LAPORAN TUGAS BESAR II

IF3270 Pembelajaran Mesin

"Convolutional Neural Network dan Recurrent Neural Network"

Dosen:

Dr. Fariska Zakhralativa Ruskanda, S.T., M.T.

Disusun Oleh:

13522126 Rizqika Mulia Pratama

13522139 Attara Majesta Ayub

13522147 Ikhwan Al Hakim

PROGRAM STUDI TEKNIK INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

SEMESTER II TAHUN 2024/2025

KATA PENGANTAR

Puji syukur kepada Tuhan Yang Maha Esa, penulis ucapkan atas kesempatan dan keberhasilan dalam menyelesaikan Tugas Besar 1 IF3270 Pembelajaran Mesin, Semester II tahun 2024/2025, yang berjudul "Convolutional Neural Network dan Recurrent Neural Network". Laporan ini merupakan dokumentasi dari proses implementasi *Convolutional Neural Network* (CNN) *dan Recurrent Neural Network* (RNN) yang dibuat dari nol *(from scratch)* sesuai dengan spesifikasi yang telah ditentukan.

Tugas ini memberikan pengalaman belajar yang sangat berharga dalam memahami konsep *deep learning*, mulai dari desain arsitektur model, pemilihan fungsi aktivasi, hingga proses *forward* dan *backward propagation*. Selain itu, berbagai eksperimen dilakukan untuk menganalisis pengaruh hyperparameter, metode inisialisasi bobot, serta perbandingan performa model dengan pustaka eksternal seperti scikit-learn.

Dengan bimbingan dari dosen, dukungan dari asisten pengajar, serta kerja sama dengan rekan satu tim, penulis dapat menyelesaikan tugas ini dengan baik. Akhir kata, penulis mengucapkan terima kasih kepada semua pihak yang telah berkontribusi dalam pengerjaan tugas ini. Semoga laporan ini dapat memberikan manfaat bagi pembaca serta menjadi referensi yang berguna dalam studi lebih lanjut mengenai pembelajaran mesin.

Bandung, 30 Mei 2025,

Kelompok 31

DAFTAR ISI

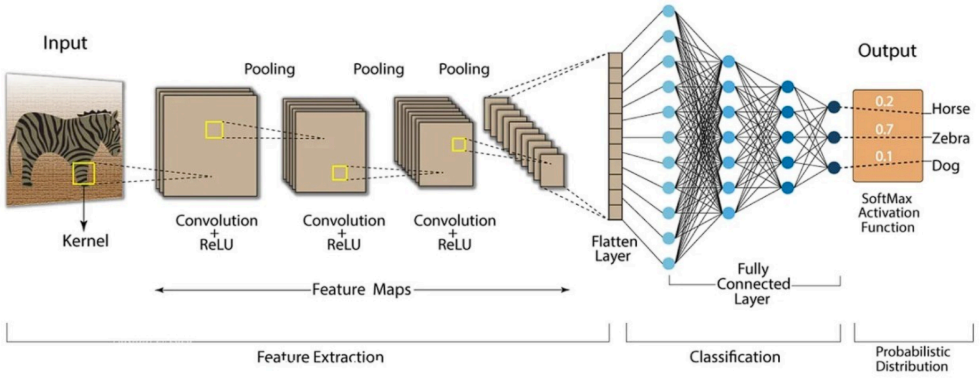
DAFTAR GAMBAR

DAFTAR TABEL

BAB I
DESKRIPSI PERSOALAN

1.1. Convolutional Neural Network

Convolutional Neural Network (CNN) adalah jenis jaringan saraf tiruan yang dirancang khusus untuk memproses data terstruktur, seperti gambar atau data grid-like (misalnya, time-series dalam format tertentu). CNN sangat efektif dalam tugas-tugas seperti pengenalan gambar, deteksi objek, dan pengolahan bahasa alami tertentu, karena kemampuannya untuk mengekstraksi fitur spasial secara otomatis dari data.



Gambar 1.1.1. Arsitektur CNN

CNN terdiri dari beberapa lapisan utama yang bekerja secara berurutan, yaitu:

1. Input Layer yang berguna untuk menerima data masukan, biasanya dalam bentuk matriks (misalnya, gambar berupa array piksel dengan dimensi tinggi × lebar × kanal warna).
2. Convolutional Layers. Lapisan ini menerapkan operasi konvolusi menggunakan filter (kernel) untuk mengekstraksi fitur seperti tepi, tekstur, atau pola dari data masukan. Setiap filter menghasilkan feature map yang menangkap informasi spasial.
3. Pooling Layers berguna untuk mengurangi dimensi spasial (downsampling) dari feature map untuk mengurangi kompleksitas komputasi dan mencegah overfitting. Contohnya adalah max pooling,

yang mengambil nilai maksimum dari wilayah tertentu dalam feature map.

4. Fully Connected Layers. Setelah fitur diekstraksi, lapisan ini (mirip dengan FFNN) menggabungkan informasi untuk menghasilkan prediksi akhir, seperti klasifikasi.
5. Output Layer bertugas untuk menghasilkan hasil akhir, seperti probabilitas kelas untuk tugas klasifikasi.

CNN memiliki beberapa keunggulan dibandingkan dengan model-model lain, yaitu:

1. Ekstraksi fitur otomatis. CNN dapat mempelajari fitur penting (seperti tepi atau tekstur) langsung dari data tanpa perlu ekstraksi manual.
2. Efisien untuk data spasial. Maksudnya adalah sangat cocok untuk data dengan struktur spasial seperti gambar, karena memanfaatkan korelasi lokal antar piksel.
3. Skalabilitas. Dengan pooling dan penggunaan filter, CNN dapat menangani data berdimensi tinggi dengan lebih efisien dibandingkan FFNN.

Namun, juga terdapat beberapa kelemahan dari CNN, yaitu:

1. Keterbatasan pada data non-spasial. CNN kurang efektif untuk data yang tidak memiliki struktur spasial, seperti teks atau data tabular.
2. Kebutuhan data besar. CNN memerlukan jumlah data pelatihan yang besar untuk mencapai performa optimal, terutama pada arsitektur yang dalam.
3. Kompleksitas komputasi. Pelatihan CNN membutuhkan sumber daya komputasi yang signifikan, terutama untuk dataset besar atau arsitektur kompleks.

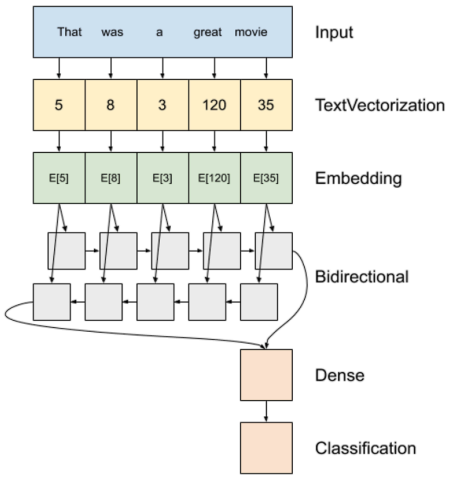
CNN menggunakan filter konvolusi untuk mengekstraksi fitur dari input, diikuti oleh fungsi aktivasi (biasanya ReLU) untuk memperkenalkan non-linearitas. Pooling membantu mengurangi dimensi data sambil mempertahankan informasi penting. Selama pelatihan, CNN menyesuaikan

bobot filter menggunakan backpropagation dan algoritma optimasi seperti gradient descent.

Fungsi aktivasi seperti ReLU memungkinkan model untuk menangkap pola yang lebih kompleks, sementara fungsi loss seperti Categorical Cross-Entropy (untuk klasifikasi) atau Mean Squared Error (untuk regresi) digunakan untuk mengukur kesalahan prediksi. Teknik regularization seperti dropout atau L2 regularization sering digunakan untuk mencegah overfitting, mirip dengan FFNN.

1.2. Recurrent Neural Network

Recurrent Neural Network (RNN) adalah jenis jaringan saraf tiruan yang dirancang untuk memproses data sekuensial, seperti teks, ucapan, atau deret waktu. RNN memiliki koneksi berulang (feedback loop) yang memungkinkan model untuk menyimpan informasi dari langkah sebelumnya, sehingga cocok untuk tugas yang melibatkan urutan atau konteks temporal.



Gambar 1.2.1. Arsitektur RNN

Lapisan utama dari RNN adalah:

1. Input Layer, yaitu untuk menerima data sekuensial, seperti kata-kata dalam kalimat atau nilai dalam deret waktu.

2. Recurrent Layers. Lapisan ini memproses setiap elemen dalam urutan sambil mempertahankan "memori" dari elemen sebelumnya melalui hidden state. Hidden state diperbarui di setiap langkah waktu.
3. Output Layer berfungsi untuk menghasilkan prediksi, baik untuk setiap langkah waktu (misalnya, dalam tugas penandaan urutan) atau untuk seluruh urutan (misalnya, klasifikasi sentimen).

RNN memiliki beberapa keunggulan dibandingkan dengan model-model lain, yaitu:

1. Kemampuan menangani data sekuensial. RNN sangat efektif untuk tugas seperti pemrosesan bahasa alami (NLP), pengenalan ucapan, dan analisis deret waktu.
2. Memori konteks. Maksudnya adalah RNN dapat menggunakan informasi dari langkah sebelumnya untuk membuat keputusan, menjadikannya cocok untuk data dengan ketergantungan temporal.
3. Fleksibilitas arsitektur. RNN dapat digunakan dalam berbagai konfigurasi, seperti one-to-many (generasi teks) atau many-to-many (terjemahan mesin).

Namun, juga terdapat beberapa kelemahan dari RNN, yaitu:

1. Masalah vanishing/exploding gradients. Selama pelatihan, gradien dapat menjadi sangat kecil atau besar, menyebabkan kesulitan dalam pembelajaran ketergantungan jarak jauh.
2. Kompleksitas pelatihan. RNN lebih sulit dan lambat untuk dilatih dibandingkan FFNN karena sifat sekuensialnya.
3. Keterbatasan memori jangka panjang. RNN standar kesulitan menangkap ketergantungan jarak jauh dalam urutan panjang.

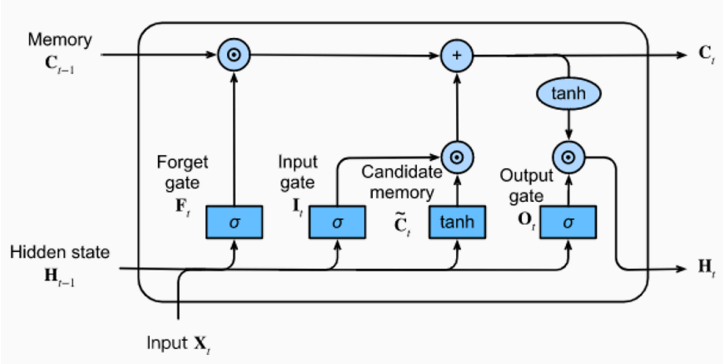
RNN memproses data secara sekuensial, memperbarui hidden state di setiap langkah waktu berdasarkan input saat ini dan hidden state sebelumnya. Bobot model disesuaikan menggunakan backpropagation through time (BPTT), yang merupakan variasi backpropagation untuk data sekuensial. Fungsi aktivasi seperti tanh atau ReLU digunakan dalam lapisan rekuren, sementara fungsi loss

seperti Categorical Cross-Entropy (untuk klasifikasi) atau Mean Squared Error (untuk regresi) digunakan untuk mengukur kesalahan.

Fungsi aktivasi seperti ReLU memungkinkan model untuk menangkap pola yang lebih kompleks, sementara fungsi loss seperti Categorical Cross-Entropy (untuk klasifikasi) atau Mean Squared Error (untuk regresi) digunakan untuk mengukur kesalahan prediksi. Teknik regularization seperti dropout atau L2 regularization sering digunakan untuk mencegah overfitting, mirip dengan FFNN.

1.2.1. Long Short-Term Memory

LSTM adalah varian khusus dari RNN yang dirancang untuk mengatasi masalah vanishing gradients dan memungkinkan pembelajaran ketergantungan jangka panjang dalam data sekuensial.



Gambar 1.2.1.1. Arsitektur LSTM

LSTM memperkenalkan memory cell dan tiga gerbang (gates) untuk mengontrol aliran informasi:

1. Forget Gate, berfungsi untuk menentukan informasi mana dari memori sebelumnya yang akan dilupakan.
2. Input Gate, berfungsi untuk menentukan informasi baru dari input yang akan disimpan dalam memory cell.
3. Output Gate, berfungsi untuk mengontrol informasi mana dari memory cell yang akan diteruskan sebagai output atau hidden state.

LSTM memproses urutan dengan memperbarui memory cell dan hidden state di setiap langkah waktu. Gerbang-forget, input, dan output memungkinkan model untuk secara selektif mengingat atau melupakan informasi, menjadikannya lebih robust untuk urutan panjang. Pelatihan dilakukan dengan BPTT, dan fungsi aktivasi seperti sigmoid (untuk gerbang) dan tanh (untuk memory cell) sering digunakan.

BAB II

PEMBAHASAN & IMPLEMENTASI

2.1 Repositori GitHub

Source code dari program ini dapat diakses melalui GitHub pada https://github.com/rizqikapratamaa/Tubes2-IF3270-ML31.

2.2 Deskripsi Kelas

2.2.1. CNN

2.2.1.1. ManualConv2DLayer

Tabel 2.2.1.1.1. Atribut ManualConv2DLayer

Atribut	Penjelasan
kernel	Matriks filter konvolusi dengan bentuk (KH, KW, C_in, C_out), digunakan untuk mengekstraksi fitur dari input.
bias	Vektor bias dengan bentuk (C_out,), ditambahkan ke hasil konvolusi untuk menyesuaikan output.
padding	Jenis padding ('same' atau 'valid') yang menentukan apakah input akan dipadding untuk mempertahankan dimensi atau tidak.
stride_h	Langkah (stride) konvolusi dalam arah tinggi (height), menentukan jarak lompatan filter pada sumbu vertikal.
stride_w	Langkah (stride) konvolusi dalam arah lebar (width), menentukan jarak lompatan filter pada sumbu horizontal.

Tabel 2.2.1.1.2. Metode ManualConv2DLayer

Metode	Penjelasan
__init__(self, kernel, bias, stride=(1,1), padding='valid')	Inisialisasi lapisan dengan kernel, bias, stride, dan padding. Mengatur stride_h dan stride_w berdasarkan input stride. Memvalidasi format stride dan mengubah padding ke huruf kecil.
forward(self, X_batch)	Melakukan operasi konvolusi pada input X_batch (bentuk: (N, H_in, W_in, C_in)). Menangani padding ('same' atau 'valid'), menghitung dimensi output, dan menghasilkan output konvolusi dengan bentuk (N, out_height, out_width, C_out) setelah menambahkan bias.

2.2.1.2. ManualMaxPooling2DLayer

Tabel 2.2.1.2.1. Atribut ManualMaxPooling2DLayer

Atribut	**Penjelasan**
pool_size	Tuple (PH, PW) yang menentukan ukuran jendela pooling untuk operasi max pooling.
padding	Jenis padding ('same' atau 'valid') yang menentukan apakah input akan dipadding.
stride_h	Langkah (stride) pooling dalam arah tinggi (height), menentukan jarak lompatan jendela pooling secara vertikal.
stride_w	Langkah (stride) pooling dalam arah lebar (width), menentukan jarak lompatan jendela pooling secara horizontal.

Tabel 2.2.1.2.2. Metode ManualMaxPooling2DLayer

Metode	**Penjelasan**
__init__(self, pool_size=(2, 2), stride=None, padding='valid')	Inisialisasi lapisan dengan ukuran pool, stride, dan padding. Jika stride tidak ditentukan, menggunakan pool_size sebagai stride. Memvalidasi format stride dan mengubah padding ke huruf kecil.
forward(self, X_batch)	Melakukan operasi max pooling pada input X_batch (bentuk: (N, H_in, W_in, C_in)). Menangani padding, menghitung dimensi output, dan menghasilkan output dengan bentuk (N, out_height, out_width, C_in) dengan mengambil nilai maksimum dari setiap jendela pooling.

2.2.1.3. ManualAveragePooling2DLayer

Tabel 2.2.1.3.1. Atribut ManualAveragePooling2DLayer

Atribut	**Penjelasan**
pool_size	Tuple (PH, PW) yang menentukan ukuran jendela pooling untuk operasi average pooling.
padding	Jenis padding ('same' atau 'valid') yang menentukan apakah input akan dipadding.
stride_h	Langkah (stride) pooling dalam arah tinggi (height), menentukan jarak lompatan jendela pooling secara vertikal.
stride_w	Langkah (stride) pooling dalam arah lebar (width), menentukan jarak lompatan jendela pooling secara horizontal.

Tabel 2.2.1.3.2. Atribut ManualAveragePooling2DLayer

Metode	**Penjelasan**
__init__(self, pool_size=(2, 2), stride=None, padding='valid')	Inisialisasi lapisan dengan ukuran pool, stride, dan padding. Jika stride tidak ditentukan, menggunakan pool_size sebagai stride. Memvalidasi format stride dan mengubah padding ke huruf kecil.
forward(self, X_batch)	Melakukan operasi average pooling pada input X_batch (bentuk: (N, H_in, W_in, C_in)). Menangani padding, menghitung dimensi output, dan menghasilkan output dengan bentuk (N, out_height, out_width, C_in) dengan menghitung rata-rata dari setiap jendela pooling.

2.2.1.4. ManualFlattenLayer

Tabel 2.2.1.4.1. Atribut ManualFlattenLayer

Atribut	**Penjelasan**
Tidak ada	Kelas ini tidak memiliki atribut karena hanya berfungsi untuk meratakan input.

Tabel 2.2.1.4.2. Metode ManualFlattenLayer

Metode	**Penjelasan**
forward(self, X_batch)	Meratakan input X_batch (bentuk: (N, H, W, C)) menjadi vektor 2D dengan bentuk (N, HWC) untuk diproses oleh lapisan berikutnya, seperti lapisan dense.

2.2.1.5. ManualGlobalAveragePooling2DLayer

Tabel 2.2.1.5.1. Atribut ManualGlobalAveragePooling2DLayer

Atribut	**Penjelasan**
Tidak ada	Kelas ini tidak memiliki atribut karena hanya melakukan operasi global average pooling.

Tabel 2.2.1.5.2. Metode ManualGlobalAveragePooling2DLayer

Metode	**Penjelasan**
forward(self, X_batch)	Melakukan global average pooling pada input X_batch (bentuk: (N, H, W, C)) dengan menghitung rata-rata pada sumbu spasial (H, W), menghasilkan output dengan bentuk (N, C).

2.2.1.6. ManualDenseLayer

Tabel 2.2.1.6.1. Atribut ManualDenseLayer

Atribut	**Penjelasan**
weights	Matriks bobot dengan bentuk (input_features, output_features) untuk transformasi linier.
bias	Vektor bias dengan bentuk (output_features,) yang ditambahkan ke hasil transformasi linier.
activation_fn	Fungsi aktivasi (misalnya, ReLU atau softmax) yang diterapkan pada hasil transformasi linier, jika ditentukan.

Tabel 2.2.1.6.2. Metode ManualDenseLayer

Metode	**Penjelasan**
__init__(self, weights, bias, activation_fn=None)	Inisialisasi lapisan dengan bobot, bias, dan fungsi aktivasi (opsional).
forward(self, X_batch)	Melakukan transformasi linier pada input X_batch (bentuk: (N, input_features)) dengan operasi dot product dan penambahan bias, diikuti oleh penerapan fungsi aktivasi (jika ada), menghasilkan output dengan bentuk (N, output_features).

2.2.1.7. CNNFromScratch

Tabel 2.2.1.7.1. Atribut CNNFromScratch

Atribut	**Penjelasan**
layers	Daftar lapisan yang membentuk arsitektur CNN, termasuk lapisan manual (Conv2D, Pooling, dll.) dan fungsi aktivasi.
keras_layer_map	Kamus yang memetakan jenis lapisan Keras (Conv2D, MaxPooling2D, dll.) ke kelas manual yang sesuai untuk konversi.

Tabel 2.2.1.7.2. Metode CNNFromScratch

Metode	**Penjelasan**
__init__(self)	Inisialisasi objek CNN dengan daftar lapisan kosong dan peta lapisan Keras ke lapisan manual.
add_keras_layer(self, keras_layer)	Menambahkan lapisan Keras ke daftar lapisan dengan mengkonversinya ke lapisan manual yang sesuai berdasarkan keras_layer_map. Menangani fungsi aktivasi (ReLU, softmax) dan mengabaikan lapisan Dropout. Memvalidasi jenis lapisan dan menangani parameter seperti bobot, bias, stride, dan padding.

load_keras_model(self, keras_model)	Mengatur ulang daftar lapisan dan memproses setiap lapisan dari model Keras yang diberikan, memanggil add_keras_layer untuk setiap lapisan.
predict(self, X_batch)	Melakukan inferensi dengan memproses input X_batch melalui semua lapisan secara berurutan, menerapkan fungsi aktivasi atau metode forward dari lapisan manual sesuai kebutuhan. Menghasilkan output prediksi akhir.

2.2.2. RNN

2.2.1.8. ManualEmbeddingLayer

Tabel 2.2.3.1.1. Atribut ManualEmbeddingLayer

Atribut	**Penjelasan**
embedding_ matrix	Matriks bobot embedding dengan bentuk (vocab_size, embed_dim), digunakan untuk mengubah token menjadi vektor dense.

Tabel 2.2.3.1.2. Metode Manual EmbeddingLayer

Metode	**Penjelasan**
__init__(self, embedding_matrix)	Menginisialisasi lapisan embedding dengan matriks embedding yang diberikan .
forward(self, X_tokens_batch)	Melakukan operasi lookup embedding pada X_tokens_batch (bentuk: (N, seq_len)) untuk menghasilkan vektor embedding.

2.2.1.9. ManualDenseLayer

Tabel 2.2.3.2.1. Atribut ManualDenseLayer

Atribut	**Penjelasan**
weights	Matriks bobot dengan bentuk (input_features, output_features) untuk transformasi linier.
bias	Vektor bias dengan bentuk (output_reatures) yang ditambahkan ke hasil transformasi linier.
activation_fn	Fungsi aktivasi (misalnya, relu atau softmax) yang diterapkan pada hasil transformasi linier, jika ditentukan.

Tabel 2.2.3.1.2. Metode ManualDenseLayer

Metode	**Penjelasan**
__init__(self, weights, bias, activation_fn=None)	Menginisialisasi lapisan dengan bobot, bias, dan fungsi aktivasi (opsional).
forward(self, X_batch)	Melakukan transformasi linier pada input X_batch (bentuk: (N, input_features)), diikuti oleh penerapan fungsi aktivasi (jika ada), menghasilkan output dengan bentuk (N, output_features).

2.2.1.10. ManualRNNCell

Tabel 2.2.3.3.1. Atribut ManualRNNCell

Atribut	**Penjelasan**
Wx	Matriks bobot input (input_dim, units) untuk input saat ini xt.
Wh	Matriks bobot recurrent (units, units) untuk hidden state sebelumnya h_prev.
b	Vektor bias (units,) yang ditambahkan pada setiap langkah waktu.
activation	Fungsi aktivasi non-linear yang digunakan (default: tanh).

Tabel 2.2.3.3.2. Metode ManualRNNCell

Metode	**Penjelasan**
__init__(self, Wx, Wh, b, activation=tanh)	Konstruktor untuk menginisialisasi bobot input, bobot hidden, bias, dan fungsi aktivasi.
forward(self, xt, h_prev)	Menghitung hidden state saat ini h_next dari input xt dan hidden state sebelumnya h_prev menggunakan formula: h_next = activation(xt·Wx + h_prev·Wh + b).

2.2.1.11. ManualRNNLayer

Tabel 2.2.3.4.1. Atribut ManualRNNLayer

Atribut	**Penjelasan**
cell	Objek ManualRNNCell yang menangani operasi per langkah waktu dalam RNN.
return_sequences	Boolean yang menentukan apakah output mencakup seluruh urutan hidden states (True) atau hanya hidden state terakhir (False).

go_backwards	Boolean yang menentukan apakah urutan input diproses dari belakang ke depan.
units	Jumlah unit RNN (dimensi hidden state).

Tabel 2.2.3.4.2. Metode ManualRNNLayer

Metode	**Penjelasan**
__init__(self, units, Wx, Wh, b, activation=tanh, return_sequences=False, go_backwards=False)	Konstruktor yang menginisialisasi layer RNN dengan parameter bobot (Wx, Wh, b), fungsi aktivasi, dan konfigurasi arah serta mode output.
forward(self, X_embedded_batch)	Melakukan propagasi maju pada input 3D (N, T, D) dengan memproses setiap timestep melalui ManualRNNCell. Mengembalikan urutan hidden states (N, T, H) jika return_sequences=True, atau hidden state terakhir (N, H) jika False. Jika go_backwards=True, urutan input dibalik sebelum diproses dan output dikembalikan ke urutan semula.

2.2.1.12. ManualBidirectionalLSTMWrapper

Tabel 2.2.3.5.1. Atribut ManualBidirectionalLSTMWrapper

Atribut	**Penjelasan**
forward_rnn_layer	Objek ManualRNNLayer yang memproses input dalam arah maju.
backward_rnn_layer	Objek ManualRNNLayer yang memproses input dalam arah mundur.
merge_mode	Cara menggabungkan output dari layer maju dan mundur.

Tabel 2.2.3.5.2. Metode ManualBidirectionalLSTMWrapper

Metode	**Penjelasan**
__init__(self, forward_rnn_layer, backward_rnn_layer, merge_mode='concat')	Konstruktor yang menginisialisasi wrapper bidirectional RNN dengan dua layer RNN arah depan dan belakang, serta mode penggabungan hasil (merge_mode).
forward(self, X_embedded_batch)	Melakukan forward pass dengan dua arah: input asli diproses oleh forward_rnn_layer, input terbalik diproses oleh backward_rnn_layer. Jika layer mengembalikan urutan (return_sequences=True), hasil backward dibalik

	lagi agar urutannya sama. Hasil akhir digabung secara concat di dimensi fitur.

2.2.1.13. SimpleRNNFromScratch

Tabel 2.2.3.6.1. Atribut SimpleRNNFromScratch

Atribut	**Penjelasan**
layers	List berisi lapisan-lapisan manual (seperti ManualEmbeddingLayer, ManualRNNLayer, dll.) yang dibangun dari model Keras.
is_built	Boolean penanda apakah model sudah dibangun dengan memanggil build_from_keras.
vectorizer	Objek vectorizer untuk melakukan tokenisasi teks. Bisa bernilai None jika hanya untuk inference numerik.
vocabulary_size	Ukuran kosakata dari vectorizer, disimpan jika tersedia.

Tabel 2.2.3.6.2. Metode SimpleRNNFromScratch

Metode	**Penjelasan**
__init__(self)	Konstruktor untuk inisialisasi awal atribut layers, is_built, dan vectorizer.
build_from_keras(self, keras_model, vectorizer=None)	Membangun layer-layer manual dari model Keras. Secara otomatis menyalin bobot dan fungsi aktivasi ke dalam bentuk manual, mendukung layer: Embedding, SimpleRNN, Bidirectional(SimpleRNN), dan Dense.
forward(self, x)	Melakukan forward pass melalui seluruh layer yang telah dibangun dalam urutan.
predict(self, x)	Melakukan prediksi terhadap input x menggunakan forward pass. Akan melempar error jika model belum dibangun dengan build_from_keras.
load_and_preprocess_nusax_sentiment(max_features, maxlen)	Memuat dataset NusaX (train/valid/test), melakukan label encoding, dan membuat TextVectorization. Output: data siap pakai, jumlah kelas, vocab size, dan vectorizer.
build_rnn_model(vocab_size, embedding_dim, max_len, num_classes, rnn_config, rnn_type, bidirectional)	Membangun model Sequential sederhana menggunakan Embedding + satu atau lebih SimpleRNN (bisa bidirectional) + Dense.
rnn_hyperparameter_analysis(x	Membuat dan melatih model SimpleRNN untuk analisis

_train, y_train, x_val, y_val, x_test, y_test, num_classes, vocab_size, rnn_type_to_test='SimpleRNN')	hyperparameter sederhana, menyimpan model terbaik.
build_simplernn_model(vocab_size, embedding_dim, maxlen, num_classes, rnn_configs, dropout_rate=0.5)	Membangun model SimpleRNN modular dari konfigurasi rnn_configs (tiap dict bisa spesifik units, bidirectional, return_sequences).
train_and_evaluate_simplernn_variant(x_train, y_train, x_val, y_val, x_test, y_test, num_classes, vocab_size, embedding_dim, max_len, rnn_config, epochs=5, batch_size=32, dropout_rate=0.2, description="")	Melatih model dengan konfigurasi rnn_config, lalu mengembalikan model, skor F1, dan history. Sudah ada callback EarlyStopping dan ReduceLROnPlateau.

2.2.3. LSTM

2.2.1.14. ManualEmbeddingLayer

Tabel 2.2.3.1.1. Atribut ManualEmbeddingLayer

Atribut	**Penjelasan**
embedding_matrix	Matriks bobot embedding dengan bentuk (vocab_size, embed_dim), digunakan untuk mengubah token menjadi vektor dense.
params	Kamun yang menyimpan parameter yang dapat dilatih oleh layer, dalam hal ini embedding_matrix
grads	Kamus yang menyimpan gradien untuk setiap parameter selama backpropagation

Tabel 2.2.3.1.2. Metode Manual EmbeddingLayer

Metode	**Penjelasan**
__init__(self, embedding_matrix)	Menginisialisasi lapisan embedding dengan matriks embedding yang diberikan .
forward(self, X_tokens_batch)	Melakukan operasi lookup embedding pada X_tokens_batch (bentuk: (N, seq_len)) untuk menghasilkan vektor embedding.
backward(self, dL_dOutput, X_Tokens_batch)	Menghitung gradien dL_dOutput terhadap embedding_matrix berdasarkan token input yang

	digunakan.

2.2.1.15. ManualDenseLayer

Tabel 2.2.3.2.1. Atribut ManualDenseLayer

Atribut	**Penjelasan**
weights	Matriks bobot dengan bentuk (input_features, output_features) untuk transformasi linier.
bias	Vektor bias dengan bentuk (output_reatures) yang ditambahkan ke hasil transformasi linier.
activation_fn	Fungsi aktivasi (misalnya, relu atau softmax) yang diterapkan pada hasil transformasi linier, jika ditentukan.
params	Kamus yang menyimpan bobot dan bias layer.
grads	Kamus yang menyimpan gradien untuk bobot dan bias.
input_	Menyimpan input terakhir yang melewati lapisan untuk digunakan dalam backpropagation.
output_	Menyimpan output terakhir lapisan (setelah fungsi aktivasi) untuk digunakan dalam backpropagation.

Tabel 2.2.3.1.2. Metode ManualDenseLayer

Metode	**Penjelasan**
__init__(self, weights, bias, activation_fn=None)	Menginisialisasi lapisan dengan bobot, bias, dan fungsi aktivasi (opsional).
forward(self, X_batch)	Melakukan transformasi linier pada input X_batch (bentuk: (N, input_features)), diikuti oleh penerapan fungsi aktivasi (jika ada), menghasilkan output dengan bentuk (N, output_features).
backward(self, dL_dOutput)	Menghitung gradien dL_dOutput terhadap input lapisan dan parameter weights dan bias, lalu mengembalikannya sebagai dL_dInput.

2.2.1.16. ManualLSTMCell

Tabel 2.2.3.3.1. Atribut ManualLSTMCell

Atribut	**Penjelasan**

Wx_i, Wx_f, Wx_c, Wx_o	Bobot input untuk gerbang input, forget, candidate memory, dan output.
Wh_i, Wh_f, Wh_c, Wh_o	Bobot rekuren (hidden state) untuk gerbang input, forget, candidate memory, dan output.
b_i, b_f, b_c, b_o	Bias untuk gerbang input, forget, candidate memory, dan output.
activation	Fungsi aktivasi untuk candidate memory dan output hidden state (biasanya tanh).
recurrent_activation	Fungsi aktivasi untuk gerbang (biasanya sigmoid).
params	Kamus yang menyimpan semua bobot dan bias dari sel LSTM.
grads	Kamus yang menyimpan gradien untuk semua bobot dan bias sel LSTM.

Tabel 2.2.3.3.2. Metode ManualLSTMCell

Metode	**Penjelasan**
__init__(self, Wx, Wh, b, activation=tanh, recurrent_activation=sigmoid)	Menginisialisasi sel LSTM dengan bobot input, bobot rekuren, bias, dan fungsi aktivasi/rekuren.
forward(self, xt, h_prev, c_prev)	Melakukan satu langkah forward pass dari sel LSTM pada input waktu xt, hidden state sebelumnya h_prev, dan cell state sebelumnya c_prev. Mengembalikan h_next, c_next, dan cache untuk backpropagation.
backward(self, dh_next, dc_next_from_tplus1, cache)	Melakukan backpropagation melalui sel LSTM untuk satu langkah waktu, menghitung gradien terhadap xt, h_prev, c_prev, dan parameter sel.
reset_grads(self)	Mereset semua gradien sel ke nol.

2.2.1.17. ManualLSTMLayer

Tabel 2.2.3.4.1. Atribut ManualLSTMLayer

Atribut	**Penjelasan**
cell	Objek ManualLSTMCell yang digunakan oleh lapisan LSTM.
return_sequences	Boolean, jika True, lapisan mengembalikan hidden state untuk setiap langkah waktu; jika False, hanya mengembalikan hidden state terakhir.

go_backwards	Boolean, jika True, urutan input diproses terbalik.
units	Jumlah unit (dimensi hidden state dan cell state) dalam lapisan LSTM.
params	Kamus yang menyimpan parameter dari sel LSTM.
grads	Kamus yang menyimpan gradien dari sel LSTM.
cell_caches_	Daftar cache dari setiap langkah waktu selama forward pass, digunakan untuk backpropagation through time (BPTT).
last_output_	Output terakhir dari forward pass lapisan.

Tabel 2.2.3.4.2. Metode ManualLSTMLayer

Metode	**Penjelasan**
__init__(self, units, Wx, Wh, b, activation=tanh, recurrent_activation=sigmoid, return_sequences=False, go_backwards=False)	Menginisialisasi lapisan LSTM dengan units, bobot, bias, fungsi aktivasi, dan konfigurasi return_sequences dan go_backwards.
forward(self, X_embedded_batch)	Melakukan forward pass melalui urutan input X_embedded_batch, memproses setiap langkah waktu menggunakan ManualLSTMCell. Mengembalikan output hidden state(s).
backward(self, dL_dOutput)	Melakukan backpropagation through time (BPTT) melalui lapisan LSTM, menghitung gradien terhadap input dan parameter lapisan.

2.2.1.18. ManualBidirectionalLSTMWrapper

Tabel 2.2.3.5.1. Atribut ManualBidirectionalLSTMWrapper

Atribut	**Penjelasan**
forward_lstm_layer	Objek ManualLSTMLayer yang memproses input dalam arah maju.
backward_lstm_layer	Objek ManualLSTMLayer yang memproses input dalam arah mundur.
merge_mode	Cara menggabungkan output dari layer maju dan mundur.
params	Kamus yang menyimpan parameter dari kedua lapisan LSTM maju dan mundur.

grads	Kamus yang menyimpan gradien dari kedua lapisan LSTM maju dan mundur.
last_output_	Output terakhir setelah penggabungan dari kedua lapisan LSTM.

Tabel 2.2.3.5.2. Metode ManualBidirectionalLSTMWrapper

Metode	**Penjelasan**
__init__(self, forward_lstm_layer, backward_lstm_layer, merge_mode='concat')	Menginisialisasi wrapper LSTM bidirectional dengan lapisan LSTM maju dan mundur, serta mode penggabungan.
forward(self, X_embedded_batch)	Melakukan forward pass melalui kedua lapisan LSTM (maju dan mundur), kemudian menggabungkan output-nya.
backward(self, dL_dOutput_concat)	Melakukan backpropagation melalui kedua lapisan LSTM (maju dan mundur) berdasarkan gradien output gabungan.

2.2.1.19. LSTMFromScratch

Tabel 2.2.3.6.1. Atribut LSTMFromScratch

Atribut	**Penjelasan**
layers	Daftar lapisan manual (seperti ManualEmbeddingLayer, ManualLSTMLayer, ManualDenseLayer, dll.) yang membentuk arsitektur LSTM.
keras_model	Referensi ke model Keras asli yang digunakan untuk memuat bobot dan arsitektur.
loss_fn	Instans fungsi kerugian yang digunakan untuk pelatihan (misalnya, ManualSparseCategoricalCrossentropy).
optimizer	Instans optimizer yang digunakan untuk memperbarui bobot model (misalnya, ManualSGD).

Tabel 2.2.3.6.2. Metode LSTMFromScratch

Metode	**Penjelasan**
__init__(self)	Menginisialisasi objek LSTMFromScratch dengan daftar lapisan kosong dan keras_model yang belum diatur.
add_keras_layer(self, keras_layer)	Menambahkan lapisan Keras ke daftar lapisan manual, mengonversinya ke representasi kelas manual yang sesuai. Menangani lapisan Embedding, LSTM, Bidirectional, dan Dense.

load_keras_model(self, keras_model)	Memuat arsitektur dan bobot dari model Keras yang diberikan ke dalam lapisan manual.
predict(self, X_batch_tokens)	Melakukan inferensi dengan memproses input X_batch_tokens melalui semua lapisan secara berurutan, menghasilkan output prediksi akhir.
compile_manual(self, loss_fn_instance, optimizer_instance)	Mengatur fungsi kerugian dan optimizer yang akan digunakan untuk pelatihan manual.
fit_manual(self, X_train_tokens, y_train_labels, epochs=1, batch_size=32)	Melakukan proses pelatihan manual (forward pass, perhitungan loss, backpropagation, dan pembaruan bobot) untuk beberapa epoch.

2.2.1.20. ManualSGD

Tabel 2.2.3.7.1. Atribut LSTMFromScratch

Atribut	**Penjelasan**
learning_rate	Tingkat pembelajaran yang digunakan untuk memperbarui bobot model.

Tabel 2.2.3.7.2. Metode LSTMFromScratch

Metode	**Penjelasan**
__init__(self, learning_rate=0.01)	Menginisialisasi optimizer SGD dengan tingkat pembelajaran yang ditentukan.
update(self, layers_list)	Memperbarui bobot dan bias semua lapisan yang memiliki parameter dan gradien, menggunakan tingkat pembelajaran yang ditentukan.

2.3 Algoritma

2.3.1. CNN

2.3.1.1. ManualConv2DLayer

Algoritma forward propagation pada ManualConv2DLayer melakukan operasi konvolusi untuk mengekstraksi fitur dari input, seperti gambar, menggunakan filter (kernel) dan bias. Proses dimulai dengan menerima input X_batch dengan bentuk (N, H_in, W_in, C_in), di mana N adalah jumlah sampel, H_in dan W_in adalah tinggi dan lebar input, dan C_in adalah jumlah kanal. Pertama, algoritma menghitung dimensi output (out_height dan out_width) berdasarkan ukuran kernel (KH, KW), stride (stride_h, stride_w), dan jenis padding ('same' atau 'valid'). Jika

padding adalah 'same', input dipadding dengan nol untuk mempertahankan dimensi output; jika 'valid', tidak ada padding. Input kemudian diubah menjadi representasi kolom menggunakan np.lib.stride_tricks.as_strided untuk memungkinkan operasi konvolusi efisien. Representasi ini memiliki bentuk (N, out_height, out_width, KH, KW, C_in). Kolom ini diratakan menjadi matriks 2D (N * out_height * out_width, KH * KW * C_in), dan kernel diratakan menjadi (KH * KW * C_in, C_out). Operasi dot product dilakukan antara matriks ini untuk menghasilkan output konvolusi, yang kemudian dibentuk kembali menjadi (N, out_height, out_width, C_out). Terakhir, vektor bias (C_out,) ditambahkan ke output untuk menghasilkan hasil akhir.

```
def forward(self, X_batch):  # X_batch shape: (N, H_in, W_in, C_in)
    N, H_in, W_in, C_in = X_batch.shape
    KH, KW, _, C_out = self.kernel.shape

    if self.padding == 'same':
        out_height = int(np.ceil(float(H_in) /
float(self.stride_h)))
        out_width = int(np.ceil(float(W_in) /
float(self.stride_w)))

        pad_h_total = max((out_height - 1) * self.stride_h + KH -
H_in, 0)
        pad_w_total = max((out_width - 1) * self.stride_w + KW -
W_in, 0)

        pad_top = pad_h_total // 2
        pad_bottom = pad_h_total - pad_top
        pad_left = pad_w_total // 2
        pad_right = pad_w_total - pad_left

        X_padded = np.pad(X_batch, ((0, 0), (pad_top, pad_bottom),
(pad_left, pad_right), (0, 0)), mode='constant',
constant_values=0.0)
    elif self.padding == 'valid':
        out_height = (H_in - KH) // self.stride_h + 1
        out_width = (W_in - KW) // self.stride_w + 1
        X_padded = X_batch
    else:
        raise ValueError(f"Unsupported padding type:
{self.padding}")

    if out_height <= 0 or out_width <= 0:
        raise ValueError(f"Output dimensions non-positive:
H={out_height}, W={out_width}. Input: H_in={H_in}, W_in={W_in},
Kernel: KH={KH}, KW={KW}, Stride:
({self.stride_h},{self.stride_w}), Padding: {self.padding}")

    shape_col = (N, out_height, out_width, KH, KW, C_in)

    strides_col = (
        X_padded.strides[0],
        self.stride_h * X_padded.strides[1],
```

```
        self.stride_w * X_padded.strides[2],
        X_padded.strides[1],
        X_padded.strides[2],
        X_padded.strides[3]
    )

    H_padded, W_padded = X_padded.shape[1:3]
    if (out_height > 0 and (out_height - 1) * self.stride_h + KH >
H_padded) or \
       (out_width > 0 and (out_width - 1) * self.stride_w + KW >
W_padded):
        raise ValueError(
            f"Calculated patch extent exceeds padded input
dimensions. "
            f"H_padded={H_padded}, W_padded={W_padded}. "
            f"out_height={out_height}, out_width={out_width}. "
            f"Required H: {(out_height - 1) * self.stride_h + KH},
"
            f"Required W: {(out_width - 1) * self.stride_w + KW}."
        )

    X_col = np.lib.stride_tricks.as_strided(X_padded,
shape=shape_col, strides=strides_col)

    X_col_reshaped = X_col.reshape(N * out_height * out_width, KH *
KW * C_in)
    kernel_reshaped = self.kernel.reshape(KH * KW * C_in, C_out)

    output_reshaped = np.dot(X_col_reshaped, kernel_reshaped)
    output = output_reshaped.reshape(N, out_height, out_width,
C_out)

    output += self.bias

    return output
```

2.3.1.2. ManualMaxPooling2DLayer

Algoritma forward propagation pada ManualMaxPooling2DLayer melakukan operasi max pooling untuk mengurangi dimensi spasial input sambil mempertahankan fitur penting. Input X_batch memiliki bentuk (N, H_in, W_in, C_in). Algoritma pertama menghitung dimensi output (out_height dan out_width) berdasarkan ukuran pool (PH, PW), stride (stride_h, stride_w), dan jenis padding ('same' atau 'valid'). Untuk padding 'same', input dipadding dengan nilai -np.inf agar operasi max pooling mengabaikan area yang dipadding; untuk 'valid', tidak ada padding. Input diproses menggunakan np.lib.stride_tricks.as_strided untuk membuat representasi patch dengan bentuk (N, out_height, out_width, C_in, PH, PW), yang mewakili jendela pooling yang akan diproses. Nilai maksimum diambil dari sumbu (4, 5) (dimensi pool PH dan PW) untuk setiap patch, menghasilkan output dengan bentuk (N,

out_height, out_width, C_in). Algoritma juga memastikan input dikonversi ke tipe data float untuk konsistensi perhitungan.

```
def forward(self, X_batch):  # X_batch shape: (N, H_in, W_in, C_in)
    N, H_in, W_in, C_in = X_batch.shape
    PH, PW = self.pool_size

    if self.padding == 'same':
        out_height = int(np.ceil(float(H_in) /
float(self.stride_h)))
        out_width = int(np.ceil(float(W_in) /
float(self.stride_w)))
        pad_h_total = max((out_height - 1) * self.stride_h + PH -
H_in, 0)
        pad_w_total = max((out_width - 1) * self.stride_w + PW -
W_in, 0)
        pad_top = pad_h_total // 2; pad_bottom = pad_h_total -
pad_top
        pad_left = pad_w_total // 2; pad_right = pad_w_total -
pad_left

        X_float = X_batch if np.issubdtype(X_batch.dtype,
np.floating) else X_batch.astype(np.float32)
        X_padded = np.pad(X_float,
((0,0),(pad_top,pad_bottom),(pad_left,pad_right),(0,0)),
                          mode='constant', constant_values=-np.inf)
    elif self.padding == 'valid':
        out_height = (H_in - PH) // self.stride_h + 1
        out_width = (W_in - PW) // self.stride_w + 1
        if not np.issubdtype(X_batch.dtype, np.floating):
             X_padded = X_batch.astype(np.float32)
        else:
             X_padded = X_batch
    else:
        raise ValueError(f"Unsupported padding: {self.padding}")

    if out_height <= 0 or out_width <= 0:
        raise ValueError(f"Invalid output dims: H_out={out_height},
W_out={out_width} for H_in={H_in}, W_in={W_in}, Pool=({PH},{PW}),
Stride=({self.stride_h},{self.stride_w})")

    S_N, S_H, S_W, S_C = X_padded.strides
    output_shape_strided = (N, out_height, out_width, C_in, PH, PW)
    output_strides = (S_N, self.stride_h * S_H, self.stride_w *
S_W, S_C, S_H, S_W)

    H_padded, W_padded = X_padded.shape[1:3]
    if (out_height > 0 and (out_height - 1) * self.stride_h + PH >
H_padded) or \
       (out_width > 0 and (out_width - 1) * self.stride_w + PW >
W_padded):
        raise ValueError(f"Patch size ({PH},{PW}) with stride
({self.stride_h},{self.stride_w}) exceeds padded input
({H_padded},{W_padded}) for output ({out_height},{out_width})")

    patches = np.lib.stride_tricks.as_strided(X_padded,
shape=output_shape_strided, strides=output_strides)
    output = np.max(patches, axis=(4, 5))
    return output
```

2.3.1.3. ManualAveragePooling2DLayer

Algoritma forward propagation pada ManualAveragePooling2DLayer melakukan operasi average pooling untuk mengurangi dimensi spasial dengan menghitung rata-rata dalam setiap jendela pooling. Input X_batch memiliki bentuk (N, H_in, W_in, C_in). Algoritma menghitung dimensi output (out_height dan out_width) berdasarkan ukuran pool (PH, PW), stride (stride_h, stride_w), dan jenis padding. Untuk padding 'same', input dipadding dengan nol; untuk 'valid', tidak ada padding. Input diproses menggunakan np.lib.stride_tricks.as_strided untuk membentuk patch dengan bentuk (N, out_height, out_width, C_in, PH, PW). Rata-rata dihitung pada sumbu (4, 5) (dimensi pool) untuk setiap patch, menghasilkan output dengan bentuk (N, out_height, out_width, C_in). Algoritma memastikan dimensi output valid dan menangani padding dengan tepat untuk menjaga konsistensi.

```
def forward(self, X_batch):  # X_batch shape: (N, H_in, W_in, C_in)
    N, H_in, W_in, C_in = X_batch.shape
    PH, PW = self.pool_size

    if self.padding == 'same':
        out_height = int(np.ceil(float(H_in) /
float(self.stride_h)))
        out_width = int(np.ceil(float(W_in) /
float(self.stride_w)))
        pad_h_total = max((out_height - 1) * self.stride_h + PH -
H_in, 0)
        pad_w_total = max((out_width - 1) * self.stride_w + PW -
W_in, 0)
        pad_top = pad_h_total // 2; pad_bottom = pad_h_total -
pad_top
        pad_left = pad_w_total // 2; pad_right = pad_w_total -
pad_left
        X_padded = np.pad(X_batch,
((0,0),(pad_top,pad_bottom),(pad_left,pad_right),(0,0)),
                          mode='constant', constant_values=0.0)
    elif self.padding == 'valid':
        out_height = (H_in - PH) // self.stride_h + 1
        out_width = (W_in - PW) // self.stride_w + 1
        X_padded = X_batch
    else:
        raise ValueError(f"Unsupported padding: {self.padding}")

    if out_height <= 0 or out_width <= 0:
        raise ValueError(f"Invalid output dims: H_out={out_height},
W_out={out_width} for H_in={H_in}, W_in={W_in}, Pool=({PH},{PW}),
Stride=({self.stride_h},{self.stride_w})")

    S_N, S_H, S_W, S_C = X_padded.strides
    output_shape_strided = (N, out_height, out_width, C_in, PH, PW)
    output_strides = (S_N, self.stride_h * S_H, self.stride_w *
S_W, S_C, S_H, S_W)
```

```
    H_padded, W_padded = X_padded.shape[1:3]
    if (out_height > 0 and (out_height - 1) * self.stride_h + PH >
H_padded) or \
       (out_width > 0 and (out_width - 1) * self.stride_w + PW >
W_padded):
        raise ValueError(f"Patch size ({PH},{PW}) with stride
({self.stride_h},{self.stride_w}) exceeds padded input
({H_padded},{W_padded}) for output ({out_height},{out_width})")

    patches = np.lib.stride_tricks.as_strided(X_padded,
shape=output_shape_strided, strides=output_strides)
    output = np.mean(patches, axis=(4, 5))
    return output
```

2.3.1.4. ManualFlattenLayer

Algoritma forward propagation pada ManualFlattenLayer meratakan input multi-dimensi menjadi vektor 2D untuk mempersiapkan data bagi lapisan berikutnya, seperti lapisan dense. Input X_batch memiliki bentuk (N, H, W, C), di mana N adalah jumlah sampel, dan H, W, C adalah dimensi spasial dan kanal. Algoritma menggunakan metode reshape untuk mengubah input menjadi bentuk (N, H*W*C), dengan mempertahankan jumlah sampel N dan meratakan semua dimensi lainnya menjadi satu dimensi. Proses ini sederhana dan tidak melibatkan operasi komputasi tambahan seperti konvolusi atau pooling.

```
def forward(self, X_batch):  # X_batch shape: (N, H, W, C)
    N = X_batch.shape[0]
    return X_batch.reshape(N, -1)  # Reshape to (N, H*W*C)
```

2.3.1.5. ManualGlobalAveragePooling2DLayer

Algoritma forward propagation pada ManualGlobalAveragePooling2DLayer melakukan operasi global average pooling untuk merangkum informasi spasial dari input. Input X_batch memiliki bentuk (N, H, W, C). Algoritma menghitung rata-rata pada sumbu spasial (1, 2) (yaitu, tinggi H dan lebar W) untuk setiap sampel dan kanal, menghasilkan output dengan bentuk (N, C). Operasi ini secara efektif mengurangi dimensi spasial menjadi satu nilai rata-rata per kanal, yang berguna untuk mengurangi kompleksitas sebelum lapisan dense.

```
def forward(self, X_batch):  # X_batch shape: (N, H, W, C)
    return np.mean(X_batch, axis=(1, 2))  # Result shape (N, C)
```

2.3.1.6. ManualDenseLayer

Algoritma forward propagation pada ManualDenseLayer melakukan transformasi linier diikuti oleh fungsi aktivasi opsional. Input X_batch memiliki bentuk (N, input_features). Pertama, algoritma melakukan operasi dot product antara X_batch dan matriks bobot weights (bentuk: (input_features, output_features)), menghasilkan matriks sementara. Vektor bias bias (bentuk: (output_features,)) ditambahkan ke hasil dot product. Jika fungsi aktivasi activation_fn (misalnya, ReLU atau softmax) ditentukan, algoritma menerapkannya pada hasil untuk memperkenalkan non-linearitas. Output akhir memiliki bentuk (N, output_features).

```
def forward(self, X_batch):  # X_batch shape: (N, input_features)
    output = np.dot(X_batch, self.weights) + self.bias
    if self.activation_fn:
        output = self.activation_fn(output)
    return output
```

2.3.1.7. CNNFromScratch

Algoritma forward propagation pada CNNFromScratch mengoordinasikan pemrosesan input melalui semua lapisan dalam arsitektur CNN. Input X_batch diteruskan secara berurutan melalui setiap elemen dalam daftar self.layers, yang dapat berupa objek lapisan manual (seperti ManualConv2DLayer, ManualMaxPooling2DLayer, dll.) atau fungsi aktivasi (seperti ReLU atau softmax). Untuk setiap lapisan, algoritma memeriksa apakah elemen tersebut adalah fungsi yang dapat dipanggil (yaitu, fungsi aktivasi) atau objek dengan metode forward. Jika itu fungsi, input langsung diteruskan ke fungsi tersebut; jika itu objek lapisan, metode forward dari lapisan tersebut dipanggil. Output dari satu lapisan menjadi input untuk lapisan berikutnya, hingga menghasilkan output akhir model. Algoritma memastikan bahwa setiap lapisan ditangani dengan benar, dengan validasi untuk jenis lapisan yang tidak didukung.

```
def predict(self, X_batch):
    output = X_batch
    for layer_idx, layer in enumerate(self.layers):
        if callable(layer) and not isinstance(layer,
(ManualConv2DLayer, ManualMaxPooling2DLayer,
ManualAveragePooling2DLayer, ManualFlattenLayer,
ManualGlobalAveragePooling2DLayer, ManualDenseLayer)):
            output = layer(output)
        elif hasattr(layer, 'forward'):
            output = layer.forward(output)
```

```
        else:
            raise TypeError(f"Layer {layer_idx} is of unhandled
type: {type(layer)}")
    return output
```

2.3.2. RNN

2.3.1.8. ManualEmbeddingLayer

Algoritma forward propagation pada ManualEmbeddingLayer berfungsi untuk mengubah indeks token input menjadi representasi vektor padat (embedding). Inputnya adalah X_batch yang berupa array integer dengan shape (N, seq_len), dimana N adalah jumlah sampel dan seq_len adalah panjang urutan token. Proses forward cukup melakukan indexing langsung pada embedding_matrix menggunakan X_batch sebagai indeks. Output berupa tensor shape (N, seq_len, embed_dim) yang merupakan embedding dari setiap token input.

```
class ManualEmbeddingLayer:
    def __init__(self, embedding_matrix):
        self.embedding_matrix = embedding_matrix

    def forward(self, X_batch):
        return self.embedding_matrix[X_batch]
```

2.3.1.9. ManualDenseLayer

ManualDenseLayer merepresentasikan layer Dense (fully connected) manual dengan bobot dan bias yang sudah diberikan. Algoritma forward-nya menghitung output linear X_batch @ weights + bias. Jika ada fungsi aktivasi (activation_fn), output dari operasi linear tersebut akan dilewatkan ke fungsi aktivasi. Outputnya berukuran (N, units) dimana units adalah jumlah neuron layer dense.

```
class ManualDenseLayer:
    def __init__(self, weights, bias, activation_fn=None):
        self.weights = weights
        self.bias = bias
        self.activation_fn = activation_fn

    def forward(self, X_batch):
        output = np.dot(X_batch, self.weights) + self.bias
        if self.activation_fn:
            output = self.activation_fn(output)
        return output
```

2.3.1.10. ManualRNNCell

ManualRNNCell merepresentasikan satu unit sel RNN sederhana dengan parameter bobot input (Wx), bobot hidden state (Wh), dan bias (b). Pada forward propagation, sel menerima input pada waktu ke-t (xt) dan state tersembunyi sebelumnya (h_prev). Kemudian menghitung h_next = activation(xt @ Wx + h_prev @ Wh + b), yang menghasilkan state tersembunyi terbaru. Fungsi aktivasi default adalah tanh, tapi bisa diganti.

```
def forward(self, X_batch):  # X_batch shape: (N, H_in, W_in, C_in)
    N, H_in, W_in, C_in = X_batch.shape
    KH, KW, _, C_out = self.kernel.shape

    if self.padding == 'same':
        out_height = int(np.ceil(float(H_in) /
float(self.stride_h)))
        out_width = int(np.ceil(float(W_in) /
float(self.stride_w)))

        pad_h_total = max((out_height - 1) * self.stride_h + KH -
H_in, 0)
        pad_w_total = max((out_width - 1) * self.stride_w + KW -
W_in, 0)

        pad_top = pad_h_total // 2
        pad_bottom = pad_h_total - pad_top
        pad_left = pad_w_total // 2
        pad_right = pad_w_total - pad_left

        X_padded = np.pad(X_batch, ((0, 0), (pad_top, pad_bottom),
(pad_left, pad_right), (0, 0)), mode='constant',
constant_values=0.0)
    elif self.padding == 'valid':
        out_height = (H_in - KH) // self.stride_h + 1
        out_width = (W_in - KW) // self.stride_w + 1
        X_padded = X_batch
    else:
        raise ValueError(f"Unsupported padding type:
{self.padding}")

    if out_height <= 0 or out_width <= 0:
        raise ValueError(f"Output dimensions non-positive:
H={out_height}, W={out_width}. Input: H_in={H_in}, W_in={W_in},
Kernel: KH={KH}, KW={KW}, Stride:
({self.stride_h},{self.stride_w}), Padding: {self.padding}")

    shape_col = (N, out_height, out_width, KH, KW, C_in)

    strides_col = (
        X_padded.strides[0],
        self.stride_h * X_padded.strides[1],
        self.stride_w * X_padded.strides[2],
```

```
        X_padded.strides[1],
        X_padded.strides[2],
        X_padded.strides[3]
    )

    H_padded, W_padded = X_padded.shape[1:3]
    if (out_height > 0 and (out_height - 1) * self.stride_h + KH >
H_padded) or \
       (out_width > 0 and (out_width - 1) * self.stride_w + KW >
W_padded):
        raise ValueError(
            f"Calculated patch extent exceeds padded input
dimensions. "
            f"H_padded={H_padded}, W_padded={W_padded}. "
            f"out_height={out_height}, out_width={out_width}. "
            f"Required H: {(out_height - 1) * self.stride_h + KH},
"
            f"Required W: {(out_width - 1) * self.stride_w + KW}."
        )

    X_col = np.lib.stride_tricks.as_strided(X_padded,
shape=shape_col, strides=strides_col)

    X_col_reshaped = X_col.reshape(N * out_height * out_width, KH *
KW * C_in)
    kernel_reshaped = self.kernel.reshape(KH * KW * C_in, C_out)

    output_reshaped = np.dot(X_col_reshaped, kernel_reshaped)
    output = output_reshaped.reshape(N, out_height, out_width,
C_out)

    output += self.bias

    return output
```

2.3.1.11. ManualRNNLayer

ManualRNNLayer adalah lapisan RNN yang membungkus sekumpulan ManualRNNCell berjalan sepanjang *sequence* waktu input.

Input adalah X_embedded_batch berbentuk (N, seq_len, input_dim).

Algoritma forward mengiterasi timestep dari input, setiap langkah memanggil ManualRNNCell.forward dengan input timestep dan hidden state sebelumnya.

Jika return_sequences=True, outputnya adalah seluruh state tersembunyi tiap timestep (N, seq_len, units), jika False hanya state terakhir (N, units).

Jika go_backwards=True, iterasi dilakukan dari akhir ke awal.

```
class ManualRNNLayer:
    def __init__(self, units, Wx, Wh, b, activation=tanh,
return_sequences=False, go_backwards=False):
        self.cell = ManualRNNCell(Wx, Wh, b, activation)
        self.return_sequences = return_sequences
        self.go_backwards = go_backwards
        self.units = units

    def forward(self, X_embedded_batch):
        N, seq_len, input_dim = X_embedded_batch.shape
        h_t = np.zeros((N, self.units))
        outputs = []

        time_steps = range(seq_len)
        if self.go_backwards:
            time_steps = reversed(time_steps)

        for t in time_steps:
            xt = X_embedded_batch[:, t, :]
            h_t = self.cell.forward(xt, h_t)
            if self.return_sequences:
                outputs.append(h_t)

        if self.return_sequences:
            stacked_outputs = np.stack(outputs, axis=1)
            return stacked_outputs[:, ::-1, :] if self.go_backwards
else stacked_outputs
        else:
            return h_t
```

2.3.1.12. ManualBidirectionalLSTMWrapper

ManualBidirectionalRNNWrapper menggabungkan dua ManualRNNLayer, satu berjalan maju dan satu berjalan mundur (reverse sequence).

Pada forward propagation, input dilewatkan ke forward_rnn_layer dan juga input dibalik pada dimensi waktu dilewatkan ke backward_rnn_layer.

Output dari backward dibalik lagi agar urutannya sama dengan forward.

Hasil kedua arah digabungkan dengan cara konkatenasi pada dimensi fitur (merge_mode='concat').

Hanya mode concat yang diimplementasikan.

```
class ManualBidirectionalRNNWrapper:
    def __init__(self, forward_rnn_layer, backward_rnn_layer,
merge_mode='concat'):
        self.forward_rnn_layer = forward_rnn_layer
```

```
        self.backward_rnn_layer = backward_rnn_layer
        if merge_mode != 'concat':
            raise NotImplementedError("Only 'concat' merge mode is
implemented for manual Bidirectional RNN.")
        self.merge_mode = merge_mode

    def forward(self, X_embedded_batch):
        # Forward
        output_forward =
self.forward_rnn_layer.forward(X_embedded_batch)
        # Backward: reverse sequence on time axis for input, then
reverse output back
        X_reversed = X_embedded_batch[:, ::-1, :]
        output_backward =
self.backward_rnn_layer.forward(X_reversed)
        # If return_sequences, output shape (N, T, units), else (N,
units)
        if self.forward_rnn_layer.return_sequences:
            # Reverse backward output on time axis to match forward
            output_backward = output_backward[:, ::-1, :]
        return np.concatenate([output_forward, output_backward],
axis=-1)
```

2.3.1.13. SimpleRNNFromScratch

SimpleRNNFromScratch adalah implementasi manual dari model RNN yang mereplikasi arsitektur dan bobot model Keras. Kelas ini memiliki metode utama build_from_keras yang mengambil model Keras dan vectorizer opsional, kemudian mengekstrak bobot dan konfigurasi dari tiap layer Keras seperti Embedding, SimpleRNN, Bidirectional, dan Dense. Untuk Embedding, bobot embedding disimpan di ManualEmbeddingLayer. Untuk SimpleRNN, bobot input (Wx), bobot rekuren (Wh), bias (b), fungsi aktivasi (tanh atau sigmoid), dan opsi return_sequences dimuat ke dalam ManualRNNLayer. Jika layer bidirectional ditemukan, kelas ini membuat dua ManualRNNLayer—satu untuk arah maju dan satu untuk mundur—yang dibungkus oleh ManualBidirectionalRNNWrapper yang menangani pembalikan input dan output serta penggabungan hasil dengan np.concatenate pada axis terakhir (concat merge mode). Untuk layer dense, bobot dan bias dimasukkan ke ManualDenseLayer dengan fungsi aktivasi seperti relu atau softmax. Metode forward menjalankan input secara berurutan melalui semua layer manual yang telah dibuat, memanggil forward pada tiap layer untuk mendapatkan output intermediate sampai hasil akhir. Metode predict memastikan model sudah dibangun dengan build_from_keras sebelum melakukan inferensi menggunakan forward. Dengan struktur ini, SimpleRNNFromScratch menggabungkan proses load bobot, pengolahan input melalui lapisan

manual, dan penggabungan hasil dari bidirectional RNN, sehingga dapat meniru perilaku model Keras secara lengkap dalam kode manual.

```
class SimpleRNNFromScratch:
    def __init__(self):
        self.layers = []
        self.is_built = False
        self.vectorizer = None

    def build_from_keras(self, keras_model, vectorizer=None):
        # Allow vectorizer to be None for inference-only use
        self.vectorizer = vectorizer
        if vectorizer is not None:
            self.vocabulary_size = vectorizer.vocabulary_size()
        else:
            self.vocabulary_size = None

        for layer in keras_model.layers:
            layer_name = layer.__class__.__name__

            if layer_name == 'Embedding':
                embedding_weights = layer.get_weights()[0]

self.layers.append(ManualEmbeddingLayer(embedding_weights))

            elif layer_name == 'SimpleRNN':
                weights = layer.get_weights()
                Wx, Wh, b = weights[0], weights[1], weights[2]
                activation_fn = tanh if layer.activation.__name__
== 'tanh' else sigmoid
                return_sequences = layer.return_sequences

                self.layers.append(ManualRNNLayer(
                    units=layer.units,
                    Wx=Wx,
                    Wh=Wh,
                    b=b,
                    activation=activation_fn,
                    return_sequences=return_sequences
                ))

            elif layer_name == 'Bidirectional':
                forward_layer = layer.forward_layer
                backward_layer = layer.backward_layer

                fw_weights = forward_layer.get_weights()
                fw_Wx, fw_Wh, fw_b = fw_weights[0], fw_weights[1],
fw_weights[2]
                fw_activation_fn = tanh if
forward_layer.activation.__name__ == 'tanh' else sigmoid
                fw_return_sequences =
forward_layer.return_sequences

                bw_weights = backward_layer.get_weights()
                bw_Wx, bw_Wh, bw_b = bw_weights[0], bw_weights[1],
bw_weights[2]
                bw_activation_fn = tanh if
backward_layer.activation.__name__ == 'tanh' else sigmoid
                bw_return_sequences =
```

```
backward_layer.return_sequences

                self.layers.append(ManualBidirectionalRNNWrapper(
                    ManualRNNLayer(
                        units=forward_layer.units,
                        Wx=fw_Wx,
                        Wh=fw_Wh,
                        b=fw_b,
                        activation=fw_activation_fn,
                        return_sequences=fw_return_sequences
                    ),
                    ManualRNNLayer(
                        units=backward_layer.units,
                        Wx=bw_Wx,
                        Wh=bw_Wh,
                        b=bw_b,
                        activation=bw_activation_fn,
                        return_sequences=bw_return_sequences
                    )
                ))

            elif layer_name == 'Dense':
                weights, bias = layer.get_weights()
                activation_fn = None
                if layer.activation.__name__ == 'relu':
                    activation_fn = relu
                elif layer.activation.__name__ == 'softmax':
                    activation_fn = softmax

                self.layers.append(ManualDenseLayer(weights, bias,
activation_fn))

        self.is_built = True

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

    def predict(self, x):
        if not self.is_built:
            raise ValueError("Model not built. Call
build_from_keras first.")
        return self.forward(x)
```

2.3.3. LSTM

2.3.1.14. ManualEmbeddingLayer

Algoritma forward propagation pada ManualEmbeddingLayer berfungsi untuk mengubah urutan token input menjadi representasi vektor padat (dense). Proses ini dimulai dengan menerima X_tokens_batch yang memiliki bentuk (N, seq_len), di mana N adalah jumlah sampel dalam batch dan seq_len adalah panjang urutan token. Algoritma kemudian melakukan operasi lookup sederhana menggunakan X_tokens_batch sebagai indeks pada embedding_matrix

yang telah diinisialisasi. Setiap token dalam urutan input akan dipetakan ke vektor embedding yang sesuai dari embedding_matrix. Hasilnya adalah output dengan bentuk (N, seq_len, embed_dim), di mana embed_dim adalah dimensi vektor embedding.
Algoritma backward propagation pada ManualEmbeddingLayer bertujuan untuk menghitung gradien yang akan digunakan untuk memperbarui embedding_matrix. Ini menerima dL_dOutput (gradien dari lapisan berikutnya) yang memiliki bentuk (N, seq_len, embed_dim) dan X_tokens_batch. Algoritma menginisialisasi gradien untuk embedding_matrix dengan nol. Kemudian, untuk setiap sampel dalam batch dan setiap langkah waktu dalam urutan, ia mengidentifikasi indeks token (token_idx) dari X_tokens_batch. Gradien dari dL_dOutput pada posisi (n, t) kemudian diakumulasikan ke baris yang sesuai di self.grads['embedding_matrix'] yang diindeks oleh token_idx. Ini memastikan bahwa hanya embedding yang benar-benar digunakan dalam forward pass yang diperbarui gradiennya.

```
class ManualEmbeddingLayer:
    def __init__(self, embedding_matrix):
        self.embedding_matrix = embedding_matrix
        self.params = {'embedding_matrix': self.embedding_matrix}
        self.grads = {'embedding_matrix':
np.zeros_like(self.embedding_matrix)}

    def forward(self, X_tokens_batch):
        return self.embedding_matrix[X_tokens_batch]

    def backward(self, dL_dOutput, X_tokens_batch):
        N, seq_len = X_tokens_batch.shape
        vocab_size, embed_dim = self.embedding_matrix.shape
        self.grads['embedding_matrix'].fill(0)
        for n in range(N):
            for t in range(seq_len):
                token_idx = X_tokens_batch[n, t]
                if token_idx < vocab_size:
                    self.grads['embedding_matrix'][token_idx] +=
dL_dOutput[n, t]
        return None
```

2.3.1.15. ManualDenseLayer

Algoritma forward propagation pada ManualDenseLayer melakukan transformasi linier pada input, diikuti oleh penerapan fungsi aktivasi opsional. Input X_batch memiliki bentuk (N, input_features). Pertama, algoritma melakukan operasi dot product antara X_batch dan matriks bobot self.weights (bentuk: (input_features, output_features)). Vektor bias self.bias (bentuk: (output_features,)) kemudian ditambahkan ke hasil dot product tersebut. Jika fungsi aktivasi self.activation_fn (seperti relu atau softmax) telah ditentukan, algoritma akan menerapkannya pada hasil

untuk memperkenalkan non-linearitas. Input dan output dari langkah forward disimpan dalam self.input_ dan self.output_ masing-masing untuk digunakan dalam perhitungan backward. Output akhir memiliki bentuk (N, output_features).

Algoritma backward propagation pada ManualDenseLayer menghitung gradien kerugian terhadap bobot, bias, dan input lapisan. Ini menerima dL_dOutput (gradien dari lapisan berikutnya) yang memiliki bentuk yang sama dengan output lapisan. Pertama, dL_dOutput diturunkan kembali melalui fungsi aktivasi (jika ada) untuk mendapatkan dL_dZ (gradien kerugian terhadap output linier/pre-aktivasi). Jika fungsi aktivasi adalah softmax, dL_dZ sama dengan dL_dOutput. Untuk relu atau sigmoid, dL_dZ dihitung dengan mengalikan dL_dOutput dengan turunan dari fungsi aktivasi pada self.output_. Selanjutnya, gradien untuk bobot (self.grads['weights']) dihitung sebagai dot product antara transpose dari input (self.input_.T) dan dL_dZ. Gradien untuk bias (self.grads['bias']) dihitung sebagai jumlah dL_dZ sepanjang sumbu sampel. Terakhir, gradien terhadap input lapisan (dL_dInput) dihitung sebagai dot product antara dL_dZ dan transpose dari bobot (self.weights.T), yang kemudian dikembalikan untuk propagasi mundur ke lapisan sebelumnya.

```
class ManualDenseLayer:
    def __init__(self, weights, bias, activation_fn=None):
        self.weights = weights
        self.bias = bias
        self.activation_fn = activation_fn
        self.params = {'weights': self.weights, 'bias': self.bias}
        self.grads = {'weights': np.zeros_like(self.weights),
'bias': np.zeros_like(self.bias)}

    def forward(self, X_batch):
        output = np.dot(X_batch, self.weights) + self.bias
        self.input_ = X_batch
        if self.activation_fn:
            output = self.activation_fn(output)
        self.output_ = output
        return output

    def backward(self, dL_dOutput):
        if self.activation_fn == softmax:
            dL_dZ = dL_dOutput
        elif self.activation_fn == relu:
            dL_dZ = dL_dOutput * d_relu_dz(self.output_)
        elif self.activation_fn == sigmoid:
            dL_dZ = dL_dOutput * d_sigmoid_dz(self.output_)
        else:
            dL_dZ = dL_dOutput
        self.grads['weights'] = np.dot(self.input_.T, dL_dZ)
```

```
        self.grads['bias'] = np.sum(dL_dZ, axis=0)
        dL_dInput = np.dot(dL_dZ, self.weights.T)
        return dL_dInput
```

2.3.1.16. ManualLSTMCell

Algoritma forward propagation pada ManualLSTMCell memproses satu langkah waktu dalam urutan untuk menghitung hidden state (h_next) dan cell state (c_next) berikutnya. Ini menerima input xt pada langkah waktu saat ini, hidden state sebelumnya h_prev, dan cell state sebelumnya c_prev. Pertama, empat kombinasi linier terpisah dihitung untuk masing-masing gerbang (input i_lin, forget f_lin, output o_lin) dan candidate memory c_tilde_lin, dengan mengalikan xt dan h_prev dengan bobot Wx dan Wh yang sesuai, lalu menambahkan bias b. Selanjutnya, gerbang i, f, dan o dihitung dengan menerapkan fungsi aktivasi rekuren (biasanya sigmoid) pada output linier masing-masing. c_tilde (candidate memory) dihitung dengan menerapkan fungsi aktivasi (biasanya tanh) pada c_tilde_lin. Cell state berikutnya c_next kemudian diperbarui menggunakan rumus gerbang forget * c_prev + input * c_tilde. Akhirnya, hidden state berikutnya h_next dihitung sebagai output * activation(c_next). Semua nilai yang relevan dari forward pass disimpan dalam cache untuk digunakan dalam backpropagation.

Algoritma backward propagation pada ManualLSTMCell menghitung gradien kerugian terhadap input xt, hidden state sebelumnya h_prev, cell state sebelumnya c_prev, dan semua parameter sel (bobot dan bias). Ini menerima gradien dh_next dan dc_next_from_tplus1 dari langkah waktu berikutnya, serta cache dari forward pass yang sesuai. Proses dimulai dengan menghitung do_lin_pre_act (gradien untuk gerbang output sebelum aktivasi), dc_next (gradien untuk cell state saat ini), df_lin_pre_act (gradien untuk gerbang forget sebelum aktivasi), dc_prev (gradien untuk cell state sebelumnya), di_lin_pre_act (gradien untuk gerbang input sebelum aktivasi), dan dc_tilde_lin_pre_act (gradien untuk candidate memory sebelum aktivasi). Gradien ini kemudian dikalikan dengan turunan fungsi aktivasi yang sesuai (d_sigmoid_dz untuk gerbang dan d_tanh_dz untuk c_tilde) untuk mendapatkan di_lin, df_lin, do_lin, dan dc_tilde_lin. Selanjutnya, gradien untuk setiap bobot (Wx_i, Wh_i, dll.) dan bias (b_i, dll.) dihitung dengan melakukan dot product atau penjumlahan yang sesuai, dan diakumulasikan dalam self.grads. Terakhir, gradien terhadap input dx_t, hidden state sebelumnya dh_prev, dan cell

state sebelumnya dc_prev dihitung dan dikembalikan untuk propagasi mundur lebih lanjut.

```
class ManualLSTMCell:
    def __init__(self, Wx, Wh, b, activation=tanh,
recurrent_activation=sigmoid):
        units = Wh.shape[0]
        self.Wx_i, self.Wx_f, self.Wx_c, self.Wx_o = np.split(Wx,
4, axis=1)
        self.Wh_i, self.Wh_f, self.Wh_c, self.Wh_o = np.split(Wh,
4, axis=1)
        self.b_i, self.b_f, self.b_c, self.b_o = np.split(b, 4)
        self.activation = activation
        self.recurrent_activation = recurrent_activation
        self.params = {
            'Wx_i': self.Wx_i, 'Wx_f': self.Wx_f, 'Wx_c':
self.Wx_c, 'Wx_o': self.Wx_o,
            'Wh_i': self.Wh_i, 'Wh_f': self.Wh_f, 'Wh_c':
self.Wh_c, 'Wh_o': self.Wh_o,
            'b_i': self.b_i, 'b_f': self.b_f, 'b_c': self.b_c,
'b_o': self.b_o
        }
        self.grads = {k: np.zeros_like(v) for k, v in
self.params.items()}

    def forward(self, xt, h_prev, c_prev):
        i_lin = np.dot(xt, self.Wx_i) + np.dot(h_prev, self.Wh_i) +
self.b_i
        f_lin = np.dot(xt, self.Wx_f) + np.dot(h_prev, self.Wh_f) +
self.b_f
        o_lin = np.dot(xt, self.Wx_o) + np.dot(h_prev, self.Wh_o) +
self.b_o
        c_tilde_lin = np.dot(xt, self.Wx_c) + np.dot(h_prev,
self.Wh_c) + self.b_c
        i = self.recurrent_activation(i_lin)
        f = self.recurrent_activation(f_lin)
        o = self.recurrent_activation(o_lin)
        c_tilde = self.activation(c_tilde_lin)
        c_next = f * c_prev + i * c_tilde
        h_next = o * self.activation(c_next)
        cache = (xt, h_prev, c_prev, i_lin, f_lin, o_lin,
c_tilde_lin, i, f, o, c_tilde, c_next, h_next)
        return h_next, c_next, cache

    def backward(self, dh_next, dc_next_from_tplus1, cache):
        xt, h_prev, c_prev, i_lin, f_lin, o_lin, c_tilde_lin, i, f,
o, c_tilde, c_next, h_next = cache
        act_c_next = self.activation(c_next)
        d_act_c_next_dz = d_tanh_dz(act_c_next) if self.activation
== tanh else None
        do_lin_pre_act = dh_next * act_c_next
        dc_next = dh_next * o * d_act_c_next_dz
        dc_next += dc_next_from_tplus1
        df_lin_pre_act = dc_next * c_prev
        dc_prev = dc_next * f
```

```
        di_lin_pre_act = dc_next * c_tilde
        dc_tilde_lin_pre_act = dc_next * i
        di_lin = di_lin_pre_act * d_sigmoid_dz(i)
        df_lin = df_lin_pre_act * d_sigmoid_dz(f)
        do_lin = do_lin_pre_act * d_sigmoid_dz(o)
        dc_tilde_lin = dc_tilde_lin_pre_act * d_tanh_dz(c_tilde)
        self.grads['Wx_i'] += np.dot(xt.T, di_lin)
        self.grads['Wh_i'] += np.dot(h_prev.T, di_lin)
        self.grads['b_i'] += np.sum(di_lin, axis=0)
        self.grads['Wx_f'] += np.dot(xt.T, df_lin)
        self.grads['Wh_f'] += np.dot(h_prev.T, df_lin)
        self.grads['b_f'] += np.sum(df_lin, axis=0)
        self.grads['Wx_o'] += np.dot(xt.T, do_lin)
        self.grads['Wh_o'] += np.dot(h_prev.T, do_lin)
        self.grads['b_o'] += np.sum(do_lin, axis=0)
        self.grads['Wx_c'] += np.dot(xt.T, dc_tilde_lin)
        self.grads['Wh_c'] += np.dot(h_prev.T, dc_tilde_lin)
        self.grads['b_c'] += np.sum(dc_tilde_lin, axis=0)
        dx_t = np.dot(di_lin, self.Wx_i.T) + np.dot(df_lin,
self.Wx_f.T) + \
               np.dot(do_lin, self.Wx_o.T) + np.dot(dc_tilde_lin,
self.Wx_c.T)
        dh_prev = np.dot(di_lin, self.Wh_i.T) + np.dot(df_lin,
self.Wh_f.T) + \
                  np.dot(do_lin, self.Wh_o.T) +
np.dot(dc_tilde_lin, self.Wh_c.T)
        return dx_t, dh_prev, dc_prev

    def reset_grads(self):
        for k in self.grads:
            self.grads[k].fill(0)
```

2.3.1.17. ManualLSTMLayer

Algoritma forward propagation pada ManualLSTMLayer memproses seluruh urutan input secara sekuensial, langkah demi langkah. Ini menerima X_embedded_batch yang memiliki bentuk (N, seq_len, embed_dim). Hidden state (h_t) dan cell state (c_t) awal diinisialisasi dengan nol. Lapisan ini akan mengiterasi melalui setiap langkah waktu dalam urutan, baik dalam urutan normal atau terbalik jika self.go_backwards adalah True. Untuk setiap langkah waktu t, input xt diekstrak dari X_embedded_batch. ManualLSTMCell dipanggil dengan xt, h_t sebelumnya, dan c_t sebelumnya untuk menghitung h_t dan c_t yang baru, bersama dengan cache yang disimpan. Output hidden state (h_t) dari setiap langkah waktu ditambahkan ke daftar outputs_h, dan cache disimpan di cell_caches_ untuk backpropagation. Jika self.return_sequences adalah True, semua hidden state yang terkumpul akan ditumpuk menjadi satu tensor output; jika tidak, hanya hidden state terakhir yang dikembalikan.

Algoritma backward propagation pada ManualLSTMLayer melakukan backpropagation through time (BPTT). Ini menerima dL_dOutput (gradien kerugian dari lapisan berikutnya) yang dapat berupa gradien dari output terakhir atau dari setiap langkah waktu, tergantung pada return_sequences. Gradien dL_dX_embedded_batch diinisialisasi dengan nol, dan dh_next_bptt serta dc_next_bptt (gradien yang akan diakumulasikan melalui waktu) juga diinisialisasi dengan nol. Lapisan ini mengiterasi mundur melalui langkah waktu. Untuk setiap langkah waktu t_proc (dalam urutan pemrosesan terbalik), cache yang sesuai diambil dari self.cell_caches_. current_dh dihitung dengan menjumlahkan dL_dOutput dari langkah waktu ini (jika return_sequences) dan dh_next_bptt yang diakumulasikan. ManualLSTMCell.backward kemudian dipanggil dengan current_dh, dc_next_bptt, dan cache untuk mendapatkan dx_t, dh_prev_bptt, dan dc_prev_bptt. dx_t (gradien terhadap input X_embedded_batch pada langkah waktu ini) disimpan, dan dh_prev_bptt serta dc_prev_bptt diperbarui untuk iterasi berikutnya. Proses ini secara efektif menyebarkan gradien mundur melalui waktu.

```
class ManualLSTMLayer:
    def __init__(self, units, Wx, Wh, b, activation=tanh,
recurrent_activation=sigmoid,
                 return_sequences=False, go_backwards=False):
        self.cell = ManualLSTMCell(Wx, Wh, b, activation,
recurrent_activation)
        self.return_sequences = return_sequences
        self.go_backwards = go_backwards
        self.units = units
        self.params = self.cell.params
        self.grads = self.cell.grads

    def forward(self, X_embedded_batch):
        N, seq_len, _ = X_embedded_batch.shape
        h_t = np.zeros((N, self.units))
        c_t = np.zeros((N, self.units))
        outputs_h = []
        cell_caches = []
        time_steps = range(seq_len)
        if self.go_backwards:
            time_steps = reversed(time_steps)
        current_X_sequence = X_embedded_batch
        if self.go_backwards:
            current_X_sequence = X_embedded_batch[:, ::-1, :]
        for t_idx_in_processing_order, t_actual_from_input in
enumerate(time_steps):
            xt = current_X_sequence[:, t_idx_in_processing_order,
:]
            h_t, c_t, cache = self.cell.forward(xt, h_t, c_t)
            outputs_h.append(h_t)
            cell_caches.append(cache)
        self.cell_caches_ = cell_caches
        if self.return_sequences:
```

```
            stacked_outputs = np.stack(outputs_h, axis=1)
            final_output_sequence = stacked_outputs[:, ::-1, :] if
self.go_backwards else stacked_outputs
            self.last_output_ = final_output_sequence
            return final_output_sequence
        else:
            self.last_output_ = h_t
            return h_t

    def backward(self, dL_dOutput):
        N, seq_len, input_dim = self.cell_caches_[0][0].shape[0],
len(self.cell_caches_), self.cell_caches_[0][0].shape[1]
        dL_dX_embedded_batch = np.zeros((N, seq_len, input_dim))
        dh_next_bptt = np.zeros((N, self.units))
        dc_next_bptt = np.zeros((N, self.units))
        self.cell.reset_grads()
        if self.return_sequences:
            dL_dOutput_processed_order = dL_dOutput[:, ::-1, :] if
self.go_backwards else dL_dOutput
        else:
            dL_dOutput_processed_order = np.zeros((N, seq_len,
self.units))
            dL_dOutput_processed_order[:, -1, :] = dL_dOutput
        time_steps_proc_order = range(seq_len)
        for t_proc in reversed(time_steps_proc_order):
            cache_t = self.cell_caches_[t_proc]
            current_dh = dL_dOutput_processed_order[:, t_proc, :] +
dh_next_bptt
            dx_t, dh_prev_bptt, dc_prev_bptt =
self.cell.backward(current_dh, dc_next_bptt, cache_t)
            actual_time_idx = (seq_len - 1 - t_proc) if
self.go_backwards else t_proc
            dL_dX_embedded_batch[:, actual_time_idx, :] = dx_t
            dh_next_bptt = dh_prev_bptt
            dc_next_bptt = dc_prev_bptt
        return dL_dX_embedded_batch
```

2.3.1.18. ManualBidirectionalLSTMWrapper

Algoritma forward propagation pada ManualBidirectionalLSTMWrapper menggabungkan output dari dua lapisan LSTM: satu yang memproses urutan maju (forward_lstm_layer) dan satu yang memproses urutan mundur (backward_lstm_layer). Ini menerima X_embedded_batch sebagai input. Pertama, input dilewatkan ke forward_lstm_layer.forward() untuk mendapatkan output maju. Kemudian, input yang sama dilewatkan ke backward_lstm_layer.forward() untuk mendapatkan output mundur. Kedua output ini, yang memiliki bentuk (N, seq_len, units) atau (N, units) tergantung pada return_sequences, kemudian digabungkan (np.concatenate) sepanjang sumbu terakhir (axis=-1). Mode penggabungan saat ini hanya mendukung 'concat', yang berarti output

dari lapisan maju dan mundur digabungkan secara berdampingan. Output gabungan ini disimpan dalam self.last_output_ dan dikembalikan.

Algoritma backward propagation pada ManualBidirectionalLSTMWrapper mendistribusikan gradien kembali ke lapisan LSTM maju dan mundur. Ini menerima dL_dOutput_concat (gradien kerugian dari lapisan berikutnya) yang merupakan gradien untuk output gabungan. Gradien dL_dOutput_concat kemudian dipisah menjadi dua bagian yang sama (dL_dOutput_fw dan dL_dOutput_bw) di sepanjang sumbu terakhir, yang sesuai dengan gradien untuk lapisan maju dan mundur. Penting untuk memastikan pemisahan ini benar, terutama jika jumlah unit tidak merata. Setelah pemisahan, forward_lstm_layer.backward() dipanggil dengan dL_dOutput_fw, dan backward_lstm_layer.backward() dipanggil dengan dL_dOutput_bw. Gradien input yang dihasilkan oleh kedua panggilan backward ini (dL_dX_fw dan dL_dX_bw) kemudian dijumlahkan, karena kedua lapisan menerima input yang sama pada forward pass. Hasil penjumlahan ini adalah gradien terhadap input asli lapisan bidirectional dan dikembalikan untuk propagasi mundur ke lapisan sebelumnya.

```
class ManualBidirectionalLSTMWrapper:
    def __init__(self, forward_lstm_layer, backward_lstm_layer,
merge_mode='concat'):
        self.forward_lstm_layer = forward_lstm_layer
        self.backward_lstm_layer = backward_lstm_layer
        if merge_mode != 'concat':
            raise NotImplementedError("Only 'concat' merge mode
implemented.")
        self.merge_mode = merge_mode
        self.params = {f'fw_{k}': v for k, v in
self.forward_lstm_layer.params.items()}
        self.params.update({f'bw_{k}': v for k, v in
self.backward_lstm_layer.params.items()})
        self.grads = {f'fw_{k}': v for k, v in
self.forward_lstm_layer.grads.items()}
        self.grads.update({f'bw_{k}': v for k, v in
self.backward_lstm_layer.grads.items()})

    def forward(self, X_embedded_batch):
        output_forward =
self.forward_lstm_layer.forward(X_embedded_batch)
        output_backward =
self.backward_lstm_layer.forward(X_embedded_batch)
        concatenated_output = np.concatenate([output_forward,
output_backward], axis=-1)
        self.last_output_ = concatenated_output
        return concatenated_output

    def backward(self, dL_dOutput_concat):
        units_fw = self.forward_lstm_layer.units
```

```
        dL_dOutput_fw, dL_dOutput_bw = np.split(dL_dOutput_concat,
2, axis=-1)
        if dL_dOutput_fw.shape[-1] != units_fw:
            dL_dOutput_fw = dL_dOutput_concat[..., :units_fw]
            dL_dOutput_bw = dL_dOutput_concat[..., units_fw:]
        dL_dX_fw = self.forward_lstm_layer.backward(dL_dOutput_fw)
        dL_dX_bw = self.backward_lstm_layer.backward(dL_dOutput_bw)
```

2.3.1.19. LSTMFromScratch

Algoritma forward propagation pada LSTMFromScratch mengoordinasikan pemrosesan input melalui semua lapisan yang membentuk arsitektur LSTM. Ini menerima X_batch_tokens sebagai input awal. Input ini kemudian diteruskan secara berurutan melalui setiap elemen dalam daftar self.layers. Setiap elemen dalam self.layers diharapkan adalah objek lapisan manual yang memiliki metode forward(). Output dari satu lapisan menjadi input untuk lapisan berikutnya, hingga semua lapisan telah diproses. Hasil akhirnya adalah output prediksi model.

Algoritma backward propagation pada LSTMFromScratch mengoordinasikan propagasi mundur gradien melalui semua lapisan model, dimulai dari lapisan output. Ini menerima d_loss (gradien kerugian dari fungsi kerugian) sebagai input awal. Gradien ini kemudian diteruskan mundur secara berurutan melalui setiap lapisan dalam daftar self.layers, tetapi dalam urutan terbalik. Setiap lapisan manual diharapkan memiliki metode backward() yang menerima gradien dari lapisan berikutnya dan mengembalikan gradien untuk lapisan sebelumnya. Output dari metode backward() satu lapisan menjadi input untuk metode backward() lapisan sebelumnya. Proses ini berlanjut hingga gradien mencapai lapisan input pertama, memungkinkan semua parameter model untuk diperbarui.

```
class LSTMFromScratch:
    def __init__(self):
        self.layers = []
        self.keras_model = None

    def add_keras_layer(self, keras_layer):
        if isinstance(keras_layer, layers.Embedding):
            embedding_matrix = keras_layer.get_weights()[0]

self.layers.append(ManualEmbeddingLayer(embedding_matrix))
        elif isinstance(keras_layer, layers.LSTM):
            weights = keras_layer.get_weights()
```

```
            Wx, Wh, b = weights[0], weights[1], weights[2]
            units = keras_layer.units
            act_cfg = keras_layer.get_config()['activation']
            rec_act_cfg =
keras_layer.get_config()['recurrent_activation']
            activation_fn = globals().get(act_cfg, tanh)
            recurrent_activation_fn = globals().get(rec_act_cfg,
sigmoid)
            return_sequences = keras_layer.return_sequences
            go_backwards = keras_layer.go_backwards
            self.layers.append(ManualLSTMLayer(units, Wx, Wh, b,
activation_fn,

recurrent_activation_fn, return_sequences, go_backwards))
        elif isinstance(keras_layer, layers.Bidirectional):
            fw_keras_layer = keras_layer.forward_layer
            fw_weights = fw_keras_layer.get_weights()
            fw_Wx, fw_Wh, fw_b = fw_weights[0], fw_weights[1],
fw_weights[2]
            fw_units = fw_keras_layer.units
            fw_act_cfg = fw_keras_layer.get_config()['activation']
            fw_rec_act_cfg =
fw_keras_layer.get_config()['recurrent_activation']
            fw_act_fn = globals().get(fw_act_cfg, tanh)
            fw_rec_act_fn = globals().get(fw_rec_act_cfg, sigmoid)
            fw_ret_seq = fw_keras_layer.return_sequences
            manual_forward_lstm = ManualLSTMLayer(fw_units, fw_Wx,
fw_Wh, fw_b, fw_act_fn,
                                                  fw_rec_act_fn,
fw_ret_seq, go_backwards=False)
            bw_keras_layer = keras_layer.backward_layer
            bw_weights = bw_keras_layer.get_weights()
            bw_Wx, bw_Wh, bw_b = bw_weights[0], bw_weights[1],
bw_weights[2]
            bw_units = bw_keras_layer.units
            bw_act_cfg = bw_keras_layer.get_config()['activation']
            bw_rec_act_cfg =
bw_keras_layer.get_config()['recurrent_activation']
            bw_act_fn = globals().get(bw_act_cfg, tanh)
            bw_rec_act_fn = globals().get(bw_rec_act_cfg, sigmoid)
            bw_ret_seq = bw_keras_layer.return_sequences
            manual_backward_lstm = ManualLSTMLayer(bw_units, bw_Wx,
bw_Wh, bw_b, bw_act_fn,
                                                   bw_rec_act_fn,
bw_ret_seq, go_backwards=True)

self.layers.append(ManualBidirectionalLSTMWrapper(manual_forward_ls
tm, manual_backward_lstm,

merge_mode=keras_layer.merge_mode))
        elif isinstance(keras_layer, layers.Dense):
            weights, bias = keras_layer.get_weights()
            activation_name =
keras_layer.get_config()['activation']
            activation_fn = globals().get(activation_name)
            self.layers.append(ManualDenseLayer(weights, bias,
activation_fn=activation_fn))
        elif isinstance(keras_layer, layers.Dropout):
            print(f"Ignoring Dropout layer: {keras_layer.name}
during manual inference/training.")
```

```
        else:
            raise ValueError(f"Unsupported Keras layer type for
LSTMFromScratch: {type(keras_layer)}")

    def load_keras_model(self, keras_model):
        self.keras_model = keras_model
        self.layers = []
        for layer in keras_model.layers:
            print(f"Processing Keras layer: {layer.name} of type
{type(layer)}")
            if isinstance(layer, tf.keras.layers.InputLayer):
                print(f"Skipping InputLayer: {layer.name}")
                continue
            self.add_keras_layer(layer)

    def predict(self, X_batch_tokens):
        output = X_batch_tokens
        for layer in self.layers:
            output = layer.forward(output)
        return output

    def compile_manual(self, loss_fn_instance, optimizer_instance):
        self.loss_fn = loss_fn_instance
        self.optimizer = optimizer_instance

    def fit_manual(self, X_train_tokens, y_train_labels, epochs=1,
batch_size=32):
        num_samples = tf.shape(X_train_tokens)[0].numpy()
        total_loss = 0
        num_batches = num_samples // batch_size

        for epoch in range(epochs):
            # Shuffle data using TensorFlow
            indices = tf.random.shuffle(tf.range(num_samples))
            X_train_shuffled = tf.gather(X_train_tokens, indices)
            y_train_shuffled = tf.gather(y_train_labels, indices)

            epoch_loss = 0
            for i in range(0, num_samples, batch_size):
                x_batch = X_train_shuffled[i:i+batch_size]
                y_batch = y_train_shuffled[i:i+batch_size]
                y_pred = self.forward(x_batch)
                loss = self.loss_fn(y_batch, y_pred)
                epoch_loss += loss.numpy()
                d_loss = self.loss_fn.backward(y_batch, y_pred)
                self.backward(d_loss)
                self.optimizer.step()

            avg_loss = epoch_loss / num_batches
            total_loss += avg_loss

        return total_loss / epochs
```

BAB III

HASIL PENGUJIAN

3.1. CNN

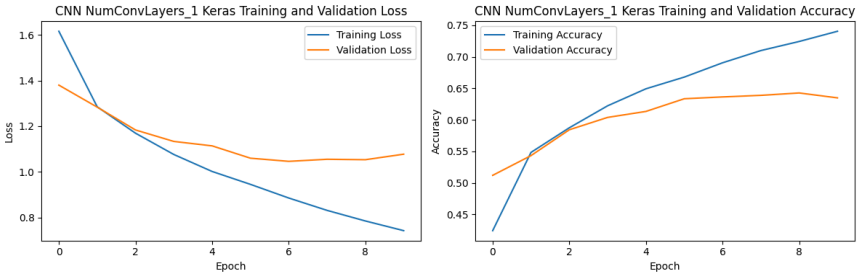
3.1.1. Pengaruh Jumlah Layer Konvolusi

Disini kami menggunakan tiga konfigurasi lapisan konvolusi yang berbeda, yaitu:

- Konfigurasi 1: Satu lapisan konvolusi menggunakan sebanyak 32 filter berukuran 3x3.
- Konfigurasi 2: Dua lapisan konvolusi menggunakan sebanyak 32 filter berukuran 3x3 dan 64 filter berukuran 3x3.
- Konfigurasi 3: Tiga lapisan konvolusi menggunakan sebanyak 32 filter berukuran 3x3, 64 filter berukuran 3x3, dan 128 filter berukuran 3x3.

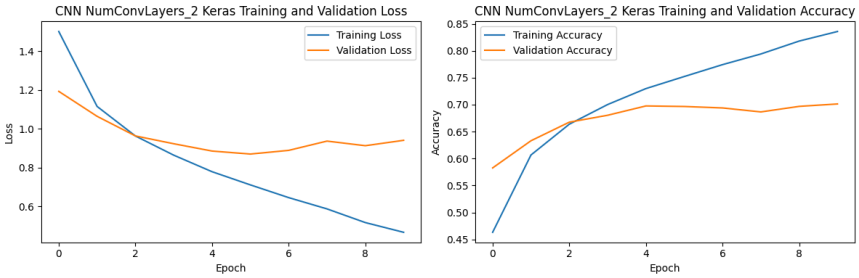
Berikut adalah hasilnya:

- Konfigurasi 1
  F1-Score: 0.6275



Gambar 3.1.1.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh jumlah layer konvolusi pada CNN

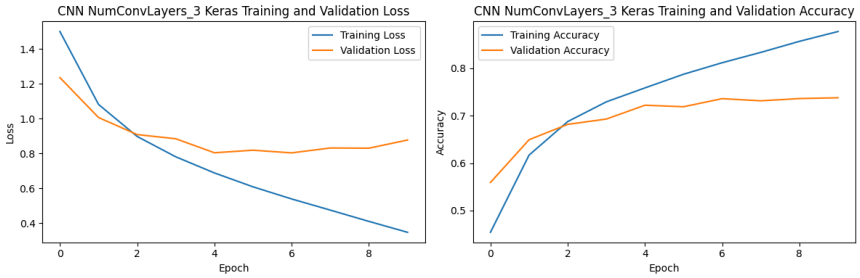
- Konfigurasi 2
  F1-Score: 0.7035



Gambar 3.1.1.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh jumlah layer konvolusi pada CNN

- Konfigurasi 3

F1-Score: 0.7415



Gambar 3.1.1.3. Grafik training loss dan validation loss konfigurasi 3 untuk pengaruh jumlah layer konvolusi pada CNN

Jumlah lapisan konvolusi mempengaruhi kemampuan model untuk mengekstraksi fitur secara hierarkis. Dari hasil, terlihat bahwa peningkatan jumlah lapisan konvolusi dari satu ke tiga lapisan meningkatkan F1-Score dari 0.6275 menjadi 0.7415. Hal ini menunjukkan bahwa model dengan lebih banyak lapisan dapat menangkap fitur yang lebih kompleks dan abstrak, karena setiap lapisan konvolusi berturut-turut memproses fitur dari lapisan sebelumnya, memungkinkan representasi yang lebih kaya. Satu lapisan konvolusi hanya menangkap fitur dasar (misalnya, tepi), sedangkan tiga lapisan memungkinkan model untuk mempelajari pola yang lebih kompleks (misalnya, bentuk atau objek). Namun, peningkatan jumlah lapisan juga meningkatkan kompleksitas komputasi dan risiko overfitting jika data pelatihan tidak cukup atau tanpa regularisasi yang memadai. Dalam kasus ini, model dengan tiga lapisan (NumConvLayers_3) memberikan performa terbaik, menunjukkan bahwa arsitektur yang lebih dalam lebih efektif untuk dataset yang digunakan (kemungkinan CIFAR-10, berdasarkan kode).

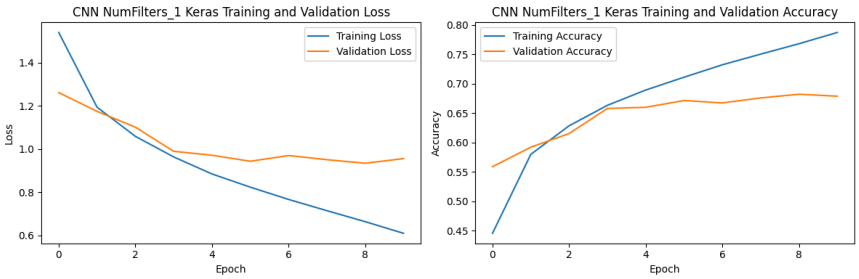
3.1.2. Pengaruh Banyak Filter per Layer Konvolusi

Disini kami menggunakan tiga konfigurasi jumlah filter yang berbeda, yaitu:

- Konfigurasi 1: Dua lapisan konvolusi menggunakan sebanyak 16 filter berukuran 3x3 dan 32 filter berukuran 3x3.
- Konfigurasi 2: Dua lapisan konvolusi menggunakan sebanyak 32 filter berukuran 3x3 dan 64 filter berukuran 3x3.
- Konfigurasi 3: Dua lapisan konvolusi menggunakan sebanyak 64 filter berukuran 3x3 dan 128 filter berukuran 3x3.

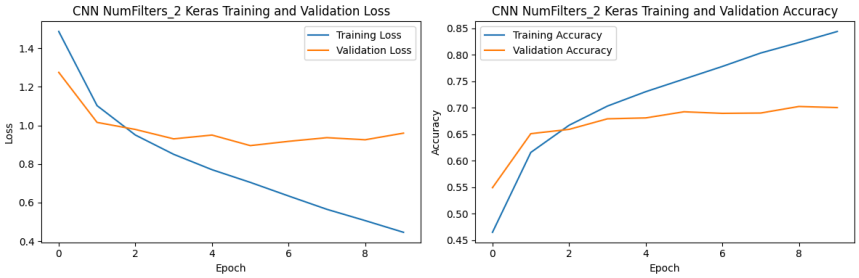
Berikut adalah hasilnya:

- Konfigurasi 1
  F1-Score: 0.6756



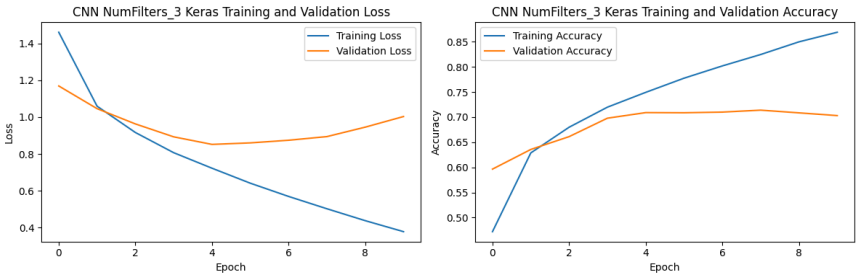
Gambar 3.1.2.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh banyak filter pada CNN

- Konfigurasi 2
  F1-Score: 0.6998



Gambar 3.1.2.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh banyak filter pada CNN

- Konfigurasi 3
  F1-Score: 0.7033



Gambar 3.1.2.3. Grafik training loss dan validation loss konfigurasi 3 untuk pengaruh banyak filter pada CNN

Jumlah filter pada setiap lapisan konvolusi menentukan jumlah fitur yang dapat diekstraksi. Hasil menunjukkan bahwa peningkatan jumlah filter dari (16, 32) ke (32, 64) meningkatkan F1-Score dari 0.6756 ke 0.6998, menunjukkan bahwa lebih banyak filter memungkinkan model untuk menangkap lebih banyak pola yang berbeda, sehingga meningkatkan kemampuan diskriminatifnya. Namun, peningkatan lebih lanjut ke (64, 128) hanya memberikan peningkatan kecil (0.7033), yang menunjukkan adanya efek diminishing returns. Dengan lebih banyak filter, model

memiliki kapasitas lebih besar, tetapi juga meningkatkan kompleksitas dan risiko overfitting, terutama jika dataset tidak cukup besar untuk mendukung parameter tambahan. Dalam hal ini, konfigurasi (64, 128) memberikan performa terbaik, tetapi keuntungannya relatif kecil dibandingkan (32, 64), menunjukkan bahwa jumlah filter yang moderat (32, 64) mungkin sudah cukup untuk menangkap fitur penting pada dataset ini.

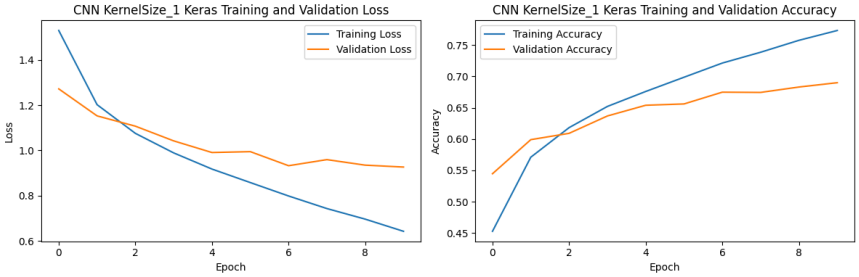
3.1.3. Pengaruh Ukuran Filter per Layer Konvolusi

Disini kami menggunakan tiga konfigurasi ukuran filter yang berbeda, yaitu:

- Konfigurasi 1: Dua lapisan konvolusi menggunakan sebanyak 32 filter berukuran 2x2 dan 64 filter berukuran 2x2.
- Konfigurasi 2: Dua lapisan konvolusi menggunakan sebanyak 32 filter berukuran 3x3 dan 64 filter berukuran 3x3.
- Konfigurasi 3: Dua lapisan konvolusi menggunakan sebanyak 32 filter berukuran 5x5 dan 64 filter berukuran 5x5.

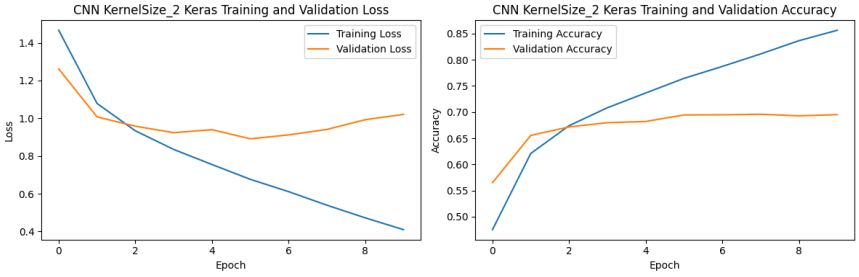
Berikut adalah hasilnya:

- Konfigurasi 1
  F1-Score: 0.6930



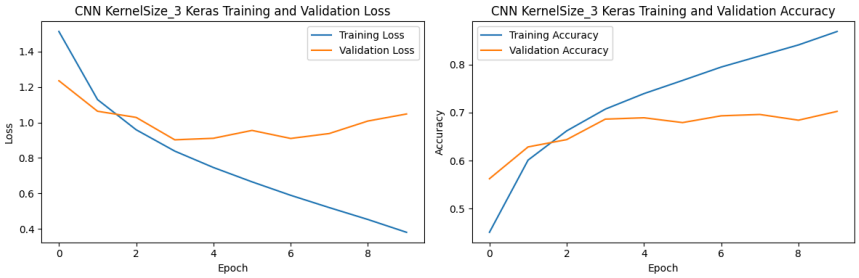
Gambar 3.1.3.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh ukuran filter pada CNN

- Konfigurasi 2
  F1-Score: 0.6964



Gambar 3.1.3.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh ukuran filter pada CNN

- Konfigurasi 3
  F1-Score: 0.7011



Gambar 3.1.3.3. Grafik training loss dan validation loss konfigurasi 3 untuk pengaruh ukuran filter pada CNN

Ukuran filter mempengaruhi jenis fitur yang dapat dideteksi oleh lapisan konvolusi. Filter yang lebih kecil (misalnya, 2x2) fokus pada detail lokal, sedangkan filter yang lebih besar (misalnya, 5x5) dapat menangkap fitur yang lebih global. Hasil menunjukkan bahwa ukuran filter 5x5 (KernelSize_3) memberikan F1-Score tertinggi (0.7011), diikuti oleh 3x3 (0.6964) dan 2x2 (0.6930). Peningkatan performa dari 2x2 ke 5x5 menunjukkan bahwa filter yang lebih besar mungkin lebih efektif untuk dataset ini, kemungkinan karena dapat menangkap pola yang lebih luas atau kontekstual (misalnya, bentuk atau struktur objek yang lebih besar pada gambar CIFAR-10). Namun, perbedaan F1-Score relatif kecil, menunjukkan bahwa ukuran filter 3x3 sudah cukup kompetitif dan sering digunakan sebagai standar karena keseimbangan antara menangkap detail dan konteks tanpa meningkatkan kompleksitas komputasi secara berlebihan. Filter 5x5 memberikan hasil terbaik, tetapi keuntungannya kecil dibandingkan 3x3, sehingga ukuran 3x3 mungkin lebih praktis untuk efisiensi komputasi.

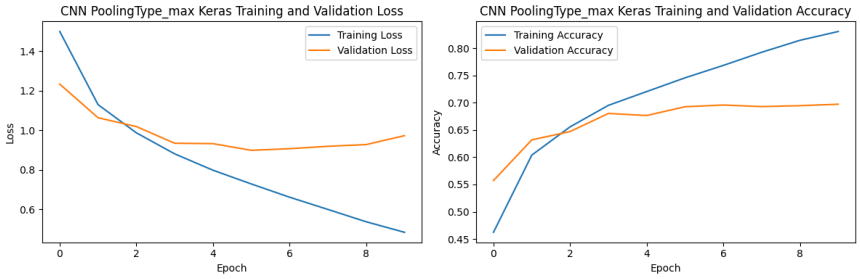
3.1.4. Pengaruh Jenis Pooling Layer

Disini kami menggunakan dua konfigurasi jenis pooling yang berbeda, yaitu:

- Konfigurasi 1: Dua lapisan konvolusi menggunakan sebanyak 32 filter berukuran 3x3 dan 64 filter berukuran 3x3 serta menggunakan max pooling layer.
- Konfigurasi 2: Dua lapisan konvolusi menggunakan sebanyak 32 filter berukuran 3x3 dan 64 filter berukuran 3x3 serta menggunakan average pooling layer.

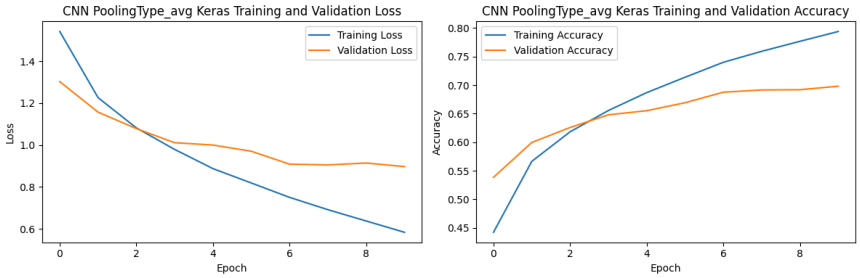
Berikut adalah hasilnya:

- Konfigurasi 1
  F1-Score: 0.6959



Gambar 3.1.4.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh jenis pooling layer pada CNN

- Konfigurasi 2
  F1-Score: 0.7040



Gambar 3.1.4.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh jenis pooling layer pada CNN

Jenis pooling layer (max pooling atau average pooling) memengaruhi bagaimana informasi spasial diringkas setelah operasi konvolusi. Max pooling mengambil nilai maksimum dari setiap jendela pooling, menonjolkan fitur yang paling dominan, sedangkan average pooling menghitung rata-rata, memberikan representasi yang lebih halus. Hasil menunjukkan bahwa average pooling (0.7040) sedikit lebih baik daripada max pooling (0.6959) dalam hal F1-Score. Ini menunjukkan bahwa untuk dataset yang digunakan, average pooling mungkin lebih efektif dalam mempertahankan informasi konteks yang lebih luas, yang dapat membantu dalam tugas klasifikasi dengan fitur yang tidak selalu bergantung pada nilai ekstrem. Max pooling cenderung lebih agresif dalam memilih fitur dominan, yang bisa mengabaikan informasi subtil, sedangkan average pooling lebih inklusif terhadap kontribusi semua piksel dalam jendela. Keunggulan average pooling dalam kasus ini menunjukkan bahwa dataset mungkin memiliki fitur yang lebih

terdistribusi secara merata, dan average pooling membantu model menangkap representasi yang lebih stabil.

3.2. RNN

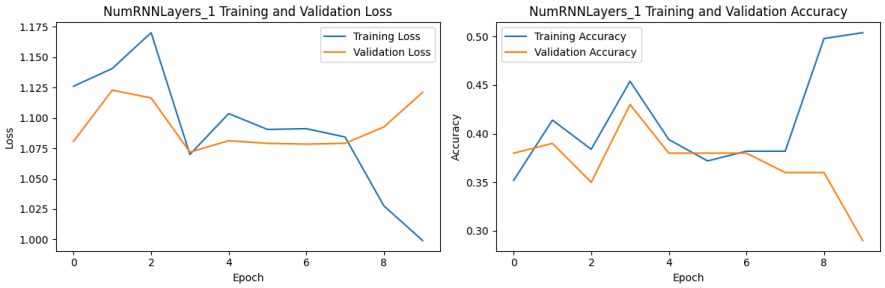
3.1.5. Pengaruh Jumlah Layer RNN

Disini kami menggunakan tiga konfigurasi lapisan RNN yang berbeda, yaitu:

- Konfigurasi 1: Satu lapisan konvolusi menggunakan sebanyak 64 unit.
- Konfigurasi 2: Dua lapisan konvolusi menggunakan sebanyak 64 unit dan 32 unit.
- Konfigurasi 3: Tiga lapisan konvolusi menggunakan sebanyak 64 unit, 32 unit, dan 16 unit.

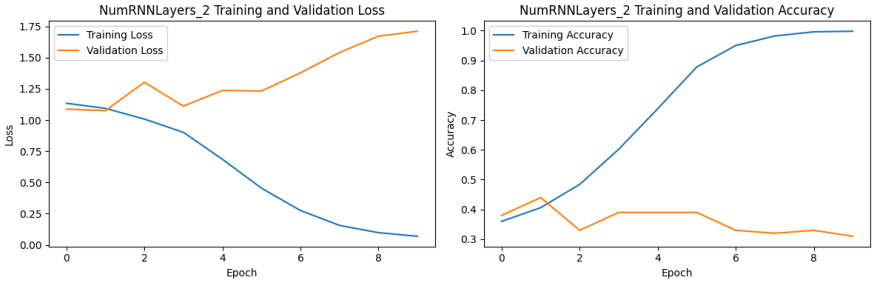
Berikut adalah hasilnya:

- Konfigurasi 1
  F1-Score: 0.3334



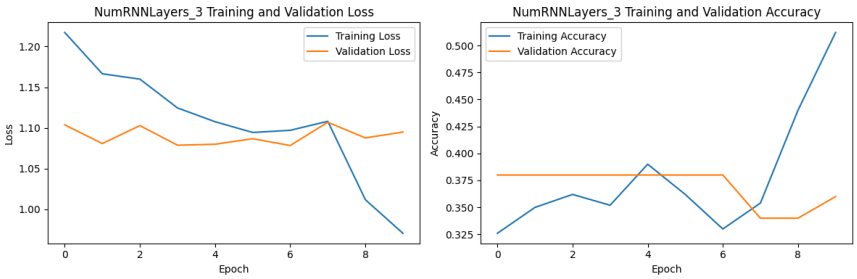
Gambar 3.1.1.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh jumlah layer konvolusi pada RNN

- Konfigurasi 2
  F1-Score: 0.3518



Gambar 3.1.1.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh jumlah layer konvolusi pada RNN

- Konfigurasi 3
  F1-Score: 0.3379



Gambar 3.1.1.3. Grafik training loss dan validation loss konfigurasi 3 untuk pengaruh jumlah layer konvolusi pada RNN

Hasil menunjukkan bahwa penambahan layer dari satu ke dua lapisan memberikan sedikit peningkatan performa (dari 0.3334 menjadi 0.3518). Namun, penambahan lapisan ketiga justru menurunkan F1-Score menjadi 0.3379. Peningkatan performa dari konfigurasi 1 ke 2 mengindikasikan bahwa tambahan lapisan dapat membantu menangkap *pattern* yang lebih kompleks. Namun, penambahan lapisan ketiga tidak memberikan *improvement* tambahan, bahkan cenderung menurunkan performa.

Kemungkinan penyebabnya adalah meningkatnya kompleksitas model yang membuat proses pelatihan menjadi lebih sulit, terutama jika ukuran dataset tidak cukup besar. Selain itu, panjang sekuens dalam dataset NusaX-Sentiment yang relatif pendek mungkin tidak memerlukan model yang sangat dalam untuk menangkap dependensi temporal secara efektif.

Dengan demikian, dua layer RNN tampaknya memberikan *trade-off* terbaik antara kompleksitas dan performa untuk tugas klasifikasi sentimen pada dataset ini.

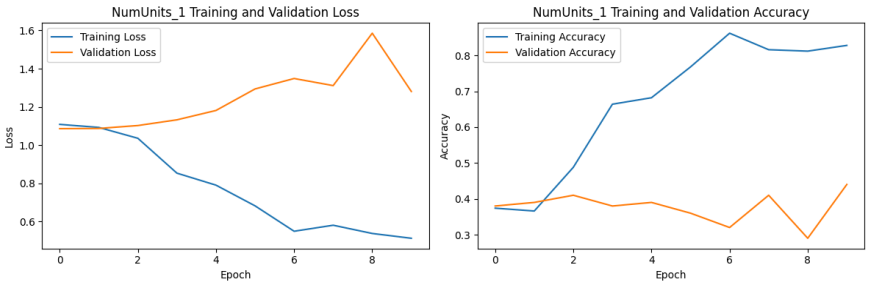
3.1.6. Pengaruh Banyak Unit (Sel) per Layer RNN

Disini kami menggunakan tiga konfigurasi jumlah cells yang berbeda, yaitu:

- Konfigurasi 1: Satu lapisan RNN dengan 32 unit.
- Konfigurasi 2: Satu lapisan RNN dengan 64 unit.
- Konfigurasi 3: Satu lapisan RNN dengan 128 unit.

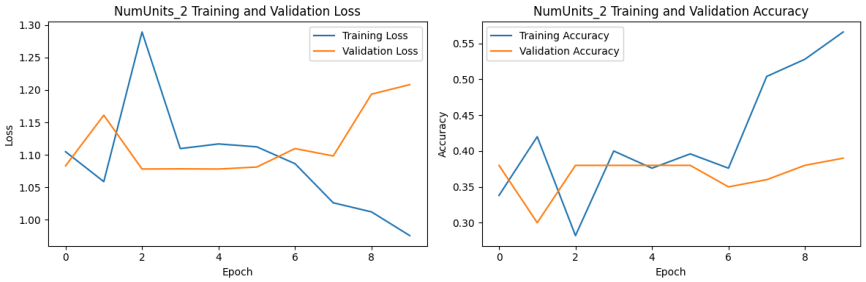
Berikut adalah hasilnya:

- Konfigurasi 1
  F1-Score: 0.3657



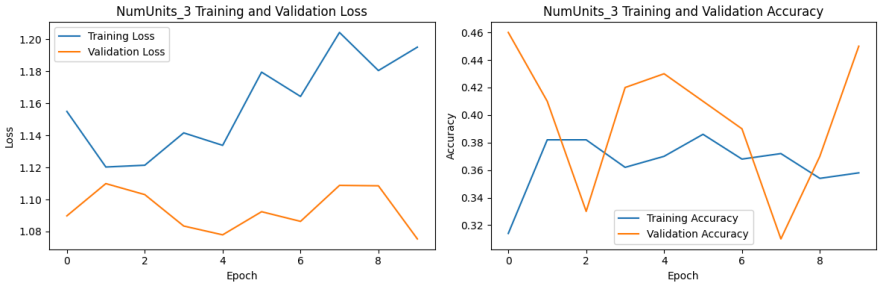
Gambar 3.1.2.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh banyak cell pada RNN

- Konfigurasi 2
  F1-Score: 0.3256



Gambar 3.1.2.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh banyak cell pada RNN

- Konfigurasi 3
  F1-Score: 0.2849



Gambar 3.1.2.3. Grafik training loss dan validation loss konfigurasi 3 untuk pengaruh banyak cell pada RNN

Hasil ini menunjukkan bahwa peningkatan jumlah unit justru menurunkan performa model. Konfigurasi dengan 32 unit menghasilkan skor F1 tertinggi, sementara konfigurasi dengan 128 unit justru memberikan performa terendah. Hal ini mengindikasikan adanya dampak negatif dari peningkatan kompleksitas model yang tidak disertai dengan kapasitas generalisasi yang memadai.

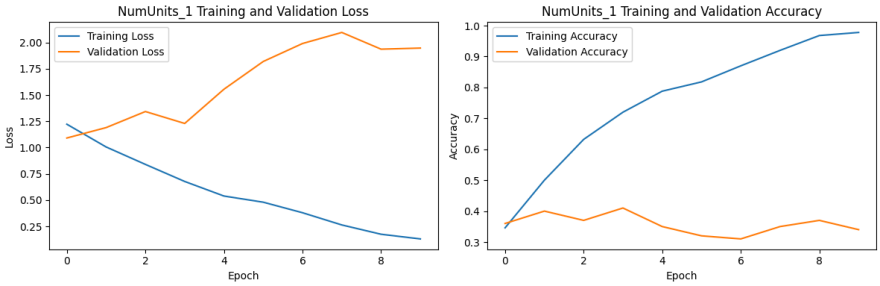
Salah satu kemungkinan penyebab hasil ini adalah *overfitting*, di mana model dengan terlalu banyak parameter menyesuaikan diri secara berlebihan terhadap data latih, tetapi gagal mempertahankan performa pada data validasi. Selain itu, dengan panjang *sequence* teks yang relatif pendek dan jumlah data yang terbatas, jumlah unit yang besar mungkin justru memperbesar *noise* daripada membantu menangkap *pattern.*

Efek *diminishing returns* juga terlihat jelas, peningkatan dari 32 ke 64 dan kemudian ke 128 unit tidak memberikan *improvement*, bahkan sebaliknya, merugikan performa. Ini menunjukkan bahwa pada dataset seperti NusaX-Sentiment, jumlah unit yang terlalu besar tidak diperlukan, dan justru membebani model tanpa *improvement* tambahan.

3.1.7. Pengaruh Jenis RNN (Unidirectional vs. Bidirectional)

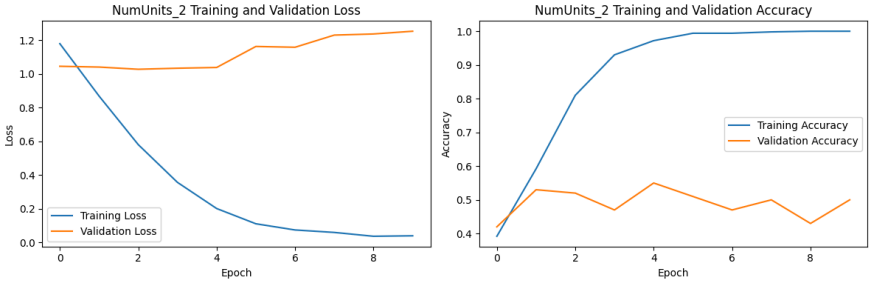
Disini kami menggunakan tiga konfigurasi ukuran filter yang berbeda, yaitu:

- Konfigurasi 1: Unidirectional.
- Konfigurasi 2: Bidirectional
- Konfigurasi 1
  F1-Score: 0.3246



Gambar 3.1.3.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh jenis RNN berdasarkan arah

- Konfigurasi 2
  F1-Score: 0.4780



Gambar 3.1.3.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh jenis RNN berdasarkan arah

Peningkatan performa yang cukup signifikan dari 0.3246 menjadi 0.4780 menunjukkan bahwa RNN bidirectional secara efektif mampu memanfaatkan informasi kontekstual secara lebih menyeluruh. Dengan model bidirectional, kata-kata di akhir *sequence* pun dapat mempengaruhi interpretasi kata-kata di awal, sesuatu yang tidak dapat dilakukan oleh model unidirectional.

Selain itu, grafik *training loss* dan *validation loss* juga memperlihatkan bahwa model bidirectional cenderung lebih stabil dan mampu mempelajari pola secara lebih baik, meskipun tetap harus diperhatikan risiko overfitting bila digunakan pada dataset kecil.

Namun, perlu dicatat bahwa RNN bidirectional memiliki kompleksitas komputasi yang lebih tinggi dan memerlukan waktu pelatihan yang lebih lama, karena dua arah harus dihitung secara paralel. Oleh karena itu, meskipun memberikan performa lebih baik, penggunaannya harus dipertimbangkan dengan mempertimbangkan *trade-off* antara *accuracy* dan *efficiency*.

3.3. LSTM

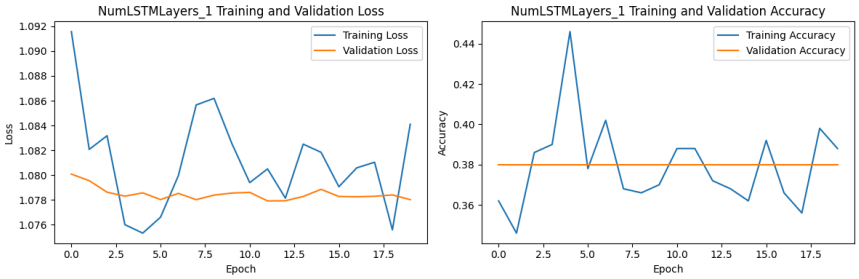
3.1.8. Pengaruh Jumlah Layer LSTM

Disini kami menggunakan tiga konfigurasi lapisan LSTM yang berbeda, yaitu:

- Konfigurasi 1: Satu lapisan konvolusi menggunakan sebanyak 64 unit.
- Konfigurasi 2: Dua lapisan konvolusi menggunakan sebanyak 64 unit dan 32 unit.
- Konfigurasi 3: Tiga lapisan konvolusi menggunakan sebanyak 64 unit, 32 unit, dan 16 unit.

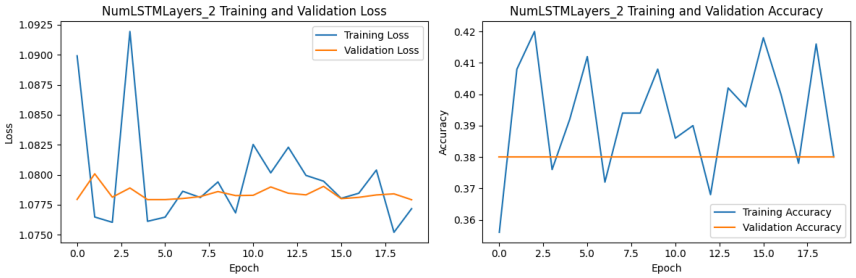
Berikut adalah hasilnya:

- Konfigurasi 1
  F1-Score: 0.1844



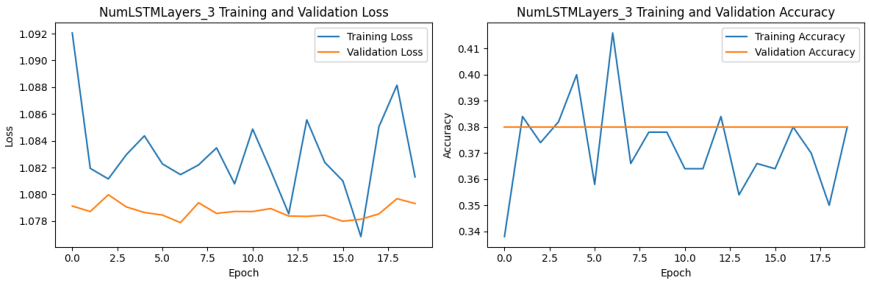
Gambar 3.1.1.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh jumlah layer LSTM

- Konfigurasi 2
  F1-Score: 0.1844



Gambar 3.1.1.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh jumlah layer LSTM

- Konfigurasi 3
  F1-Score: 0.1844



Gambar 3.1.1.3. Grafik training loss dan validation loss konfigurasi 3 untuk pengaruh jumlah layer LSTM

Jumlah layer LSTM memengaruhi kemampuan model untuk menangkap dependensi temporal yang lebih kompleks dalam data teks. Hasil eksperimen menunjukkan bahwa penambahan jumlah layer dari satu layer (64 unit), dua layer (64+32 unit), hingga tiga layer (64+32+16 unit) menghasilkan F1-score yang identik, yaitu 0.1844 untuk ketiga konfigurasi. Hal ini menunjukkan bahwa penambahan layer tidak memberikan peningkatan performa pada dataset NusaX-Sentiment.

Kemungkinan besar, dataset ini memiliki panjang sekuens yang relatif pendek (maksimum 200 token, tetapi rata-rata lebih rendah), sehingga satu layer LSTM sudah cukup untuk menangkap pola temporal yang relevan untuk klasifikasi sentimen. Penambahan layer meningkatkan kompleksitas model, tetapi tidak menghasilkan keuntungan signifikan, mungkin karena keterbatasan ukuran dataset atau sifat tugas klasifikasi sentimen yang tidak memerlukan pemodelan dependensi jarak jauh. Dalam hal ini, konfigurasi satu layer tampaknya sudah optimal untuk efisiensi dan performa.

3.1.9. Pengaruh Banyak Unit (Sel) per Layer LSTM

Disini kami menggunakan tiga konfigurasi jumlah filter yang berbeda, yaitu:

- Konfigurasi 1: Satu lapisan LSTM dengan 32 unit.
- Konfigurasi 2: Satu lapisan LSTM dengan 64 unit.
- Konfigurasi 3: Satu lapisan LSTM dengan 128 unit.

Berikut adalah hasilnya:

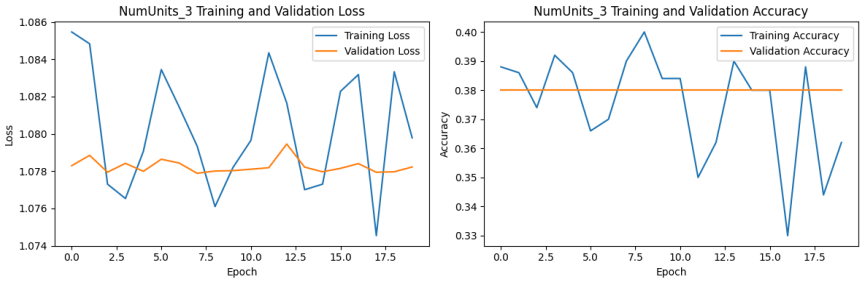
- Konfigurasi 1
  F1-Score: 0.1844

Gambar 3.1.2.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh banyak unit pada layer LSTM

- Konfigurasi 2
  F1-Score: 0.1827

Gambar 3.1.2.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh banyak unit pada layer LSTM

- Konfigurasi 3
  F1-Score: 0.1827



Gambar 3.1.2.3. Grafik training loss dan validation loss konfigurasi 3 untuk pengaruh banyak unit pada layer LSTM

Jumlah unit dalam layer LSTM menentukan kapasitas model untuk merepresentasikan fitur dalam sekuens teks. Hasil eksperimen menunjukkan bahwa peningkatan jumlah unit dari 32 unit (F1-score 0.1844) ke 64 unit (F1-score 0.1827) dan 128 unit (F1-score 0.1827) tidak memberikan peningkatan performa yang berarti, bahkan sedikit menurun. Penurunan kecil ini mungkin disebabkan oleh kapasitas model yang berlebih pada 64 dan 128 unit, yang dapat menyebabkan overfitting pada dataset NusaX-Sentiment yang relatif sederhana. Dengan sekuens pendek dan tugas klasifikasi sentimen yang tidak terlalu kompleks, 32 unit tampaknya sudah cukup untuk menangkap pola sentimen utama. Peningkatan jumlah unit menambah parameter model, tetapi tanpa data yang cukup besar atau kompleksitas tugas yang lebih tinggi, efeknya cenderung minimal. Konfigurasi 32 unit memberikan keseimbangan terbaik antara kapasitas dan efisiensi dalam eksperimen ini.

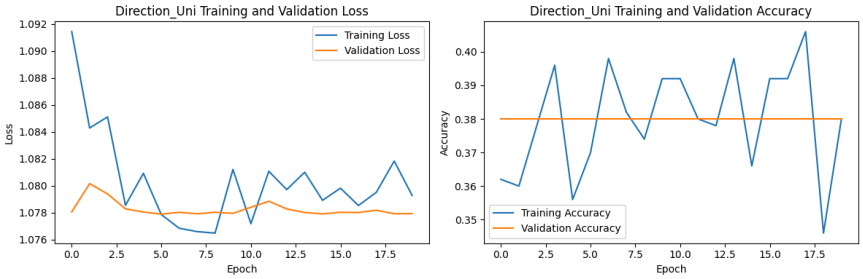
3.1.10. Pengaruh Jenis LSTM (Unidirectional vs. Bidirectional)

Disini kami menggunakan tiga konfigurasi ukuran filter yang berbeda, yaitu:

- Konfigurasi 1: Unidirectional.
- Konfigurasi 2: Bidirectional

Berikut adalah hasilnya:

- Konfigurasi 1
  F1-Score: 0.1844



Gambar 3.1.3.1. Grafik training loss dan validation loss konfigurasi 1 untuk pengaruh jenis LSTM pada LSTM

- Konfigurasi 2
  F1-Score: 0.7428

Gambar 3.1.3.2. Grafik training loss dan validation loss konfigurasi 2 untuk pengaruh jenis LSTM pada LSTM

Jenis LSTM, apakah unidirectional atau bidirectional, memengaruhi cara model memproses konteks dalam sekuens teks. Hasil eksperimen menunjukkan bahwa perubahan dari LSTM unidirectional (F1-score 0.1844) ke bidirectional (F1-score 0.7428) memberikan peningkatan performa yang sangat signifikan. Bidirectional LSTM memungkinkan model untuk menangkap konteks dari arah maju dan mundur, yang sangat penting untuk klasifikasi sentimen, di mana kata-kata di awal dan akhir kalimat sering kali saling mempengaruhi. Peningkatan F1-score yang besar menunjukkan bahwa konteks dua arah memungkinkan model untuk lebih akurat mengidentifikasi pola sentimen dalam dataset NusaX-Sentiment. Meskipun bidirectional LSTM meningkatkan kompleksitas komputasi karena menggandakan jumlah parameter, manfaatnya jelas terlihat pada tugas ini. Konfigurasi bidirectional memberikan performa terbaik, menjadikannya pilihan optimal untuk dataset ini.

BAB IV

KESIMPULAN & SARAN

4.1. KESIMPULAN

Berdasarkan eksperimen pada tugas besar kali ini, implementasi CNN pada dataset CIFAR-10 menunjukkan bahwa peningkatan jumlah layer konvolusi (dari 1 ke 3) meningkatkan F1-Score dari 0.6275 ke 0.7415, dengan tigas lapisan memberikan performa terbaik karena menangkap fitur kompleks. Jumlah filter (64, 128) menghasilkan F1-Score tertinggi (0.7033), namun dengan efek diminishing returns. Ukuran filter 5x5 (F1-Score 0.7011) sedikit lebih baik dibandingkan 3x3 dan 2x2, sementara average pooling (F1-Score 0.7040) mengungguli max pooling (0.6959) karena mempertahankan informasi konteks lebih baik. Untuk RNN dan LSTM pada dataset NusaX-Sentiment, bidirectional LSTM memberikan performa terbaik (F1-Score 0.7428) dibandingkan unidirectional (0.1844), karena menangkap konteks dua arah. Namun, penambahan lapisan RNN (maksimal F1-Score 0.3518 pada 2 lapisan) dan LSTM (F1-Score stagnan di 0.1844) tidak selalu meningkatkan performa, kemungkinan karena dataset yang relatif sederhana. Jumlah unit RNN dan LSTM juga menunjukkan efek *diminishing returns*, dengan 32 unit sering kali cukup untuk tugas ini.

4.2. SARAN

1. Optimalkan arsitektur model dengan memilih jumlah lapisan dan unit yang seimbang untuk menghindari overfitting, terutama pada dataset kecil seperti NusaX-Sentiment.
2. Gunakan bidirectional LSTM untuk tugas klasifikasi sentimen yang memerlukan konteks dua arah, tetapi pertimbangkan efisiensi komputasi.
3. Untuk CNN, prioritaskan ukuran filter 3x3 dan average pooling untuk efisiensi dan performa yang seimbang pada dataset CIFAR-10.
4. Terapkan teknik regularisasi tambahan (misalnya dropout lebih agresif) untuk mencegah overfitting pada model yang lebih kompleks.
5. Eksplorasi augmentasi data untuk meningkatkan generalisasi model, terutama pada dataset dengan sekuens pendek atau variasi terbatas.

BAB V

PEMBAGIAN TUGAS

Nama	NIM	Pembagian Tugas
Rizqika Mulia Pratama	13522126	Semua yang berhubungan dengan LSTM
Attara Majesta Ayub	13522139	Semua yang berhubungan dengan RNN
Ikhwan Al Hakim	13522147	Semua yang berhubungan dengan CNN

REFERENSI

- https://keras.io/api/layers/
- https://d2l.ai/chapter_recurrent-modern/lstm.html
- https://d2l.ai/chapter_recurrent-modern/deep-rnn.html
- https://d2l.ai/chapter_recurrent-modern/bi-rnn.html
- https://d2l.ai/chapter_recurrent-neural-networks/index.html
- https://d2l.ai/chapter_convolutional-neural-networks/index.html
- https://numpy.org/doc/2.1/reference/generated/numpy.einsum.html
- https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/
- https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/
- https://github.com/IndoNLP/nusax/tree/main/datasets/sentiment/indonesian