]]

```

```

+ "bias"]

        bwd_W = self.weights[config_entry["bwd_prefix"]
+ "kernel"]
        bwd_U = self.weights[config_entry["bwd_prefix"]
+ "recurrent_kernel"]
        bwd_b = self.weights[config_entry["bwd_prefix"]
+ "bias"]

        # Forward pass LSTM
        h_fwd = self._lstm_pass(
            current_tensor,
            fwd_W,
            fwd_U,
            fwd_b,
            return_sequences=True,
            go_backwards=False,
        )

        # Backward pass LSTM
        h_bwd = self._lstm_pass(
            current_tensor,
            bwd_W,
            bwd_U,
            bwd_b,
            return_sequences=True,
            go_backwards=True,
        )

        # Gabungkan output
        if (
            return_sequences_for_this_pass
        ): # Jika Keras layer ini
    return_sequences=True
            current_tensor = np.concatenate([h_fwd,
h_bwd], axis=-1)
        else:
            current_tensor = np.concatenate(
                [h_fwd[:, -1, :], h_bwd[:, 0, :]],
axis=-1
        )

        elif config_entry["type"] == "unidirectional":
            W = self.weights[config_entry["prefix"]] +

```

```

"kernel"]
        U = self.weights[config_entry["prefix"]] +
"recurrent_kernel"]
        b = self.weights[config_entry["prefix"]] +
"bias"]

        current_tensor = self._lstm_pass(
            current_tensor,
            W,
            U,
            b,

return_sequences=return_sequences_for_this_pass,
            go_backwards=False,
        )

lstm_output = current_tensor

# 3. Output Dense Layer
output = self.dense_forward(
    lstm_output, self.weights["output_kernel"],
self.weights["output_bias"]
)
return output

def predict(self, texts):
    sequences = self.vectorizer(texts).numpy()
    return self.forward(sequences)

def compare_with_keras(self, texts):
    start_time_scratch = time.time()
    scratch_preds = self.predict(texts)
    scratch_time = time.time() - start_time_scratch

    start_time_keras = time.time()
    sequences_tf = self.vectorizer(texts)
    keras_preds = self.keras_model.predict(sequences_tf)
    keras_time = time.time() - start_time_keras

    scratch_classes = np.argmax(scratch_preds, axis=1)
    keras_classes = np.argmax(keras_preds, axis=1)

    implementation_accuracy = 0.0
    mae = float("inf")

```

```

        if not (np.any(np.isnan(scratch_preds)) or
np.any(np.isinf(scratch_preds))):
            implementation_accuracy = np.mean(scratch_classes
== keras_classes)
            mae = np.mean(np.abs(scratch_preds - keras_preds))
        else:
            print("WARNING: NaN or Inf found in scratch
predictions!")

        print(f"\nPerbandingan waktu prediksi (LSTM):")
        print(f"From scratch: {scratch_time:.4f} detik")
        print(f"Keras: {keras_time:.4f} detik")
        if keras_time > 0:
            print(f"Ratio waktu (scratch/keras):
{scratch_time/keras_time:.2f}x")
        else:
            print(f"Ratio waktu (scratch/keras): N/A (Keras
time is zero)")

        print(
            f"\nAkurasi kecocokan implementasi (kelas):
{implementation_accuracy:.6f}"
        )
        print(f"Mean absolute error antar probabilitas
prediksi: {mae:.8f}")

        if (
            mae > 1e-4
        ):
            print(" MAE tinggi, memeriksa beberapa perbedaan
prediksi:")
            for i in range(min(3, scratch_preds.shape[0])): # Tampilkan beberapa contoh
                print(
                    f"      Sample {i}:
Scratch={scratch_preds[i]}, Keras={keras_preds[i]}, Diff
MAE={np.mean(np.abs(scratch_preds[i] - keras_preds[i]))}"
                )

    return scratch_preds, keras_preds,
implementation_accuracy, mae

```

BAB III

Hasil Pengujian

3.1. Convolutional Neural Network (CNN)

3.1.1. Pengaruh jumlah layer konvolusi

Eksperimen pertama bertujuan untuk mengamati bagaimana perbedaan jumlah layer konvolusi memengaruhi kinerja model CNN dan waktu pelatihan model CNN. Variasi yang diuji adalah model dengan 1, 2, dan 3 layer konvolusi.

Tabel 3.1.1.1. Pengujian Pengaruh Jumlah Layer Konvolusi CNN

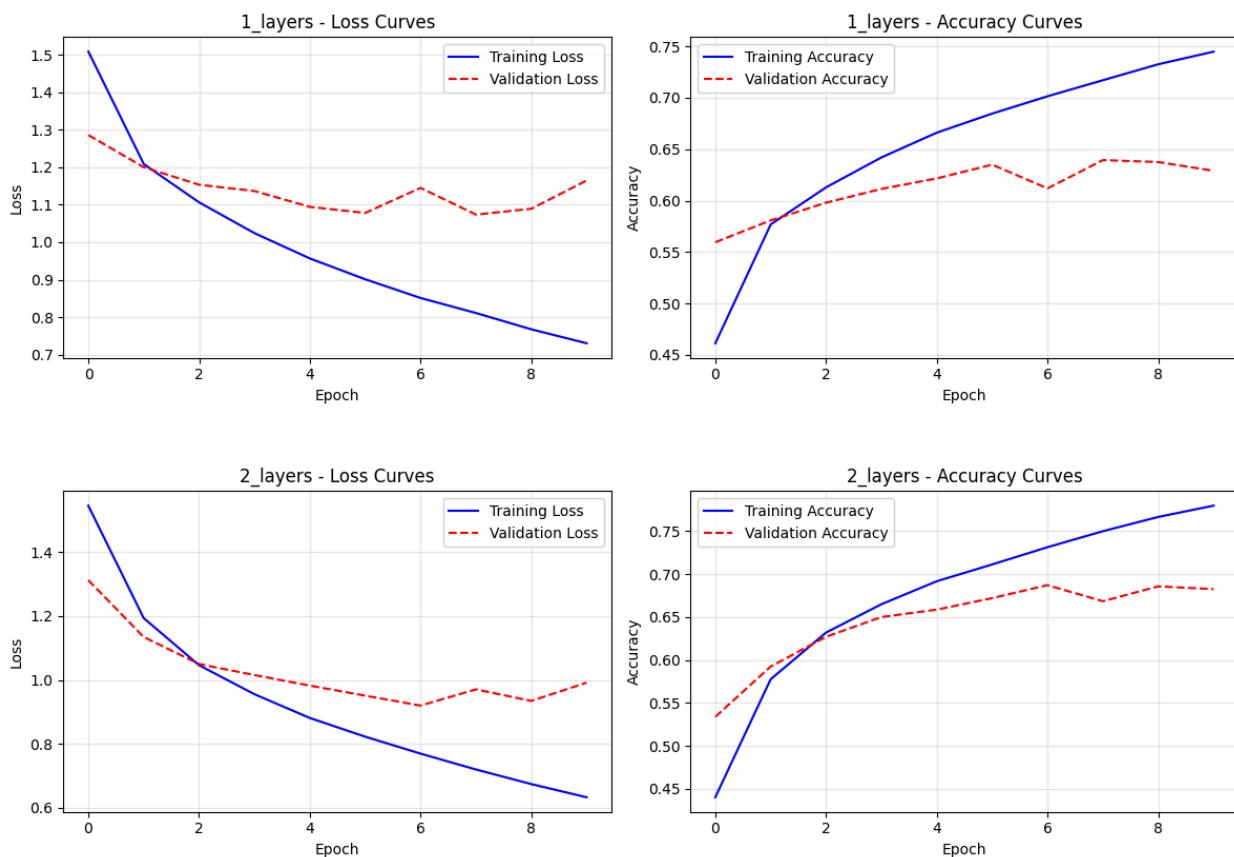
Variabel Tetap	
Jenis Pooling Layer	MaxPooling2D
Ukuran Pooling Window	(2,2)
Jumlah Unit Hidden Dense Layer	64
Jumlah Unit Output Dense Layer	10
Fungsi Aktivasi Output Dense Layer	softmax
Optimizer	Adam
Loss Function	sparse_categorical_crossentropy
Ukuran batch (batch_size)	32
epochs	10
Early Stopping Patience	5
Learning Rate Reduction Factor	0.5
Learning Rate Reduction Patience	3
Minimum Learning Rate (MIN_LR)	1e-7
Padding Conv2D	valid
Fungsi Aktivasi Conv2D	relu
Variabel Bebas (jumlah layer Konvolusi : Jumlah Layer, Filter, Kernel)	
Percobaan ke-1	1, [32], [3]
Percobaan ke-2	2, [32, 64], [3, 3]

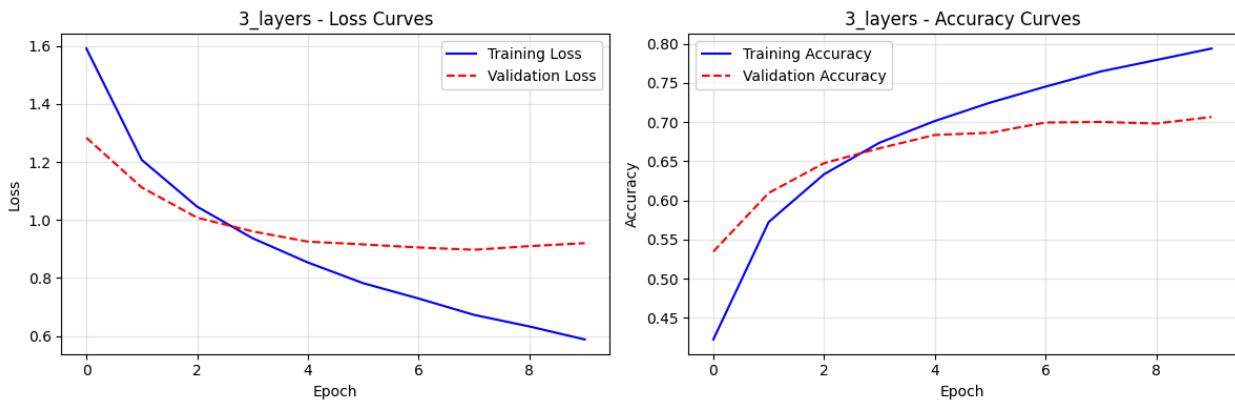
Percobaan ke-3	3, [32, 64, 128], [3, 3, 3]
----------------	-----------------------------

Tabel 3.1.1.2. Hasil Eksperimen Pengujian Pengaruh Jumlah Layer Konvolusi CNN

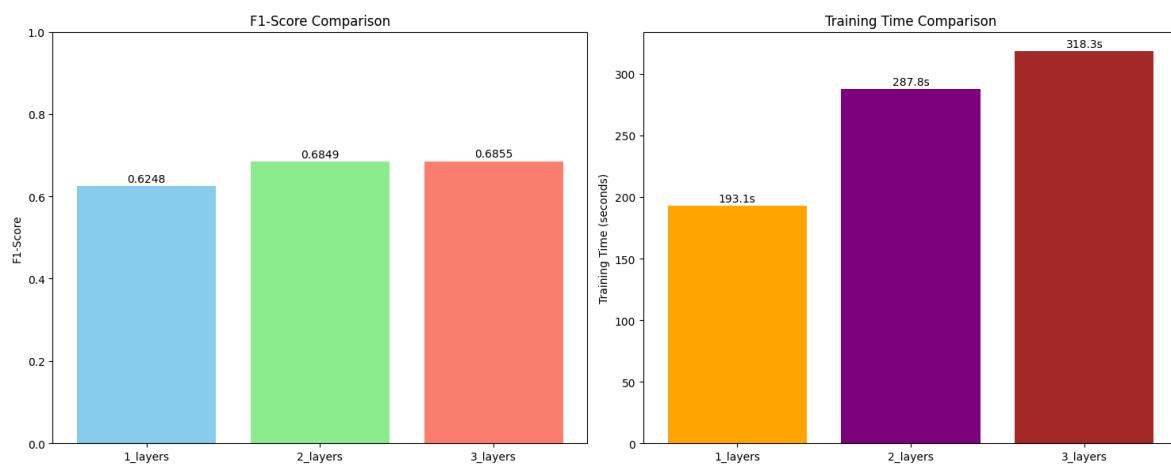
Jumlah Layer Konvolusi	Akurasi Test	F1-Score (Macro) Test	Loss Test	Waktu Training (detik)
1	6.303	0.6248	1.0754	193.11
2	6.850	0.6849	9.222	287.75
3	6.868	0.6855	9.260	318.27

Berdasarkan tabel hasil pengujian, penambahan jumlah layer konvolusi dari satu ke dua layer menunjukkan peningkatan performa yang signifikan, dengan F1-score naik dari 0.6248 menjadi 0.6849 dan loss test menurun. Namun, penambahan lebih lanjut ke tiga layer hanya memberikan peningkatan F1-score yang sangat marginal menjadi 0.6855, disertai sedikit kenaikan pada loss test dan peningkatan waktu training yang cukup besar. Hal ini mengindikasikan bahwa model dengan dua layer konvolusi menawarkan keseimbangan terbaik antara performa dan efisiensi komputasi untuk konfigurasi ini, sementara penambahan ke tiga layer menunjukkan adanya diminishing returns.





Observasi kurva pembelajaran untuk model dengan 1, 2, dan 3 layer konvolusi menunjukkan bahwa semua model mulai menunjukkan tanda-tanda overfitting seiring berjalannya epoch. Hal ini terlihat dari training loss yang terus menurun sementara validation loss cenderung stagnan atau bahkan sedikit meningkat setelah beberapa epoch awal, dan training accuracy yang terus meningkat melampaui validation accuracy yang mencapai plateau. Model dengan 1 layer konvolusi menunjukkan validation accuracy yang berfluktuasi dan mencapai plateau lebih rendah dibandingkan model dengan 2 dan 3 layer. Model dengan 2 dan 3 layer konvolusi mampu mencapai validation accuracy yang lebih tinggi dan validation loss yang lebih rendah, mengindikasikan kapasitas belajar yang lebih baik. Meskipun demikian, generalization gap (perbedaan antara performa training dan validasi) tetap terlihat jelas pada model 2 dan 3 layer, menandakan bahwa mereka juga mengalami overfitting, meskipun performa validasi puncaknya lebih baik daripada model 1 layer.



Berdasarkan plot "F1-Score Comparison" dan "Training Time Comparison" untuk eksperimen pengaruh jumlah layer, dapat diamati bahwa penambahan jumlah layer dari satu ke dua layer (1_layers ke 2_layers) menghasilkan peningkatan F1-score yang cukup signifikan, dari 0.6248 menjadi 0.6849. Peningkatan lebih lanjut ke tiga layer (3_layers)

hanya memberikan kenaikan F1-score yang sangat kecil, mencapai 0.6855. Di sisi lain, waktu pelatihan menunjukkan peningkatan yang konsisten seiring bertambahnya jumlah layer. Model dengan satu layer (1_layers) membutuhkan waktu 193.1 detik, model dengan dua layer (2_layers) membutuhkan 287.8 detik, dan model dengan tiga layer (3_layers) membutuhkan waktu pelatihan terlama yaitu 318.3 detik. Peningkatan waktu ini sejalan dengan bertambahnya jumlah parameter dan kompleksitas komputasi pada model yang lebih dalam.

3.1.2. Pengaruh banyak filter per layer konvolusi

Eksperimen kedua bertujuan untuk mengamati bagaimana perbedaan banyak filter per layer konvolusi mempengaruhi kinerja model CNN dan waktu pelatihan model CNN. Variasi yang diuji adalah model dengan filter sebagai berikut [16, 32, 64], [32, 64, 128], [64, 128, 256].

Tabel 3.1.2.1. Pengujian Pengaruh Banyak Filter per Layer Konvolusi CNN

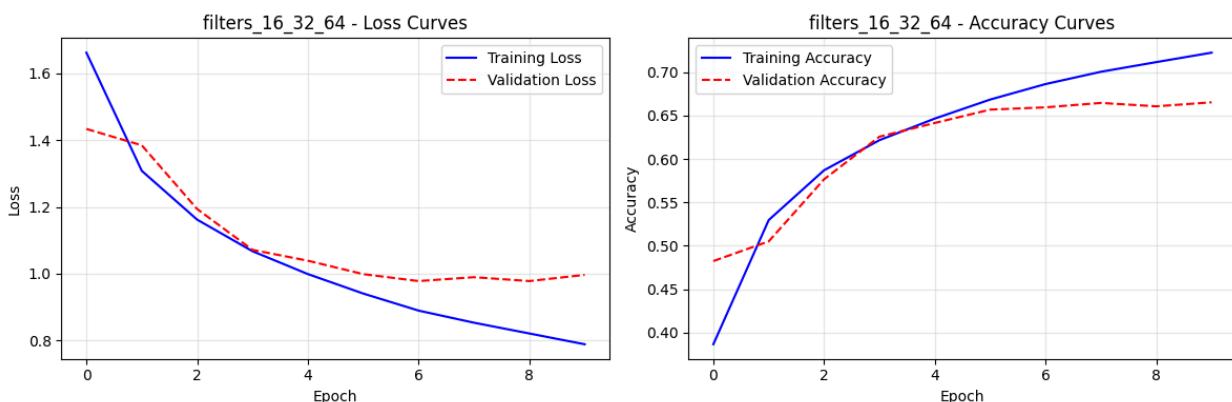
Variabel Tetap	
Jumlah Layer Konvolusi	3
Ukuran Kernel per Layer Konvolusi	[3,3,3]
Jenis Pooling Layer	MaxPooling2D
Ukuran Pooling Window	(2,2)
Jumlah Unit Hidden Dense Layer	64
Jumlah Unit Output Dense Layer	10
Fungsi Aktivasi Output Dense Layer	softmax
Optimizer	Adam
Loss Function	sparse_categorical_crossentropy
Ukuran batch (batch_size)	32
epochs	10
Early Stopping Patience	5
Learning Rate Reduction Factor	0.5
Learning Rate Reduction Patience	3
Minimum Learning Rate (MIN_LR)	1e-7

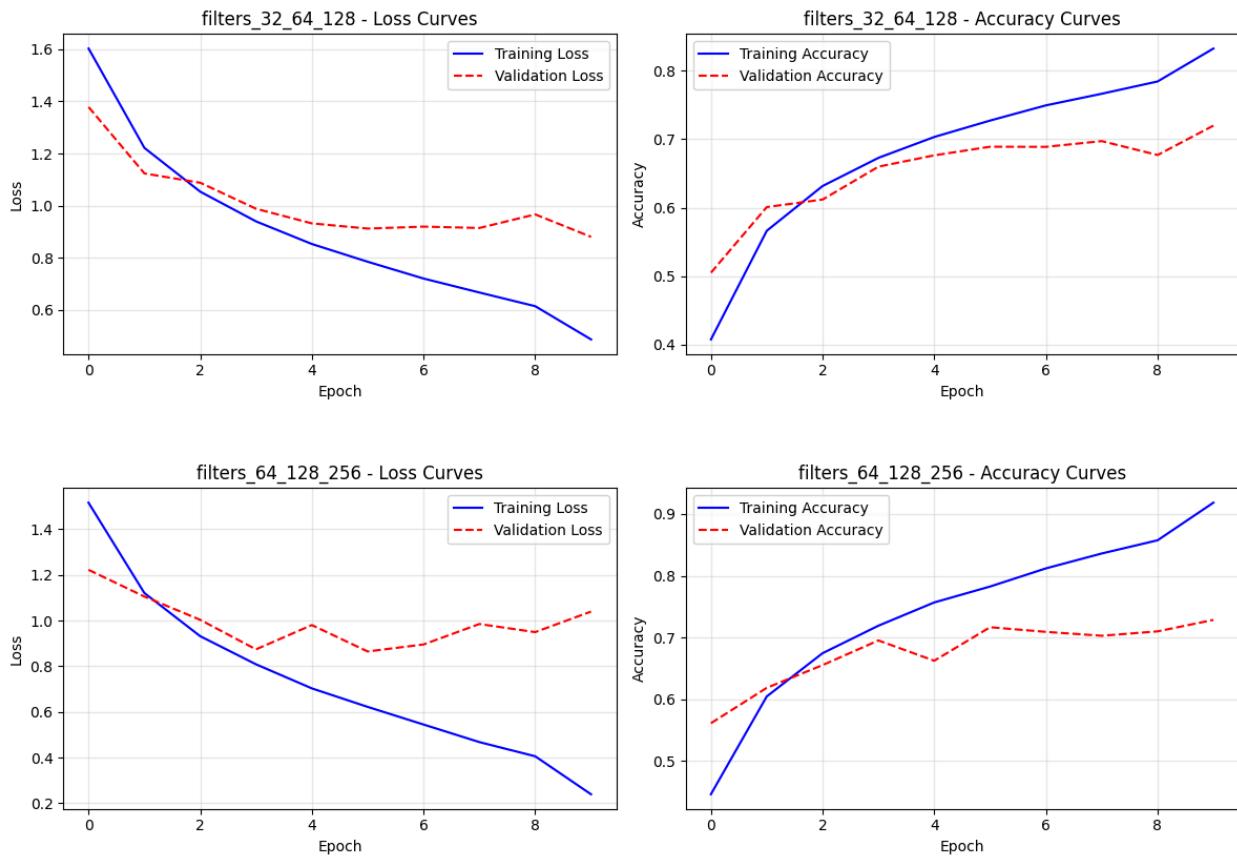
Padding Conv2D	valid
Fungsi Aktivasi Conv2D	relu
Variabel Bebas (Jumlah Filter per Layer Konvolusi)	
Percobaan ke-1	[16, 32, 64]
Percobaan ke-2	[32, 64, 128]
Percobaan ke-3	[64, 128, 256]

Tabel 3.1.2.2. Hasil Eksperimen Pengujian Pengaruh Banyak Filter per Layer Konvolusi CNN

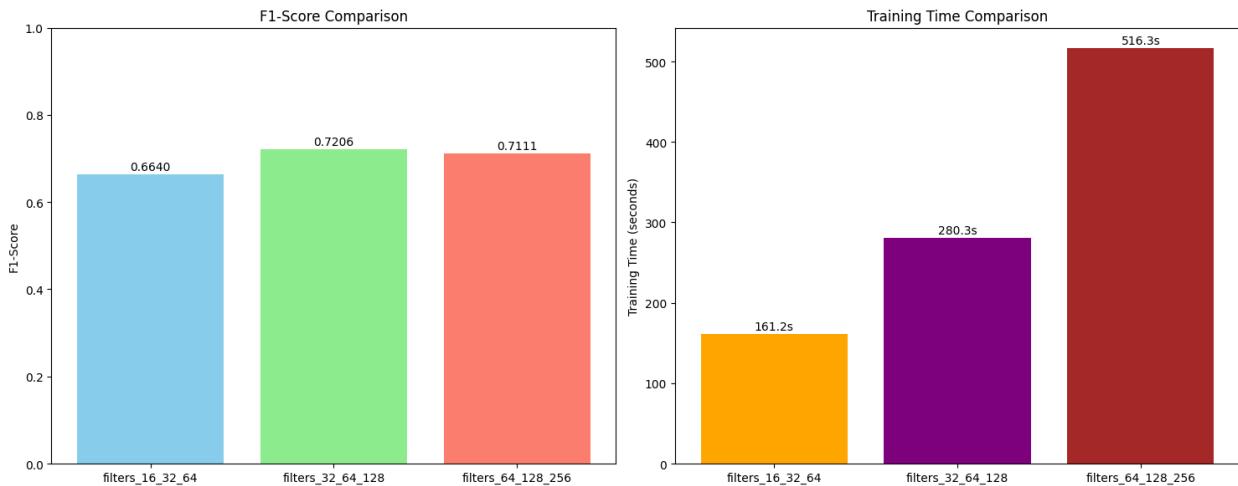
Konfigurasi Filter (filter_config)	Akurasi (accuracy)	F1-Score (f1_score)	Loss (loss)	Waktu Training (train_time)
16-32-64	6.670	6.640	9.760	161.23
32-64-128	7.206	7.206	8.884	280.32
64-128-256	7.142	7.111	8.650	516.29

Berdasarkan tabel hasil pengujian, peningkatan jumlah filter dari konfigurasi 16-32-64 ke 32-64-128 secara signifikan meningkatkan akurasi (dari 0.6670 menjadi 0.7206) dan F1-score (dari 0.6640 menjadi 0.7206), serta menurunkan loss. Namun, penambahan filter lebih lanjut ke 64-128-256 justru sedikit menurunkan akurasi dan F1-score menjadi 0.7142 dan 0.7111, meskipun loss sedikit membaik. Di sisi lain, waktu training meningkat drastis seiring bertambahnya jumlah filter, dari 161.23 detik hingga 516.29 detik. Dengan demikian, konfigurasi filter 32-64-128 menunjukkan keseimbangan terbaik antara performa dan efisiensi komputasi, karena peningkatan lebih lanjut tidak memberikan keuntungan performa yang sebanding dengan kenaikan biaya komputasi.





Berdasarkan kurva pembelajaran untuk tiga konfigurasi jumlah filter yang berbeda (filters_16_32_64, filters_32_64_128, dan filters_64_128_256), dapat diobservasi beberapa tren terkait performa training dan validasi. Secara umum, semua konfigurasi menunjukkan tanda-tanda overfitting, di mana training loss terus menurun dan training accuracy terus meningkat, sementara metrik validasi (loss dan akurasi) mencapai plateau atau bahkan memburuk setelah beberapa epoch. Konfigurasi dengan jumlah filter lebih banyak (filters_32_64_128 dan filters_64_128_256) cenderung memiliki training loss yang lebih rendah dan training accuracy yang lebih tinggi, mengindikasikan kapasitas model yang lebih besar untuk mempelajari data training. Namun, pada kurva validasi, konfigurasi filters_16_32_64 menunjukkan validation accuracy yang stagnan pada level yang lebih rendah. Konfigurasi filters_32_64_128 mencapai validation accuracy yang lebih tinggi sebelum menunjukkan tanda overfitting yang lebih jelas. Sementara itu, konfigurasi filters_64_128_256 menunjukkan validation loss yang lebih fluktuatif dan validation accuracy yang tidak secara konsisten melampaui konfigurasi filters_32_64_128, bahkan cenderung lebih tidak stabil, menandakan overfitting yang lebih parah meskipun kapasitas modelnya paling besar.



Berdasarkan plot yang tersedia, khususnya "F1-Score Comparison" dan "Training Time Comparison", dapat dianalisis pengaruh variasi jumlah filter terhadap performa model dan waktu pelatihan. Terlihat bahwa F1-score meningkat ketika konfigurasi filter diubah dari filters_16_32_64 (F1-score 0.6640) ke filters_32_64_128 (F1-score 0.7206). Namun, peningkatan jumlah filter lebih lanjut ke filters_64_128_256 justru menghasilkan sedikit penurunan F1-score menjadi 0.7111. Sebaliknya, waktu pelatihan menunjukkan peningkatan yang konsisten dan signifikan seiring dengan bertambahnya jumlah filter pada setiap layer. Model dengan konfigurasi filters_16_32_64 membutuhkan waktu sekitar 161.2 detik, konfigurasi filters_32_64_128 membutuhkan 280.3 detik, dan konfigurasi filters_64_128_256 membutuhkan waktu paling lama yaitu 516.3 detik. Peningkatan waktu pelatihan ini wajar terjadi karena penambahan jumlah filter secara langsung meningkatkan jumlah parameter dalam model dan kompleksitas komputasi pada setiap operasi konvolusi. Dengan demikian, konfigurasi filters_32_64_128 tampaknya menawarkan keseimbangan terbaik antara performa F1-score dan waktu pelatihan dalam eksperimen ini.

3.1.3. Pengaruh ukuran filter per layer konvolusi

Eksperimen ketiga bertujuan untuk mengamati bagaimana perbedaan ukuran filter per layer konvolusi mempengaruhi kinerja model CNN dan waktu pelatihan model CNN. Variasi yang diuji adalah model dengan ukuran filter sebagai berikut [3, 3, 3], [4, 3, 3], [3, 4, 5].

Tabel 3.1.3.1. Pengujian Pengaruh Ukuran Filter per Layer Konvolusi CNN

Variabel Tetap	
Jumlah Layer Konvolusi	3
Jumlah Filter per Layer	[32, 64, 128]

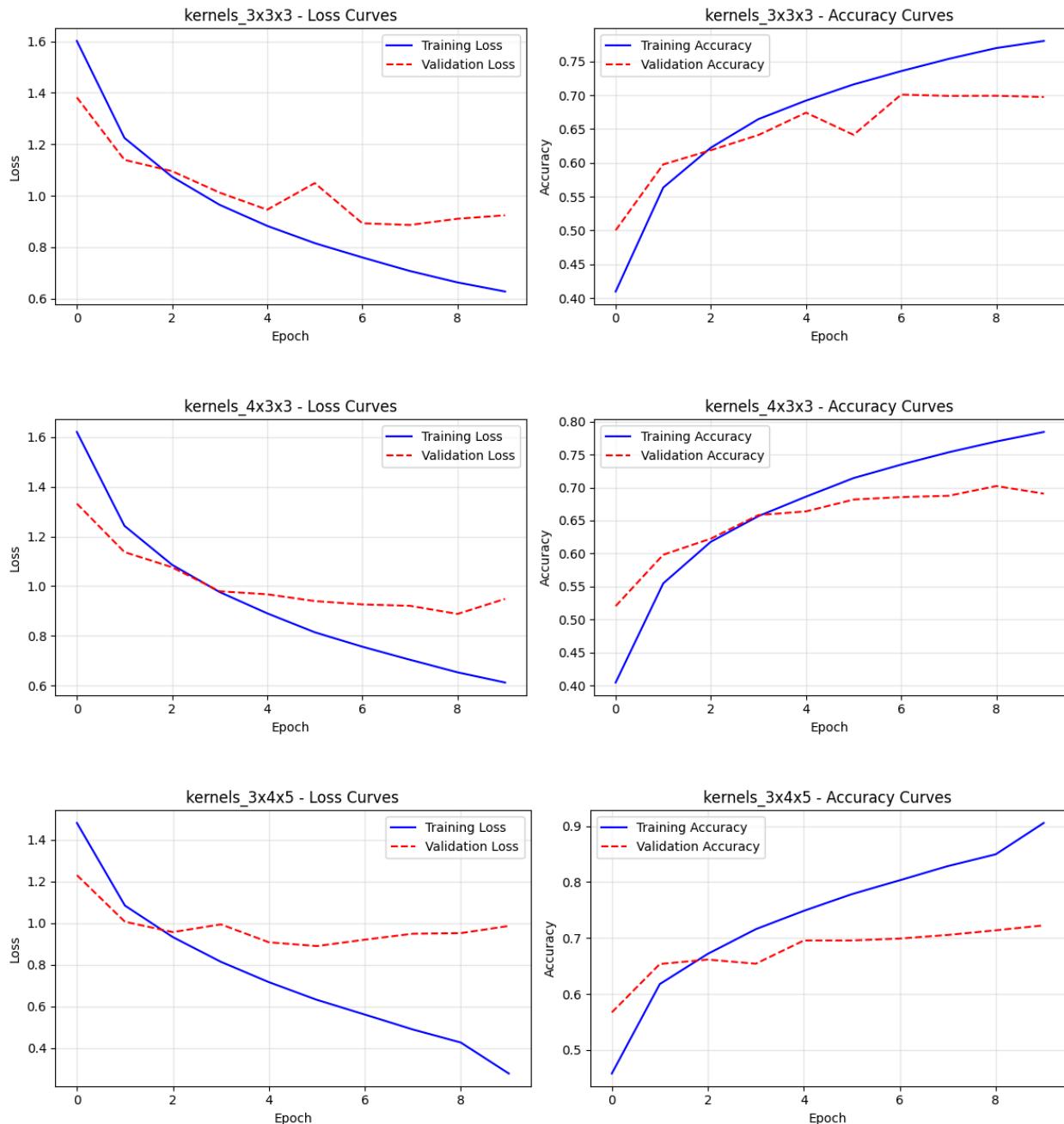
Jenis Pooling Layer	MaxPooling2D
Ukuran Pooling Window	(2,2)
Jumlah Unit Hidden Dense Layer	64
Jumlah Unit Output Dense Layer	10
Fungsi Aktivasi Output Dense Layer	softmax
Optimizer	Adam
Loss Function	sparse_categorical_crossentropy
Ukuran batch (batch_size)	32
epochs	10
Early Stopping Patience	5
Learning Rate Reduction Factor	0.5
Learning Rate Reduction Patience	3
Minimum Learning Rate (MIN_LR)	1e-7
Padding Conv2D	valid
Fungsi Aktivasi Conv2D	relu
Variabel Bebas (Ukuran Kernel per Layer Konvolusi)	
Percobaan ke-1	[3, 3, 3]
Percobaan ke-2	[4, 3, 3]
Percobaan ke-3	[3, 4, 5]

Tabel 3.1.3.2. Hasil Eksperimen Pengujian Pengaruh Ukuran Filter per Layer Konvolusi CNN

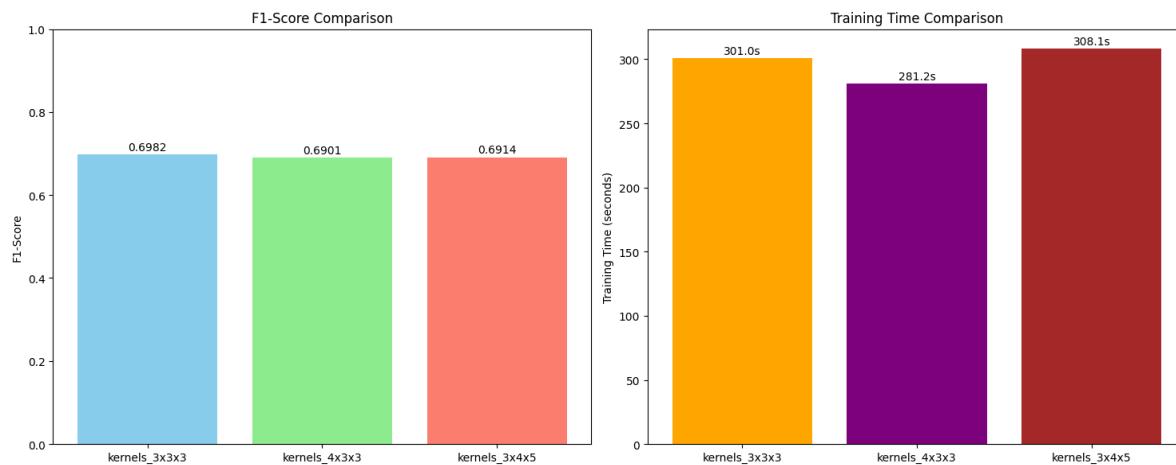
Model Name	Kernel Configuration	Test Accuracy	F1-Score	Test Loss	Training Time (seconds)
kernels_3x3x3	3x3x3	7.016	6.982	8.953	301.04
kernels_4x3x3	4x3x3	6.924	6.901	9.242	281.23
kernels_3x4x5	3x4x5	6.950	6.914	9.104	308.11

Berdasarkan tabel hasil eksperimen, konfigurasi kernel kernels_3x3x3 (semua kernel berukuran 3x3) menunjukkan performa prediktif terbaik dengan Test Accuracy tertinggi

(0.7016), F1-Score tertinggi (0.6982), dan Test Loss terendah (0.8953), dengan waktu training 301.04 detik. Menggunakan kernel 4x4 pada layer pertama (kernels_4x3x3) sedikit menurunkan akurasi dan F1-score namun menghasilkan waktu training tercepat (281.23 detik), kemungkinan karena pengurangan dimensi feature map awal akibat padding='valid'. Sebaliknya, konfigurasi kernel campuran yang membesar (kernels_3x4x5) juga tidak melampaui performa konfigurasi 3x3x3 dan memiliki waktu training terlama (308.11 detik). Ini menunjukkan bahwa untuk arsitektur dan dataset ini, kernel 3x3 yang seragam adalah pilihan paling efektif untuk mencapai performa klasifikasi optimal.



Berdasarkan kurva pembelajaran untuk variasi ukuran kernel, semua konfigurasi (kernels_3x3x3, kernels_4x3x3, dan kernels_3x4x5) menunjukkan tanda-tanda overfitting, di mana metrik training terus membaik sementara metrik validasi mencapai plateau dan menunjukkan divergensi. Konfigurasi kernels_3x3x3 (semua kernel 3x3) dan kernels_4x3x3 (kernel 4x4 di awal) menampilkan kurva validation accuracy yang relatif stabil setelah mencapai puncaknya di sekitar 0.69-0.70, dengan validation loss yang juga menunjukkan plateau sebelum sedikit meningkat. Sebaliknya, konfigurasi kernels_3x4x5 (kernel campuran membesar) memperlihatkan training loss terendah dan training accuracy tertinggi, namun validation accuracy-nya mencapai plateau lebih awal dan validation loss-nya mulai meningkat lebih cepat dan lebih signifikan, mengindikasikan overfitting atau generalisasi yang kurang stabil.



Berdasarkan plot "F1-Score Comparison" dan "Training Time Comparison" untuk eksperimen ukuran kernel, konfigurasi kernels_3x3x3 mencapai F1-score tertinggi (0.6982) dengan waktu pelatihan 301.0 detik. Konfigurasi kernels_4x3x3 menunjukkan waktu pelatihan tercepat (281.2 detik), namun dengan F1-score yang sedikit lebih rendah (0.6901). Sementara itu, konfigurasi kernels_3x4x5 memiliki F1-score (0.6914) yang juga sedikit di bawah konfigurasi kernels_3x3x3 tetapi membutuhkan waktu pelatihan terlama (308.1 detik). Dengan demikian, penggunaan kernel 3x3 yang seragam (kernels_3x3x3) memberikan performa F1-score terbaik, meskipun konfigurasi dengan kernel 4x4 di layer pertama (kernels_4x3x3) menawarkan keunggulan dalam efisiensi waktu pelatihan dengan sedikit pengorbanan pada F1-score.

3.1.4. Pengaruh jenis pooling layer

Eksperimen keempat bertujuan untuk mengamati bagaimana perbedaan jenis pooling layer mempengaruhi kinerja model CNN dan waktu pelatihan model CNN. Variasi yang diuji adalah average pooling dan juga max pooling.

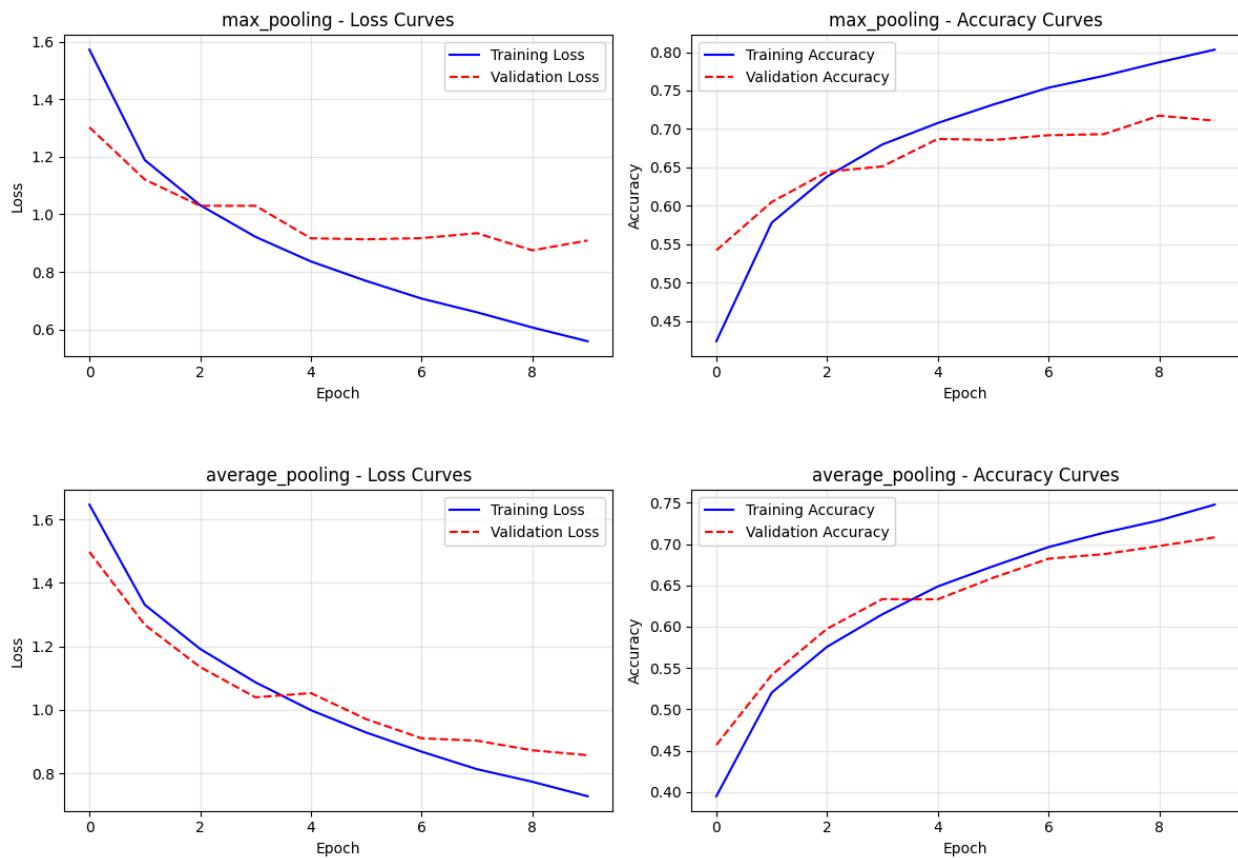
Tabel 3.1.4.1. Pengujian Pengaruh Jenis Pooling Layer CNN

Variabel Tetap	
Jumlah Layer Konvolusi	3
Jumlah Filter per Layer	[32, 64, 128]
Ukuran Kernel	[3 ,3, 3]
Ukuran Pooling Window	(2,2)
Jumlah Unit Hidden Dense Layer	64
Jumlah Unit Output Dense Layer	10
Fungsi Aktivasi Output Dense Layer	softmax
Optimizer	Adam
Loss Function	sparse_categorical_crossentropy
Ukuran batch (batch_size)	32
epochs	10
Early Stopping Patience	5
Learning Rate Reduction Factor	0.5
Learning Rate Reduction Patience	3
Minimum Learning Rate (MIN_LR)	1e-7
Padding Conv2D	valid
Fungsi Aktivasi Conv2D	relu
Variabel Bebas (Jenis Pooling Layer)	
Percobaan ke-1	MaxPooling2D
Percobaan ke-2	AveragePooling2D

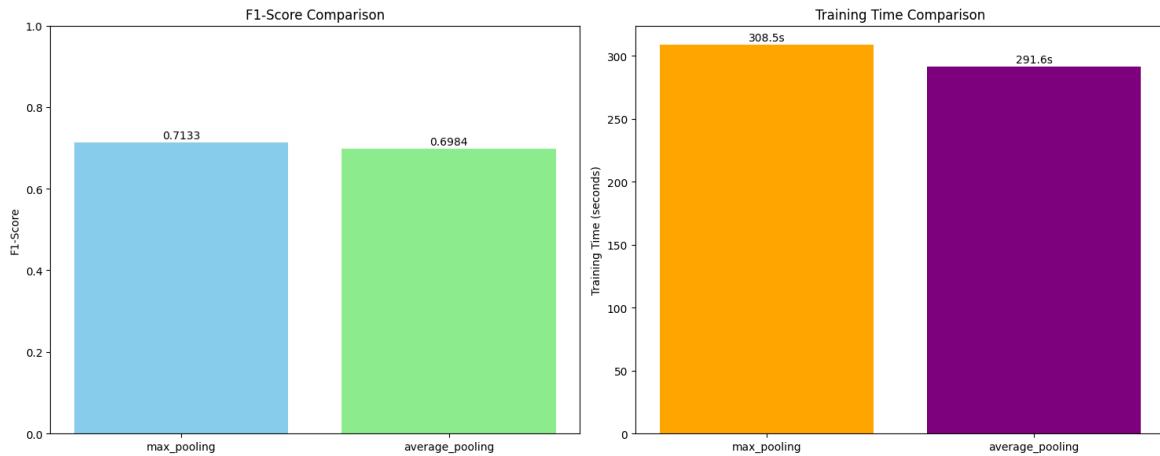
Tabel 3.1.4.2. Hasil Eksperimen Pengujian Pengaruh Jenis Pooling Layer CNN

Model Name	Jenis Pooling (Pooling_Type)	Akurasi Test (Test_Accuracy)	F1-Score (F1_Score)	Loss Test (Test_Loss)	Waktu Training (Training_Time_seconds)
max_pooling	Max	7.155	7.133	8.821	308.55
average_pooling	Average	6.989	6.984	8.597	291.61

Berdasarkan tabel hasil eksperimen, Max Pooling (max_pooling) menunjukkan performa klasifikasi yang lebih unggul dengan Test Accuracy 0.7155 dan F1-Score 0.7133, dibandingkan Average Pooling (average_pooling) yang mencapai Akurasi 0.6989 dan F1-Score 0.6984. Meskipun Average Pooling mencatatkan Test Loss sedikit lebih rendah (0.8597 vs 0.8821) dan waktu training yang sedikit lebih cepat (291.61 detik vs 308.55 detik), Max Pooling lebih efektif dalam menghasilkan keputusan kelas yang akurat, menjadikannya pilihan yang lebih baik karena F1-score adalah prioritas utama untuk dataset cifar-10.



Berdasarkan perbandingan kurva pembelajaran untuk Max Pooling dan Average Pooling, kedua jenis pooling menunjukkan tanda-tanda overfitting, di mana metrik training terus membaik melampaui metrik validasi yang mencapai plateau. Kurva validasi untuk Max Pooling menunjukkan fluktuasi yang lebih signifikan, terutama pada validation loss yang membentuk pola V sebelum sedikit meningkat lagi di akhir epoch, meskipun validation accuracy-nya tampak mencapai puncak sedikit lebih tinggi (sekitar 0.71-0.72). Sebaliknya, Average Pooling menampilkan kurva validasi yang lebih stabil dan halus, baik untuk loss maupun akurasi, dengan validation loss yang cenderung terus menurun atau stagnan di level yang lebih rendah pada akhir epoch, dan validation accuracy yang mencapai plateau di sekitar 0.70-0.71.



Berdasarkan plot "F1-Score Comparison" dan "Training Time Comparison" yang membandingkan kinerja model dengan max_pooling dan average_pooling, terlihat bahwa max_pooling mencapai F1-score yang lebih tinggi (0.7133) dibandingkan dengan average_pooling (0.6984). Dari segi waktu pelatihan, average_pooling sedikit lebih cepat dengan waktu 291.6 detik, sementara max_pooling membutuhkan 308.5 detik. Meskipun average_pooling menawarkan keunggulan kecil dalam efisiensi waktu dan kurva validation loss yang tampak lebih stabil dan berakhir lebih rendah pada plot "Validation Loss Comparison", max_pooling unggul dalam metrik performa klasifikasi utama (F1-score). Ini mengindikasikan bahwa untuk dataset dan arsitektur yang diuji, max_pooling lebih efektif dalam mengekstraksi fitur-fitur penting yang mendukung keputusan klasifikasi yang lebih baik, meskipun dengan sedikit tambahan waktu komputasi.

3.2. Simple Recurrent Neural Network (Simple RNN)

3.2.1. Pengaruh Jumlah Layer RNN

Eksperimen pertama bertujuan untuk mengamati bagaimana perbedaan jumlah layer RNN memengaruhi kinerja model dan waktu pelatihan. Variasi yang diuji adalah model dengan 1, 2, dan 3 layer RNN.

Tabel 3.2.1.1. Pengujian Pengaruh Jumlah Layer RNN

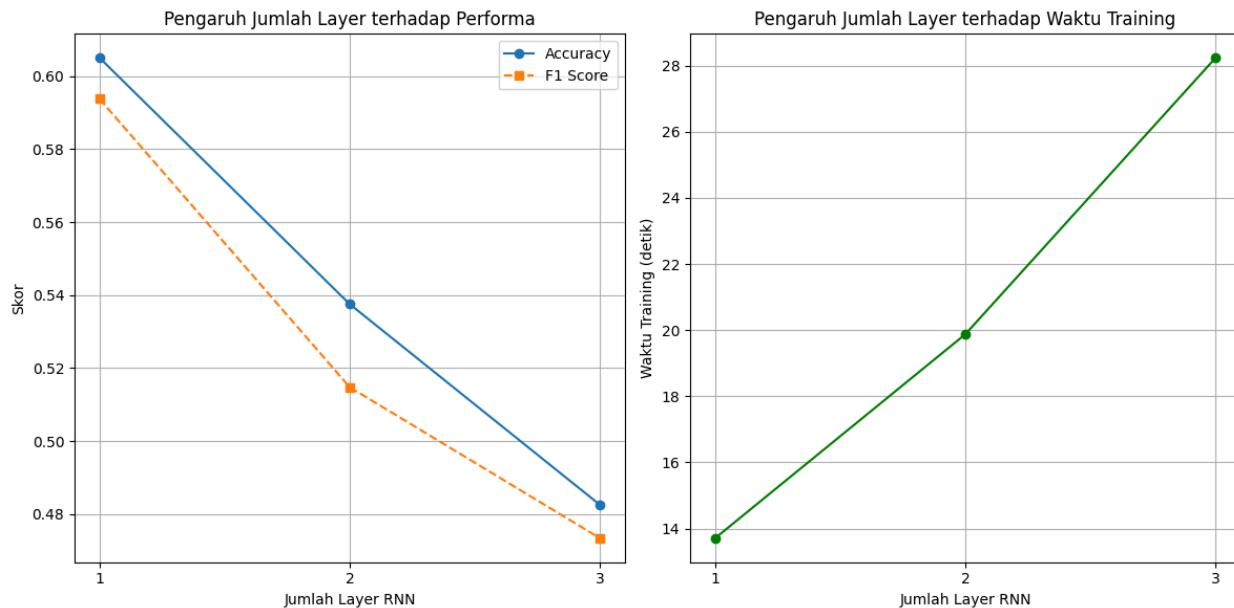
Variabel Tetap	
Jumlah sel per layer (rnn_units)	48
Dimensi embedding (embedding_dim)	96
Arah RNN	Bidirectional
Tingkat dropout setelah layer RNN (dropout_rate)	0.3
Tingkat dropout setelah layer embedding	0.2

(embedding_dropout)	
Tingkat dropout untuk koneksi rekuren dalam sel RNN (recurrent_dropout)	0.1
Faktor regularisasi L2 (l2_reg)	0.01
Tingkat pembelajaran awal untuk optimizer Adam (learning_rate)	0.001
Ukuran batch (batch_size)	24
epochs	30
Variabel Bebas (jumlah layer RNN)	
Percobaan ke-1	1
Percobaan ke-2	2
Percobaan ke-3	3

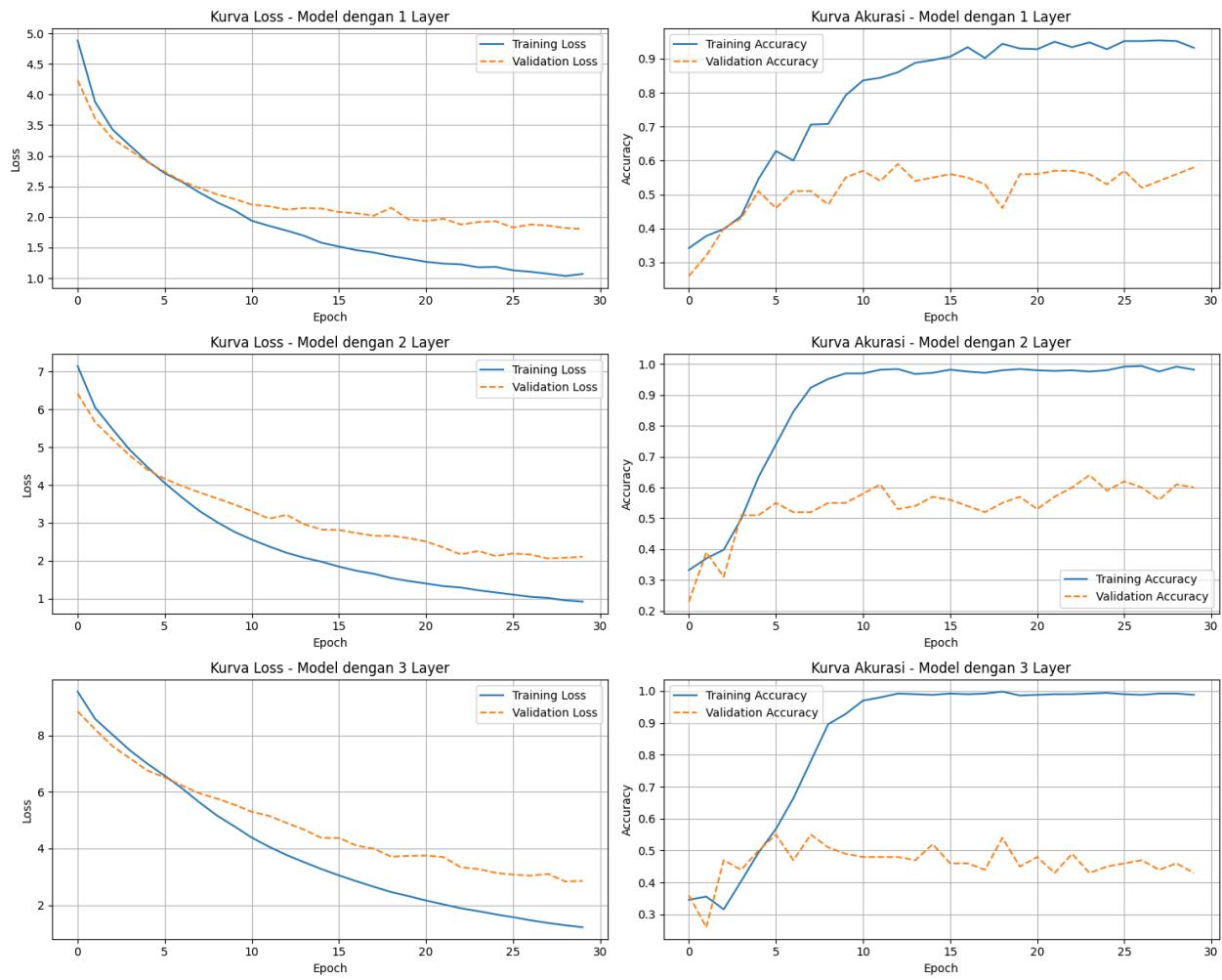
Tabel 3.2.1.2. Hasil Eksperimen Pengaruh Jumlah Layer RNN

num_layers	accuracy	f1_score	loss	train_time
0	1	0.6050	0.593736	1.803163
1	2	0.5375	0.514738	2.287036
2	3	0.4825	0.473370	2.691531

Berdasarkan tabel 3.2.1.2, model dengan 1 layer RNN menunjukkan performa terbaik dengan F1-score mencapai 0.5937 dan akurasi 0.6050. Penambahan jumlah layer menjadi 2 dan 3 justru mengakibatkan penurunan F1-score dan akurasi secara bertahap.



Dari plot "Pengaruh Jumlah Layer terhadap Performa", terlihat jelas bahwa F1-score dan akurasi tertinggi dicapai oleh model dengan 1 layer, kemudian menurun untuk model dengan 2 layer, dan menurun lebih lanjut untuk model dengan 3 layer. Sebaliknya, plot "Pengaruh Jumlah Layer terhadap Waktu Training" menunjukkan bahwa waktu pelatihan meningkat secara signifikan seiring dengan bertambahnya jumlah layer. Model dengan 1 layer membutuhkan sekitar 13.7 detik, model 2 layer sekitar 19.9 detik, dan model 3 layer sekitar 28.2 detik. Peningkatan waktu ini sejalan dengan bertambahnya jumlah parameter dan kompleksitas komputasi pada model yang lebih dalam.



Observasi kurva pembelajaran untuk model dengan 1, 2, dan 3 layer menunjukkan bahwa seiring bertambahnya kedalaman layer, generalization gap antara performa training dan validasi cenderung melebar. Model dengan 1 layer menunjukkan gap yang paling kecil, sementara model dengan 2 dan 3 layer memperlihatkan validation loss yang relatif lebih tinggi dan validation accuracy yang lebih rendah serta lebih berfluktuasi dibandingkan kurva trainingnya. Hal ini mengindikasikan bahwa model yang lebih dalam (2 dan 3 layer) mengalami overfitting yang lebih signifikan dan kesulitan melakukan generalisasi pada dataset yang digunakan.

Berdasarkan hasil eksperimen, dapat disimpulkan bahwa model Simple RNN dengan 1 layer memberikan performa terbaik, menghasilkan F1-score macro tertinggi (0.5937) dan akurasi (0.6050). Penambahan jumlah layer menjadi 2 atau 3 tidak hanya menurunkan F1-score dan akurasi secara progresif tetapi juga secara signifikan meningkatkan waktu pelatihan. Model yang lebih dalam menunjukkan kecenderungan overfitting yang lebih kuat, kemungkinan karena kompleksitas model yang lebih tinggi tidak diimbangi dengan jumlah data training yang cukup.

3.2.2. Pengaruh Banyak Cell RNN per Layer

Eksperimen kedua difokuskan untuk menganalisis dampak dari variasi jumlah unit (cell) dalam layer Simple RNN terhadap performa model. Variasi jumlah unit yang diuji adalah 24, 48, dan 96 unit.

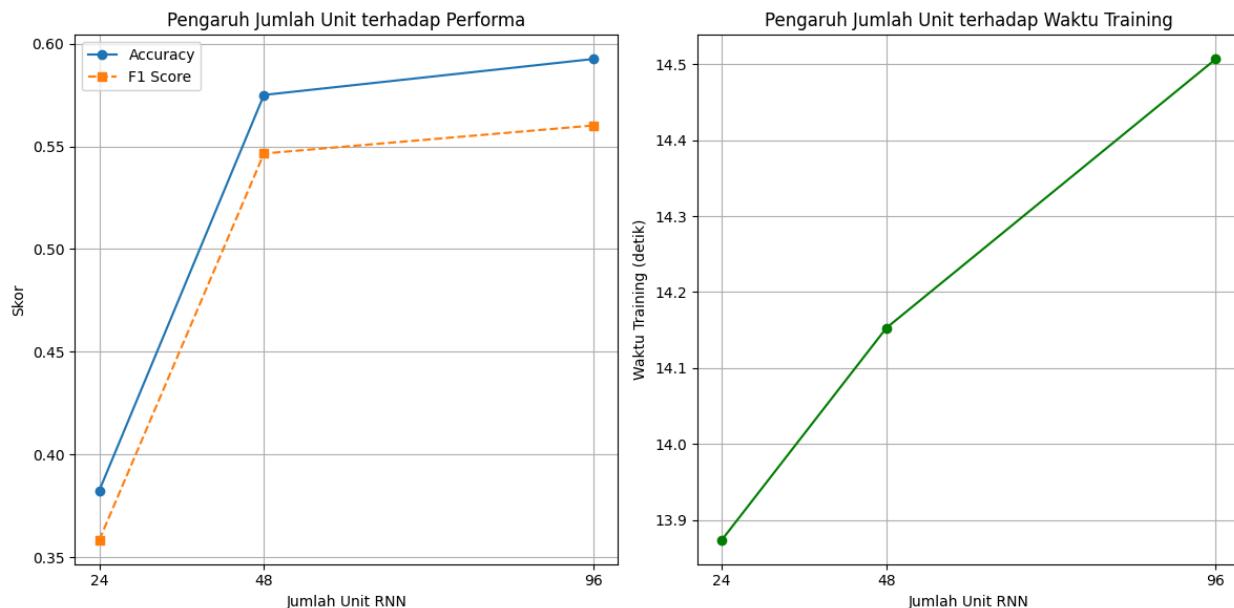
Tabel 3.2.2.1. Pengujian Pengaruh Banyak Cell RNN per Layer

Variabel Tetap	
Jumlah layer RNN (num_rnn_layers)	1
Dimensi embedding (embedding_dim)	96
Arah RNN	Bidirectional
Tingkat dropout setelah layer RNN (dropout_rate)	0.3
Tingkat dropout setelah layer embedding (embedding_dropout)	0.2
Tingkat dropout untuk koneksi rekuren dalam sel RNN (recurrent_dropout)	0.1
Faktor regularisasi L2 (l2_reg)	0.01
Tingkat pembelajaran awal untuk optimizer Adam (learning_rate)	0.001
Ukuran batch (batch_size)	24
epochs	30
Variabel Bebas (jumlah sel per layer)	
Percobaan ke-1	24
Percobaan ke-2	48
Percobaan ke-3	96

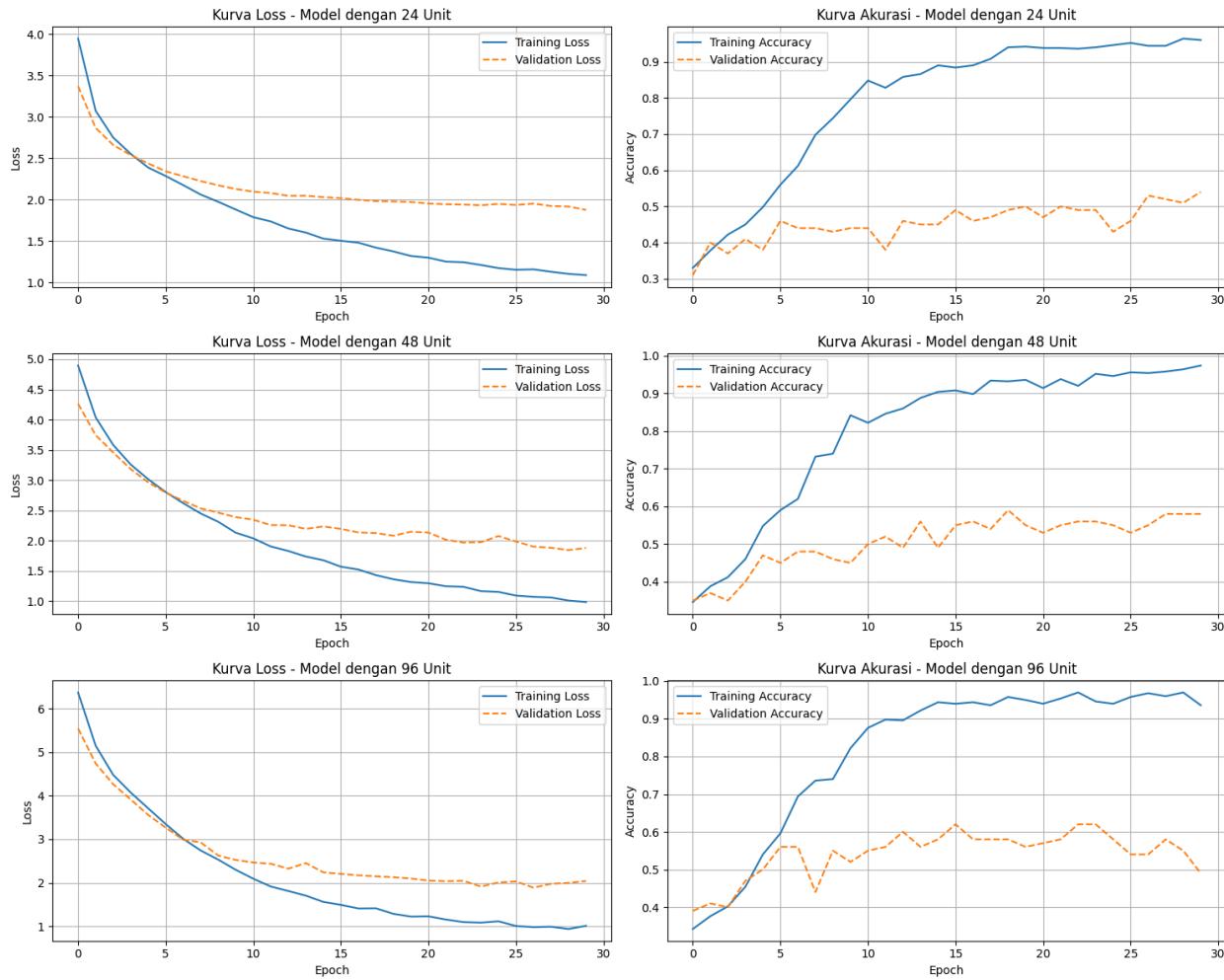
Tabel 3.2.2.2. Hasil Eksperimen Pengaruh Banyak Cell RNN per Layer

	num_units	accuracy	f1_score	loss	train_time
0	24	0.3825	0.358394	2.050279	13.873381
1	48	0.5750	0.546508	1.787028	14.152757
2	96	0.5925	0.560144	1.880376	14.506904

Dari tabel 3.2.2.1, terlihat bahwa peningkatan jumlah unit RNN dari 24 ke 48 unit menghasilkan lonjakan performa yang signifikan, dengan F1-score meningkat dari 0.3584 menjadi 0.5465. Peningkatan lebih lanjut ke 96 unit juga memberikan sedikit kenaikan F1-score menjadi 0.5601. Model dengan 96 unit RNN menunjukkan F1-score tertinggi dalam rangkaian eksperimen ini.



Plot "Pengaruh Jumlah Unit terhadap Performa" secara visual mengonfirmasi tren pada Tabel 2. Kenaikan performa (F1-score dan akurasi) paling drastis terjadi saat jumlah unit ditingkatkan dari 24 ke 48. Peningkatan dari 48 ke 96 unit masih menunjukkan sedikit perbaikan, namun dengan margin yang lebih kecil. Plot "Pengaruh Jumlah Unit terhadap Waktu Training" menunjukkan bahwa waktu pelatihan sedikit meningkat seiring dengan bertambahnya jumlah unit, meskipun peningkatannya tidak terlalu signifikan dibandingkan dengan dampak penambahan layer. Model dengan 24 unit membutuhkan sekitar 13.87 detik, 48 unit sekitar 14.15 detik, dan 96 unit sekitar 14.51 detik.



Analisis kurva menunjukkan bahwa model dengan 24 unit memiliki kapasitas yang paling terbatas, terlihat dari performa training dan validasi yang lebih rendah serta generalization gap yang signifikan. Ketika jumlah unit ditingkatkan menjadi 48 dan kemudian 96, terjadi perbaikan yang jelas pada kurva loss dan akurasi baik untuk data training maupun validasi; model mampu mencapai loss yang lebih rendah dan akurasi yang lebih tinggi. Meskipun generalization gap (perbedaan antara performa training dan validasi) tetap ada pada model dengan 48 dan 96 unit, tidak terlihat adanya peningkatan overfitting yang drastis pada 96 unit dibandingkan 48 unit, mengindikasikan bahwa kapasitas model hingga 96 unit masih dapat ditoleransi dengan baik oleh dataset yang digunakan.

Dari eksperimen ini, dapat disimpulkan bahwa jumlah unit RNN per layer memiliki pengaruh yang signifikan terhadap performa model. Model dengan 24 unit memiliki kapasitas yang terlalu kecil dan menghasilkan F1-score yang rendah. Peningkatan jumlah unit menjadi 48 memberikan lonjakan performa yang besar. Penambahan lebih lanjut menjadi 96 unit memberikan sedikit peningkatan tambahan pada F1-score, menunjukkan adanya titik jenuh atau diminishing returns setelah kapasitas tertentu tercapai untuk

dataset dan arsitektur ini. Meskipun waktu pelatihan sedikit meningkat dengan penambahan unit, peningkatannya tidak sebesar dampak penambahan jumlah layer. Untuk konfigurasi 1 layer bidirectional RNN, jumlah unit sekitar 48 hingga 96 tampaknya memberikan keseimbangan yang baik antara kompleksitas model dan kemampuan generalisasi pada dataset NusaX-Sentiment.

3.2.3. Pengaruh Jenis Layer RNN Berdasarkan Arah

Eksperimen ketiga bertujuan untuk mengevaluasi pengaruh penggunaan arsitektur RNN unidirectional (satu arah) dibandingkan dengan bidirectional (dua arah) terhadap performa klasifikasi sentimen.

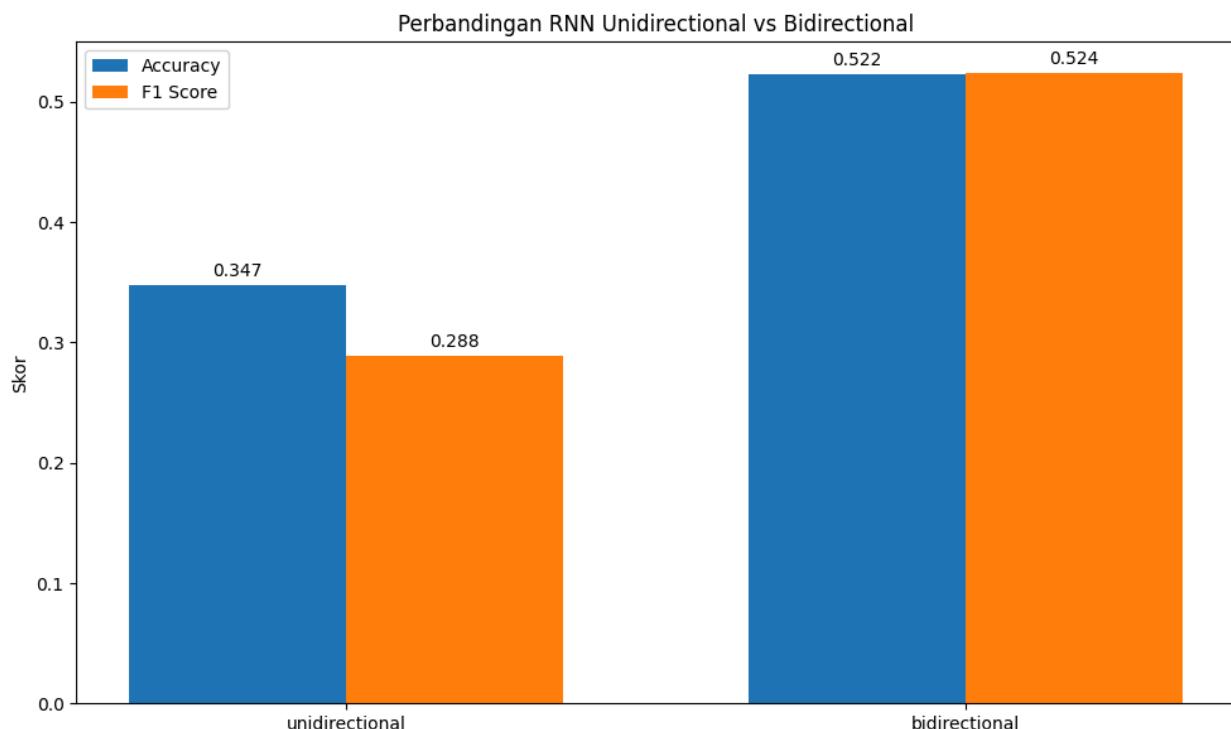
Tabel 3.2.3.1. Pengujian Pengaruh Jenis Layer RNN Berdasarkan Arah

Variabel Tetap	
Jumlah layer RNN (num_rnn_layers)	1
Dimensi embedding (embedding_dim)	96
Jumlah sel per layer (rnn_units)	48
Tingkat dropout setelah layer RNN (dropout_rate)	0.3
Tingkat dropout setelah layer embedding (embedding_dropout)	0.2
Tingkat dropout untuk koneksi rekuren dalam sel RNN (recurrent_dropout)	0.1
Faktor regularisasi L2 (l2_reg)	0.01
Tingkat pembelajaran awal untuk optimizer Adam (learning_rate)	0.001
Ukuran batch (batch_size)	24
epochs	30
Variabel Bebas (jenis layer RNN berdasarkan arah)	
Percobaan ke-1	Unidirectional
Percobaan ke-2	Bidirectional

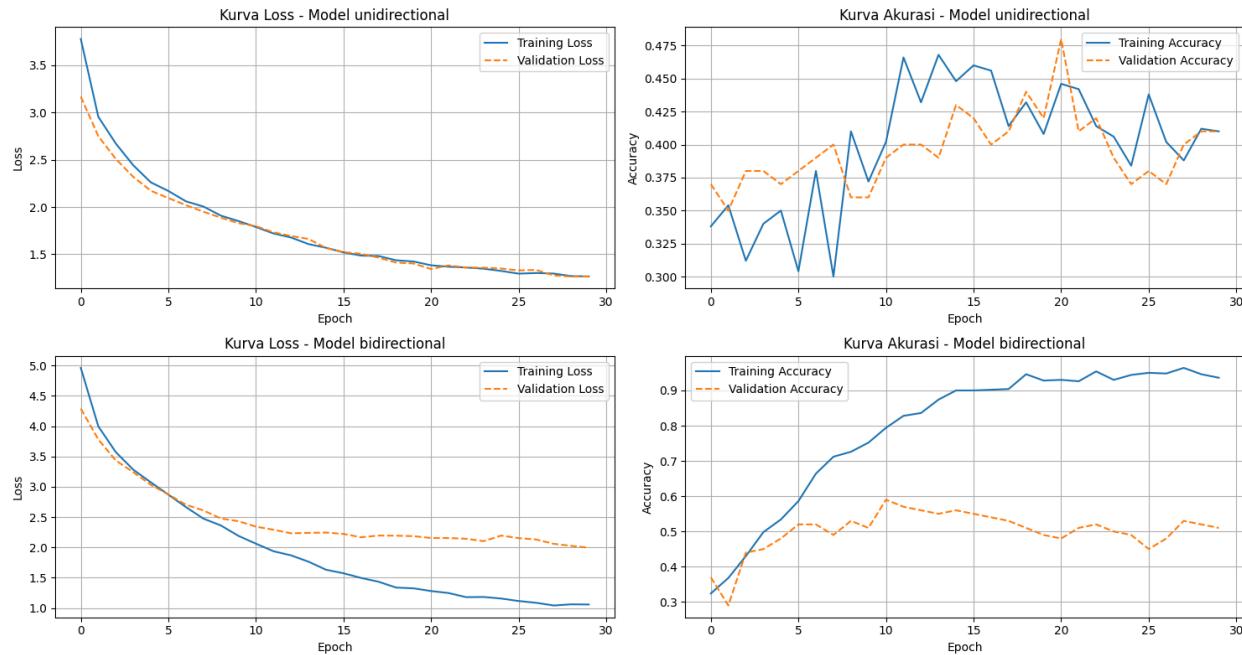
Tabel 3.2.3.2. Hasil Eksperimen Pengaruh Jenis Layer RNN Berdasarkan Arah

	is_bidirectional	direction	accuracy	f1_score	loss	train_time
0	False	unidirectional	0.3475	0.288469	1.316145	8.657205
1	True	bidirectional	0.5225	0.523727	2.052382	14.615227

Berdasarkan tabel 3.2.3.2, terlihat perbedaan performa yang sangat signifikan antara kedua jenis arsitektur. Model bidirectional RNN menghasilkan F1-score macro 0.5237, jauh mengungguli model unidirectional RNN yang hanya mencapai F1-score 0.2885. Akurasi model bidirectional (0.5225) juga jauh lebih tinggi dibandingkan unidirectional (0.3475). Waktu pelatihan untuk model bidirectional (sekitar 14.6 detik) lebih lama dibandingkan unidirectional (sekitar 8.7 detik), yang wajar mengingat model bidirectional memproses sekuens dari dua arah sehingga komputasinya lebih banyak.



Grafik batang dengan jelas menunjukkan superioritas model bidirectional. Baik F1-score maupun akurasi untuk model bidirectional secara substansial lebih tinggi daripada model unidirectional.



Perbandingan kurva pembelajaran antara model unidirectional dan bidirectional secara jelas memperlihatkan perbedaan signifikan dalam kemampuan belajar. Model unidirectional menunjukkan kurva loss (baik training maupun validasi) yang cenderung tinggi dan tidak banyak menurun, serta kurva akurasi yang stagnan pada level rendah, mengindikasikan kesulitan model untuk mempelajari pola representatif dari data secara efektif. Sebaliknya, model bidirectional menampilkan penurunan training loss yang lebih konsisten dan peningkatan training accuracy yang lebih baik, disertai dengan performa kurva validasi (loss dan akurasi) yang jauh lebih superior dibandingkan model unidirectional, meskipun generalization gap antara performa training dan validasi masih teramat. Hal ini menegaskan bahwa kemampuan model bidirectional untuk memproses informasi dari kedua arah sekvens memberikan keunggulan dalam menangkap pola data yang lebih kompleks.

Hasil eksperimen dengan sangat jelas menunjukkan bahwa arsitektur bidirectional RNN secara signifikan lebih unggul daripada unidirectional RNN untuk tugas klasifikasi sentimen pada dataset NusaX-Sentiment dengan konfigurasi yang diuji. Model bidirectional mampu menangkap informasi kontekstual dari kedua arah sekvens (masa lalu dan masa depan), yang krusial untuk pemahaman makna dalam teks. Peningkatan F1-score yang lebih dari 0.23 (dari 0.2885 menjadi 0.5237) menegaskan pentingnya pemrosesan dua arah ini. Meskipun waktu pelatihannya lebih lama, peningkatan performa yang dihasilkan sangat signifikan, menjadikannya pilihan yang jauh lebih baik untuk kasus ini. Kegagalan model unidirectional untuk mempelajari beberapa kelas secara efektif semakin memperkuat kesimpulan ini.

3.3. Long-Short Term Memory Network (LSTM)

3.3.1. Pengaruh Jumlah Layer LSTM

Eksperimen pertama untuk model LSTM bertujuan mengamati bagaimana variasi jumlah layer LSTM—yaitu 1, 2, dan 3 layer—memengaruhi kinerja klasifikasi dan waktu yang dibutuhkan untuk pelatihan.

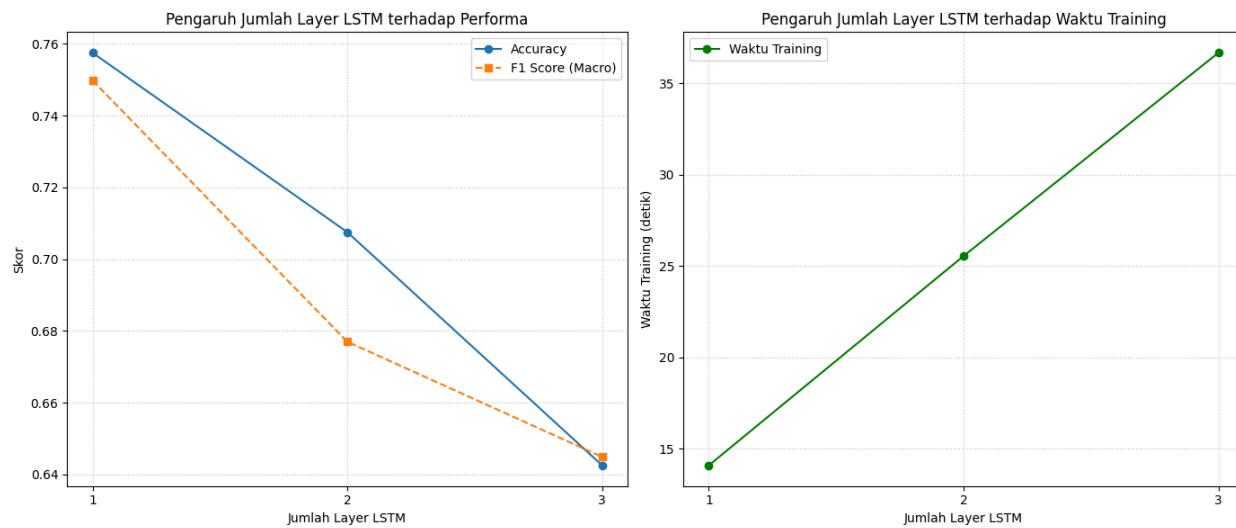
Tabel 3.3.1.1. Pengujian Pengaruh Jumlah Layer LSTM

Variabel Tetap	
Jumlah sel per layer (lstm_units)	64
Dimensi embedding (embedding_dim)	128
Arah LSTM	Bidirectional
Tingkat dropout setelah layer LSTM (dropout_rate)	0.2
Tingkat dropout setelah layer embedding (embedding_dropout_rate)	0.1
Tingkat dropout untuk koneksi rekuren dalam sel LSTM (recurrent_dropout_rate)	0.05
Faktor regularisasi L2 (l2_reg)	0.0001
Tingkat pembelajaran awal untuk optimizer Adam (learning_rate)	0.001
Ukuran batch (batch_size)	32
epochs	24
Variabel Bebas (jumlah layer LSTM)	
Percobaan ke-1	1
Percobaan ke-2	2
Percobaan ke-3	3

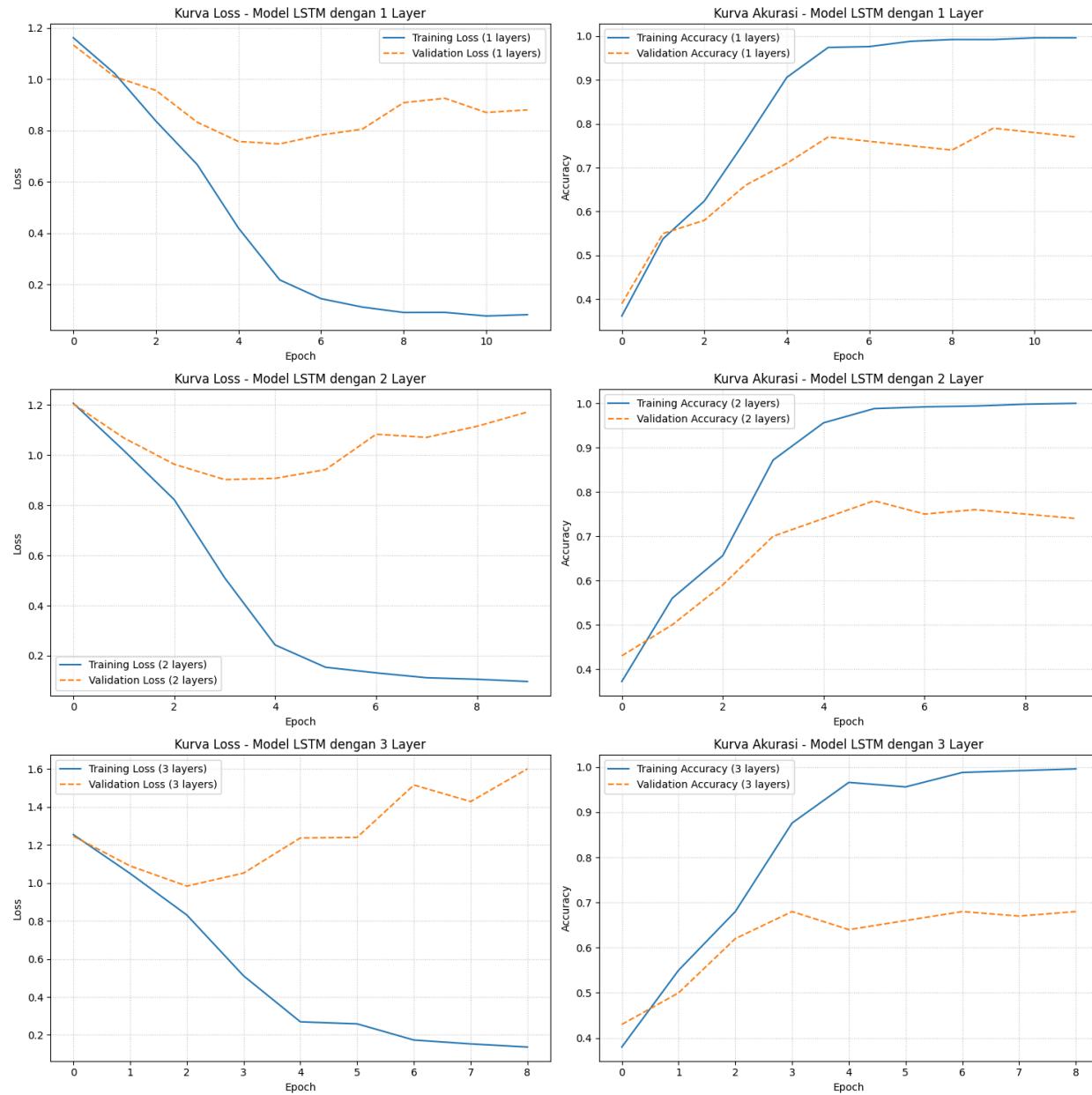
Tabel 3.3.1.2. Hasil Eksperimen Pengaruh Jumlah Layer LSTM

num_lstm_layers	accuracy	f1_score	loss	train_time
0	1	0.7575	0.749760	0.711254
1	2	0.7075	0.676969	0.826143
2	3	0.6425	0.644862	0.909501
				36.665589

Berdasarkan tabel 3.3.1.2, model LSTM dengan 1 layer menunjukkan performa terbaik dengan F1-score macro mencapai 0.7498 dan akurasi 0.7575. Penambahan jumlah layer menjadi 2 dan 3 layer mengakibatkan penurunan F1-score dan akurasi secara signifikan.



Dari plot terlihat jelas bahwa F1-score dan akurasi tertinggi dicapai oleh model dengan 1 layer LSTM. Kedua metrik ini menurun ketika jumlah layer ditingkatkan menjadi 2, dan terus menurun untuk model dengan 3 layer. Sementara itu, plot "Pengaruh Jumlah Layer LSTM terhadap Waktu Training" menunjukkan bahwa waktu pelatihan meningkat secara linear dan cukup signifikan seiring dengan bertambahnya jumlah layer. Model dengan 1 layer memerlukan sekitar 14.1 detik, model 2 layer sekitar 25.5 detik, dan model 3 layer sekitar 36.7 detik untuk pelatihan.



Kurva pembelajaran untuk setiap variasi jumlah layer LSTM menunjukkan bahwa model dengan 1 layer memiliki validation loss yang paling stabil dan paling rendah di antara ketiganya, serta validation accuracy yang tertinggi dan juga paling stabil. Meskipun masih ada generalization gap (performa training lebih baik dari validasi), model 1 layer menunjukkan kemampuan generalisasi yang lebih baik. Seiring dengan penambahan layer menjadi 2 dan 3, kurva validation loss menunjukkan tren meningkat setelah beberapa epoch awal (terutama pada model 3 layer yang sangat jelas), dan validation accuracy cenderung stagnan atau bahkan menurun pada level yang lebih rendah. Hal ini mengindikasikan bahwa model LSTM yang lebih dalam (2 dan 3 layer) mengalami overfitting yang lebih parah pada dataset ini dan kesulitan untuk melakukan generalisasi.

Berdasarkan hasil eksperimen ini, dapat disimpulkan bahwa untuk dataset NusaX-Sentiment dan dengan konfigurasi hyperparameter LSTM lainnya yang dijaga konstan, model LSTM dengan 1 layer memberikan performa klasifikasi terbaik, ditinjau dari F1-score macro (0.7498) dan akurasi (0.7575). Penambahan jumlah layer LSTM menjadi 2 atau 3 layer tidak hanya menurunkan performa secara signifikan tetapi juga memperpanjang waktu pelatihan secara substansial. Model yang lebih dalam menunjukkan kecenderungan overfitting yang lebih kuat, yang mungkin disebabkan oleh kompleksitas model yang berlebihan untuk ukuran dataset yang digunakan. Oleh karena itu, arsitektur dengan satu layer LSTM bidirectional lebih disarankan untuk skenario ini.

3.3.2. Pengaruh Banyak Cell LSTM per Layer

Eksperimen kedua adalah untuk menginvestigasi dampak variasi jumlah unit (atau sel memori) dalam setiap layer LSTM terhadap performa model.

Tabel 3.3.2.1. Pengujian Pengaruh Banyak Cell LSTM per Layer

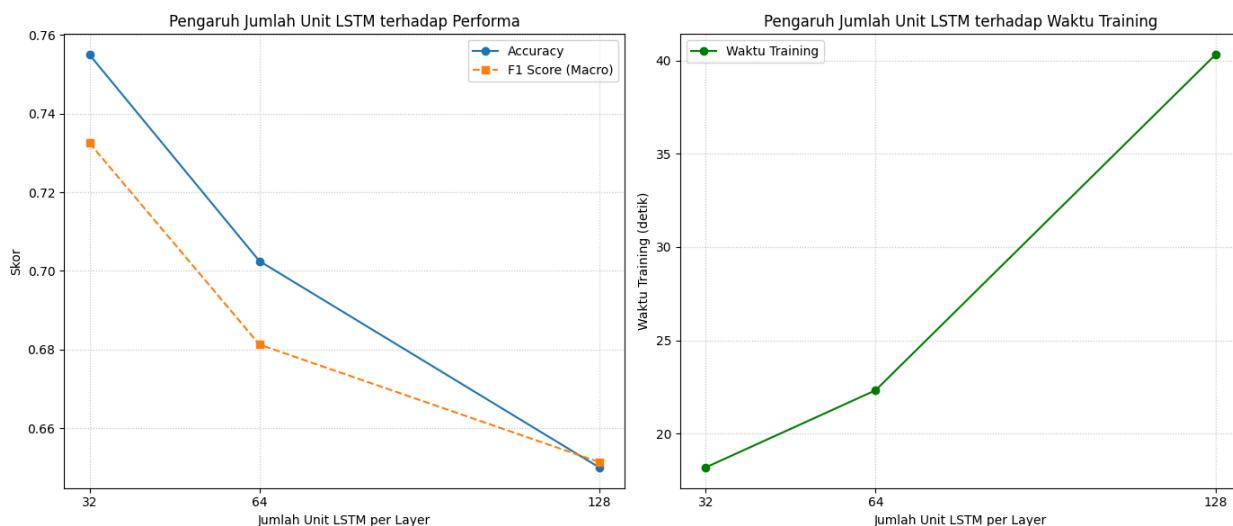
Variabel Tetap	
Jumlah layer LSTM (num_lstm_layers)	2
Dimensi embedding (embedding_dim)	128
Arah LSTM	Bidirectional
Tingkat dropout setelah layer LSTM (dropout_rate)	0.2
Tingkat dropout setelah layer embedding (embedding_dropout_rate)	0.1
Tingkat dropout untuk koneksi rekuren dalam sel LSTM (recurrent_dropout_rate)	0.05
Faktor regularisasi L2 (l2_reg)	0.0001
Tingkat pembelajaran awal untuk optimizer Adam (learning_rate)	0.001
Ukuran batch (batch_size)	32
epochs	24
Variabel Bebas (jumlah sel per layer LSTM)	
Percobaan ke-1	32
Percobaan ke-2	64

Percobaan ke-3	128
----------------	-----

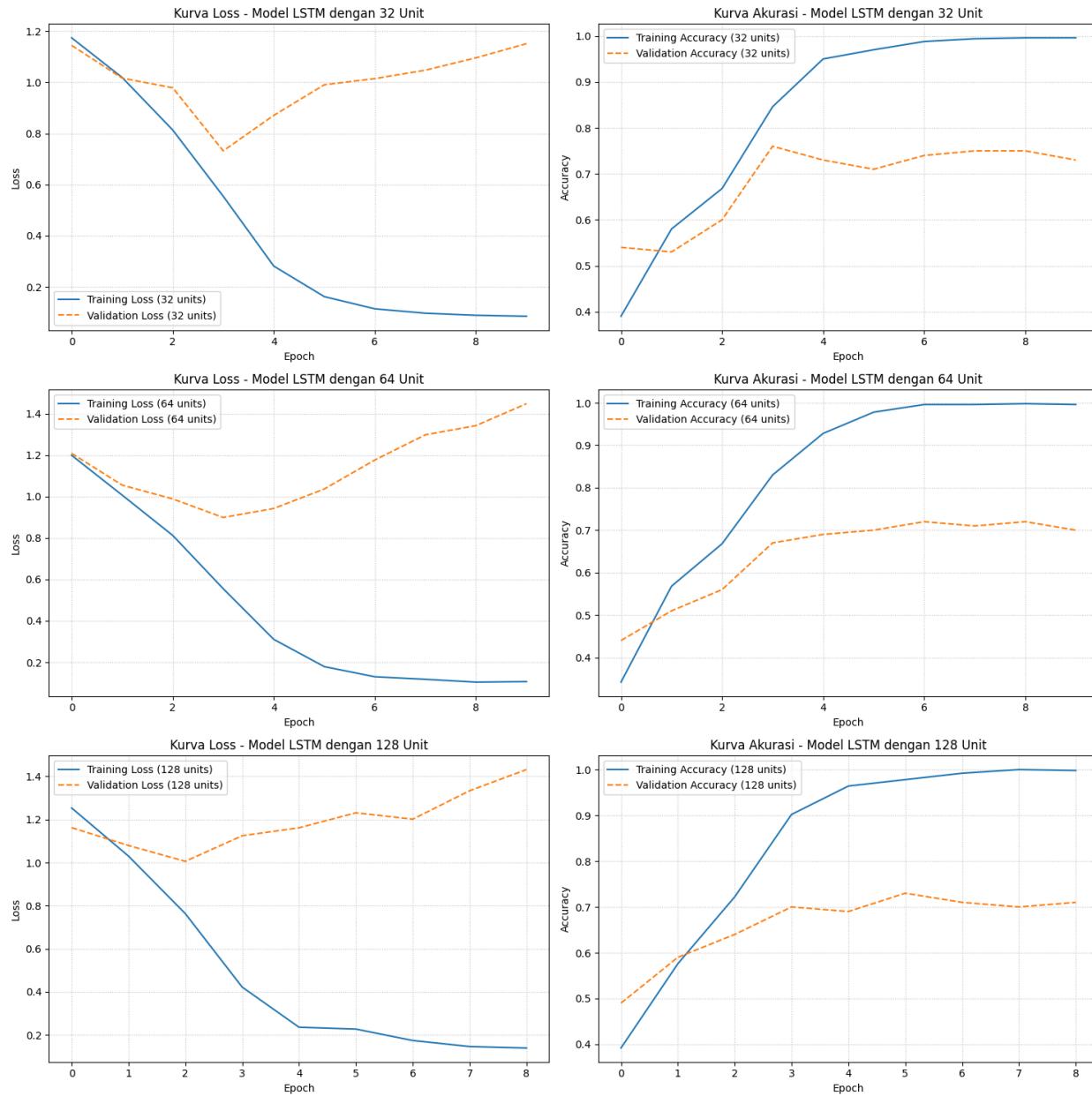
Tabel 3.3.2.2. Hasil Eksperimen Pengaruh Banyak Cell LSTM per Layer

num_lstm_units	accuracy	f1_score	loss	train_time
0	32	0.7550	0.732519	18.185164
1	64	0.7025	0.681263	22.322235
2	128	0.6500	0.651342	40.311298

Dari tabel 3.3.2.2, hasil yang cukup menarik adalah model dengan 32 unit LSTM per layer menunjukkan performa terbaik, dengan F1-score macro mencapai 0.7325 dan akurasi 0.7550. Berbeda dengan ekspektasi teoritis umum atau hasil pada Simple RNN, penambahan jumlah unit menjadi 64 dan kemudian 128 justru mengakibatkan penurunan F1-score dan akurasi secara bertahap untuk konfigurasi 2 layer bidirectional LSTM ini.



Plot "Pengaruh Jumlah Unit LSTM terhadap Performa" mengonfirmasi tren yang tertera pada tabel 3.3.2.2. F1-score dan akurasi tertinggi dicapai oleh model dengan 32 unit, kemudian menurun untuk model dengan 64 unit, dan menurun lebih lanjut untuk model dengan 128 unit. Sebaliknya, plot "Pengaruh Jumlah Unit LSTM terhadap Waktu Training" menunjukkan bahwa waktu pelatihan meningkat secara signifikan seiring dengan bertambahnya jumlah unit. Model dengan 32 unit memerlukan sekitar 18.2 detik, model 64 unit sekitar 22.3 detik, dan model 128 unit memerlukan waktu paling lama, yaitu sekitar 40.3 detik.



Kurva pembelajaran untuk setiap variasi jumlah unit LSTM memperlihatkan dinamika yang menarik. Model dengan 32 unit menunjukkan validation loss yang relatif stabil dan validation accuracy yang paling tinggi di antara ketiganya, dengan generalization gap yang tampak paling terkontrol. Seiring dengan peningkatan jumlah unit menjadi 64 dan 128, meskipun training loss terus menurun dan training accuracy mencapai tingkat tinggi, kurva validation loss untuk model 64 dan 128 unit cenderung meningkat setelah beberapa epoch awal, dan validation accuracynya lebih rendah serta lebih fluktuatif. Ini mengindikasikan bahwa model dengan jumlah unit yang lebih besar (64 dan 128 unit) untuk arsitektur 2 layer ini lebih cepat mengalami overfitting pada dataset yang digunakan.

Berdasarkan hasil eksperimen ini, untuk arsitektur 2 layer bidirectional LSTM pada dataset NusaX-Sentiment, penggunaan 32 unit LSTM per layer memberikan performa terbaik dengan F1-score macro 0.7325. Berlawanan dengan intuisi bahwa lebih banyak unit selalu lebih baik hingga titik tertentu, dalam kasus ini, penambahan jumlah unit menjadi 64 dan 128 justru menurunkan performa dan secara signifikan meningkatkan waktu pelatihan. Analisis kurva pembelajaran menunjukkan bahwa model dengan jumlah unit yang lebih besar lebih rentan terhadap overfitting. Hal ini kemungkinan disebabkan oleh kombinasi dari kedalaman model (2 layer) dan peningkatan kapasitas per layer (lebih banyak unit) yang menjadi terlalu kompleks untuk ukuran dataset yang relatif kecil, sehingga model mulai menghafal data training alih-alih belajar pola yang dapat digeneralisasi.

3.3.3. Pengaruh Jenis Layer LSTM Berdasarkan Arah

Eksperimen terakhir untuk model LSTM ini adalah membandingkan performa antara arsitektur LSTM unidirectional (satu arah) dengan LSTM bidirectional (dua arah).

Tabel 3.3.3.1. Pengujian Pengaruh Jenis Layer LSTM Berdasarkan Arah

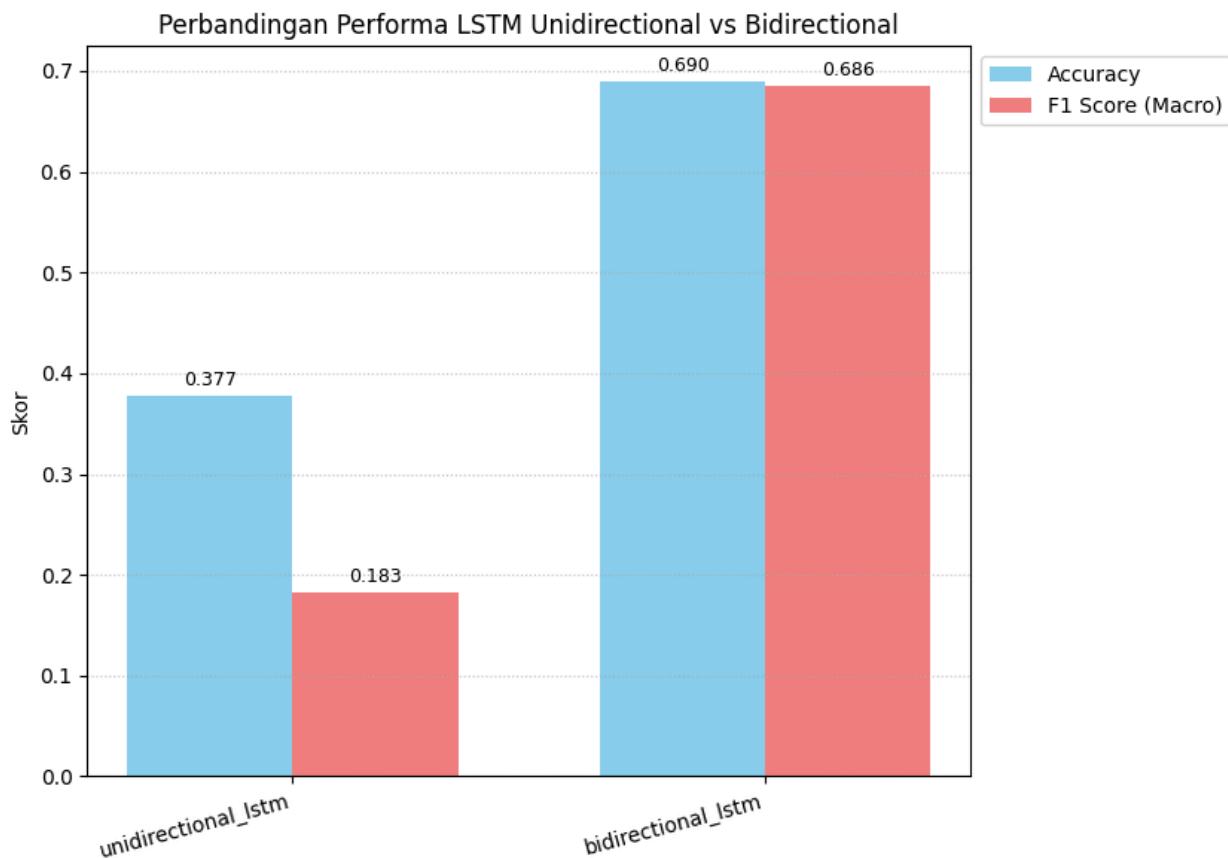
Variabel Tetap	
Jumlah layer LSTM (num_lstm_layers)	2
Dimensi embedding (embedding_dim)	128
Jumlah sel per layer (lstm_units)	64
Tingkat dropout setelah layer LSTM (dropout_rate)	0.2
Tingkat dropout setelah layer embedding (embedding_dropout_rate)	0.1
Tingkat dropout untuk koneksi rekuren dalam sel LSTM (recurrent_dropout_rate)	0.05
Faktor regularisasi L2 (l2_reg)	0.0001
Tingkat pembelajaran awal untuk optimizer Adam (learning_rate)	0.001
Ukuran batch (batch_size)	32
epochs	24
Variabel Bebas (Jenis Layer LSTM Berdasarkan Arah)	
Percobaan ke-1	Unidirectional

Percobaan ke-2	Bidirectional
----------------	---------------

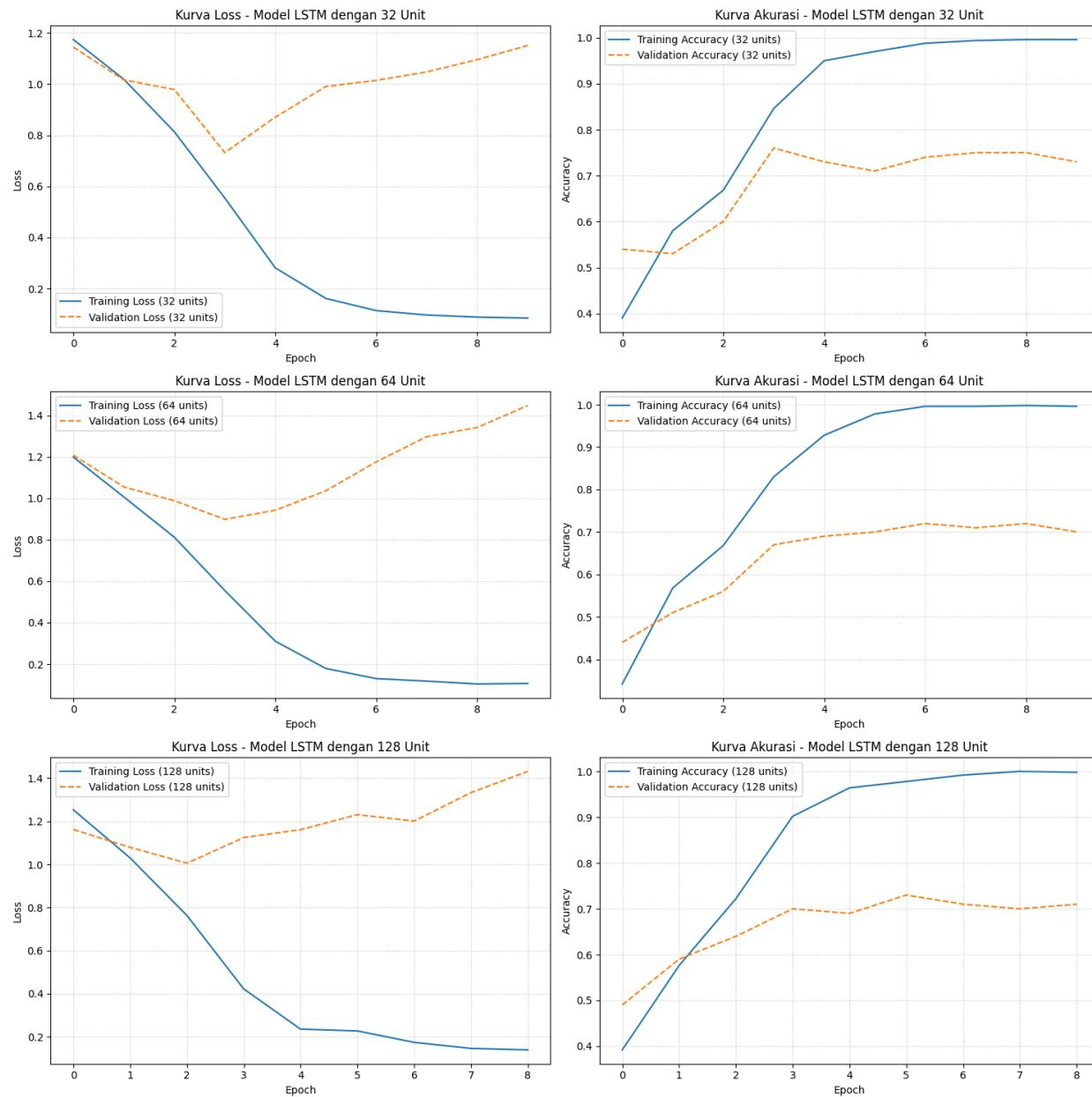
Tabel 3.3.3.2. Hasil Eksperimen Pengaruh Jenis Layer LSTM Berdasarkan Arah

	is_bidirectional_lstm	direction_lstm	accuracy	f1_score	loss	train_time
0	False	unidirectional_lstm	0.3775	0.182698	1.112922	21.134746
1	True	bidirectional_lstm	0.6900	0.685504	0.833162	22.314827

Hasil pada Tabel 3.3.3.2 menunjukkan perbedaan performa yang sangat mencolok. Model bidirectional LSTM mencapai F1-score macro 0.6855, yang secara drastis lebih tinggi dibandingkan model unidirectional LSTM dengan F1-score hanya 0.1827. Akurasi model bidirectional (0.6900) juga jauh melampaui model unidirectional (0.3775). Dari segi waktu pelatihan, model bidirectional (sekitar 22.3 detik) hanya sedikit lebih lama dibandingkan model unidirectional (sekitar 21.1 detik) untuk konfigurasi 2 layer ini, yang menunjukkan bahwa penambahan komputasi untuk pemrosesan dua arah tidak terlalu membebani secara signifikan dibandingkan manfaat performa yang didapatkan.



Grafik batang dengan jelas mengilustrasikan keunggulan signifikan dari model bidirectional LSTM. Terdapat lonjakan besar pada F1-score dan akurasi ketika beralih dari unidirectional ke bidirectional.



Perbandingan kurva pembelajaran antara model LSTM unidirectional dan bidirectional mengungkapkan perbedaan mendasar dalam kemampuan belajar kedua model. Model unidirectional menunjukkan kurva loss (training dan validasi) yang relatif datar dan tinggi, serta kurva akurasi yang sangat berfluktuasi dan berada pada level rendah, mengindikasikan bahwa model ini sangat kesulitan untuk mempelajari pola yang bermakna dari data dan bahkan gagal mengklasifikasikan beberapa kelas secara efektif.

(seperti terlihat pada laporan klasifikasi di output teks yang menunjukkan presisi dan recall 0.00 untuk kelas 'negative' dan 'neutral'). Sebaliknya, model bidirectional LSTM menunjukkan kurva training loss yang menurun dengan baik dan training accuracy yang meningkat secara konsisten. Meskipun kurva validation loss untuk model bidirectional mulai menunjukkan tanda-tanda kenaikan (indikasi overfitting) setelah beberapa epoch, dan validation accuracy cenderung mendatar, performa validasinya secara keseluruhan jauh lebih superior dibandingkan model unidirectional.

Eksperimen ini dengan tegas menunjukkan bahwa arsitektur bidirectional LSTM secara signifikan lebih superior daripada unidirectional LSTM untuk tugas klasifikasi sentimen pada dataset NusaX-Sentiment dengan konfigurasi 2 layer dan 64 unit per layer. Kemampuan model bidirectional untuk memproses informasi sekuens dari kedua arah (konteks masa lalu dan masa depan) terbukti sangat krusial untuk pemahaman makna teks dan menghasilkan performa klasifikasi yang jauh lebih baik. Peningkatan F1-score yang sangat besar (lebih dari 0.50, dari 0.1827 menjadi 0.6855) menggarisbawahi pentingnya pemrosesan dua arah ini. Meskipun waktu pelatihannya sedikit lebih lama, peningkatan performa yang dramatis menjadikannya pilihan yang tidak diragukan lagi untuk kasus ini. Model unidirectional gagal total dalam mempelajari dataset secara efektif.

BAB IV

Kesimpulan dan Saran

4.1. Kesimpulan

Berdasarkan serangkaian eksperimen yang dilakukan pada Tugas Besar 2, dapat ditarik beberapa kesimpulan penting mengenai pengaruh berbagai *hyperparameter* terhadap kinerja model Convolutional Neural Network (CNN), Simple Recurrent Neural Network (Simple RNN), dan Long-Short Term Memory (LSTM). Untuk CNN, berdasarkan hasil pengujian, ditemukan bahwa penggunaan **dua layer konvolusi** menawarkan keseimbangan terbaik antara performa dan efisiensi. Peningkatan dari satu ke dua *layer* menaikkan F1-score secara signifikan (dari 0.6248 menjadi 0.6849), namun penambahan ke tiga *layer* hanya memberikan peningkatan marginal (F1-score 0.6855) dengan biaya waktu *training* yang lebih tinggi, menunjukkan adanya **diminishing returns**. Jumlah *filter* optimal untuk tiga *layer* konvolusi adalah pada konfigurasi [32, 64, 128], yang menghasilkan F1-score 0.7206, lebih unggul dari konfigurasi *filter* yang lebih sedikit ([16, 32, 64] dengan F1-score 0.6640) maupun yang lebih banyak ([64, 128, 256] dengan F1-score 0.7111), di mana penambahan *filter* berlebih juga meningkatkan waktu *training* secara drastis. Penggunaan **kernel 3x3 yang seragam** pada semua *layer* konvolusi terbukti paling efektif, mencapai F1-score 0.6982 dan *Test Accuracy* 0.7016. Selain itu, **Max Pooling** secara konsisten menghasilkan F1-score yang lebih tinggi (0.7133) dibandingkan *Average Pooling* (0.6984) untuk *dataset* CIFAR-10. Penting dicatat bahwa semua konfigurasi CNN menunjukkan tanda-tanda **overfitting**.

Pada model rekuren, baik Simple Recurrent Neural Network (Simple RNN) maupun Long-Short Term Memory (LSTM), **arsitektur Bidirectional** secara signifikan mengungguli arsitektur **Unidirectional** dalam tugas klasifikasi sentimen pada *dataset* NusaX-Sentiment. Untuk Simple RNN, model **Bidirectional** mencapai F1-score 0.5237 dibandingkan 0.2885 untuk **Unidirectional**, dan untuk LSTM, perbedaannya lebih dramatis dengan F1-score 0.6855 untuk **Bidirectional** melawan 0.1827 untuk **Unidirectional**. Hal ini menegaskan pentingnya pemrosesan konteks dari kedua arah sekuens. Untuk kedua jenis RNN ini, **model yang lebih dangkal (satu layer)** cenderung memberikan performa terbaik; Simple RNN satu *layer* mencapai F1-score 0.5937 dan LSTM satu *layer* mencapai F1-score 0.7498, serta lebih tahan terhadap **overfitting** dibandingkan model yang lebih dalam (dua atau tiga *layer*). Jumlah **cell** atau **unit** per *layer* juga menunjukkan pengaruh signifikan. Untuk Simple RNN satu *layer*, peningkatan dari 24 **unit** (F1-score 0.3584) ke **48 unit (F1-score 0.5465)** memberikan lonjakan performa besar, dengan sedikit peningkatan lagi pada 96 **unit** (F1-score 0.5601). Sementara itu, untuk LSTM dua *layer* **bidirectional**, **32 unit** secara mengejutkan memberikan hasil terbaik (F1-score 0.7325), dan penambahan **unit** lebih lanjut justru menurunkan performa, kemungkinan karena kompleksitas model yang berlebihan untuk *dataset* yang digunakan. Fenomena

overfitting juga menjadi tantangan konsisten pada model RNN dan LSTM, terutama pada arsitektur yang lebih dalam atau lebih kompleks.

4.2. Saran

Dalam pengembangan model *deep learning* untuk tugas klasifikasi gambar serupa, disarankan untuk memulai dengan **arsitektur yang relatif dangkal, seperti dua layer konvolusi**, yang terbukti memberikan keseimbangan performa-efisiensi. Kombinasikan ini dengan **jumlah filter yang moderat, misalnya skema seperti [32, 64, 128] untuk tiga layer**, yang menunjukkan hasil F1-score yang baik tanpa membebani waktu komputasi secara berlebihan. Penggunaan **kernel 3x3 yang seragam** lebih diutamakan karena efektivitasnya, dan **Max Pooling** sebaiknya dipilih jika F1-score menjadi metrik evaluasi prioritas.

Untuk model pemrosesan sekuens seperti Simple RNN dan LSTM pada tugas klasifikasi teks, sangat disarankan untuk **memprioritaskan penggunaan arsitektur Bidirectional** karena kemampuannya yang signifikan dalam menangkap konteks dari kedua arah sekuens. Mengingat model yang lebih dalam (dua atau tiga *layer*) cenderung cepat mengalami **overfitting** dan menunjukkan penurunan performa pada dataset NusaX-Sentiment yang diuji, maka dianjurkan untuk **memulai dengan satu layer rekuren** dan mengevaluasi hasilnya secara cermat sebelum mempertimbangkan penambahan kedalaman. Jumlah **cell** atau **unit** per *layer* juga perlu **disedesuaikan secara cermat melalui eksperimen (tune)**, karena jumlah optimal dapat bervariasi tergantung kedalaman, tipe arsitektur (Simple RNN atau LSTM), dan karakteristik *dataset*.

Secara umum, mengingat fenomena **overfitting** yang konsisten muncul di berbagai eksperimen baik pada CNN, Simple RNN, maupun LSTM, sangat penting untuk **menerapkan dan memperkuat strategi untuk mengatasinya**, seperti penggunaan teknik regularisasi yang tepat dan mungkin *early stopping* yang lebih agresif jika diperlukan. Pengembang model harus selalu **mempertimbangkan trade-off antara peningkatan performa (F1-score, akurasi) dengan kompleksitas model dan waktu training** yang dibutuhkan. Melakukan eksperimen **hyperparameter** secara sistematis, dengan mengubah satu parameter pada satu waktu sambil menjaga yang lain konstan sebagaimana telah dicontohkan dalam laporan ini akan membantu dalam mengisolasi dampak masing-masing dan menemukan konfigurasi yang optimal untuk tugas spesifik yang dihadapi.

BAB V

Referensi

Tim Pengajar IF3270 Pembelajaran Mesin. 2024/2025. Salindia Perkuliahan Convolutional Neural Network (CNN)

Tim Pengajar IF3270 Pembelajaran Mesin. 2024/2025. Salindia Perkuliahan Recurrent Neural Network (RNN) Bagian 1

Tim Pengajar IF3270 Pembelajaran Mesin. 2024/2025. Salindia Perkuliahan Recurrent Neural Network (RNN) Bagian 2

Keras. (n.d.). SimpleRNN layer. Retrieved May 30, 2025, from https://keras.io/api/layers/recurrent_layers/simple_rnn/

TensorFlow. (n.d.). SparseCategoricalCrossentropy. Retrieved May 30, 2025, from https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Keras. (n.d.). Adam optimizer. Retrieved May 30, 2025, from <https://keras.io/api/optimizers/adam/>

Spectrum IEEE. (n.d.). Deep learning's computational cost. Retrieved May 30, 2025, from <https://spectrum.ieee.org/deep-learning-computational-cost>

Lampiran

Repository GitHub

https://github.com/alandmprtma/IF3270_Tubes2_62

Pembagian Kerja Kelompok

NIM	Nama	Pekerjaan
13522124	Aland Mulia Pratama	Mengimplementasikan CNN (Convolutional Neural Network) yang mencakup pembuatan file konfigurasi, preprocessing data, model, train, dan juga implementasi forward propagation from scratch. Mengerjakan laporan Deskripsi Persoalan, Implementasi CNN, Pengujian CNN, serta Daftar Pustaka.
13522132	Hafizh Hananta Akbari	Tidak Mengerjakan Sama Sekali
13522135	Christian Justin Hendrawan	Mengimplementasikan Simple RNN dan LSTM yang mencakup pembuatan file konfigurasi, preprocessing, model, train, dan implementasi from scratch. Mengerjakan laporan implementasi dan hasil pengujian Simple RNN dan LSTM, Kesimpulan & Saran, serta merapihkan format laporan.