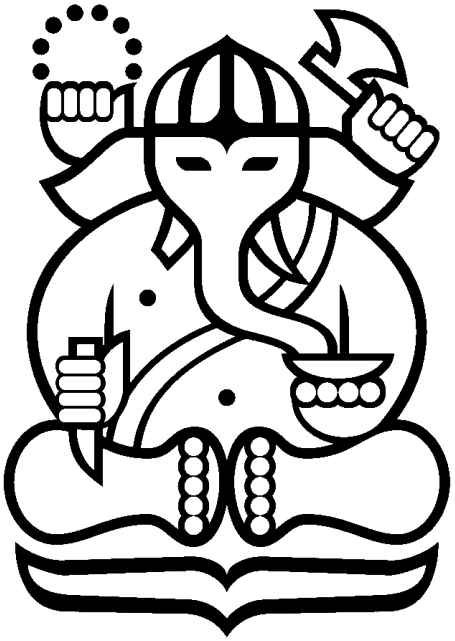
LAPORAN TUGAS BESAR 2

IF3270 PEMBELAJARAN MESIN

Convolutional Neural Network dan Recurrent Neural Network



Disusun Oleh:

13522069 Nabila Shikoofa Muida

13522087 Shulha

13522109 Azmi Mahmud Bazeid

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

DAFTAR TABEL

BAB I
DESKRIPSI PERSOALAN

Dalam era data yang semakin kompleks dan beragam, kebutuhan akan pemrosesan data visual dan teks secara otomatis menjadi sangat penting, khususnya dalam bidang klasifikasi. Convolutional Neural Network (CNN) telah terbukti efektif dalam menangani tugas-tugas yang berkaitan dengan data visual seperti klasifikasi gambar, sedangkan Recurrent Neural Network (RNN) dan Long Short-Term Memory (LSTM) banyak digunakan untuk memproses data sekuensial seperti teks.

Tugas Besar ini dirancang untuk mengeksplorasi dan memahami mekanisme kerja dari CNN, RNN, dan LSTM, khususnya pada proses *forward propagation* yang merupakan inti dari pengambilan keputusan dalam jaringan saraf. Mahasiswa diminta untuk tidak hanya membangun dan melatih model menggunakan library Keras, tetapi juga mengimplementasikan kembali mekanisme *forward propagation* dari awal (*from scratch*) menggunakan library dasar seperti NumPy, guna memperdalam pemahaman terhadap alur data dan operasi matematis dalam jaringan saraf tersebut.

Permasalahan utama yang diangkat adalah bagaimana membangun model klasifikasi yang efektif dan akurat pada dua domain data yang berbeda, yaitu gambar dan teks, serta bagaimana mengevaluasi pengaruh berbagai *hyperparameter*terhadap performa model. Selain itu, tantangan tambahan juga terdapat pada pembuatan modul *forward propagation*modular yang dapat meniru hasil dari model yang dilatih dengan Keras, sebagai validasi terhadap keakuratan implementasi manual yang dilakukan.

BAB II
IMPLEMENTASI

2.1. Implementasi Convolutional Neural Network

Implementasi Convolutional Neural Network (CNN) yang dikembangkan terdiri dari beberapa komponen utama yang bekerja secara terintegrasi untuk melakukan klasifikasi gambar pada dataset CIFAR-10. Implementasi ini mencakup arsitektur CNN yang fleksibel, sistem pelatihan model, eksperimen hyperparameter, dan implementasi forward propagation kustom dari scratch menggunakan NumPy.

Sistem ini dirancang untuk melakukan eksperimen komprehensif terhadap berbagai konfigurasi CNN, termasuk jumlah layer konvolusi, jumlah filter, ukuran kernel, dan jenis pooling. Selain itu, implementasi forward propagation kustom dibuat untuk memverifikasi bahwa pemahaman terhadap operasi CNN sudah benar dengan membandingkan hasilnya dengan implementasi Keras.

2.1.1. Deskripsi Kelas, Atribut, dan Method

2.1.1.1 Kelas CNNCustomForwardPropagation

Kelas CNNCustomForwardPropagation merupakan kelas utama yang mengimplementasikan forward propagation CNN secara manual menggunakan NumPy.

Atribut:

- `model_path`: String yang menyimpan path ke file weights model (.h5)
- `architecture_path`: String yang menyimpan path ke file arsitektur model (.json)
- `weights`: Dictionary yang menyimpan weights untuk setiap layer yang memiliki parameter
- `biases`: Dictionary yang menyimpan bias untuk setiap layer yang memiliki parameter
- `architecture`: Menyimpan konfigurasi arsitektur model dari Keras

Method:

- `__init__(model_path, architecture_path)`: Constructor yang menginisialisasi path model dan memanggil load_model()
- `load_model()`: Memuat model dari file JSON dan mengekstrak weights serta biases dari setiap layer
- `relu(x)`: Implementasi fungsi aktivasi ReLU (Rectified Linear Unit)
- `softmax(x)`: Implementasi fungsi aktivasi softmax dengan stabilisasi numerik
- `convolutional_2d_forward()`: Operasi konvolusi 2D dengan dukungan padding 'same' dan 'valid'
- `max_pooling_2d_forward()`: Operasi max pooling 2D
- `average_pooling_2d_forward()`: Operasi average pooling 2D
- `dense_forward()`: Operasi fully connected layer
- `global_average_pooling_2d_forward()`: Operasi global average pooling
- `flatten_forward()`: Operasi flatten untuk mengubah tensor 4D menjadi 2D
- `forward(x)`: Method utama untuk menjalankan forward propagation lengkap
- `predict(x)`: Method wrapper untuk melakukan prediksi

2.1.1.2 Kelas CompileCNNArchitecture

Kelas CompileCNNArchitecture bertanggung jawab untuk membangun arsitektur CNN yang fleksibel dengan berbagai konfigurasi hyperparameter.

Atribut: Kelas ini tidak memiliki atribut instance tapi memakai konstanta-konstanta dari config.py.

Method:

- `create_cnn_model(conv_layers, filters_per_layer, filter_sizes, pooling_type)`: Method utama untuk membuat model CNN Sequential dengan parameter:
  - `conv_layers`: Jumlah layer konvolusi
  - `filters_per_layer`: List jumlah filter untuk setiap layer
  - `filter_sizes`: List ukuran kernel untuk setiap layer
  - `pooling_type`: Jenis pooling ('max' atau 'average')

2.1.1.3 Kelas CNNModelTrainer

Kelas CNNModelTrainer menangani proses pelatihan model CNN menggunakan Keras/TensorFlow dan evaluasi performa model.

Atribut:

- `x_train`: Data training (fitur)
- `y_train`: Label training
- `x_val`: Data validasi (fitur)
- `y_val`: Label validasi
- `x_test`: Data testing (fitur)
- `y_test`: Label testing

Method:

- `__init__(data_splits)`: Constructor yang menerima dictionary berisi split data
- `train_model(model, experiment_name, epochs, batch_size)`: Method utama untuk melatih model dengan:
  - Proses training menggunakan model.fit()
  - Evaluasi pada test set untuk mendapatkan loss dan accuracy
  - Perhitungan F1-score macro
  - Penyimpanan weights dan arsitektur model
  - Mengembalikan dictionary berisi hasil training

2.1.1.4 Kelas CNNExperiments

Kelas CNNExperiments mengatur dan menjalankan berbagai eksperimen hyperparameter secara sistematis untuk membandingkan performa berbagai konfigurasi CNN.

Atribut:

- `trainer`: Instance CNNModelTrainer untuk proses training
- `model_builder`: Instance CompileCNNArchitecture untuk membangun model
- `visualizer`: Instance ExperimentVisualizer untuk visualisasi hasil

- `results`: Dictionary yang menyimpan hasil semua eksperimen

Method:

- `__init__(data_splits)`: Constructor yang menginisialisasi semua komponen
- `experiment_conv_layers()`: Eksperimen efek jumlah layer konvolusi
- `experiment_num_filters()`: Eksperimen efek jumlah filter per layer
- `experiment_filter_sizes()`: Eksperimen efek ukuran kernel/filter
- `experiment_pooling_types()`: Eksperimen efek jenis pooling layer
- `run_all_experiments()`: Menjalankan semua eksperimen secara berurutan
- `_save_results()`: Method private untuk menyimpan hasil ke file JSON

2.1.1.5 Kelas ExperimentVisualizer

Kelas ExperimentVisualizer bertanggung jawab untuk visualisasi hasil eksperimen dalam bentuk grafik dan summary statistik.

Atribut: Kelas ini tidak memiliki atribut instance.

Method:

- `plot_comparison(results, title, filename)`: Membuat visualisasi perbandingan dengan tiga subplot:
  - Training loss comparison
  - Validation loss comparison
  - F1-score comparison
- `print_summary(results)`: Mencetak ringkasan hasil semua eksperimen dalam format tabel

2.1.1.6 Kelas LoadCifar10

Kelas LoadCifar10 menangani loading dan preprocessing dataset CIFAR-10, termasuk normalisasi dan pembagian data.

Atribut:

- `x_train`: Data training setelah split

- `y_train`: Label training setelah split
- `x_val`: Data validasi
- `y_val`: Label validasi
- `x_test`: Data testing
- `y_test`: Label testing

Method:

- `__init__()`: Constructor yang menginisialisasi atribut dengan None
- `load_and_prepare_data()`: Method utama untuk:
    - Loading CIFAR-10 dari keras.datasets
    - Normalisasi pixel values ke range [0,1]
    - Flattening labels
    - Random split untuk training dan validasi
    - Validasi ukuran dataset
- `get_data_splits()`: Mengembalikan dictionary berisi semua split data
- `get_test_subset(size)`: Mengembalikan subset dari test data dengan ukuran tertentu

2.1.2. Penjelasan Forward Propagation

Forward propagation diimplementasikan sebagai pipeline sequential yang memproses input melalui setiap layer dalam arsitektur CNN. Implementasi ini mengikuti prinsip bahwa setiap layer menerima output dari layer sebelumnya dan menghasilkan output untuk layer selanjutnya.

i. Convolutional Layer

Implementasi layer konvolusi menggunakan pendekatan manual dengan nested loops untuk menghitung setiap posisi output. Proses meliputi:

1. Perhitungan dimensi output berdasarkan formula: output_size = (input_size - filter_size + 2*padding) / stride + 1.

2. Penerapan padding 'same' dan 'valid' dengan perhitungan otomatis.
3. Operasi konvolusi menggunakan element-wise multiplication dan summation.
4. Penambahan bias pada setiap output channel.

ii. Pooling Layers

Dua jenis pooling diimplementasikan:

- Max Pooling: Mengambil nilai maksimum dari setiap window
- Average Pooling: Menghitung rata-rata dari setiap window
- Global Average Pooling: Menghitung rata-rata seluruh spatial dimension

iii. Dense Layer

Implementasi fully connected layer menggunakan operasi matrix multiplication standar dengan formula: output = f(input × weights + bias) dengan f adalah fungsi aktivasi.

Sistem validasi membandingkan output custom implementation dengan Keras implementation dengan menggunakan model dan weights yang sama. Lalu, dihitung F1-score macro untuk evaluasi performa dan untuk memastikan bahwa hasil forward propagationnya sama dilakukan assertion untuk memastikan output identik. Testing dilakukan dengan subset 1000 sampel dari test set.

2.1.3. Penjelasan Implementasi Backward Propagation

Backward propagation adalah proses menghitung gradien error terhadap setiap parameter dalam network untuk melakukan update weights. Dalam implementasi ini, backward propagation tidak diimplementasikan secara manual karena fokus tugas adalah pada forward propagation saja. Implementasi menggunakan library Keras/TensorFlow untuk menangani backward propagation**.**

2.2. Implementasi Text Preprocessing untuk Simple RNN dan LSTM

Sebelum melatih dan mengimplementasikan model Simple RNN maupun LSTM, terlebih dahulu dilakukan *preprocessing* terhadap dataset sentimen NusaX-Sentiment (Bahasa Indonesia). Implementasi dilakukan pada file ‘text_eda.py’ dan ‘text_preprocessor.py’ pada direktori ‘text-preprocessing’.

1. Eksplorasi Awal (EDA)

   Kami melakukan exploratory data analysis (EDA) terhadap kolom teks untuk memahami karakteristik panjang kalimat dalam dataset. Dari hasil analisis distribusi jumlah kata per kalimat, kami menemukan bahwa 95% kalimat memiliki panjang ≤ 55 kata. Berdasarkan itu, kami menetapkan SEQUENCE_LENGTH = 55, agar sebagian besar teks bisa diproses tanpa banyak informasi yang terpotong.

2. Tokenization

   Tokenization adalah proses mengubah kalimat menjadi unit-unit yang lebih kecil (disebut token), yang kemudian direpresentasikan sebagai bilangan bulat (integer). Untuk melakukannya, kami menggunakan TextVectorization layer dari Keras, yang secara otomatis memetakan setiap kata dalam teks menjadi indeks unik sesuai urutan frekuensi kemunculan kata dalam dataset.

   Parameter output_mode='int' dipilih karena kami membutuhkan representasi urutan kata asli sebagai input ke model RNN, bukan representasi agregat seperti bag-of-words atau TF-IDF. Layer ini kami adaptasi dengan data teks pelatihan, agar ia mempelajari frekuensi dan membuat *vocab*-nya berdasarkan data asli. Setelah tokenisasi, setiap kalimat diubah menjadi array berisi indeks kata dengan padding jika kurang dari 55 kata karena tidak mengimplementasikan ragged.

3. Embedding

   Setelah teks diubah menjadi token integer, kami mengkonversi token-token tersebut menjadi vektor berdimensi tetap melalui *embedding layer* dari Keras. Proses ini penting untuk menangkap informasi semantik dari kata-kata, sehingga kata-kata yang maknanya mirip akan berada dekat di ruang vektor.

   Pada tahap ini, setiap token integer diubah menjadi vektor berdimensi 128 (atau sesuai nilai EMBEDDING_DIM), yang nilainya akan dilatih selama proses training. Kami mengaktifkan ‘mask_zero=True’ karena padding (token 0) tidak memiliki makna dan perlu diabaikan oleh layer RNN.

   Untuk memverifikasi proses ini, kami juga membangun pipeline *preprocessing* dan mendemonstrasikannya terhadap sampel teks. Hasilnya kami cetak untuk memastikan alur ‘text → token → vector’ berjalan sesuai harapan.

2.3. Implementasi Simple Recurrent Neural Network

Secara umum, kami menyelesaikan proyek kami dengan membuat file ‘keras_rnn.py’ untuk model implementasi dengan Keras, ‘custom_rnn.py’ untuk model implementasi dari *scratch*, dan notebook untuk analisis dan eksplorasi.

2.3.1. Deskripsi Kelas, Atribut, dan Method

2.3.1.1. Kelas RNNSentimentClassifier

Kelas ini merupakan kelas implementasi RNN dengan Keras

1. Atribut ‘vocab_size’

   Atribut ini diset dengan nilai ukuran *vocabulary* dataset, default 5000 namun akan menggunakan nilai aktual ukuran *vocabulary* berdasarkan informasi dari tahap pemrosesan teks yang sudah dilakukan.

2. Atribut ‘embedding_dim’

   Atribut ini untuk dipakai untuk nilai ‘output_dim’ atau dimensi output pada layer Embedding.

3. Atribut ‘max_sequence_length’

   Atribut ini diambil dari rata-rata panjang kalimat pada dataset untuk nilai ‘input_dim’ pada layer Embedding.

4. Atribut ‘model’

   Atribut ini mengacu pada model RNN dari objek kelas ini.

5. Atribut ‘history’

   Atribut ini mengacu pada *history training* model dari objek kelas ini.

6. Fungsi ‘build_model’

   Fungsi ini menerima parameter berupa jumlah layer RNN, jumlah unit RNN per layer, arah, jumlah unit pada layer Dense, *dropout rate*, dan *learning rate*. Menggunakan ‘Sequential()’ dari Keras, fungsi ini menambahkan layer Embedding dengan parameter yang sudah didefinisikan pada atribut kelas dan dengan ‘mask_zero=True’ agar model tidak belajar dari *padding*. Setelah itu, menambahkan layer Dropout pertama setelah layer Embedding untuk meregulasi vektor Embedding membantu model tidak overfitting pada kata tertentu atau kombinasi token tertentu sehingga lebih general.

   Fungsi ini kemudian menambah layer RNN sesuai jumlah layer, unit, dan arah pada parameter fungsi. Jika bukan merupakan layer terakhir, menggunakan parameter ‘return_sequences=True’ agar layer RNN berikutnya

dapat membaca urutan penuh dari layer sebelumnya. Sementara itu jika layer RNN terakhir akan menggunakan ‘return_sequences=False agar outputnya hanya satu per kalimat dan vektor bisa diproses di layer Dense. Setelah masing-masing layer selalu ditambahkan layer Dropout.

Setelah itu, ditambah layer Dense yang pertama (FC Hidden 1) barulah layer Output dengan Dense untuk 3 kelas output. Dengan Optimizer Adam, model Keras ini kemudian di-*compile* dan nilainya disimpan pada atribut ‘model’ objek.

7. Fungsi ‘train’

   Fungsi ini menerima parameter berupa dataset training, dataset validasi, jumlah epoch, dan *callbacks*. Jika *callbacks* tidak dijadikan argumen, maka akan menggunakan default yakni ‘EarlyStopping’ jika ‘val_loss’ tidak membaik selama 10 epoch berturut-turut dan ‘ReduceLROnPlateau’ untuk menurunkan learning rate jika model berhenti membaik tanpa peningkatan selama 5 epoch. Kemudian, akan dilakukan *fit* pada model dengan dataset training dan dataset validasi tersebut dan hasil pelatihan (loss, accuracy, dsb.) akan disimpan pada atribut ‘history’ model objek.

8. Fungsi ‘evaluate’

   Fungsi ini menerima parameter berupa dataset test dan nilai label yang sebenarnya (opsional, jika tidak maka akan ekstrak dari dataset test). Model akan memprediksi ‘predict’ masing-masing *probability* nilai masing-masing kelas label dan menentukan label teks berdasarkan *probability* terbesar. Kemudian, dilakukan perhitungan metrik evaluasi *test loss, test accuracy, dan macro F1-score*. Fungsi ini mengembalikan *dictionary* berisi metrik dan hasil prediksi, yang berguna untuk analisis performa model lebih lanjut.

9. Fungsi ‘plot_training_history’

   Fungsi ini menggunakan nilai atribut ‘history’ objek untuk menggambarkan *training and validation loss* dan *training and validation loss accuracy*.

10. Fungsi ‘get_model_summary’

    Fungsi ini mengembalikan ringkasan struktur model, termasuk jumlah total parameter, daftar layer, serta referensi ke metode ‘model.summary’.

11. Fungsi ‘save_model’, ‘load_model’, ‘save_weights’, dan ‘load_weights’

Fungsi ‘save_model’ dengan parameter filepath menyimpan seluruh model (arsitektur, bobot, dan konfigurasi) ke file .keras atau .h5. Fungsi ‘load_model’ dengan parameter filepath memuat kembali model lengkap dari file yang disimpan sebelumnya.

Fungsi ‘save_weights’ dengan parameter filepath menyimpan hanya bobot model ke file dengan ekstensi .weights.h5. Fungsi ‘load_weights’ dengan parameter filepath memuat bobot model dari file ke model yang telah dibangun sebelumnya.

2.3.1.2. Kelas ModelComparator

Kelas ini adalah untuk membandingkan satu model RNN dengan model RNN lainnya untuk kebutuhan visualisasi, analisis, dan eksplorasi.

1. Atribut ‘results’

   Atribut ini adalah array atau list untuk menyimpan results yakni config, metrik, dan history model.

2. Fungsi ‘add_result’

   Fungsi ini menerima parameter config, metrik, dan history untuk ditambahkan pada list ‘results’.

3. Fungsi ‘compare_results’

   Fungsi ini membandingkan *accuracy*, *loss*, dan F1-Score metrik model dan mengembalikannya dalam bentuk dataframe terurut.

4. Fungsi ‘plot_comparison’

   Fungsi ini menampilkan *bar chart* perbandingan model berdasarkan metrik.

5. Fungsi ‘plot_training_curves_comparison’

   Fungsi ini menampilkan *line chart* perbandingan *loss* dan *accuracy* dari model sepanjang epoch pelatihan.

2.3.1.3. Kelas EmbeddingLayer

Kelas ini adalah untuk penerapan layer Embedding *from scratch*.

1. __init__(...)

   Mendeklarasi awal atribut ‘vocab_size’, ‘embedding_dim’, ‘mask_zero’, dan ‘weights’ serta ‘input_shape’ yang merupakan (batch_size,

sequence_length) dari tahap *tokenization* dan 'output_shape' yang merupakan dimensi keluaran layer Embedding ini (batch_size, sequence_length, embedding_dim).

2. Fungsi 'forward'

   Fungsi ini menerima parameter berupa input_token dengan *shape* batch_size, sequence_length. Fungsi ini mengeluarkan output berupa vektor hasil *embedding* dengan *shape* batch_size, sequence_length, embedding_dim. Penjelasan pada bagian berikutnya.

3. Fungsi 'backward'

   Fungsi ini menerima parameter berupa gradien output dengan *shape* batch_size, sequence_length, embedding_dim. Fungsi ini mengeluarkan output berupa gradien weights dengan *shape* sequence_length, embedding_dim.

4. Fungsi 'update_weights'

   Fungsi ini menerima parameter berupa gradien weights dari fungsi backward dan melakukan *update* terhadap nilai *weights* untuk selanjutnya.

2.3.1.4. Kelas SimpleRNNCell

Kelas ini adalah untuk penerapan unit RNN *from scratch*.

1. __init__(...)

   Mendeklarasi awal atribut 'input_size' yakni berapa unit input, 'hidden_size' yakni berapa unit hidden cell ini dan 'activation' yaitu pilihan fungsi aktivasi (default: tanh). Juga mendefinisikan atribut bobot yaitu W_ih (bobot input ke hidden), W_hh (bobot hidden ke hidden recurrent), dan b_hh (bias hidden ke hidden recurrent).

2. Fungsi 'load_weights'

   Fungsi ini menyalin nilai *weights* dari model Keras.

3. Fungsi 'tanh' dan 'tanh_derivative'

   Fungsi ini adalah fungsi aktivasi dan turunan fungsi aktivasi

4. Fungsi 'forward_step'

   Fungsi ini menerima parameter berupa 'x_t' yaitu input pada timestep t dan 'h_prev' yaitu nilai hidden state sebelumnya. Fungsi ini mengeluarkan

output berupa nilai hidden state untuk timestep ini. Penjelasan pada bagian berikutnya.

5. Fungsi ‘backward_step’

   Fungsi ini menerima parameter berupa ‘grad_h_t’ yaitu nilai gradien hidden state pada timestep t. Fungsi ini mengeluarkan output berupa ‘grad_x_t’ yaitu gradien terhadap input timestep t, ‘grad_h_prev’ yaitu gradien terhadap nilai hidden step sebelumnya, dan gradien bobot ‘grad_W_ih’, ‘grad_W_hh’, dan ‘grad_b_hh’.

2.3.1.5. Kelas SimpleRNNLayer

Kelas ini adalah untuk penerapan layer RNN *from scratch*.

1. __init__(...)

   Mendeklarasi awal atribut ‘input_size’ yakni berapa unit input, ‘hidden_size’ yakni berapa unit hidden cell (bisa dalam bentuk list), return sequence dan bidirectional, dan nilai *dropout*. Juga mendefinisikan atribut ‘forward_cell’ dan ‘backward_cell’ jika bidirectional.

2. Fungsi ‘load_weights’

   Fungsi ini menyalin nilai *weights* dari model Keras.

3. Fungsi ‘forward’

   Fungsi ini menerima parameter berupa ‘x’ yaitu beberapa *sequences* input. Fungsi ini mengeluarkan output berupa nilai output dari layer RNN tersebut. Penjelasan pada bagian berikutnya.

4. Fungsi ‘backward’

   Fungsi ini menerima parameter berupa ‘grad_output’ yaitu nilai gradien dari layer berikutnya. Fungsi ini mengembalikan nilai berupa ‘grad_input’ yaitu gradien terhadap input.

2.3.1.6. Kelas DropoutLayer

Kelas ini adalah untuk penerapan layer Dropout *from scratch*.

1. __init__(...)

   Mendeklarasi awal atribut ‘dropout_rate’ sesuai argumen, serta boolean ‘mask’ dan ‘training’.

2. Fungsi ‘set_training’

Fungsi ini untuk merubah nilai atribut training objek. Menandakan apakah ini sedang training atau tidak.

3. Fungsi ‘forward’

   Fungsi ini menerima parameter berupa ‘x’ yaitu beberapa *sequences* input. Fungsi ini mengeluarkan output berupa x hasil *dropout*. Penjelasan pada bagian berikutnya.

4. Fungsi ‘backward’

   Fungsi ini menerima parameter berupa ‘grad_output’ yaitu nilai gradien dari layer berikutnya. Fungsi ini mengembalikan nilai berupa ‘grad_output’ yang nilainya sudah disesuaikan dengan nilai dropout.

2.3.1.7. Kelas DenseLayer

Kelas ini adalah untuk penerapan layer Dense *from scratch*.

1. __init__(...)

   Mendeklarasi awal atribut ‘layer_sizes’ sesuai argumen dan juga fungsi aktivasi. Terdapat atribut ‘nn’ yakni objek kelas NeuralNetwork dari implementasi FFNN sebelumnya. Pada inisialisasi ini juga dideklarasikan atribut ‘input_shape’ dan *cache* untuk kebutuhan *backward pass* misalnya.

2. Fungsi ‘load_keras_weights’

   Fungsi ini untuk menyalin nilai bobot dan bias dari model Keras.

3. Fungsi ‘forward’

   Fungsi ini menerima parameter berupa ‘x’ yaitu input dari layer RNN. Fungsi ini mengembalikan nilai yaitu hasil output dari layer ini. Juga menyimpan *cache* yang mungkin dibutuhkan.

4. Fungsi ‘backward’

   Fungsi ini menerima parameter berupa ‘grad_output’ yaitu nilai gradien dari layer berikutnya. Fungsi ini mengembalikan nilai berupa ‘grad_input’ dan ‘grad_weights’, gradien terhadap input dan terhadap bobot.

2.3.1.8. Kelas RNNFromScratch

Kelas ini adalah untuk penerapan layer Dense *from scratch*.

1. __init__(...)

Mendeklarasi awal atribut ‘layers’ dan ‘layers_type’ yakni layer apa saja yang ada di keseluruhan model RNN ini, beserta tipenya. Juga atribut ‘training’ (boolean) apakah dilakukan training atau tidak.

2. Fungsi ‘set_training’

   Fungsi ini untuk merubah nilai atribut training objek. Menandakan apakah ini sedang training atau tidak, untuk *dropout layer*.

3. Fungsi ‘add_embedding’, ‘add_rnn’, ‘add_dropout’ dan ‘add_dense’

   Fungsi-fungsi ini nantinya akan menambahkan layer-layer objek kelas EmbeddingLayer, SimpleRNNLayer, DropoutLayer, dan DenseLayer sesuai dengan parameter dan argumen untuk membentuk masing-masing objek kelas tersebut.

4. Fungsi ‘forward’ dan ‘forward_single_batch’

   Fungsi ini menerima parameter berupa ‘grad_output’ yaitu nilai gradien dari layer berikutnya. Fungsi ini mengembalikan nilai berupa ‘grad_output’ yang nilainya sudah disesuaikan dengan nilai dropout. Penjelasan pada bagian berikutnya.

5. Fungsi ‘backward’

   Fungsi ini menerima parameter berupa ‘grad_output’ yaitu nilai gradien dari fungsi loss. Fungsi ini mengembalikan nilai berupa ‘gradients’ yaitu *dictionary* gradien dari seluruh masing-masing layer.

6. Fungsi ‘train_step’

   Fungsi ini melakukan satu tahap training (*forward*, *backward*, dan pembaruan bobot). Fungsi ini menerima input berupa input (per batch), label input (per batch), *learning rate*. Fungsi ini mengembalikan nilai *training loss* untuk batch ini.

7. Fungsi ‘fit’

   Fungsi ini melakukan keseluruhan *training* dengan memanggil fungsi ‘train_step’ . Fungsi ini menerima input berupa data train (‘x_train’, ‘y_train’), data validasi (‘x_val’, ‘y_val’), *epoch*, ‘batch_size’ . Fungsi ini mengembalikan nilai *training loss* dan *validation loss* untuk keseluruhan *training*.

8. Fungsi ‘predict’

   Fungsi ini memanggil fungsi ‘forward’ namun bukan untuk *training*.

9. Fungsi ‘evaluate’

   Fungsi ini menerima parameter berupa dataset test dan nilai label yang sebenarnya. Model akan memprediksi ‘predict’ masing-masing probability nilai masing-masing kelas label dan menentukan label teks berdasarkan probability terbesar. Kemudian, dilakukan perhitungan metrik evaluasi *test loss, test accuracy,* dan *macro F1-score*. Fungsi ini mengembalikan dictionary berisi metrik dan hasil prediksi, yang berguna untuk analisis performa model lebih lanjut.

2.3.1.9. Fungsi untuk Loading Model Keras

Terdapat tiga fungsi berikut:

1. Fungsi ‘extract_keras_architecture’

   Fungsi ini menerima parameter berupa model Keras dan mengembalikan bentuk arsitekturnya untuk kemudian bisa dibuat model *from scratch* dengan arsitektur yang sama. Fungsi ini mencatat jumlah total layer, bentuk input dan output model, serta mengumpulkan detail tiap layer seperti nama, tipe, konfigurasi, bentuk input/output, dan jumlah parameter yang dapat dilatih. Untuk setiap layer tertentu, fungsi ini juga menambahkan informasi spesifik seperti ukuran vocab, jumlah unit, jenis RNN, fungsi aktivasi, atau tingkat dropout (tergantung layer).

2. Fungsi ‘build_scratch_model_from_keras_auto’

   Fungsi ini menerima parameter berupa model Keras dan mengembalikan model *scratch* yang sesuai. Pertama, fungsi ini memanggil fungsi ‘extract_keras_architecture’ untuk mengetahui arsitektur model Keras.

   Fungsi ini memproses setiap layer Keras satu per satu. Jika layer berupa Embedding, maka ditambahkan embedding layer ke model scratch. Jika layer berupa RNN (seperti SimpleRNN atau Bidirectional), maka ditambahkan RNN layer dengan detail seperti jumlah unit, arah, dan apakah mengembalikan sekuens. Jika Dropout, maka ditambahkan dropout layer. Untuk layer Dense, semua layer dikumpulkan terlebih dahulu dalam list dense_layers_info dan kemudian dibangun sebagai satu blok pada akhir proses, termasuk mengatur ukuran input/output dan fungsi aktivasi. Setelah

seluruh layer ditambahkan, fungsi melanjutkan dengan memuat bobot model Keras ke dalam model scratch menggunakan ‘load_keras_weights_auto’.

3. Fungsi ‘load_keras_weights_auto’

   Fungsi ini menerima parameter berupa model Keras, model Scratch dengan arsitektur sama dengan model Keras, dan juga bentuk arsitektur yang tadi sudah didefinisikan. Fungsi ini melakukan iterasi antara layer Keras dan layer model scratch. Untuk tiap jenis layer, Embedding, RNN (baik unidirectional maupun bidirectional), dan Dense, fungsi ini menyesuaikan cara pengambilan bobot. Layer Embedding mengambil langsung matriks embedding, layer RNN menangani bobot berbeda untuk forward dan backward (jika bidirectional), dan layer Dense mengumpulkan semua layer Dense berturut-turut, mengabaikan Dropout di antaranya, lalu memuatnya sekaligus ke dalam Dense block model manual.

2.3.2. Penjelasan Implementasi Forward Propagation

Pada implementasi model RNN yang kami rancang, berikut adalah implementasi dari *forward propagation*

2.3.2.1. Embedding Layer

Pertama, dilakukan pengecekan terhadap bentuk input dan memastikannya 2 dimensi (batch, sequence_length) seperti standar embedding layer. Kemudian dilakukan persiapan dengan membuat matriks list kosong berbentuk 3D (batch_size, sequence_length, embedding_dim) dengan tipe data float32. Barulah matriks embedding tersebut diisi dengan *weights* dari model Keras yang sudah dilatih. (Kecuali jika dilakukan implementasi keseluruhan dengan *backward propagation* dan *training* juga dilakukan oleh model ini, baru bisa dibuat inisialisasi *weight* dengan random atau formula tertentu)

2.3.2.2. RNN Layer

Pada setiap sel di RNN layer dilakukan perhitungan dengan rumus $h_t = tanh\ (Ux_t + (Wh_{t-1} + bx_h))$ dan menyimpan nilainya pada *cache* jika dibutuhkan untuk keperluan *backward*. Pada setiap layer, pertama dilakukan pengecekan input agar berbentuk 3 dimensi (jika masih 2 dimensi akan diproses panjang 1). Kemudian, diinisialisasi nilai hidden forward dan hidden

backward (jika *bidirectional*) dengan matriks berukuran *input shape* dengan nilai 0.

Dilakukan *loop forward*, untuk setiap timestep, ambil input pada timestep ke-t dan dilakukan perhitungan sel dan nilainya disimpan di array 'forward_outputs'. Hal yang sama untuk jika *bidirectional* hanya saja memproses nilai input dari belakang (timestep 1 input terakhir, dst.) dan nilainya disimpan di 'backward_outputs'. Jika 'return_sequence=True', maka output berbentuk tensor 3D (batch, seq_length, hidden_size) sementara jika tidak hanya mengembalikan nilai terakhir. Nilai *cache* disimpan untuk kebutuhan *backward*.

2.3.2.3. Dropout Layer

Saat training, buat mask dari distribusi binomial (0 atau 1), lalu normalisasi dengan 1 / (1 - dropout_rate) agar ekspektasi output tetap sama. Saat inferensi (jika atribut training False), nilai input diteruskan.

2.3.2.4. Dense Layer

Implementasi menggunakan FFNN yang pernah dibuat. Sebelumnya, dilakukan pengecekan terhadap bentuk input. Kemudian, menggunakan method 'forward_propagation' dari FFNN. Hasilnya adalah activations[-1] atau hasil output dari layer terakhir FFNN atau Dense Layer tersebut.

2.3.2.5. Keseluruhan

Implementasi *forward propagation* keseluruhan adalah dengan menyusuri satu per satu setiap layer dengan *forward propagation*. Fungsi 'forward' digunakan untuk melakukan proses forward pass pada input, baik secara langsung maupun dibagi dalam batch jika 'batch_size' ditentukan. Jika batch_size diberikan dan jumlah data lebih besar dari ukuran batch, input akan dibagi ke dalam beberapa batch, lalu setiap batch diproses menggunakan fungsi '_forward_single_batch', dan hasilnya digabung kembali menggunakan np.vstack. Fungsi '_forward_single_batch' melakukan iterasi pada setiap layer, menjalankan fungsi forward dari masing-masing layer. Proses ini menghasilkan output akhir dari jaringan untuk satu batch input. Keseluruhan akan menghasilkan probabilitas prediksi terhadap input.

2.4. Implementasi Long-Short Term Memory Neural Network

Secara umum, kami menyelesaikan proyek kami dengan membuat satu file implementasi model Long-Short Term Memory.

2.4.1. Deskripsi Kelas, Atribut, dan Method

2.4.1.1. Kelas LSTMCell

Kelas LSTMCell akan menangani komputasi satu langkah waktu dari LSTM cell tunggal.

1. Atribut ‘input_size’

   Atribut untuk dimensi input per timestep.

2. Atribut ‘hidden_size’

   Atribut untuk ukuran dari hidden state.

3. Atribut ‘W_f, W_i, W_c, W_o’

   Matriks bobot untuk masing-masing gate: forget gate, input gate, cell state, dan output gate.

4. Atribut ‘b_f, b_i, b_c, b_o’

   Vektor bias untuk tiap gate.

5. Method ‘forward_step(x_t, h_prev, c_prev)’
   - Input: x_t, h_prev, c_prev
   - Output: h_t, c_t
   - Proses: Hitung gate values → update cell state → update hidden state.

2.4.1.2. Kelas LSTMLayer

Kelas LSTMLayer akan menyusun *sequence input* menggunakan LSTMCell secara berurutan, sehingga dapat melakukan *unidirectional* dan *bidirectional*.

1. Atribut ‘forward_cell’

   Objek LSTMCell untuk arah maju.

2. Atribut ‘backward_cell’

   Objek LSTMCell untuk arah mundur (saat bidirectional)

3. Method ‘forward(x, mask=None)’
   - Input: Sequence input (batch_size, seq_len, input_size)
   - Output: Sequence output (batch_size, seq_len, hidden_size * directions)
   - Proses: Iterasi LSTMCell untuk setiap timestep (dan arah).

2.4.1.3. LSTMScratchModel

Kelas LSTMLayer akan mewakili keseluruhan model LSTM dari embedding hingga klasifikasi.

1. Atribut ‘embedding’

   Layer untuk mengubah token menjadi vektor.

2. Atribut ‘lstm_layers’

   Daftar dari layer-layer LSTM.

3. Atribut ‘dense’

   Dense layer sebelum output akhir.

4. Atribut ‘output_layers’

   Layer akhir dengan aktivasi softmax.

2.4.2. Penjelasan Forward Propagation

2.4.2.1. Arsitektur LSTM Cell (per *timestep*)

Setiap LSTM cell dirancang untuk memproses input pada satu langkah waktu, sambil mempertahankan informasi penting dari langkah sebelumnya melalui dua komponen utama: *hidden state* dan *cell state*. Hidden state (h_t) adalah representasi output yang akan diteruskan ke layer lain atau ke timestep berikutnya, sedangkan cell state (c_t) berfungsi sebagai memori jangka panjang yang membawa informasi penting dari masa lalu.

Di dalam setiap cell, terdapat tiga gerbang utama: forget gate, input gate, dan output gate. Forget gate bertugas mengatur informasi apa dari cell state sebelumnya (c_{t-1}) yang perlu dilupakan. Input gate menentukan informasi baru mana yang akan ditambahkan ke memori, sementara candidate values (c̃_t) dihasilkan untuk mewakili potensi informasi baru yang akan disimpan. Cell state kemudian diperbarui dengan menggabungkan dua bagian tersebut—yang dilupakan dari masa lalu dan yang ditambahkan dari input saat ini. Setelah cell state diperbarui, output gate mengontrol bagian mana dari cell state yang akan digunakan untuk menghitung hidden state (h_t). Hidden state ini didapatkan dengan mengalikan output gate dengan aktivasi tanh dari cell state, menghasilkan informasi yang kaya konteks untuk diteruskan.

2.4.2.2. Alur Forward Propagation

Forward propagation dalam LSTM dimulai ketika seluruh *input sequence*, yang terdiri dari beberapa *timestep*, diberikan ke model. Misalnya dalam klasifikasi teks, input diubah menjadi vektor melalui proses *embedding*. Model kemudian memproses input tersebut secara berurutan satu per satu, setiap timestep diumpankan ke LSTM cell, bersama dengan *hidden state* dan *cell state* dari langkah sebelumnya.

Pada langkah pertama, karena tidak ada informasi sebelumnya, *hidden state* dan *cell state* diinisialisasi sebagai vektor nol. Input pertama (x_1) bersama dengan state awal diproses oleh LSTM cell, menghasilkan hidden state (h_1) dan cell state (c_1) baru. Kemudian pada langkah kedua, input kedua (x_2) diproses bersama dengan h_1 dan c_1, dan begitu seterusnya hingga seluruh sequence selesai.

Di setiap langkah ini, nilai hidden state h_t yang dihasilkan bisa disimpan untuk keperluan lanjutan seperti sequence labeling, atau hanya hidden state terakhir (h_T) yang digunakan sebagai representasi sequence secara keseluruhan—biasanya dalam tugas klasifikasi. Setelah forward propagation selesai, hidden state terakhir dilewatkan ke fully connected layer, diaktifkan dengan ReLU atau fungsi aktivasi lainnya, lalu diakhiri dengan layer softmax untuk menghasilkan distribusi probabilitas dari kelas-kelas target.

2.4.3. Penjelasan Implementasi Backward Propagation

Pada implementasi model FFNN yang kami rancang, jika metode lalalala lalalala

2.4.3.1. XXX Kalo misal ada poin2

ZLorem ipsum

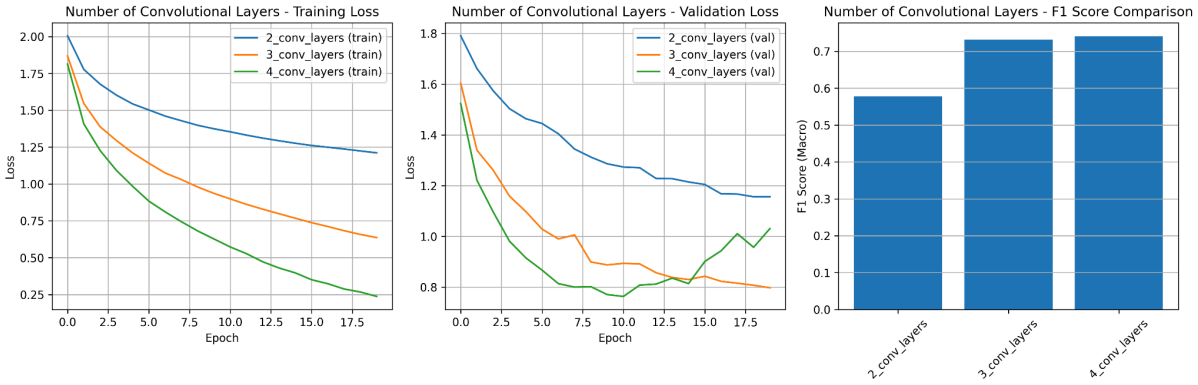
2.4.3.2. XXX Kalo misal ada poin2

BAB III
PENGUJIAN DAN ANALISIS

3.1. Hasil Pengujian Concurrent Neural Network

3.1.1. Pengaruh Jumlah Layer Konvolusi

Model	Accuracy	F1 (Macro)	Loss
2 conv layers	0.5848	0.5781	1.1373
3 conv layers	0.7304	0.7318	0.8057
4 conv layers	0.7374	0.7404	1.0579



Hasil dapat dijelaskan dengan teori *Hierarchical Feature Learning.*

- 2 Layer vs 3 Layer (+15% F1-score):
    1. Layer 1: Mendeteksi fitur dasar (edges, corners, textures)
    2. Layer 2: Kombinasi fitur dasar menjadi shapes sederhana
    3. Layer 3: Kombinasi shapes menjadi object parts yang kompleks

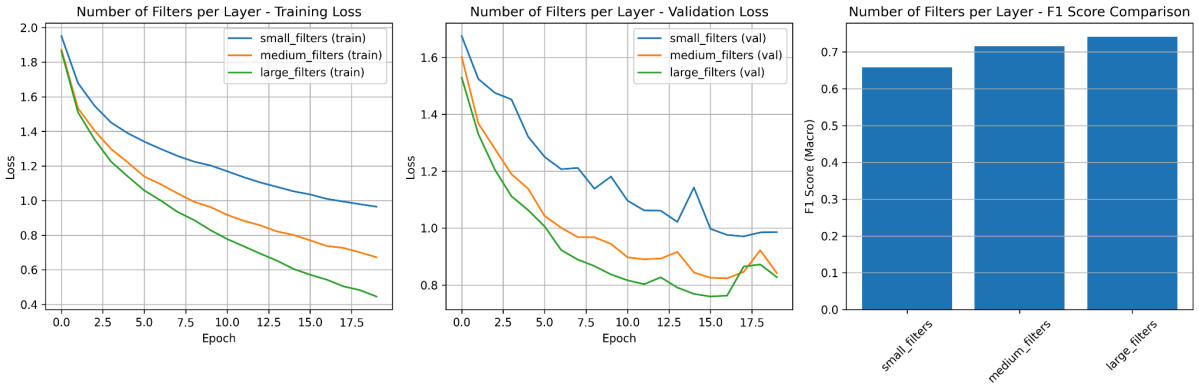
2 layer terlalu shallow untuk CIFAR-10 yang kompleks. 3 layer mencapai sweet spot untuk representasi hierarkis yang memadai. Setiap kelas CIFAR-10 memiliki struktur visual yang berbeda misalnya pesawat dengan kucing sehingga terkadang butuh representasi yang lebih dalam.

- 3 Layer vs 4 Layer (+0.8% F1-score):

Diminishing returns terjadi; kedalaman tambahan tidak memberikan representasi baru yang signifikan. Terdapat juga overfitting risk yaitu model lebih kompleks mungkin mulai menghafal training data. Potensi gradient degradation semakin besar yaitu sinyal gradient mulai melemah di layer yang sangat dalam.

3.1.2. Pengaruh Banyak Filter per Layer Konvolusi

Filters Config	Accuracy	F1 (Macro)	Loss
Small Filters	0.6567	0.6575	0.9707
Medium Filters	0.7171	0.7157	0.8299
Large Filters	0.7427	0.7413	0.8539



Hasil dapat dijelaskan dengan teori *Feature Capacity & Representational Power.*
Filter yang lebih banyak menghasilkan F1-score lebih tinggi karena

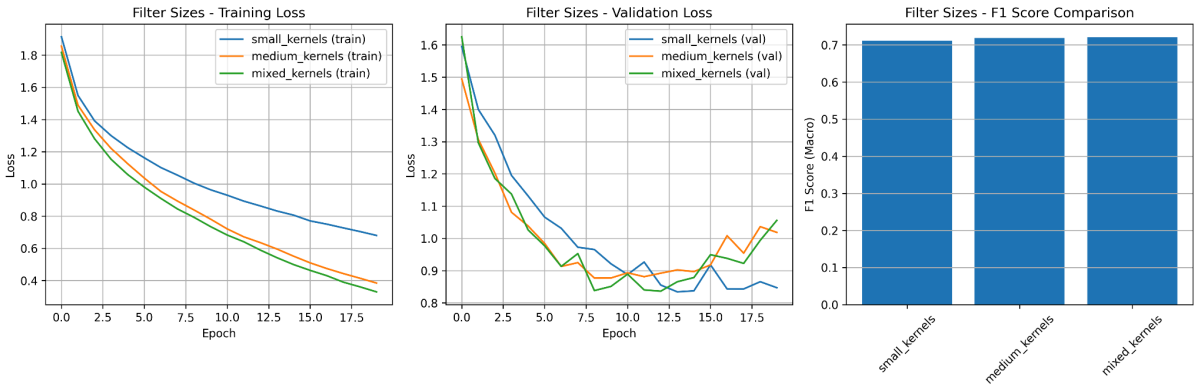
- Feature Diversity: yaitu lebih banyak filter berarti lebih banyak jenis fitur yang bisa dideteksi. Setiap filter menangkap pola yang berbeda (horizontal lines, curves, textures).
- Class Discrimination: CIFAR-10 memiliki 10 kelas dengan karakteristik visual yang beragam. Kelas seperti "bird" vs "airplane" butuh fitur yang sangat spesifik sehingga filter lebih banyak berarti kemampuan diskriminasi antar kelas lebih baik.

Kelas yang sulit (seperti cat vs dog) mendapat manfaat dari representasi yang lebih kaya. Peningkatan yang terjadi kecil dikarenakan

- Parameter explosion: yaitu model menjadi sangat besar sehingga butuh data lebih banyak.
- Computation overhead: yaitu training menjadi lebih sulit.

3.1.3. Pengaruh Ukuran Filter per Layer Konvolusi

Kernel Size	Accuracy	F1 (Macro)	Loss
Small Kernels	0.7109	0.7115	0.8601
Medium Kernels	0.7198	0.7185	1.0206
Mixed Kernels	0.7202	0.7205	1.0757



Hasil dapat dijelaskan dengan teori *Receptive Field* & *Computational Efficiency.*

Dampaknya terhadap F1-score kecil yaitu sekitar ~1% F1-score dikarenakan:

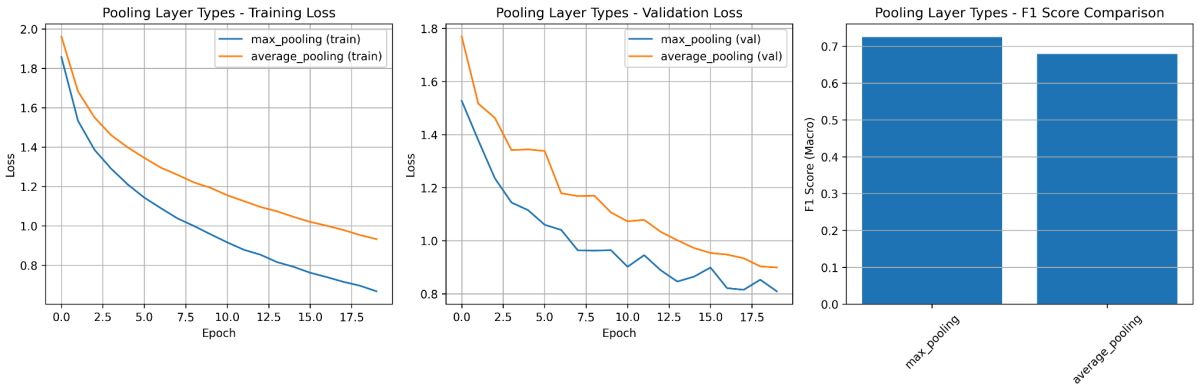
- Receptive Field Trade-off:
    1. 3x3 kernels: Receptive field kecil, tapi efisien dan bisa ditumpuk.
    2. 5x5 kernels: Receptive field lebih besar, menangkap konteks spatial yang lebih luas.
    3. Mixed kernels: Multi-scale feature detection.
- CIFAR-10 Constraint: Gambar hanya 32x32 pixel maka ukurannya dikatakan relatif kecil. Perbedaan antara 3x3 vs 5x5 tidak sesignifikan pada gambar kecil. Fitur penting sudah bisa ditangkap dengan 3x3.

- Depth vs Width Trade-off: Kernel besar berarti parameter lebih banyak tapi depth sama. Untuk budget parameter yang sama, lebih baik tambah depth daripada kernel size

Mixed kernels sedikit lebih baik dikarenakan multi-scale feature extraction yaitu layer berbeda menangkap detail di skala berbeda dan architectural diversity yaitu mengurangi inductive bias yang berlebihan.

3.1.4. Pengaruh Jenis Pooling Layer

Pooling Type	Accuracy	F1 (Macro)	Loss
Max Pooling	0.7228	0.7246	0.8099
Average Pooling	0.6794	0.6790	0.8924



Hasil dapat dijelaskan dengan teori *Information Preservation & Noise Reduction*
Hasil mengatakan bahwa Max Pooling lebih baik dari Average Pooling dengan +4.4% F1-score. Hal ini dikarenakan (dalam kasus ini dan dalam konteks pengujian ini):

- Feature Preservation:
    1. Max pooling: Mempertahankan fitur yang paling kuat/prominent.
    2. Average pooling: Menghaluskan dan mungkin menghilangkan fitur penting.
- Class discrimination:
    1. Fitur yang strong (yang dipertahankan max pooling) lebih diskriminatif.
    2. Average pooling bisa "mengaburkan" perbedaan antar kelas.

- Macro F1 *Context*:
    1. Kelas yang mirip (seperti dog vs cat) butuh fitur yang sangat spesifik.
    2. Max pooling lebih baik mempertahankan fitur diskriminatif ini.
    3. Average pooling mungkin membuat kelas yang mirip semakin sulit dibedakan.

3.2. Hasil Pengujian Simple Recurrent Neural Network

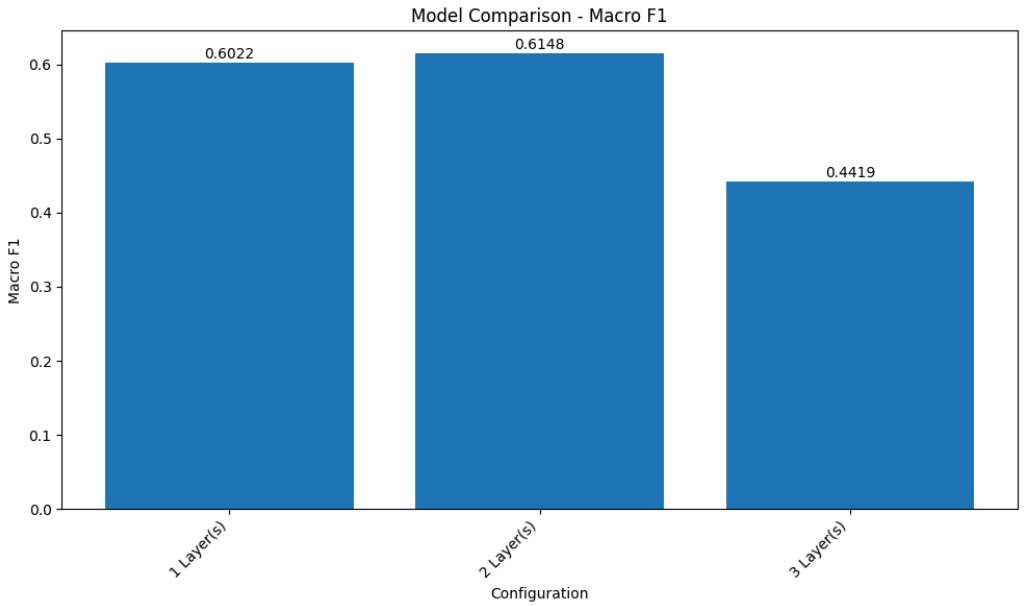
3.2.1. Pengaruh jumlah layer RNN

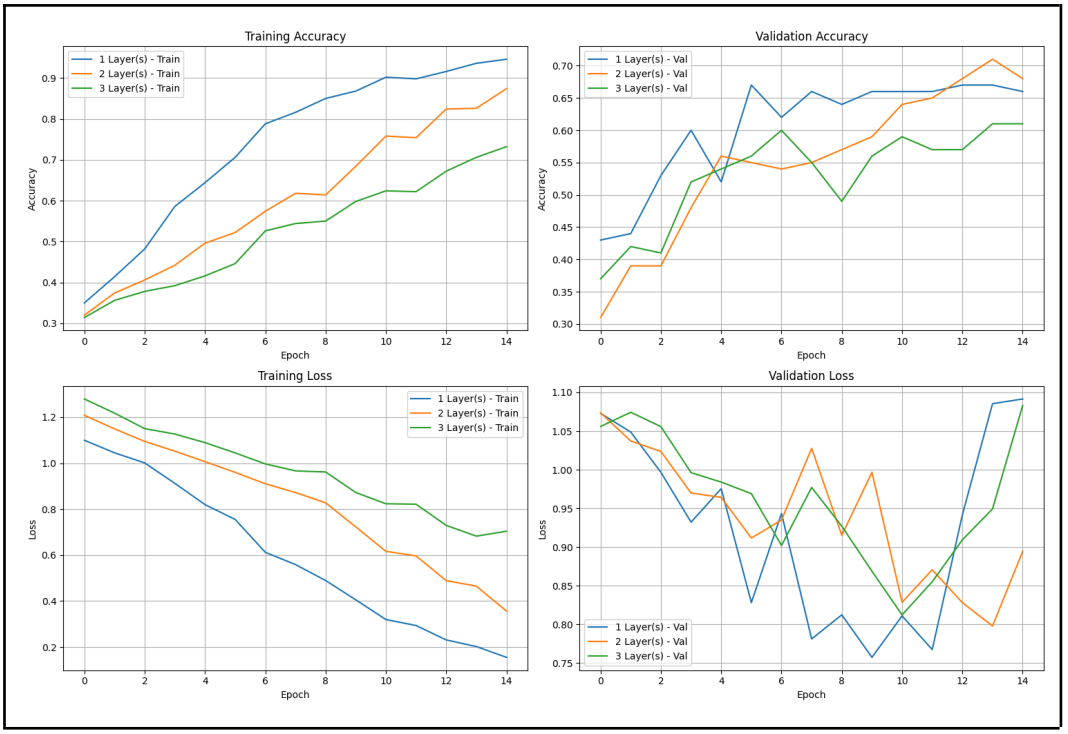
Dalam pengujian ini, berbagai jumlah layer (1, 2, 3) diuji untuk mengetahui pengaruhnya terhadap performa model RNN dalam memprediksi dataset sentimen kalimat di NusaX. Dilakukan dua kali percobaan karena pernah mencoba beberapa kali hasilnya berbeda, namun yang paling konsisten dan sering adalah 2 layer terbaik.

1. Percobaan Pertama

Comparison of Different Numbers of RNN Layers:

	Configuration	Test Accuracy	Test Loss	Macro F1-Score
1	2 Layer(s)	0.6475	1.032030	0.614787
0	1 Layer(s)	0.6600	0.799009	0.602168
2	3 Layer(s)	0.5575	0.878547	0.441926

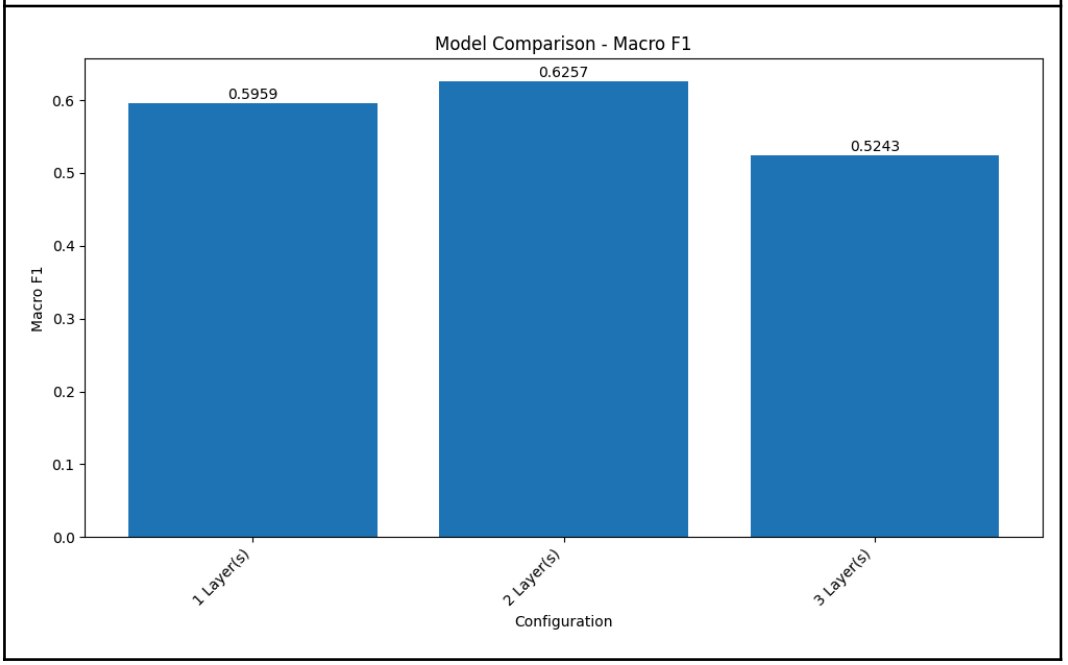


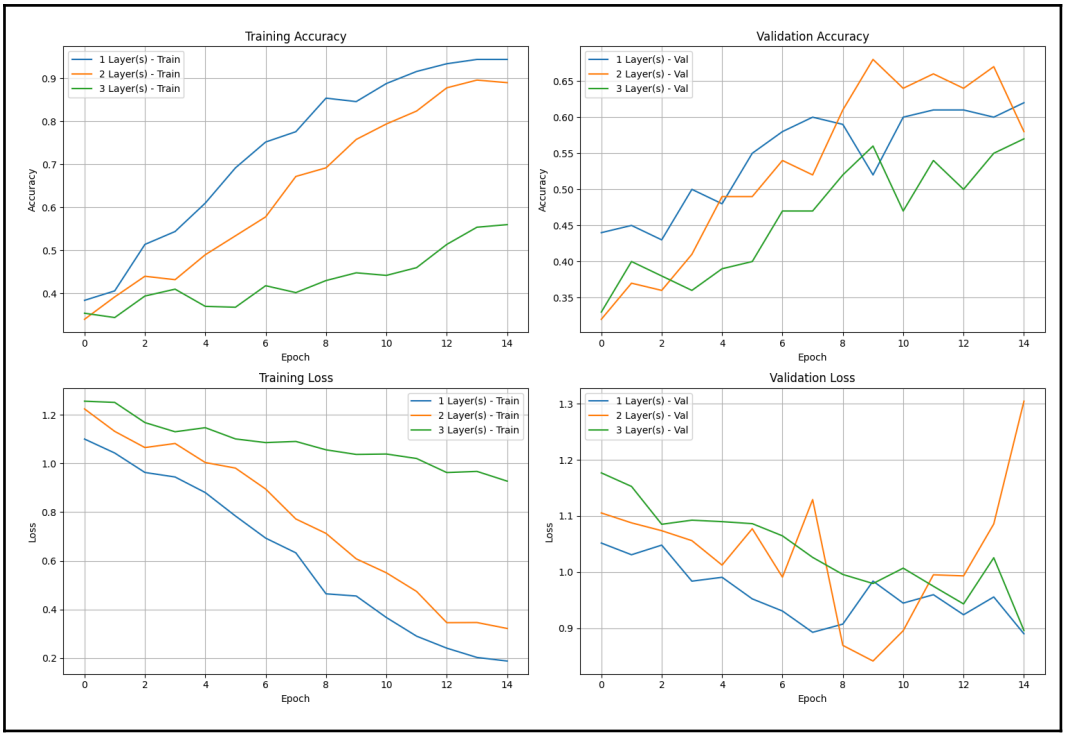


2. Percobaan Kedua

Comparison of Different Numbers of RNN Layers:

	Configuration	Test Accuracy	Test Loss	Macro F1-Score
1	2 Layer(s)	0.6525	0.865226	0.625698
0	1 Layer(s)	0.6125	1.006416	0.595852
2	3 Layer(s)	0.5425	0.905089	0.524307





Berdasarkan grafik hasil training dan validasi, dapat disimpulkan bahwa model dengan dua RNN layer memberikan performa terbaik secara keseluruhan. Pada grafik akurasi dan loss selama training, model dengan satu layer menunjukkan laju pembelajaran tercepat dan akurasi training tertinggi, namun paling sering model dua layer yang paling baik. Model dengan tiga layer mengalami akurasi yang stagnan di bawah 0.7 dan loss yang menurun sangat lambat, model ini cenderung underfitting terhadap data yang tersedia.

Jika dibandingkan dari segi hasil uji pada data test, model dengan dua layer secara konsisten mencetak metrik yang lebih seimbang. Pada percobaan pertama, meskipun model satu layer unggul dalam akurasi dan loss test, model dua layer memberikan nilai macro F1-score yang lebih tinggi, menunjukkan kemampuannya yang lebih baik dalam mengklasifikasi seluruh kelas secara merata. Sementara itu, model tiga layer menunjukkan performa yang paling rendah. Hal ini semakin diperkuat pada percobaan kedua, di mana model dua layer mendominasi hampir semua metrik utama: akurasi meningkat, loss menurun, dan F1-score tetap stabil.

Dari keseluruhan hasil yang diperoleh, dapat disimpulkan bahwa model dengan dua RNN layer memberikan trade-off terbaik antara kapasitas dan generalisasi.

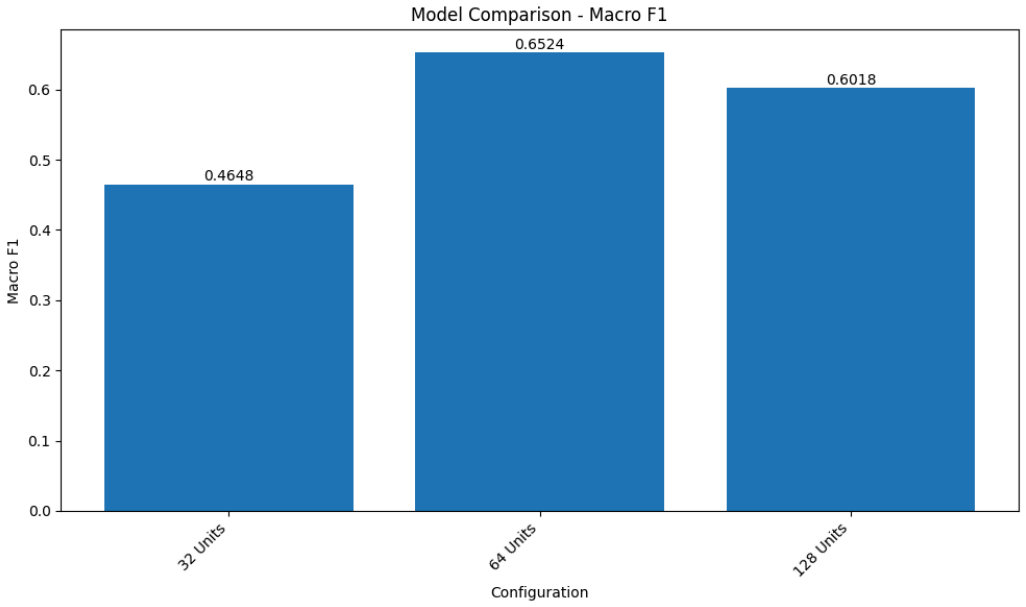
3.2.2. Pengaruh banyak cell RNN per layer

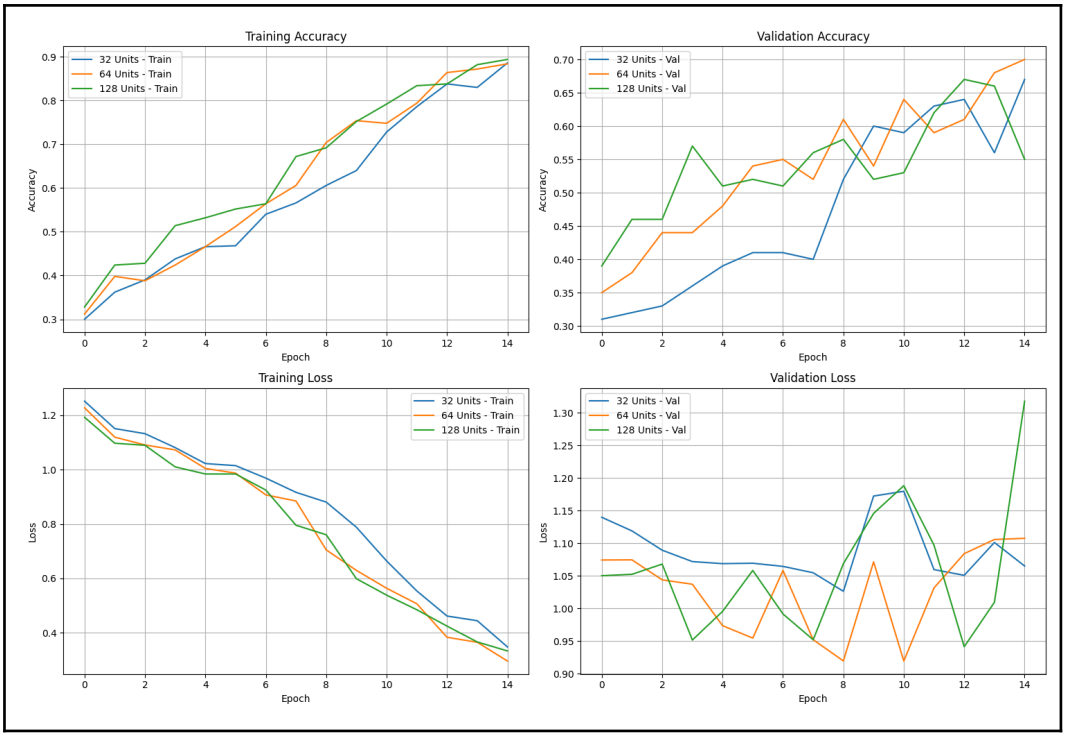
Dalam pengujian ini, berbagai jumlah unit RNN per layer (32, 64, 128) diuji untuk mengetahui pengaruhnya terhadap performa model RNN dalam memprediksi dataset sentimen kalimat di NusaX. Dilakukan dua kali percobaan karena pernah mencoba beberapa kali hasilnya berbeda, namun yang paling konsisten dan sering adalah 64 unit terbaik.

1. Percobaan Pertama

Comparison of Different Numbers of RNN Units:

	Configuration	Test Accuracy	Test Loss	Macro F1-Score
1	64 Units	0.6675	0.851489	0.652385
2	128 Units	0.6275	1.036651	0.601833
0	32 Units	0.5225	0.982528	0.464780

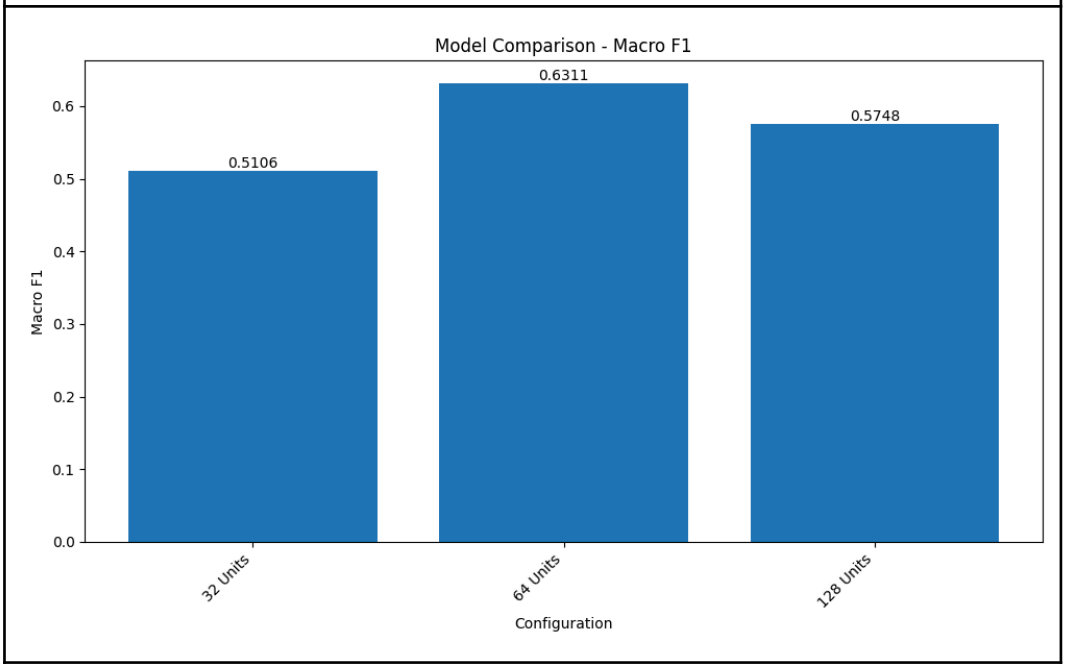


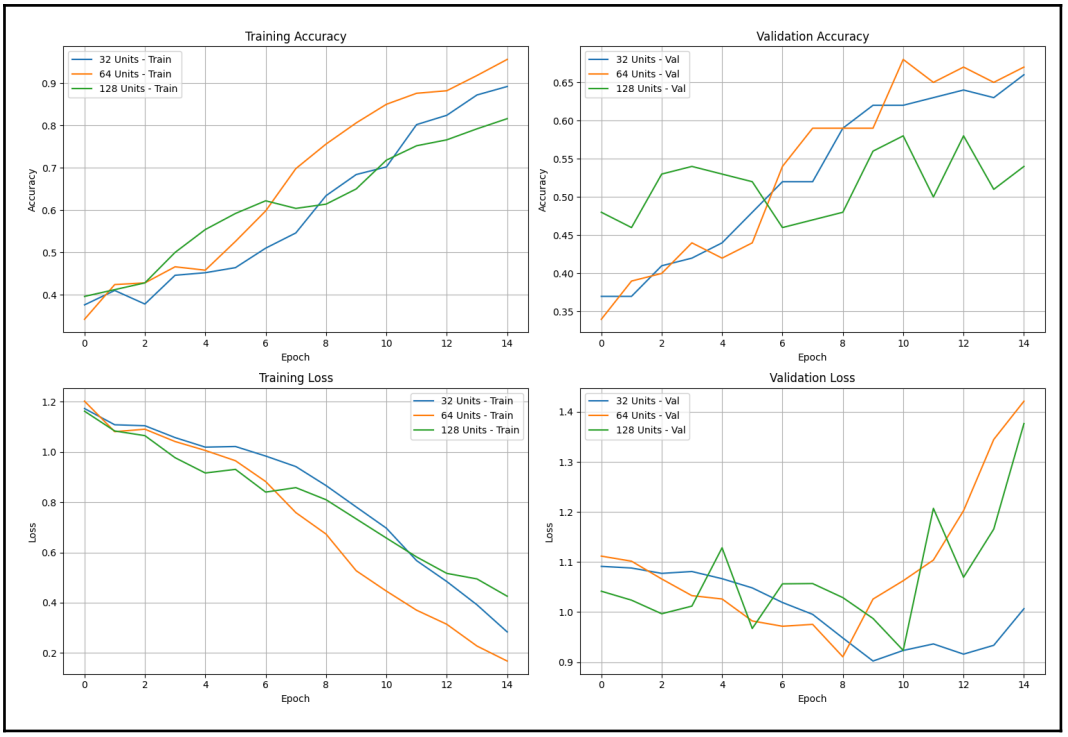


2. Percobaan Kedua

Comparison of Different Numbers of RNN Units:

	Configuration	Test Accuracy	Test Loss	Macro F1-Score
1	64 Units	0.6675	0.804901	0.631054
2	128 Units	0.6100	0.910828	0.574839
0	32 Units	0.6025	0.919227	0.510566





Dari kedua percobaan, konfigurasi dengan 64 unit menunjukkan performa paling stabil dan unggul, dengan test accuracy tertinggi (66.75%) serta macro F1-score terbaik di kedua percobaan (masing-masing 0.652 dan 0.631).

Berdasarkan grafik hasil pelatihan dan validasi, terlihat bahwa model dengan 64 unit secara umum mengalami peningkatan yang konsisten dalam hal akurasi dan penurunan loss selama epoch berlangsung. Model dengan 128 unit memang mencapai akurasi pelatihan yang lebih tinggi pada awalnya, namun pada validasi performanya justru fluktuatif dan menunjukkan indikasi overfitting. Sementara itu, model dengan 32 unit cenderung mengalami kesulitan mencapai performa yang tinggi, dengan nilai akurasi dan F1-score yang konsisten lebih rendah dibanding dua konfigurasi lainnya.

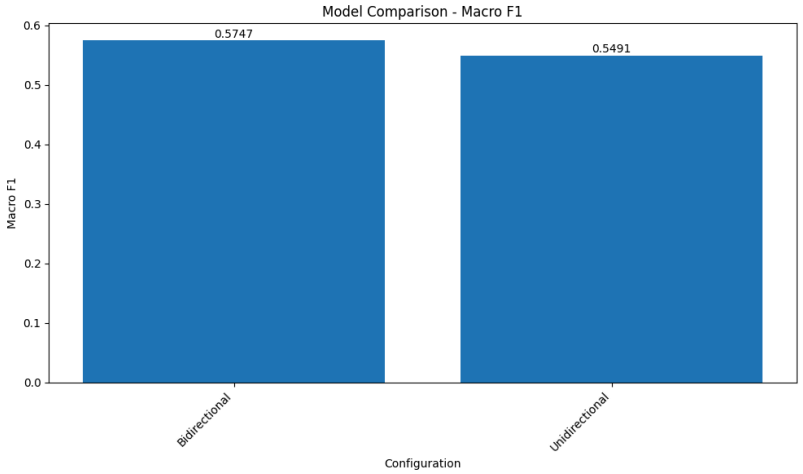
Dalam konteks dataset dan konfigurasi arsitektur ini, jumlah unit yang terlalu sedikit tidak cukup menangkap kompleksitas data, sementara jumlah unit yang terlalu banyak justru berisiko menyebabkan overfitting. Konfigurasi 64 unit terlihat menawarkan keseimbangan terbaik antara kompleksitas model dan generalisasi terhadap data validasi.

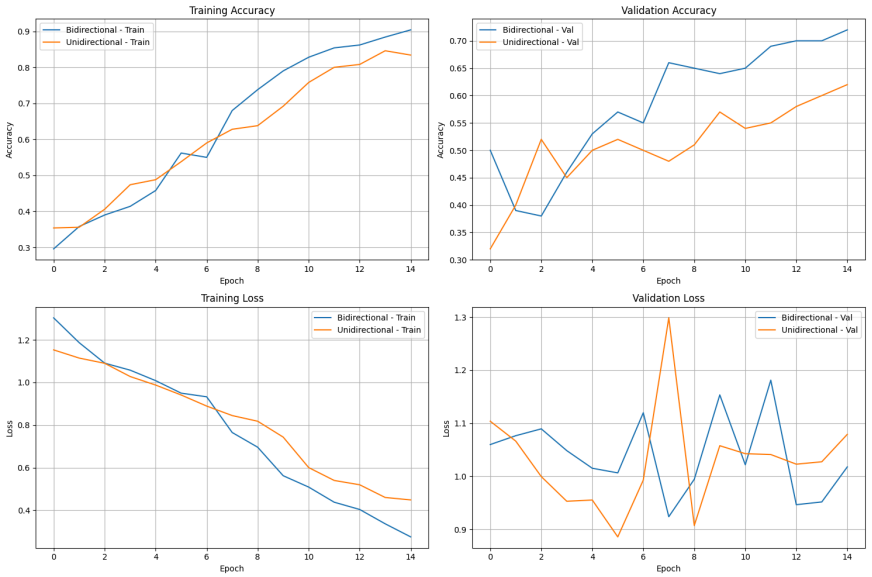
3.2.3. Pengaruh jenis layer RNN berdasarkan arah

Dalam pengujian ini, arah RNN (unidirectional vs bidirectional) diuji untuk mengetahui pengaruhnya terhadap performa model RNN dalam memprediksi dataset sentimen kalimat di NusaX.

Comparison of Bidirectional vs Unidirectional RNN:

	Configuration	Test Accuracy	Test Loss	Macro F1-Score
0	Bidirectional	0.610	0.912895	0.574747
1	Unidirectional	0.585	0.864377	0.549066





Bidirectional RNN menunjukkan performa yang konsisten lebih baik dibandingkan Unidirectional RNN, baik pada fase pelatihan maupun validasi. Grafik training dan validation accuracy menunjukkan bahwa Bidirectional RNN mencapai akurasi yang lebih tinggi dan lebih stabil seiring bertambahnya epoch.

Meskipun fluktuasi masih terlihat terutama pada validation loss, model Bidirectional cenderung lebih cepat dan konsisten mencapai performa optimal.

Secara kuantitatif, Bidirectional RNN mencatat test accuracy sebesar 61.0% dan macro F1-score 0.5747, sedikit lebih tinggi dibandingkan Unidirectional RNN yang mencatat test accuracy 58.5% dan F1-score 0.5491. Hal ini menegaskan bahwa pemanfaatan informasi konteks dari dua arah dalam Bidirectional RNN memberi keuntungan dalam memahami konteks kalimat secara lebih menyeluruh, meskipun bedanya tidak terlalu jauh.

3.2.4. Perbandingan hasil dengan model *from scratch*

```
Keras Model Performance:
  Accuracy: 0.6900
  Macro F1: 0.6593

From-Scratch Model Performance:
  Accuracy: 0.6775
  Macro F1: 0.6319

Implementation Similarity:
  Prediction Match Rate: 0.7850
  Probability MSE: 0.083943
```

Saya telah berusaha semaksimal mungkin untuk mereplikasi model Keras *from-scratch*, dengan mengambil bobot langsung dari model Keras sebagai dasar awal. Meskipun secara arsitektur dan parameter model seharusnya identik, hasil evaluasi menunjukkan bahwa performa model *from-scratch* belum sepenuhnya menyamai model Keras. Terdapat selisih pada akurasi, macro F1-score, dan perbedaan prediksi sebesar ±21%, yang menunjukkan adanya ketidaksesuaian dalam implementasi. Kemungkinan besar, kesalahan terjadi pada detail-detail teknis seperti perhitungan forward pass yang tidak identik (misal torch.matmul dengan np.dot), penanganan dimensi atau urutan matriks yang berbeda, atau perbedaan numerik fungsi aktivasi.

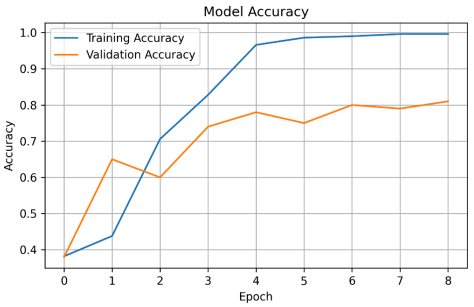
Selain itu, proses pemetaan dan penggunaan bobot dari model Keras mungkin belum sepenuhnya akurat. Saya sudah melakukan *debugging* (per layer, dengan data dummy, dll.) namun ini sudah yang terbaik dari sebelumnya perbedaan lebih dari 50% dari percobaan saya.

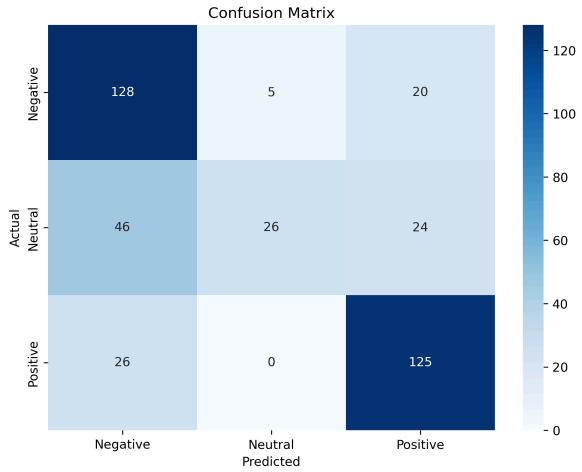
3.3. Hasil Pengujian LSTM

3.3.1. Pengaruh Jumlah Layer LSTM

3.3.1.1. 1 Layer

LSTM 1 Layer



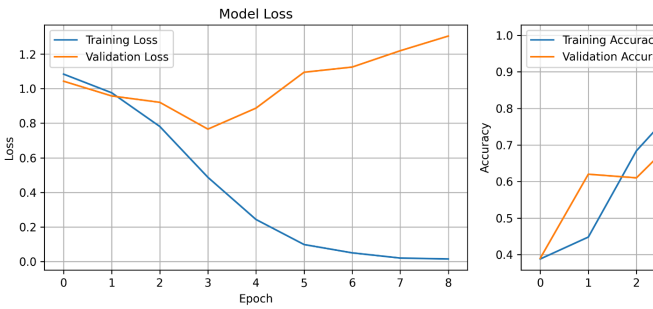


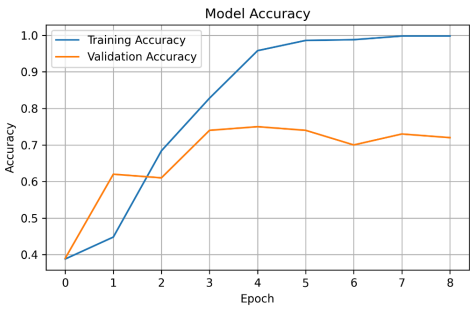
Validation Results:
Macro F1: 0.7078
Accuracy: 0.7400

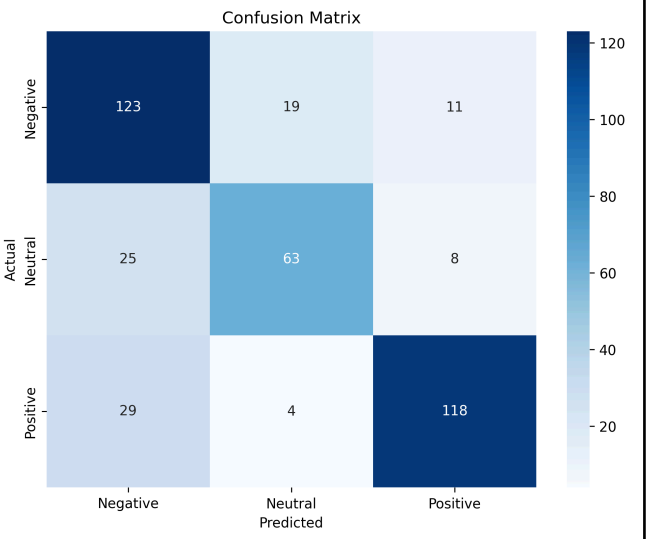
Test Results:
Macro F1: 0.6386
Accuracy: 0.6975

3.3.1.2. 2 Layer

LSTM 2 Layer





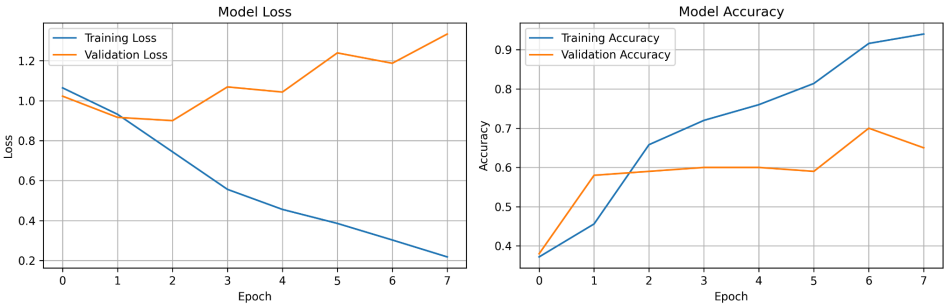


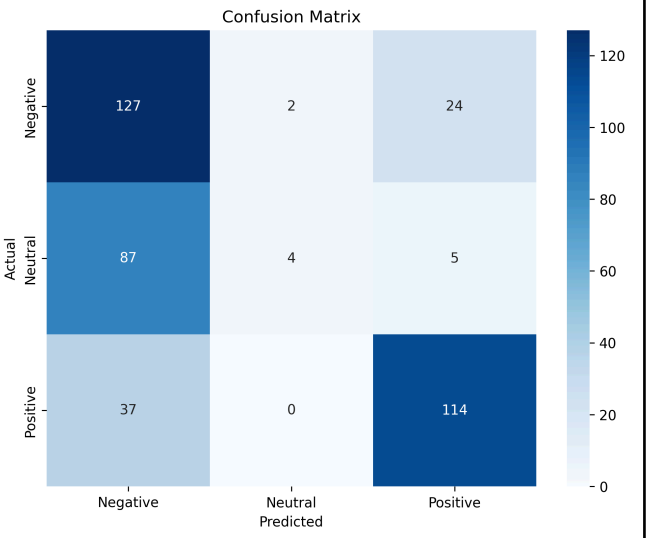
Validation Results:
Macro F1: 0.7381
Accuracy: 0.7400

Test Results:
Macro F1: 0.7524
Accuracy: 0.7600

3.3.1.3. 3 Layer

LSTM 3 Layer





Validation Results:
Macro F1: 0.4537
Accuracy: 0.5900

Test Results:
Macro F1: 0.4942
Accuracy: 0.6125

Dalam pengujian ini, dilakukan eksperimen terhadap model LSTM dengan variasi jumlah layer, yaitu 1, 2, dan 3 layer, untuk mengevaluasi dampaknya terhadap performa model. Hasil pengujian diukur menggunakan metrik **Macro F1-score**dan **Accuracy** pada data validasi dan data uji.

Pada konfigurasi **LSTM 1 layer**, model menunjukkan performa yang cukup stabil. Nilai F1-score di data validasi sebesar 0.7078 dengan akurasi 74%, dan di data uji F1-score sebesar 0.6386 dengan akurasi 69.75%. Hal ini mencerminkan bahwa model mampu menangkap pola dasar dari data namun masih memiliki ruang untuk perbaikan, terutama dalam generalisasi ke data yang belum terlihat (test set).

Ketika jumlah layer ditingkatkan menjadi **2 layer**, performa model meningkat secara signifikan, terutama di data uji. F1-score di test set naik ke 0.7524 dengan akurasi mencapai 76%, menandakan bahwa penambahan lapisan memperkuat kemampuan model dalam menangkap pola kompleks dalam data sekuensial. LSTM dua layer berhasil mengatasi keterbatasan kapasitas dari satu lapisan tanpa menyebabkan overfitting.

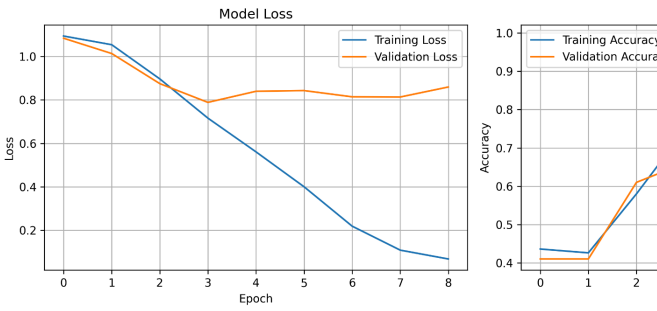
Namun, saat jumlah layer ditambah lagi menjadi **3 layer**, performa model justru menurun drastis. F1-score pada data validasi dan uji masing-masing turun ke 0.4537 dan 0.4942, dengan akurasi juga menurun menjadi sekitar 59% dan 61.25%. Penurunan ini kemungkinan besar disebabkan oleh **overfitting** atau **kesulitan dalam pelatihan** jaringan yang lebih dalam, terutama jika data latih tidak cukup besar atau regularisasi dan tuning hyperparameter belum optimal.

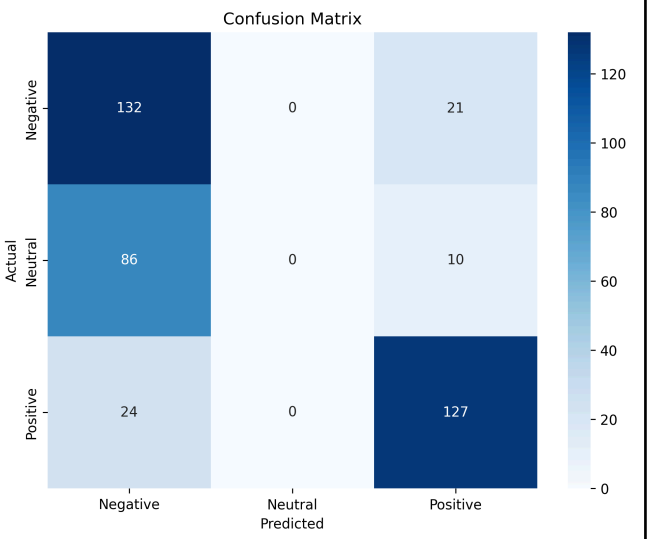
Secara keseluruhan, hasil ini menunjukkan bahwa menambah jumlah layer pada LSTM memang dapat meningkatkan performa, namun hanya sampai titik tertentu. Dalam konteks pengujian ini, **LSTM 2 layer memberikan performa terbaik dan paling seimbang**, sementara konfigurasi dengan 3 layer malah menurunkan kemampuan generalisasi model.

3.3.2. Pengaruh banyak cell LSTM per layer

3.3.2.1. 32 units

LSTM 32 units



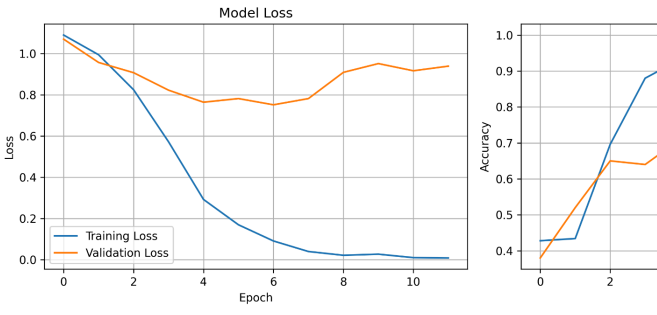


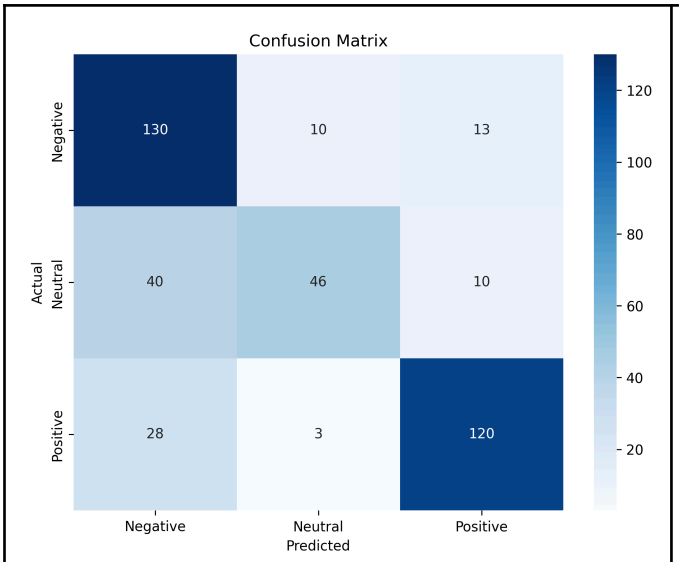
Validation Results:
Macro F1: 0.5037
Accuracy: 0.6600

Test Results:
Macro F1: 0.4968
Accuracy: 0.6475

3.3.2.2. 64 units

LSTM 64 units



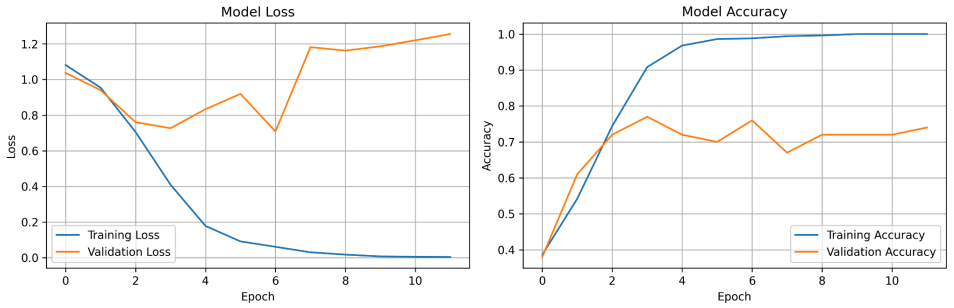


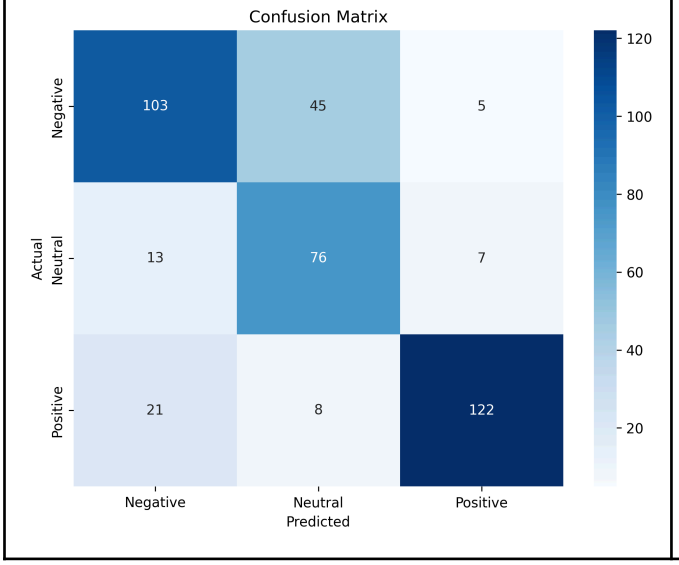
Validation Results:
Macro F1: 0.7566
Accuracy: 0.7700

Test Results:
Macro F1: 0.7169
Accuracy: 0.7400

3.3.2.3. 128 units

LSTM 128 units





Validation Results:
Macro F1: 0.7557
Accuracy: 0.7600

Test Results:
Macro F1: 0.7473
Accuracy: 0.7525

Ketika model menggunakan 32 unit per layer, performa yang didapat relatif rendah. Macro F1-score di validasi hanya sekitar 0.5037 dengan akurasi

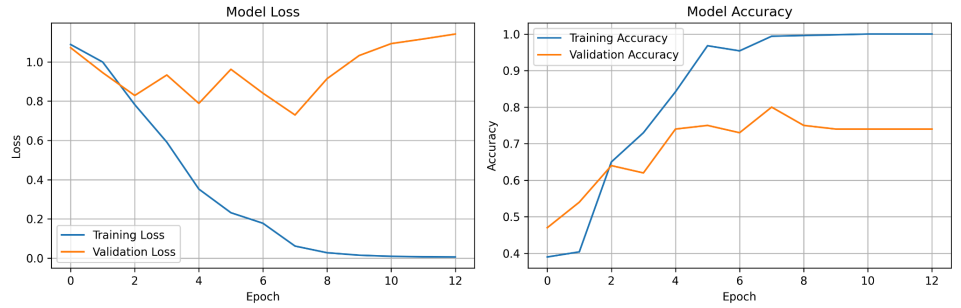
66%, sementara pada data uji F1-score berada di 0.4968 dengan akurasi 64.75%. Peningkatan signifikan terlihat saat jumlah unit diperbesar menjadi 64 units. Di sini, performa model naik secara jelas, dengan Macro F1-score validasi mencapai 0.7566 dan akurasi 77%, sedangkan pada data uji F1-score mencapai 0.7169 dengan akurasi 74%.

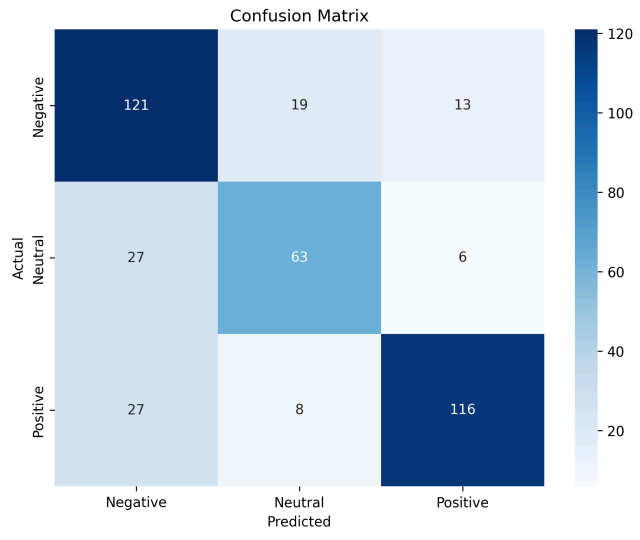
Menambah unit lagi menjadi 128 units menunjukkan hasil yang sedikit lebih baik atau setara dibandingkan dengan 64 units. Secara keseluruhan, eksperimen ini menunjukkan bahwa peningkatan jumlah unit LSTM dari 32 ke 64 memberikan dampak signifikan terhadap performa model, sedangkan penambahan dari 64 ke 128 unit memberikan peningkatan yang lebih kecil dan cenderung mendatar.

3.3.3. Pengaruh jenis layer LSTM berdasarkan arah

3.3.3.1. Unidirectional

LSTM Unidirectional



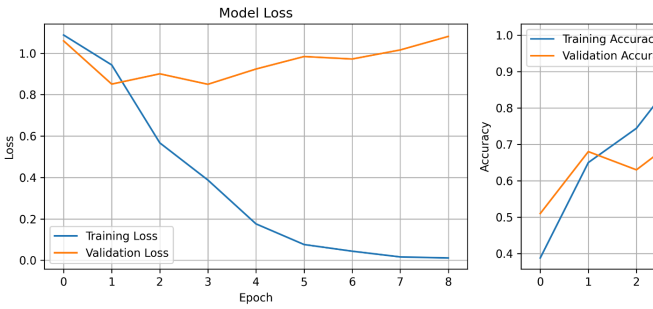


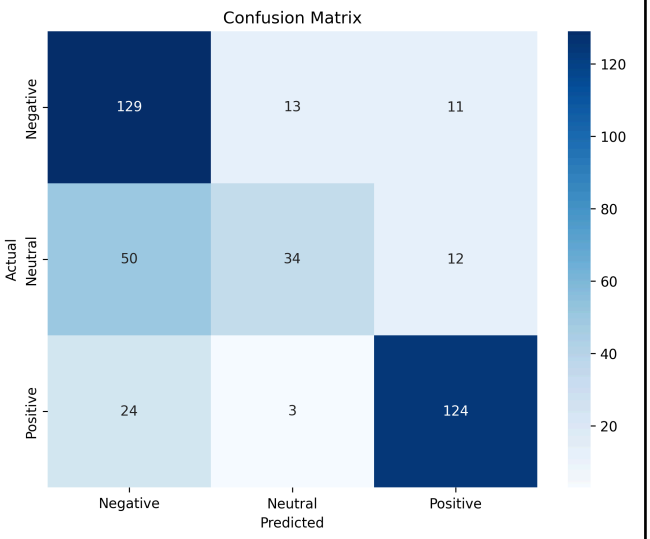
Validation Results:
Macro F1: 0.7972
Accuracy: 0.8000

Test Results:
Macro F1: 0.7421
Accuracy: 0.7500

3.3.3.2. Bidirectional

LSTM Bidirectional





Validation Results:
Macro F1: 0.6997
Accuracy: 0.7300

Test Results:
Macro F1: 0.6742
Accuracy: 0.7175

Pada pengujian menggunakan LSTM unidirectional, model menunjukkan performa yang lebih baik dibandingkan dengan bidirectional. Pada data validasi, Macro F1 mencapai 0.7972 dengan akurasi 0.8000, sedangkan pada data uji Macro F1 sebesar 0.7421 dan akurasi 0.7500. Performa ini mengindikasikan bahwa LSTM unidirectional mampu menangkap pola urutan data dengan baik untuk kasus ini.

Sebaliknya, model dengan LSTM bidirectional memiliki performa yang lebih rendah. Validasi menunjukkan Macro F1 sebesar 0.6997 dengan akurasi 0.7300, dan pada data uji Macro F1 sebesar 0.6742 dengan akurasi 0.7175. Meskipun secara teori bidirectional LSTM memiliki keuntungan karena dapat melihat konteks dari kedua arah (maju dan mundur), dalam kasus ini model bidirectional kurang optimal dibandingkan unidirectional.

BAB IV
PENUTUP

3.1. Kesimpulan

Kami berhasil mengimplementasikan CNN, RNN, dan LSTM from scratch maupun menggunakan Keras.

3.2. Saran

Mengerjakan dari jauh-jauh hari sehingga output dapat lebih optimal

3.3. Pembagian Tugas

NIM	Penanggung Jawab	Tugas
13522069	Nabila Shikoofa Muida	- LSTM, Laporan
13522087	Shulha	- RNN, Laporan
13522109	Azmi Mahmud Bazeid	- CNN, Laporan