INFORMATICS ENGINEERING STUDY PROGRAM
SCHOOL OF ELECTRICAL ENGINEERING DAN INFORMATICS
BANDUNG INSTITUTE OF TECHNOLOGY

Laporan Tugas Besar #2

Convolutional Neural Network dan Recurrent Neural Network

Disusun Oleh:
Kelompok 1

Anggota:
1. Marzuli Suhada M - 13522070
2. Ahmad Mudabbir Arif - 13522072
3. Naufal Adnan - 13522116

IF3270 Pembelajaran Mesin

2025

DAFTAR ISI

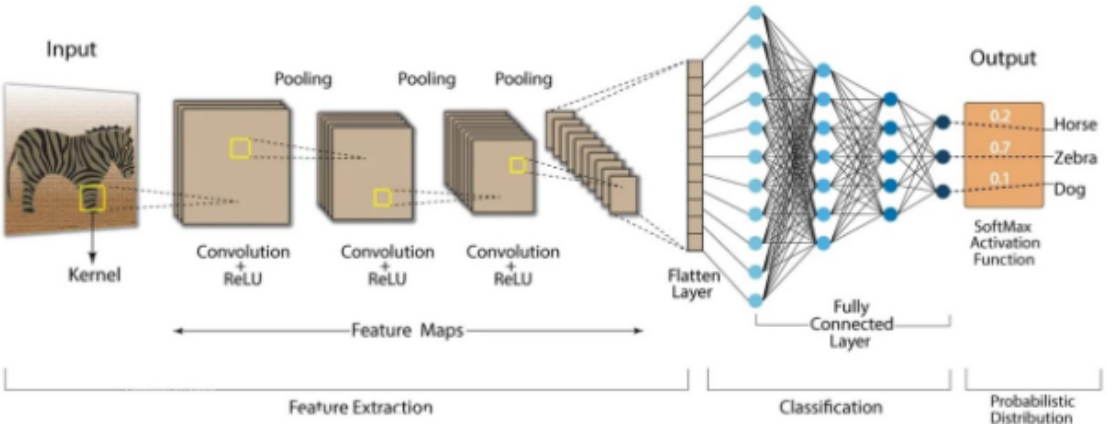
DAFTAR TABEL

DAFTAR GAMBAR

I. DESKRIPSI PERSOALAN

1. CNN

Convolutional Neural Network (CNN) merupakan salah satu arsitektur *deep learning* yang paling efektif untuk pemrosesan data visual seperti gambar. CNN dirancang khusus untuk menangkap pola-pola spasial dalam data gambar melalui operasi konvolusi yang dapat mengekstrak fitur-fitur hierarkis dari tingkat rendah hingga tingkat tinggi. Dalam rangka memahami konsep dasar machine learning secara mendalam dan penerapannya pada data visual, mahasiswa ditugaskan untuk mengimplementasikan dan melatih model CNN dari awal (from scratch) untuk klasifikasi gambar, sekaligus melakukan perbandingan dengan implementasi menggunakan Keras.



Gambar 1.1 Arsitektur Convolutional Neural Network

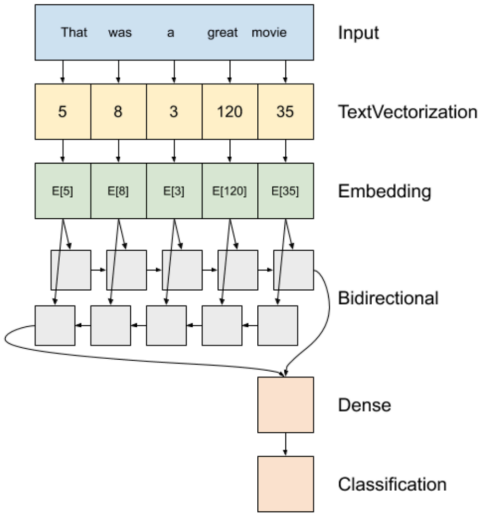
Kami diminta untuk membuat implementasi CNN yang mampu:

- Memproses data gambar CIFAR-10 dengan melakukan preprocessing yang meliputi:
  - Normalisasi pixel values ke rentang [0,1]
  - Pembagian dataset menjadi training (40k), validation (10k), dan test (10k) sesuai rasio 4:1
- Membangun dan melatih model CNN untuk klasifikasi gambar dengan dataset CIFAR-10 menggunakan Keras, dengan ketentuan:
  - Model minimal memiliki Conv2D layer, Pooling layers, Flatten/Global Pooling layer, dan Dense layer

    - Menggunakan Sparse Categorical Crossentropy sebagai loss function untuk menangani klasifikasi multikelas
    - Menggunakan Adam sebagai optimizer dengan learning rate adaptif
- Melakukan variasi pelatihan untuk analisis pengaruh beberapa hyperparameter:
    - Pengaruh jumlah layer konvolusi (3 variasi: 2, 3, dan 4 layer)
    - Pengaruh banyak filter per layer konvolusi (3 variasi kombinasi: small, medium, large)
    - Pengaruh ukuran filter per layer konvolusi (3 variasi: 3x3, mixed, 5x5)
    - Pengaruh jenis pooling layer yang digunakan (2 variasi: max pooling vs average pooling)
- Evaluasi dan analisis yang meliputi:
    - Membandingkan hasil akhir prediksi menggunakan macro f1-score
    - Membandingkan grafik training loss dan validation loss tiap epoch
    - Memberikan kesimpulan mengenai pengaruh hyperparameter tersebut terhadap kinerja model
- Implementasi from scratch yang mencakup:
    - Menyimpan weight hasil pelatihan model Keras
    - Membuat modul forward propagation from scratch dengan kemampuan membaca bobot dari model Keras
    - Implementasi forward propagation secara modular per layer
    - Membandingkan hasil forward propagation from scratch dengan Keras menggunakan macro f1-score
- Fitur bonus yang diimplementasikan:
    - Implementasi backward propagation from scratch untuk training capability
    - Support untuk batch inference dengan berbagai ukuran batch

2. Simple RNN (*Recurrent Neural Network*)

Recurrent Neural Network (RNN) merupakan salah satu arsitektur penting dalam bidang deep learning, khususnya untuk data sekuensial seperti teks. Dalam rangka memahami konsep dasar *machine learning* secara mendalam dan penerapannya pada data teks, mahasiswa ditugaskan untuk mengimplementasikan dan melatih model RNN dari awal (*from scratch*) untuk klasifikasi teks, sekaligus melakukan perbandingan dengan implementasi menggunakan Keras.



Gambar 1.2 Arsitektur Recurrent Neural Network

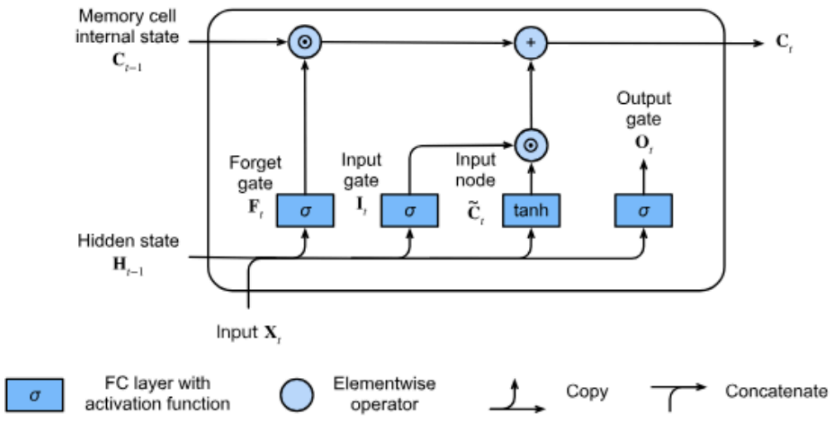
Kami diminta untuk membuat implementasi RNN yang mampu:

- Menerima data teks dan melakukan preprocessing yang meliputi:
  - *Tokenization* menggunakan *TextVectorization layer*.
  - *Embedding function* menggunakan *Embedding layer*.
- Membangun dan melatih model RNN untuk klasifikasi teks dengan dataset NusaX-Sentiment (Bahasa Indonesia) menggunakan Keras, dengan ketentuan:
  - Model minimal memiliki *Embedding layer*, *Bidirectional RNN layer* dan/atau *Unidirectional RNN layer*, *Dropout layer*, dan *Dense layer.*
  - Menggunakan *Sparse Categorical Crossentropy* sebagai *loss function*.
  - Menggunakan Adam sebagai *optimizer*.
- Melakukan variasi pelatihan untuk analisis pengaruh beberapa *hyperparameter*:
  - Pengaruh jumlah layer RNN (3 variasi).
  - Pengaruh banyak *cell* RNN per layer (3 variasi kombinasi).
  - Pengaruh jenis layer RNN berdasarkan arah (*bidirectional* atau *unidirectional*) (2 variasi).
- Membandingkan hasil akhir prediksi menggunakan macro f1-score.
- Membandingkan grafik *training loss* dan *validation loss* tiap epoch.
- Memberikan kesimpulan mengenai pengaruh *hyperparameter* tersebut terhadap kinerja model.
- Menyimpan *weight* hasil pelatihan model Keras.

- Membuat modul *forward propagation from scratch* dari model yang telah dibuat dengan ketentuan:
    - Dapat membaca model hasil pelatihan dengan Keras (bobot sama).
    - Implementasi *forward propagation* secara modular (per layer).
    - Membandingkan hasil *forward propagation from scratch* dengan hasil *forward propagation* menggunakan Keras menggunakan *split data test* dan metrik macro f1-score.
- Terdapat beberapa fitur bonus yang dapat diimplementasikan:
    - Implementasi fungsi *backward propagation from scratch* untuk seluruh layer yang digunakan.
    - Seluruh implementasi *forward propagation* harus bisa menangani kasus *batch inference*, dimana model dapat menerima lebih dari satu input untuk satu kali *forward propagation*.

3. LSTM

Long Short-Term Memory (LSTM) merupakan salah satu arsitektur penting dalam bidang *deep learning*, khususnya untuk data sekuensial seperti teks. LSTM dikembangkan untuk mengatasi masalah *vanishing gradient* yang dialami oleh *Recurrent Neural Network* (RNN) tradisional, sehingga mampu menangkap dependensi jangka panjang dalam sequence data. Dalam rangka memahami konsep dasar *machine learning* secara mendalam dan penerapannya pada data teks, mahasiswa ditugaskan untuk mengimplementasikan dan melatih model LSTM dari awal (*from scratch*) untuk klasifikasi teks, sekaligus melakukan perbandingan dengan implementasi menggunakan Keras.



Gambar 1.3 Arsitektur Long Short-Term Memory (LSTM)

Kami diminta untuk membuat implementasi LSTM yang mampu:

- Menerima data teks dan melakukan *preprocessing* yang meliputi:
    - *Tokenization* menggunakan *TextVectorization layer* untuk mengubah teks menjadi *sequence of integers*
    - *Embedding function* menggunakan *Embedding layer* untuk mengkonversi token menjadi *dense vector representation*
- Membangun dan melatih model LSTM untuk klasifikasi teks dengan dataset NusaX-Sentiment (Bahasa Indonesia) menggunakan Keras, dengan ketentuan:
    - Model minimal memiliki *Embedding layer*, Bidirectional LSTM layer dan/atau Unidirectional LSTM layer, *Dropout layer*, dan *Dense layer*
    - Menggunakan *Sparse Categorical Crossentropy* sebagai loss function untuk menangani klasifikasi multikelas
    - Menggunakan Adam sebagai *optimizer* dengan *learning rate* adaptif
- Melakukan variasi pelatihan untuk analisis pengaruh beberapa *hyperparameter*:
    - Pengaruh jumlah layer LSTM (3 variasi: 1, 2, dan 3 layer)
    - Pengaruh banyak cell LSTM per layer (3 variasi: 32, 64, dan 128 cells)
    - Pengaruh jenis layer LSTM berdasarkan arah (2 variasi: *unidirectional* vs *bidirectional*)
- Evaluasi dan analisis yang meliputi:
    - Membandingkan hasil akhir prediksi menggunakan *macro f1-score*
    - Membandingkan grafik *training loss* dan *validation loss* tiap *epoch*
    - Memberikan kesimpulan mengenai pengaruh *hyperparameter* tersebut terhadap kinerja model
- Implementasi *from scratch* yang mencakup:
    - Menyimpan *weight* hasil pelatihan model Keras
    - Membuat modul *forward propagation from scratch* dengan kemampuan membaca bobot dari model Keras
    - Implementasi *forward propagation* secara modular per layer
    - Membandingkan hasil *forward propagation from scratch* dengan Keras menggunakan *macro f1-score*
- Fitur bonus yang diimplementasikan:
    - Implementasi *backward propagation from scratch* untuk *training capability*

- Support untuk *batch inference* dengan berbagai ukuran *batch*

II. PEMBAHASAN

1. Penjelasan Implementasi

A. Deskripsi Kelas, Atribut, dan Method

1. CNN

a. Kelas BaseLayer

Kelas BaseLayer merupakan kelas abstrak dasar (base class) untuk semua layer dalam arsitektur CNN. Kelas ini dirancang sebagai blueprint yang menyediakan struktur umum dan kontrak dasar untuk semua layer turunan seperti convolutional layer, pooling layer, dan layer lainnya. BaseLayer tidak memiliki implementasi aktual untuk metode forward dan backward, namun menyediakan atribut umum seperti bobot, bias, dan cache yang akan digunakan oleh layer-layer turunan.

Tabel 2.1.1.1 Atribut Kelas BaseLayer

No	Atribut	Deskripsi
1.	`weights`	*Dictionary* untuk menyimpan parameter bobot layer (kernel, bias, dll)
2.	`biases`	*Dictionary* untuk menyimpan parameter bias layer
3.	`cache`	*Dictionary* untuk menyimpan informasi intermediate yang dibutuhkan saat *backward propagation*

Tabel 2.1.1.2 Method Kelas BaseLayer

No	Method	Deskripsi
1.	`__init__()`	Inisialisasi struktur dasar layer dengan *dictionary* kosong untuk *weights*, *biases*, dan *cache*
2.	`forward(x, training=False)`	Fungsi abstrak untuk *forward propagation* dan harus

		diimplementasikan oleh *subclass*
3.	`backward(grad_output)`	Fungsi abstrak untuk *backward propagation* dan harus diimplementasikan oleh *subclass*

b. Kelas Conv2DLayer

Kelas Conv2DLayer adalah implementasi layer konvolusi 2D yang merupakan komponen inti dalam arsitektur CNN. Layer ini bertanggung jawab untuk melakukan operasi konvolusi pada input tensor untuk mengekstrak fitur spatial dari data gambar. Conv2DLayer menerima parameter seperti jumlah filter, ukuran kernel, stride, padding, dan fungsi aktivasi. Layer ini menggunakan He initialization untuk inisialisasi bobot yang optimal dan mendukung berbagai fungsi aktivasi seperti ReLU, tanh, dan sigmoid.

Tabel 2.1.1.3 Atribut Kelas Conv2DLayer

No	Atribut	Deskripsi
1.	`input_shape`	Bentuk input tensor (height, width, channels)
2.	`filters`	Jumlah filter/kernel yang digunakan untuk konvolusi
3.	`kernel_size`	Ukuran kernel konvolusi (height, width)
4.	`strides`	Nilai stride untuk pergeseran kernel
5.	`padding`	Jenis padding yang digunakan ('same' atau 'valid')
6.	`activation`	Fungsi aktivasi yang diterapkan setelah konvolusi
7.	`weights['kernel']`	Matriks bobot kernel berukuran (kh, kw, channels, filters)
8.	`biases['bias']`	Vektor bias berukuran (filters,)
9.	`output_shape`	Bentuk output yang dihitung berdasarkan parameter layer

10.	`initialized`	Status inisialisasi layer

Tabel 2.1.1.4 Method Kelas Conv2DLayer

No	Method	Deskripsi
1.	`__init__(input_shape, filters, kernel_size, strides, padding, activation)`	Konstruktor layer konvolusi dengan inisialisasi parameter dan perhitungan output shape
2.	`_initialize_weights()`	Menginisialisasi bobot menggunakan He initialization untuk distribusi optimal
3.	`_calculate_output_shape()`	Menghitung dimensi output berdasarkan parameter konvolusi
4.	`_add_padding(x)`	Menambahkan padding pada input sesuai konfigurasi layer
5.	`_apply_activation(x)`	Menerapkan fungsi aktivasi pada output linear
6.	`forward(x, training=False)`	Melakukan operasi konvolusi lengkap dari input hingga output teraktivasi
7.	`backward(self, grad_output)`	Melakukan backward propagation
8.	`set_weights(weights, biases)`	Mengatur bobot dan bias layer dari sumber eksternal

c. Kelas MaxPooling2D

Kelas MaxPooling2D adalah implementasi layer max pooling yang melakukan downsampling pada feature maps dengan mengambil nilai maksimum dalam setiap window pooling. Layer ini mengurangi dimensi spatial sambil mempertahankan informasi penting, membantu mengurangi overfitting dan computational cost.

Tabel 2.1.1.5 Atribut Kelas MaxPooling2D

No	Atribut	Deskripsi
1.	`pool_size`	Ukuran window pooling (height, width)
2.	`strides`	Nilai stride untuk pergeseran window pooling
3.	`padding`	Jenis padding yang digunakan

Tabel 2.1.1.6 Method Kelas MaxPooling2D

No	Method	Deskripsi
1.	`__init__(pool_size, strides, padding)`	Konstruktor untuk layer max pooling
2.	`_calculate_output_shape(input_shape)`	Menghitung bentuk output setelah operasi pooling
3.	`forward(x, training=False)`	Melakukan operasi max pooling untuk downsampling feature maps
4.	`backward(self, grad_output)`	Melakukan backward propagation

d. Kelas AveragePooling2D

Kelas AveragePooling2D adalah implementasi layer average pooling yang melakukan downsampling dengan mengambil nilai rata-rata dalam setiap window pooling. Berbeda dengan max pooling, average pooling mempertahankan informasi dari seluruh window secara proporsional, memberikan representasi yang lebih smooth.

Tabel 2.1.1.7 Atribut Kelas MaxPooling2D

No	Atribut	Deskripsi
1.	`pool_size`	Ukuran window pooling (height, width)
2.	`strides`	Nilai stride untuk pergeseran window pooling
3.	`padding`	Jenis padding yang digunakan

Tabel 2.1.1.8 Method Kelas MaxPooling2D

No	Method	Deskripsi
1.	`__init__(pool_size, strides, padding)`	Konstruktor untuk layer average pooling
2.	`_calculate_output_shape(input_shape)`	Menghitung bentuk output setelah operasi pooling
3.	`forward(x, training=False)`	Melakukan operasi average pooling untuk downsampling smooth
4.	`backward(self, grad_output)`	Melakukan backward propagation

e. Kelas FlattenLayer

Kelas FlattenLayer adalah layer yang mengkonversi tensor multi-dimensi menjadi tensor 1D. Layer ini umumnya digunakan sebagai transisi antara layer konvolusional dan layer dense dalam arsitektur CNN, memungkinkan data spatial untuk diproses oleh fully connected layers.

Tabel 2.1.1.9 Method Kelas FlattenLayer

No	Method	Deskripsi
1.	`forward(x, training=False)`	Mengubah tensor multi-dimensi menjadi vektor 1D sambil mempertahankan dimensi batch
2.	`backward(self, grad_output)`	Melakukan backward propagation

f. Kelas DenseLayer

Kelas DenseLayer adalah implementasi layer fully connected yang melakukan transformasi linear pada input dan menerapkan fungsi aktivasi. Layer ini umumnya digunakan pada bagian classifier dalam arsitektur CNN untuk menghasilkan prediksi akhir.

Tabel 2.1.1.10 Atribut Kelas DenseLayer

No	Atribut	Deskripsi
1.	`input_size`	Jumlah neuron input
2.	`output_size`	Jumlah neuron output
3.	`activation`	Nama fungsi aktivasi yang digunakan
4.	`weights['kernel']`	Matriks bobot berukuran (input_size, output_size)
5.	`biases['bias']`	Vektor bias berukuran (output_size,)
6.	`initialized`	Status inisialisasi layer

Tabel 2.1.1.11 Method Kelas DenseLayer

No	Method	Deskripsi
1.	`__init__(input_size, output_size, activation)`	Konstruktor untuk layer dense dengan konfigurasi ukuran dan aktivasi
2.	`_initialize_weights()`	Menginisialisasi bobot menggunakan He initialization
3.	`build(input_shape)`	*Dynamic weight initialization* berdasarkan bentuk input
4.	`_apply_activation(x)`	Menerapkan fungsi aktivasi termasuk softmax untuk klasifikasi
5.	`forward(x, training=False)`	Melakukan transformasi linear dan aktivasi
6.	`backward(self, grad_output)`	Melakukan backward propagation
7.	`set_weights(weights, biases)`	Mengatur bobot dan bias dari sumber eksternal

g. Kelas CNNFromScratch

CNNFromScratch adalah kelas utama yang mengelola seluruh arsitektur CNN, loading dari Keras, dan evaluasi model. Kelas ini menyediakan API tingkat tinggi untuk membangun, memuat, dan menggunakan model CNN from scratch.

Tabel 2.1.1.12 Atribut Kelas CNNFromScratch

No	Atribut	Deskripsi
1.	`layers`	List berisi semua layer dalam model sesuai urutan
2.	`layer_types`	List berisi tipe setiap layer corresponding dengan layers
3.	`compiled`	Status kompilasi/loading model

Tabel 2.1.1.13 Method Kelas CNNFromScratch

No	Method	Deskripsi
1.	`add_layer(layer, layer_type)`	Menambahkan layer ke dalam model CNN
2.	`load_from_keras(keras_model)`	Memuat bobot dan arsitektur dari model Keras
3.	`forward(x)`	Forward propagation melalui semua layer secara berurutan
4.	`backward(grad_output)`	Backward propagation melalui tiap layer
5.	`update_weights()`	Update bobot
6.	`predict(x, batch_size)`	Batch prediction dengan support untuk berbagai ukuran batch
7.	`predict_proba(x, batch_size)`	Prediksi probabilitas dengan batch processing
8.	`evaluate(x, y, batch_size)`	Evaluasi performa model dengan berbagai metrik
9.	`sparse_categorical_crossentro`	Fungsi loss sparse categorical

	`py(y_true, y_pred)`	crossentropy
10.	`sparse_categorical_crossentropy_gradient(self, y_true, y_pred)`	Menghitung gradien dari sparse categorical crossentropy
11.	`train_step(self, x_batch, y_batch)`	Fungsi satu langkah pelatihan
12.	`fit(x_train, y_train, epochs, batch_size, validation_data, verbose)`	Melatih model
13.	`compare_with_keras(keras_model, x_test, y_test)`	Perbandingan komprehensif dengan model Keras

2. Simple RNN (*Recurrent Neural Network*)

a. Kelas BaseLayer

Kelas BaseLayer merupakan kelas abstrak dasar (*base class*) untuk semua layer dalam arsitektur jaringan saraf. Kelas ini dirancang sebagai *blueprint* yang menyediakan struktur umum dan kontrak dasar untuk semua layer turunan seperti *embedding layer*, simple RNN layer, dan layer lainnya. BaseLayer tidak memiliki implementasi aktual untuk metode *forward* dan *backward*, namun menyediakan atribut umum seperti bobot, bias, dan *cache* yang akan digunakan oleh layer-layer turunan.

Tabel 2.1.2.1 Atribut Kelas BaseLayer

No	Atribut	Deskripsi
4.	`weights`	*Dictionary* untuk menyimpan parameter bobot layer (jika ada)
5.	`biases`	*Dictionary* untuk menyimpan parameter bias layer (jika ada)
6.	`cache`	*Dictionary* untuk menyimpan informasi yang dibutuhkan saat *backward* (misalnya output dari *forward*)

Tabel 2.1.2.2 Method Kelas BaseLayer

No	Method	Deskripsi
2.	`__init__()`	Inisialisasi struktur dasar layer dengan *dictionary* kosong untuk *weights*, *biases*, dan *cache*
3.	`forward(x, training=False)`	Fungsi abstrak untuk *forward propagation* dan harus diimplementasikan oleh *subclass*
4.	`backward(grad_output)`	Fungsi abstrak untuk *backward propagation* dan harus diimplementasikan oleh *subclass*

b. Kelas EmbeddingLayer

Kelas EmbeddingLayer adalah layer representasi kata yang digunakan untuk mengonversi token (dalam bentuk indeks integer) menjadi vektor *embedding* berdimensi tetap. Layer ini penting dalam arsitektur jaringan saraf untuk pemrosesan bahasa alami, seperti RNN. EmbeddingLayer menerima ukuran kosakata (vocab_size) dan dimensi embedding (embedding_dim) sebagai parameter. Layer ini juga mendukung inisialisasi bobot embedding secara manual maupun otomatis menggunakan metode Xavier Initialization.

Selama *forward pass*, layer ini mengambil indeks token dari input dan mengambil vektor *embedding* yang sesuai. Dalam *backward pass*, gradien dihitung hanya untuk indeks yang digunakan selama *forward*, menjadikannya efisien dan ringan terhadap memori.

Tabel 2.1.2.3 Atribut Kelas EmbeddingLayer

No	Atribut	Deskripsi
1.	`vocab_size`	Ukuran total kosakata atau jumlah token unik
2.	`embedding_dim`	Daftar fungsi aktivasi untuk setiap layer

3.	`weights['embedding']`	Matriks *embedding* berukuran (vocab_size, embedding_dim) yang merepresentasikan vektor untuk setiap token
4.	`cache['input_indices']`	Input token dalam bentuk indeks, disimpan untuk *backward pass*
5.	`cache['batch_size']`	Ukuran batch saat *forward pass*
6.	`cache['seq_len']`	Panjang urutan input saat *forward pass*
7.	`gradients['embedding']`	Gradien terhadap matriks *embedding* yang dihitung selama *backward pass*

Tabel 2.1.2.4 Method Kelas EmbeddingLayer

No	Method	Deskripsi
1.	`__init__(vocab_size, embedding_dim, weights=None)`	Konstruktor layer. Jika bobot disediakan, digunakan langsung. Jika tidak, bobot diinisialisasi dengan metode Xavier
2.	`forward(x, training=False)`	Mengubah input berupa indeks token menjadi representasi vektor *embedding*, menyimpan input untuk keperluan *backward*
3.	`backward(grad_output)`	Menghitung gradien terhadap matriks *embedding* berdasarkan posisi indeks input yang digunakan

c. Kelas SimpleRNNLayer

SimpleRNNLayer adalah implementasi dasar dari layer *Recurrent Neural Network* (RNN). Layer ini bertanggung jawab memproses data sekuensial dari waktu ke waktu, mempertahankan memori jangka pendek melalui keadaan tersembunyi (*hidden state*). Layer ini menerima input sekuensial berdimensi (*batch_size*, *sequence_length*, *input_dim*) dan mengembalikan output sekuensial atau hanya hasil akhir, tergantung nilai parameter *return_sequences*.

Layer ini menggunakan fungsi aktivasi tanh dan memungkinkan inisialisasi bobot secara manual atau otomatis (menggunakan Xavier initialization).

Tabel 2.1.2.5 Atribut Kelas SimpleRNNLayer

No	Atribut	Deskripsi
1.	`units`	Jumlah unit dalam layer RNN; menentukan dimensi dari *hidden state*
2.	`return_sequences`	Jika *True*, akan mengembalikan semua *hidden state* untuk setiap langkah waktu, dan jika *False*, hanya mengembalikan state terakhir
3.	`weights['kernel']`	Matriks bobot input terhadap *hidden*, berukuran (input_dim, units)
4.	`weights['recurrent_kernel']`	Matriks bobot hidden terhadap *hidden*, berukuran (units, units)
5.	`biases['bias']`	Vektor bias, berukuran (units,)
6.	`cache['input']`	Input sekuensial selama *forward pass*
7.	`cache['hidden_states']`	Daftar *hidden state* dari waktu ke waktu, termasuk state awal
8.	`cache['outputs']`	Daftar output h_t dari setiap langkah waktu
9.	`cache['batch_size']`	Ukuran *batch* input
10.	`cache['seq_len']`	Panjang sekuens input
11.	`gradients['kernel']`	Gradien terhadap *weights*['kernel']
12.	`gradients['recurrent_kernel']`	Gradien terhadap *weights*['recurrent_kernel']
13.	`gradients['bias']`	Gradien terhadap *biases*['bias']

Tabel 2.1.2.6 Method Kelas SimpleRNNLayer

No	Method	Deskripsi
1.	`__init__(units, return_sequences=True, weights=None)`	Konstruktor untuk layer RNN. Jika bobot disediakan, layer akan menggunakannya. Jika tidak, layer

		akan menginisialisasi bobot sendiri saat *forward* pertama kali
2.	`forward(x, training=False)`	Melakukan proses *forward* pada data sekuensial, menghitung *hidden state* pada setiap waktu menggunakan persamaan: h_t = tanh(x_t @ W + h_{t-1} @ U + b)
3.	`backward(grad_output)`	Menghitung propagasi balik terhadap semua bobot dan input, melalui mekanisme *Backpropagation Through Time* (BPTT). Hanya menghitung gradien untuk bobot yang digunakan, tidak memperbarui nilai secara langsung

d. Kelas BidirectionalRNNLayer

BidirectionalRNNLayer adalah implementasi layer *Bidirectional Recurrent Neural Network* (BiRNN) yang memproses input sekuensial dalam dua arah: maju (*forward*) dan mundur (*backward*). Layer ini memungkinkan model untuk menangkap konteks dari masa lalu maupun masa depan pada setiap langkah waktu, yang meningkatkan performa pada tugas seperti analisis sentimen, pengenalan suara, dan lainnya.

Layer ini menerima input sekuensial berdimensi (*batch_size*, *sequence_length*, *input_dim*) dan mengembalikan hasil gabungan dari dua arah, baik sebagai urutan penuh maupun hanya hasil terakhir tergantung nilai *return_sequences*.

Layer ini menggunakan fungsi aktivasi tanh dan dapat menerima bobot inisialisasi secara manual maupun otomatis menggunakan Xavier initialization.

Tabel 2.1.2.7 Atribut Kelas BidirectionalRNNLayer

No	Atribut	Deskripsi
1.	`units`	Total jumlah unit dalam layer (dibagi menjadi forward_units dan backward_units)
2.	`forward_units`	Jumlah unit pada arah *forward*
3.	`backward_units`	Jumlah unit pada arah *backward*
4.	`return_sequences`	Jika *True*, layer mengembalikan seluruh urutan output; jika *False*, hanya mengembalikan gabungan dari output terakhir
5.	`weights['forward_kernel']`	Bobot input untuk arah maju, berdimensi (input_dim, forward_units)
6.	`weights['forward_recurrent']`	Bobot rekuren untuk arah maju, berdimensi (forward_units, forward_units)
7.	`biases['forward_bias']`	Bias untuk arah maju, berdimensi (forward_units,)
8.	`weights['backward_kernel']`	Bobot input untuk arah mundur, berdimensi (input_dim, backward_units)
9.	`weights['backward_recurrent']`	Bobot rekuren untuk arah mundur, berdimensi (backward_units, backward_units)
10.	`biases['backward_bias']`	Bias untuk arah mundur, berdimensi (backward_units,)
11.	`cache['input']`	Input sekuensial selama *forward pass*
12.	`cache['forward_hidden_states']`	Daftar *hidden state* arah maju dari waktu ke waktu, termasuk state awal
13.	`cache['backward_hidden_states']`	Daftar *hidden state* arah mundur dari waktu ke waktu, termasuk state awal
14.	`cache['forward_outputs']`	Daftar output h_t arah maju dari setiap langkah waktu
15.	`cache['backward_outputs']`	Daftar output h_t arah mundur dari setiap langkah waktu

16.	`cache['batch_size']`	Ukuran batch input
17.	`cache['seq_len']`	Panjang sekuens input
18.	`gradients['forward_kernel']`	Gradien terhadap weights['forward_kernel']
19.	`gradients['forward_recurrent']`	Gradien terhadap weights['forward_recurrent']
20.	`gradients['forward_bias']`	Gradien terhadap biases['forward_bias']
21.	`gradients['backward_kernel']`	Gradien terhadap weights['backward_kernel']
22.	`gradients['backward_recurrent']`	Gradien terhadap weights['backward_recurrent']
23.	`gradients['backward_bias']`	Gradien terhadap biases['backward_bias']

Tabel 2.1.2.8 Method Kelas BidirectionalRNNLayer

No	Method	Deskripsi
1.	`__init__(units, return_sequences=True, weights=None)`	Konstruktor layer BiRNN. Mengatur jumlah unit, arah propagasi, serta menerima bobot inisialisasi opsional. Jika bobot tidak disediakan, maka bobot akan diinisialisasi otomatis saat forward() pertama kali dipanggil
2.	`forward(x, training=False)`	Melakukan proses propagasi maju dan mundur pada input sekuensial. Menghitung *hidden state* untuk kedua arah, lalu menggabungkan hasil output keduanya
3.	`backward(grad_output)`	Menghitung *backpropagation through time* (BPTT) untuk kedua arah. Mengembalikan gradien terhadap input serta menyimpan gradien terhadap semua parameter bobot dan bias

e. Kelas DropoutLayer

DropoutLayer adalah implementasi lapisan *dropout* yang sering digunakan dalam jaringan saraf untuk mencegah *overfitting*. Layer ini secara acak mengatur sebagian input menjadi nol selama pelatihan, yang membantu mengurangi ketergantungan antar neuron dan mendorong model untuk mempelajari fitur yang lebih *robust*.

Tabel 2.1.2.9 Atribut Kelas DropoutLayer

No	Atribut	Deskripsi
1.	`rate`	Tingkat *dropout*, yaitu fraksi input yang akan diatur menjadi nol. Nilai ini harus antara 0 dan 1
2.	`cache['mask']`	Masker biner yang dihasilkan selama *forward pass* (saat pelatihan) dan digunakan kembali selama *backward pass*

Tabel 2.1.2.10 Method Kelas DropoutLayer

No	Method	Deskripsi
1.	`__init__(self, rate)`	Inisialisasi DropoutLayer dengan tingkat dropout tertentu
2.	`forward(self, x, training=False)`	Melakukan operasi *forward propagation*. Saat mode *training* (*training* = *True*), layer ini menerapkan *dropout* pada input x dengan probabilitas *rate*. Jika bukan dalam mode *training*, layer hanya meneruskan input apa adanya
3.	`backward(self, grad_output)`	Melakukan operasi *backward propagation*. Gradien dari output dikalikan dengan *mask dropout* yang sama seperti pada f*orward propagation*.

f. Kelas DenseLayer

DenseLayer adalah implementasi dari lapisan *fully-connected* (dense) dalam jaringan saraf tiruan. Layer ini menerima input, melakukan transformasi linier menggunakan bobot dan bias, dan menerapkan fungsi aktivasi tertentu.

Tabel 2.1.2.11 Atribut Kelas DenseLayer

No	Atribut	Deskripsi
1.	`layers`	Jumlah neuron dalam *dense layer* (*output size*).
2.	`activation_name`	Nama fungsi aktivasi yang digunakan (misal: 'relu', 'softmax', dan lain lain).
3.	`activation`	Fungsi aktivasi yang diambil dari modul Activation.
4.	`d_activation`	Turunan fungsi aktivasi, untuk proses *backward*.
5.	`weights`	Menyimpan kernel (bobot) dan bias layer.
6.	`cache`	Menyimpan nilai input dan hasil perhitungan untuk *backward pass*.
7.	`gradients`	Menyimpan gradien terhadap kernel dan bias setelah backward.

Tabel 2.1.2.12 Method Kelas DenseLayer

No	Method	Deskripsi
1.	`__init__(self, units, activation='softmax', weights=None)`	Konstruktor untuk inisialisasi layer. Menyimpan parameter jumlah unit, fungsi aktivasi, dan (opsional) bobot awal.
2.	`forward(self, x, training=False)`	Melakukan propagasi maju, yaitu menghitung output lapisan dari input yang diberikan. Seluruh nilai penting seperti input mentah, output linear, dan hasil aktivasi disimpan dalam cache untuk digunakan saat backpropagation.

3.	`backward(self, grad_output)`	Melakukan propagasi mundur, menghitung gradien dari output terhadap input dan parameter lapisan.

g. Kelas RNNFromScratch

Kelas utama untuk membangun, melatih, dan menyimpan model RNN secara manual (tanpa framework training seperti Keras/TensorFlow). Kelas ini mendukung berbagai jenis layer seperti embedding, RNN, bidirectional RNN, dropout, dan dense, serta fungsi-fungsi training dan evaluasi.

Tabel 2.1.2.13 Atribut Kelas RNNFromScratch

No	Atribut	Deskripsi
1.	`layers`	List dari layer-layer *neural network* yang ditambahkan.
2.	`learning_rate`	*Float*, laju pembelajaran untuk *update* bobot.
3.	`encoder`	OneHotEncoder opsional untuk encoding label saat *training*

Tabel 2.1.2.14 Method Kelas RNNFromScratch

No	Atribut	Deskripsi
1.	`add_embedding_layer(self, vocab_size, embedding_dim, weights=None)`	Menambahkan layer sesuai jenis ke dalam model.
2.	`add_simple_rnn_layer(self, units, return_sequences=True, weights=None)`	
3.	`add_bidirectional_rnn_layer(self, units, return_sequences=True, weights=None)`	
4.	`add_dropout_layer(self, rate)`	
5.	`add_dense_layer(self,`	

	`units, activation='softmax', weights=None)`	
6.	`forward(self, x, training=False):`	Melakukan propagasi maju ke seluruh layer
7.	`backward(grad_output)`	Melakukan propagasi balik (gradien) ke semua layer yang mendukung
8.	`categorical_crossentropy_loss(y_true, y_pred)`	Menghitung *loss* klasifikasi untuk prediksi *multiclass*
9.	`categorical_crossentropy_gradient(y_true, y_pred)`	Menghitung gradien dari *loss* terhadap output
10.	`update_weights()`	Update bobot dan bias berdasarkan gradien dan learning rate
11.	`train_step(x_batch, y_batch)`	Satu langkah *training* (*forward → backward → update*).
12.	`fit(self, X_train, y_train, epochs=10, batch_size=32, validation_data=None, verbose=1)`	Melatih model dengan data *training* dan *validasi* secara *batch*
13.	`predict(x, batch_size=32)`	Menghasilkan prediksi dari input menggunakan model
14.	`save_model(filepath)`	Menyimpan struktur dan bobot model ke file JSON
15.	`load_model(filepath)`	Memuat model dari file JSON
16.	`load_keras_model(keras_model)`	Mengkonversi model Keras menjadi RNNFromScratch
17.	`plot_history(history)`	Menampilkan grafik hasil *training* (*loss* & F1-score)

3. LSTM

a. Kelas BaseLayer

Kelas BaseLayer merupakan kelas abstrak dasar (*base class*) yang menyediakan struktur umum untuk semua layer dalam arsitektur LSTM. Kelas ini dirancang sebagai *blueprint* yang menyediakan kontrak dasar dan atribut umum yang akan digunakan oleh semua layer turunan.

Tabel 2.1.3.1 Atribut Kelas BaseLayer

No	Atribut	Deskripsi
1.	`weights`	*Dictionary* untuk menyimpan parameter bobot layer (*kernel*, *recurrent kernel*, dll)
2.	`biases`	*Dictionary* untuk menyimpan parameter bias *layer*
3.	`cache`	*Dictionary* untuk menyimpan informasi *intermediate* yang dibutuhkan saat *backward propagation*

Tabel 2.1.3.2 Method Kelas BaseLayer

No	Method	Deskripsi
1.	`__init__()`	Inisialisasi struktur dasar *layer* dengan *dictionary* kosong untuk *weights*, *biases*, dan *cache*
2.	`forward(x, training=False)`	Fungsi abstrak untuk *forward propagation* yang harus diimplementasikan oleh *subclass*
3.	`backward(grad_output)`	Fungsi abstrak untuk *backward propagation* yang harus diimplementasikan oleh *subclass*

b. Kelas EmbeddingLayer

EmbeddingLayer adalah *layer* yang mengkonversi token (dalam bentuk indeks integer) menjadi vektor *embedding* berdimensi tetap. Layer ini menggunakan *lookup table* yang memetakan setiap token ke dalam ruang vektor *dense*,

memungkinkan model untuk mempelajari representasi semantik dari kata-kata.

Tabel 2.1.3.3 Atribut Kelas EmbeddingLayer

No	Atribut	Deskripsi
1.	`vocab_size`	Ukuran total kosakata atau jumlah token unik dalam dataset
2.	`embedding_dim`	Dimensi vektor *embedding* untuk setiap token
3.	`weights['embedding']`	Matriks *embedding* berukuran (*vocab_size*, *embedding_dim*) yang diinisialisasi dengan Xavier initialization
4.	`cache['input_indices']`	Input token dalam bentuk indeks, disimpan untuk *backward pass*
5.	`cache['batch_size']`	Ukuran *batch* saat *forward pass*
6.	`cache['seq_len']`	Panjang urutan input saat *forward pass*

Tabel 2.1.3.4 Method Kelas EmbeddingLayer

No	Method	Deskripsi
1.	`__init__(vocab_size, embedding_dim, weights=None)`	Konstruktor yang menginisialisasi dimensi *embedding* dan bobot (manual atau Xavier)
2.	`forward(x, training=False)`	Mengubah input berupa indeks token menjadi vektor *embedding* melalui *lookup operation*
3.	`backward(grad_output)`	Menghitung gradien terhadap matriks *embedding* hanya untuk indeks yang digunakan (*sparse update*)

c. Kelas LSTMLayer

LSTMLayer adalah implementasi inti dari *Long Short-Term Memory* yang memproses data sekuensial dengan kemampuan mengatasi *vanishing gradient problem*. Layer ini menggunakan tiga gate mechanism (*input*, *forget*, *output*) dan *cell state* untuk mengontrol aliran informasi.

Tabel 2.1.3.5 Atribut Kelas LSTMLayer

No	Atribut	Deskripsi
1.	`units`	Jumlah *hidden units* dalam *layer* LSTM, menentukan dimensi *cell state* dan *hidden state*
2.	`return_sequences`	Boolean untuk menentukan output (*True*: semua *timesteps*, *False*: *timestep* terakhir)
3.	`weights['kernel']`	Matriks bobot *input-to-hidden* berukuran (input_dim, 4*units) untuk 4 *gates*
4.	`weights['recurrent_kernel']`	Matriks bobot *hidden-to-hidden* berukuran (units, 4*units) untuk *recurrent connections*
5.	`biases['bias']`	Vektor bias berukuran (4*units,) untuk semua gates
6.	`cache['hidden_states']`	*List hidden states* dari setiap *timestep* termasuk state awal
7.	`cache['cell_states']`	*List cell states* dari setiap *timestep* untuk *backward propagation*
8.	`cache['gates_history']`	*History* dari semua *gate activations* untuk setiap *timestep*

Tabel 2.1.3.6 Method Kelas LSTMLayer

No	Method	Deskripsi
1.	`__init__(units, return_sequences=True, weights=None)`	Konstruktor untuk inisialisasi parameter LSTM dengan bobot opsional
2.	`_sigmoid(x)`	Fungsi aktivasi sigmoid dengan *numerical clipping* untuk stabilitas (-500, 500)
3.	`_tanh(x)`	Fungsi aktivasi tanh dengan *numerical clipping* untuk mencegah *overflow*
4.	`forward(x, training=False)`	Implementasi *forward propagation* LSTM dengan *gate mechanism* dan *state updates*
5.	`backward(grad_output)`	Implementasi *Backpropagation Through Time*

		(BPTT) untuk menghitung gradien

d. Kelas BidirectionalLSTMLayer

BidirectionalLSTMLayer memproses input sekuensial dalam dua arah secara simultan: *forward* (kiri ke kanan) dan *backward* (kanan ke kiri). Hasil dari kedua arah kemudian dikoncatenasi untuk memberikan representasi yang lebih kaya akan konteks.

Tabel 2.1.3.7 Atribut Kelas BidirectionalRNNLayer

No	Atribut	Deskripsi
1.	`units`	Total jumlah *output units* (dibagi menjadi *forward_units* dan *backward_units*)
2.	`forward_units`	Jumlah *units* untuk LSTM arah *forward* (*units* // 2)
3.	`backward_units`	Jumlah *units* untuk LSTM arah *backward* (*units* // 2)
4.	`weights['forward_kernel']`	Matriks bobot input untuk arah *forward*
5.	`weights['forward_recurrent']`	Matriks bobot recurrent untuk arah *forward*
6.	`weights['backward_kernel']`	Matriks bobot input untuk arah *backward*
7.	`weights['backward_recurrent']`	Matriks bobot recurrent untuk arah *backward*
8.	`cache['forward_outputs']`	*Output sequences* dari *forward* LSTM
9.	`cache['backward_outputs']`	*Output sequences* dari *backward* LSTM

Tabel 2.1.3.8 Method Kelas BidirectionalLSTMLayer

No	Method	Deskripsi

1.	`__init__(units, return_sequences=True, weights=None)`	Konstruktor yang membagi *units* untuk kedua arah dan inisialisasi bobot
2.	`forward(x, training=False)`	*Forward pass* untuk kedua arah dengan *concatenation* hasil *output*
3.	`backward(grad_output)`	*Backward pass* yang menghitung gradien untuk kedua arah secara terpisah

e. Kelas DropoutLayer

DropoutLayer adalah teknik regularisasi yang secara acak "mematikan" sebagian neuron selama *training* untuk mencegah *overfitting* dan meningkatkan generalisasi model.

Tabel 2.1.3.9 Atribut Kelas DropoutLayer

No	Atribut	Deskripsi
1.	`rate`	Probabilitas *dropout* (0.0-1.0), menentukan fraksi input yang akan di-nol-kan
2.	`cache['mask']`	*Binary mask* yang digunakan untuk *dropout*, disimpan untuk *backward pass*

Tabel 2.1.3.10 Method Kelas DropoutLayer

No	Method	Deskripsi
1.	`__init__(rate)`	Inisialisasi dengan tingkat *dropout* tertentu
2.	`forward(x, training=False)`	Aplikasi *dropout* hanya saat *training*, dengan *scaling* (1-*rate*) untuk kompensasi
3.	`backward(grad_output)`	Propagasi gradien melalui *mask* yang sama seperti *forward pass*

f. Kelas DenseLayer

DenseLayer adalah layer yang melakukan transformasi linier penuh antara input dan output, umumnya digunakan sebagai *classifier layer* terakhir dalam arsitektur LSTM.

Tabel 2.1.3.11 Atribut Kelas DenseLayer

No	Atribut	Deskripsi
1.	`units`	Jumlah neuron *output* dalam *dense layer*
2.	`activation_name`	Nama fungsi aktivasi yang digunakan (softmax, relu, dll)
3.	`activation`	*Function object* dari aktivasi yang dipilih
4.	`d_activation`	*Function object* untuk *derivative* aktivasi (*backward pass*)
5.	`weights['kernel']`	Matriks bobot berukuran (input_dim, units)
6.	`weights['bias']`	Vektor bias berukuran (units,)

Tabel 2.1.3.12 Method Kelas DenseLayer

No	Method	Deskripsi
1.	`__init__(units, activation='softmax', weights=None)`	Konstruktor dengan inisialisasi parameter dan fungsi aktivasi
2.	`build(input_shape)`	*Dynamic weight initialization* berdasarkan ukuran input
3.	`forward(x, training=False)`	Linear transformation (x @ W + b) diikuti aktivasi
4.	`backward(grad_output)`	Menghitung gradien terhadap weights, bias, dan input

g. Kelas LSTMFromScratch

LSTMFromScratch adalah kelas orkestrator yang mengelola seluruh arsitektur LSTM, *training loop*, dan evaluasi model. Kelas ini menyediakan API tingkat tinggi untuk membangun, melatih, dan menggunakan model LSTM.

Tabel 2.1.3.13 Atribut Kelas LSTMFromScratch

No	Atribut	Deskripsi
1.	`layers`	List berisi semua layer dalam model sesuai urutan

2.	`learning_rate`	*Learning rate* untuk *gradient descent optimization*
3.	`encoder`	OneHotEncoder untuk *handling categorical labels*

Tabel 2.1.3.14 Method Kelas LSTMFromScratch

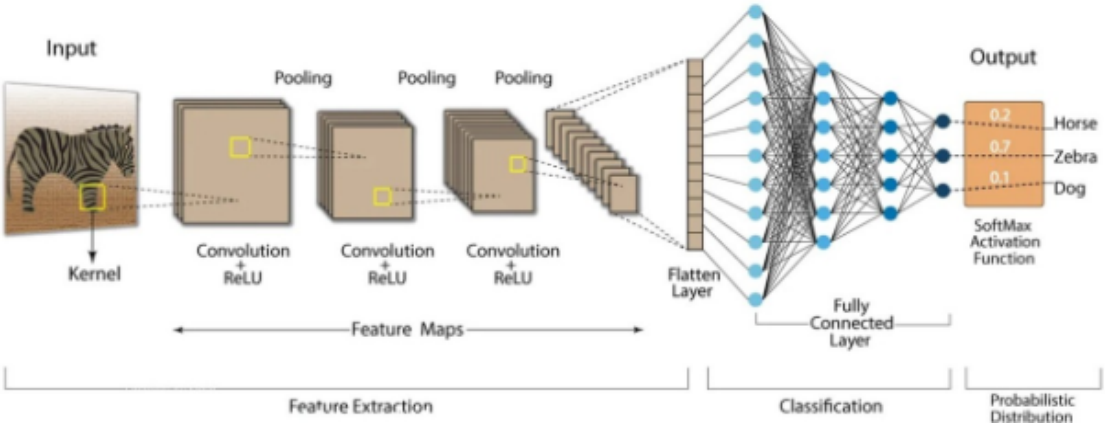
No	Atribut	Deskripsi
1.	`add_*_layer()`	*Suite method* untuk menambahkan berbagai jenis layer ke model
2.	`forward(x, training=False)`	*Forward propagation* melalui semua *layer* secara berurutan
3.	`backward(grad_output)`	*Backward propagation* melalui semua *layer* secara terbalik
4.	`fit(X_train, y_train, ...)`	*Training loop* dengan *mini-batch* SGD, *validation*, dan *metrics tracking*
5.	`predict(x, batch_size=32)`	*Batch prediction* dengan *support* untuk berbagai ukuran *batch*
6.	`load_keras_model(keras_model)`	*Transfer learning* dari model Keras ke implementasi *from scratch*
7.	`compare_with_keras(...)`	*Utility* untuk membandingkan performa dengan model Keras

B. *Forward Propagation*

1. CNN

CNN (Convolutional Neural Network) adalah arsitektur neural network yang dirancang khusus untuk memproses data dengan struktur grid seperti gambar. Berbeda dengan feedforward network tradisional, CNN menggunakan operasi konvolusi untuk mengekstrak fitur spatial secara hierarkis dari data input. Untuk masalah image classification ini, arsitektur CNN terdiri dari beberapa jenis layer yang bekerja secara berurutan: convolutional layers untuk ekstraksi fitur, pooling

layers untuk downsampling, flatten layer untuk transisi, dan dense layers untuk klasifikasi.



Gambar 2.1 Arsitektur Convolutional Neural Network

Pada implementasinya di kelas CNNFromScratch, forward propagation merupakan sequential pipeline dimana output dari satu layer menjadi input untuk layer selanjutnya. Model melakukan iterasi melalui semua layers yang telah didefinisikan dan memanggil method forward() dari masing-masing layer.

```
Input: x_0 ∈ R^(B×H×W×C)                          [gambar CIFAR-10]
↓
Conv2D: x_1 = Conv2D(x_0)                         [B×H'×W'×F1]
↓
Pooling: x_2 = Pooling(x_1)                       [B×H''×W''×F1]
↓
...
↓
Flatten: x_n = Flatten(x_{n-1})                   [B×Features]
↓
Dense: y = Dense(x_n)                             [B×Classes]

Dimana:
B = batch_size, H,W = height,width
C = channels, F = filters
Classes = jumlah kelas CIFAR-10 (10)
```

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

Berikut mekanisme forward propagation untuk masing-masing layer dalam arsitektur CNN:

1. ***Convolutional Layer* - Ekstraksi Fitur Spatial**

   Convolutional layer merupakan komponen inti CNN yang bertanggung jawab mengekstrak fitur spatial dari input gambar melalui operasi konvolusi. Layer ini menggunakan sekumpulan filter (kernel) yang dapat dipelajari untuk mendeteksi berbagai pola seperti edges, textures, dan shapes pada berbagai posisi dalam gambar.

   a. **Konsep Dasar Operasi Konvolusi**

      Operasi konvolusi dilakukan dengan menggeser filter di atas input image dengan stride tertentu. Pada setiap posisi, dilakukan element-wise multiplication antara filter dan region input yang bersesuaian, kemudian dijumlahkan untuk menghasilkan satu nilai pada feature map. Proses ini diulang untuk semua posisi dan semua filter, menghasilkan multiple feature maps.

```
Operasi Konvolusi:
Output[b,h,w,f] = σ(∑∑∑ Input[b, h*s+i, w*s+j, c] *
Kernel[i,j,c,f] + bias[f])
Dimana:
   ● b = batch index
   ● h,w = spatial coordinates di output
   ● f = filter index
   ● s = stride value
   ● σ = activation function (ReLU, tanh, sigmoid)
   ● i,j,c = kernel dimensions (height, width, channels)
```

   b. **Parameter dan Konfigurasi Layer**

      Input Shape: (batch_size, height, width, channels)

      ● Untuk CIFAR-10: (batch_size, 32, 32, 3)

      Kernel/Filter: (kernel_height, kernel_width, input_channels, output_filters)

      ● Contoh: (3, 3, 3, 32) untuk 32 filter 3x3 pada input RGB

      Stride: Menentukan pergeseran filter

      ● Stride = 1: overlap maksimum, output size hampir sama dengan input

- Stride = 2: overlap 50%, output size setengah dari input

Padding: Menangani edge effects

- 'valid': no padding, output size berkurang
- 'same': zero padding, output size dipertahankan

c. Perhitungan Output Shape

```
def _calculate_output_shape(self):
    h, w, c = self.input_shape
    kh, kw = self.kernel_size
    sh, sw = self.strides

    if self.padding == 'same':
        oh = int(np.ceil(h / sh))
        ow = int(np.ceil(w / sw))
    else:  # valid padding
        oh = int((h - kh) / sh) + 1
        ow = int((w - kw) / sw) + 1

    return (oh, ow, self.filters)
```

d. Implementasi Padding

Untuk menjaga dimensi spatial atau mengontrol ukuran output, padding dapat ditambahkan di sekitar input. Padding 'same' mempertahankan dimensi input, sedangkan 'valid' menghasilkan output yang lebih kecil.

```
def _add_padding(self, x):
    if self.padding == 'same':
        h, w = x.shape[1], x.shape[2]
        kh, kw = self.kernel_size
        sh, sw = self.strides

        pad_h = max(0, (self.output_shape[0] - 1) * sh + kh
- h)
        pad_w = max(0, (self.output_shape[1] - 1) * sw + kw
- w)

        pad_top, pad_bottom = pad_h // 2, pad_h - pad_h //
2
        pad_left, pad_right = pad_w // 2, pad_w - pad_w //
2

        x = np.pad(x, ((0, 0), (pad_top, pad_bottom),
                       (pad_left, pad_right), (0, 0)))
    return x
```

e. Fungsi Aktivasi

Setelah operasi konvolusi linear, fungsi aktivasi diterapkan untuk menambahkan non-linearity:

```
def _apply_activation(self, x):
    if self.activation == 'relu':
        return np.maximum(0, x)  # ReLU: f(x) = max(0, x)
    elif self.activation == 'tanh':
        return np.tanh(x)  # Tanh: f(x) = (e^x - e^-x)/(e^x
+ e^-x)
    elif self.activation == 'sigmoid':
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))  #
Sigmoid: f(x) = 1/(1+e^-x)
    else:
        return x
```

f. Forward Propagation

```
def forward(self, x, training=False):
    batch_size = x.shape[0]
    x_padded = self._add_padding(x)

    oh, ow, _ = self.output_shape
    output = np.zeros((batch_size, oh, ow, self.filters))

    kh, kw = self.kernel_size
    sh, sw = self.strides

    # Operasi konvolusi untuk setiap batch, filter, dan
posisi spatial
    for b in range(batch_size):
        for f in range(self.filters):
            for h in range(oh):
                for w in range(ow):
                    h_start = h * sh
                    h_end = h_start + kh
                    w_start = w * sw
                    w_end = w_start + kw

                    if h_end <= x_padded.shape[1] and w_end
<= x_padded.shape[2]:
                        # Ekstrak region dari input yang
akan di-konvolusi
                        region = x_padded[b, h_start:h_end,
w_start:w_end, :]

                        # Hitung output sebagai dot product
dengan kernel + bias
                        conv_result = np.sum(region *
self.weights['kernel'][:, :, :, f])
                        output[b, h, w, f] = conv_result +
self.biases['bias'][f]

    return self._apply_activation(output)
```

g. Backward Propagation Convolutional Layer

Backward propagation pada convolutional layer melibatkan perhitungan gradien terhadap tiga komponen: input, kernel weights, dan bias. Proses ini menggunakan chain rule untuk mempropagasi error dari layer berikutnya kembali ke layer sebelumnya.

- Gradien untuk Kernel (∂L/∂K):

  Gradien terhadap kernel dihitung dengan melakukan konvolusi antara input dan gradient output:

  $\partial L/\partial K[i,j,c,f] = \sum\sum \text{Input}[b, h*s+i, w*s+j, c] * \partial L/\partial \text{Output}[b,h,w,f]$

- Gradien untuk Bias (∂L/∂b):

  Gradien terhadap bias adalah jumlah dari semua gradient output untuk filter yang bersangkutan:

  $\partial L/\partial b[f] = \sum\sum\sum \partial L/\partial \text{Output}[b,h,w,f]$

- Gradien untuk Input (∂L/∂Input):

  Gradien terhadap input dihitung dengan melakukan full convolution menggunakan kernel yang di-flip:

  $\partial L/\partial \text{Input}[b,h,w,c] = \sum\sum\sum K[i,j,c,f] * \partial L/\partial \text{Output}[b, (h-i)/s, (w-j)/s, f]$

```
def backward(self, grad_output):
        if 'input' not in self.cache:
            raise ValueError("Forward pass harus
dilakukan terlebih dahulu")

        input_data = self.cache['input']
        batch_size, input_h, input_w, input_c =
input_data.shape
        _, output_h, output_w, output_c =
grad_output.shape

        grad_kernel =
np.zeros_like(self.weights['kernel'])
        grad_bias = np.zeros_like(self.biases['bias'])
        grad_input = np.zeros_like(input_data)

        input_padded = self._add_padding(input_data)

        kh, kw = self.kernel_size
        sh, sw = self.strides
```

```
        for b in range(batch_size):
            for f in range(self.filters):
                for h in range(output_h):
                    for w in range(output_w):
                        h_start = h * sh
                        h_end = h_start + kh
                        w_start = w * sw
                        w_end = w_start + kw

                        if h_end <=
input_padded.shape[1] and w_end <=
input_padded.shape[2]:
                            region = input_padded[b,
h_start:h_end, w_start:w_end, :]
                            grad_kernel[:, :, :, f] +=
region * grad_output[b, h, w, f]

                            grad_bias[f] +=
grad_output[b, h, w, f]

                            grad_input_region =
self.weights['kernel'][:, :, :, f] * grad_output[b, h,
w, f]

                            if self.padding == 'same':
                                pad_h = max(0, (output_h
- 1) * sh + kh - input_h)
                                pad_w = max(0, (output_w
- 1) * sw + kw - input_w)
                                pad_top = pad_h // 2
                                pad_left = pad_w // 2

                                input_h_start = max(0,
h_start - pad_top)
                                input_h_end =
min(input_h, h_end - pad_top)
                                input_w_start = max(0,
w_start - pad_left)
                                input_w_end =
min(input_w, w_end - pad_left)

                                if (input_h_start <
input_h_end and input_w_start < input_w_end):
                                    kernel_h_start =
max(0, pad_top - h_start)
                                    kernel_h_end =
kernel_h_start + (input_h_end - input_h_start)
                                    kernel_w_start =
max(0, pad_left - w_start)
                                    kernel_w_end =
kernel_w_start + (input_w_end - input_w_start)

                                    grad_input[b,
input_h_start:input_h_end, input_w_start:input_w_end, :]
```

```
+= \

grad_input_region[kernel_h_start:kernel_h_end,
kernel_w_start:kernel_w_end, :]
                            else:
                                grad_input[b,
h_start:h_end, w_start:w_end, :] += grad_input_region

        self.gradients['kernel'] = grad_kernel
        self.gradients['bias'] = grad_bias

        return grad_input
```

2. *Pooling Layer - Spatial Downsampling*

Pooling layer melakukan downsampling pada feature maps untuk mengurangi dimensi spatial sambil mempertahankan informasi penting. Hal ini membantu mengurangi computational cost, mencegah overfitting, dan memberikan translation invariance. Terdapat dua jenis pooling yang diimplementasikan: Max Pooling dan Average Pooling.

a. Max Pooling

Max Pooling mengambil nilai maksimum dalam setiap window pooling, mempertahankan fitur yang paling kuat dan memberikan robustness terhadap small translations:

```
MaxPool[b,h,w,c] = max(Input[b, h*s+i, w*s+j, c])
                   i,j ∈ pooling_window
```

Karakteristik Max Pooling:

- Feature Detection: Mempertahankan strongest activations
- Translation Invariance: Robust terhadap small shifts
- Sparsity: Menghasilkan representasi yang sparse
- Sharp Features: Mempertahankan edge dan texture yang jelas

b. Average Pooling

Average Pooling menghitung rata-rata nilai dalam window pooling, memberikan representasi yang lebih smooth dan mempertahankan informasi global:

```
AvgPool[b,h,w,c] = mean(Input[b, h*s+i, w*s+j, c])
                        i,j ∈ pooling_window
```

Karakteristik Average Pooling:

- Smooth Representation: Menghasilkan output yang halus
- Information Preservation: Mempertahankan lebih banyak informasi
- Global Context: Memberikan representasi yang lebih global
- Less Discriminative: Kurang efektif untuk feature detection

c. Perhitungan Output Shape Pooling

```
def _calculate_output_shape(self, input_shape):
    h, w, c = input_shape
    ph, pw = self.pool_size
    sh, sw = self.strides

    if self.padding == 'same':
        oh = int(np.ceil(h / sh))
        ow = int(np.ceil(w / sw))
    else:  # valid padding
        oh = int((h - ph) / sh) + 1
        ow = int((w - pw) / sw) + 1

    return (oh, ow, c)
```

d. Implementasi *Max Pooling*

```
def forward(self, x, training=False):
    batch_size, h, w, c = x.shape
    oh, ow, _ = self._calculate_output_shape((h, w, c))
    output = np.zeros((batch_size, oh, ow, c))

    ph, pw = self.pool_size
    sh, sw = self.strides

    # Operasi max pooling untuk setiap batch, channel, dan
```

```
posisi spatial
    for b in range(batch_size):
        for ch in range(c):
            for i in range(oh):
                for j in range(ow):
                    h_start = i * sh
                    h_end = min(h_start + ph, h)
                    w_start = j * sw
                    w_end = min(w_start + pw, w)

                    if h_end > h_start and w_end > w_start:
                        # Ambil nilai maksimum dalam window
pooling
                        pooling_region = x[b,
h_start:h_end, w_start:w_end, ch]
                        output[b, i, j, ch] =
np.max(pooling_region)

    return output
```

e. Implementasi *Average Pooling*

```
def forward(self, x, training=False):
    batch_size, h, w, c = x.shape
    oh, ow, _ = self._calculate_output_shape((h, w, c))
    output = np.zeros((batch_size, oh, ow, c))

    ph, pw = self.pool_size
    sh, sw = self.strides

    # Operasi average pooling untuk setiap batch, channel,
dan posisi spatial
    for b in range(batch_size):
        for ch in range(c):
            for i in range(oh):
                for j in range(ow):
                    h_start = i * sh
```

```
                    h_end = min(h_start + ph, h)
                    w_start = j * sw
                    w_end = min(w_start + pw, w)

                    if h_end > h_start and w_end > w_start:
                        # Hitung nilai rata-rata dalam
window pooling
                        pooling_region = x[b,
h_start:h_end, w_start:w_end, ch]
                        output[b, i, j, ch] =
np.mean(pooling_region)

    return output
```

f. Backward Propagation Pooling Layers

Backward propagation pada pooling layers memiliki karakteristik yang berbeda antara Max Pooling dan Average Pooling karena sifat operasinya yang berbeda.

- Max Pooling Backward:

  Pada max pooling, gradient hanya dipropagasi ke posisi yang memiliki nilai maksimum dalam pooling window. Posisi lain mendapat gradient nol karena tidak berkontribusi terhadap output.

  $\partial L/\partial Input[i,j] = \{\partial L/\partial Output[h,w]\}$

  jika Input[i,j] = max(pooling_window)

  otherwise 0

- Average Pooling Backward:

  Pada average pooling, gradient didistribusikan secara merata ke semua posisi dalam pooling window:

  $\partial L/\partial Input[i,j] = \partial L/\partial Output[h,w] / (pool\_height \times pool\_width)$

```
def backward(self, grad_output):
        if 'input' not in self.cache or 'mask' not in
self.cache:
            raise ValueError("Forward pass harus
dilakukan terlebih dahulu")
```

```
        input_data = self.cache['input']
        mask = self.cache['mask']

        grad_input = np.zeros_like(input_data)
        batch_size, oh, ow, c = grad_output.shape
        ph, pw = self.pool_size
        sh, sw = self.strides

        for b in range(batch_size):
            for ch in range(c):
                for i in range(oh):
                    for j in range(ow):
                        h_start = i * sh
                        h_end = min(h_start + ph,
input_data.shape[1])
                        w_start = j * sw
                        w_end = min(w_start + pw,
input_data.shape[2])

                        max_mask = mask[b, i, j, ch]
                        if max_mask is not None:
                            max_h, max_w = max_mask
                            grad_input[b, max_h,
max_w, ch] += grad_output[b, i, j, ch]

        return grad_input


def backward(self, grad_output):
        if 'input' not in self.cache:
            raise ValueError("Forward pass harus
dilakukan terlebih dahulu")

        input_data = self.cache['input']
        grad_input = np.zeros_like(input_data)

        batch_size, oh, ow, c = grad_output.shape
        ph, pw = self.pool_size
        sh, sw = self.strides

        for b in range(batch_size):
            for ch in range(c):
                for i in range(oh):
                    for j in range(ow):
                        h_start = i * sh
                        h_end = min(h_start + ph,
input_data.shape[1])
                        w_start = j * sw
                        w_end = min(w_start + pw,
input_data.shape[2])

                        pool_size = (h_end - h_start)
* (w_end - w_start)
                        if pool_size > 0:
```

```
                                grad_per_element =
grad_output[b, i, j, ch] / pool_size
                                grad_input[b,
h_start:h_end, w_start:w_end, ch] += grad_per_element

        return grad_input
```

3. **Flatten Layer - Dimensional Transition**

   Flatten layer mengkonversi feature maps multi-dimensional menjadi vector 1D untuk dapat diproses oleh dense layers. Layer ini berfungsi sebagai bridge antara convolutional layers yang memproses data spatial dan dense layers yang memproses data linear.

   a. **Operasi Flatten**

   ```
   Flatten: (B, H, W, C) → (B, H×W×C)
   ```

   Contoh:

   - Input: (32, 8, 8, 64) → Output: (32, 4096)
   - Input: (10, 4, 4, 128) → Output: (10, 2048)

   b. **Implementasi Flatten**

   ```
   def forward(self, x, training=False):
       batch_size = x.shape[0]
       flattened_size = np.prod(x.shape[1:])
       return x.reshape(batch_size, flattened_size)
   ```

   c. **Backward Propagation Flatten Layer**

   Backward propagation pada flatten layer sangat sederhana karena operasinya hanya melakukan reshaping tanpa mengubah nilai. Gradient cukup di-reshape kembali ke bentuk input original.

   ```
   ∂L/∂Input = reshape(∂L/∂Output, input_shape)
   ```

   ```
   def backward(self, grad_output):
           if 'input_shape' not in self.cache:
   ```

```
            raise ValueError("Forward pass harus dilakukan
terlebih dahulu")

        original_shape = self.cache['input_shape']
        return grad_output.reshape(original_shape)
```

4. Dense Layer - Classification Output

Dense layer (fully connected layer) melakukan transformasi linear pada input dan menerapkan fungsi aktivasi. Pada layer terakhir, biasanya menggunakan aktivasi softmax untuk menghasilkan distribusi probabilitas untuk klasifikasi multi-class.

a. Linear Transformation

```
z = x @ W + b
Dimana:
```

- x ∈ R^(batch_size × input_features): input features
- W ∈ R^(input_features × output_units): weight matrix
- b ∈ R^(output_units): bias vector
- z ∈ R^(batch_size × output_units): pre-activation output

b. Inisialisasi Bobot (He Initialization)

```
def _initialize_weights(self):
    # He initialization untuk ReLU activations
    std = np.sqrt(2.0 / self.input_size)
    self.weights['kernel'] = np.random.normal(0, std,
                             (self.input_size,
self.output_size))
    self.biases['bias'] = np.zeros((1, self.output_size))
```

c. Fungsi Aktivasi Dense Layer

```
if self.activation == 'relu':
        return np.maximum(0, x)
```

```
elif self.activation == 'tanh':
        return np.tanh(x)
elif self.activation == 'sigmoid':
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))
elif self.activation == 'softmax':
        exp_x = np.exp(x - np.max(x, axis=1,
keepdims=True))
        return exp_x / np.sum(exp_x, axis=1,
keepdims=True)
else:
        return x
```

d. Forward Propagation Dense Layer

```
def forward(self, x, training=False):
    if not self.initialized:
        self.build(x.shape)

    if len(x.shape) > 2:
        batch_size = x.shape[0]
        x = x.reshape(batch_size, -1)

    linear_output = np.dot(x, self.weights['kernel']) +
self.biases['bias']

    activated_output =
self._apply_activation(linear_output)

    return activated_output
```

e. Backward Propagation Dense Layer

Backward propagation pada dense layer melibatkan perhitungan gradien terhadap weights, bias, dan input. Chain rule diterapkan untuk mempropagasi gradien melalui fungsi aktivasi dan transformasi linear.

- Gradien melalui Fungsi Aktivasi:
  ReLU Derivative:

$\partial ReLU/\partial z = \{1 \text{ if } z > 0, 0 \text{ if } z \leq 0\}$

Softmax Derivative (dengan Cross-Entropy):

$\partial(\text{softmax} + \text{cross-entropy})/\partial z = \text{softmax\_output} - \text{one\_hot\_target}$

Sigmoid Derivative:

$\partial\sigma/\partial z = \sigma(z) \times (1 - \sigma(z))$

Tanh Derivative:

$\partial\tanh/\partial z = 1 - \tanh^2(z)$

- Gradien untuk Weights dan Bias:

$\partial L/\partial W = X^T @ \partial L/\partial z$

$\partial L/\partial b = \text{sum}(\partial L/\partial z, \text{axis}=0)$

$\partial L/\partial X = \partial L/\partial z @ W^T$

```
def backward(self, grad_output):
        if 'input' not in self.cache:
            raise ValueError("Forward pass harus
dilakukan terlebih dahulu")

        input_data = self.cache['input']
        linear_output = self.cache.get('linear_output',
None)

        if self.activation == 'relu':
            grad_activation = (linear_output >
0).astype(float)
            grad_linear = grad_output * grad_activation
        elif self.activation == 'softmax':
            grad_linear = grad_output
        elif self.activation == 'sigmoid':
            sigmoid_output = self.cache['output']
            grad_linear = grad_output * sigmoid_output
* (1 - sigmoid_output)
        elif self.activation == 'tanh':
            tanh_output = self.cache['output']
            grad_linear = grad_output * (1 -
tanh_output**2)
        else:
            grad_linear = grad_output

        self.gradients['kernel'] = np.dot(input_data.T,
grad_linear)
        self.gradients['bias'] = np.sum(grad_linear,
axis=0, keepdims=True)

        grad_input = np.dot(grad_linear,
self.weights['kernel'].T)

        return grad_input
```

5. **Loss Function dan Training Loop**

   Sparse Categorical Cross-Entropy Loss

   Untuk klasifikasi multi-class, digunakan sparse categorical cross-entropy loss yang cocok untuk label dalam bentuk integer (bukan one-hot encoded).

   Loss Function:

   `L = -(1/N) × Σ log(y_pred[i, y_true[i]])`

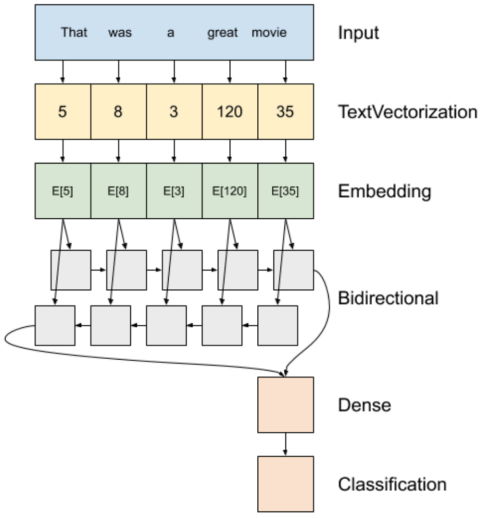
   Gradient:

   `∂L/∂y_pred = (y_pred - y_true_one_hot) / N`

```
def sparse_categorical_crossentropy(self, y_true, y_pred):
        batch_size = y_pred.shape[0]
        y_pred_clipped = np.clip(y_pred, 1e-15, 1 - 1e-15)
        log_likelihood =
-np.log(y_pred_clipped[range(batch_size), y_true])
        return np.mean(log_likelihood)

    def sparse_categorical_crossentropy_gradient(self, y_true,
y_pred):
        batch_size = y_pred.shape[0]
        y_true_one_hot = np.zeros_like(y_pred)
        y_true_one_hot[range(batch_size), y_true] = 1
        return (y_pred - y_true_one_hot) / batch_size
```

2. **Simple RNN (*Recurrent Neural Network*)**

   RNN (*Recurrent Neural Network*) adalah arsitektur *neural network* yang dirancang untuk memproses data *sequential* dengan mempertahankan informasi dari *timestep* sebelumnya melalui *hidden state*. Tidak seperti *feedforward network*, RNN memiliki koneksi rekuren yang memungkinkan informasi mengalir dari *output* kembali ke *input* pada *timestep* berikutnya. Untuk masalah *text classification* ini memiliki jenis layer-layer di dalamnya terdiri dari *embedding layer*, *bidirectional layer* dan/atau *unidirectional* RNN *layer*, *dropout layer*, dan *dense* layer.



Gambar 2.2 Arsitektur Recurrent Neural Network

Pada implementasinya di kelas *RNNFromScratch*, *forward propagation* merupakan *sequential pipeline* dimana *output* dari satu layer menjadi *input* untuk *layer* selanjutnya. Model melakukan iterasi melalui semua layers yang telah didefinisikan dan memanggil method forward() dari masing-masing *layer*.

```
Input: x_0 ∈ R^(B×S)                          [integer indices]
↓
Embedding: x_1 = Embed(x_0)                   [B×S×E]
↓
RNN: x_2 = RNN(x_1)                           [B×S×H] atau [B×H]
↓
Dropout: x_3 = Dropout(x_2)                   [same shape as x_2]
↓
Dense: y = Dense(x_3)                         [B×C]

Dimana:
B = batch_size, S = sequence_length
E = embedding_dim, H = hidden_units
C = num_classes
```

```
def forward(self, x, training=False):
        current_output = x

        for layer in self.layers:
            current_output = layer.forward(current_output,
```

```
training)

        return current_output
```

Berikut mekanisme *forward propagation* untuk masing-masing *layer*-nya.

1. ***Embedding Layer* - Transformasi Token ke *Dense Vector***

   Embedding layer merupakan pintu gerbang pertama dalam arsitektur RNN yang mengkonversi representasi diskrit (*integer indices*) menjadi representasi kontinu (*dense vectors*). Proses dimulai ketika input berupa *sequence of integers* dengan dimensi batch_size × seq_len masuk ke dalam *layer* ini. Setiap integer dalam *sequence* merepresentasikan satu token dari *vocabulary* yang telah di-*encode* sebelumnya.

   Embedding matrix yang berukuran vocab_size × embedding_dim berfungsi sebagai *lookup table*. Untuk setiap posisi dalam *sequence*, *layer* melakukan operasi *lookup* sederhana dimana nilai integer digunakan sebagai indeks untuk mengambil *row* yang sesuai dari *embedding matrix*. Proses ini dilakukan secara paralel untuk seluruh *batch* dan seluruh *timestep* dalam *sequence*.

   Inisialisasi *embedding matrix* menggunakan *Xavier initialization* untuk memastikan distribusi *weight* yang optimal pada awal *training*. Setiap token *vocabulary* memiliki representasi unik dalam ruang embedding yang dapat dipelajari selama proses *training*. *Output* dari layer ini adalah tensor 3D dengan dimensi batch_size × seq_len × embedding_dim, dimana setiap token telah dikonversi menjadi *dense vector* yang kaya informasi.

```
E ∈ R^(V×d)  (Embedding matrix)
x_embedded[i,j] = E[x[i,j]]
Output shape: (batch_size, seq_len, embedding_dim)
```

```
def forward(self, x, training=False):
        batch_size, seq_len = x.shape
        embedded = np.zeros((batch_size, seq_len,
                             self.embedding_dim))
```

```
        for i in range(batch_size):
            for j in range(seq_len):
                embedded[i, j] =
                        self.weights['embedding'][x[i, j]]

        # Cache for backward pass
        self.cache['input_indices'] = x
        self.cache['batch_size'] = batch_size
        self.cache['seq_len'] = seq_len

        return embedded
```

2. *Simple* RNN *Layer - Sequential Information Processing*

Simple RNN *Layer* merupakan inti dari arsitektur yang memproses informasi secara *sequential* dengan mempertahankan *memory* dari *timestep* sebelumnya. Layer ini menerima *input* dari *embedding layer* dan mulai memproses *sequence* dari *timestep* pertama hingga terakhir (*left-to-right processing*).

Pada setiap *timestep*, RNN melakukan dua operasi transformasi *linear* utama. Pertama, *input* pada *timestep* saat ini dikalikan dengan *input weight matrix* atau *kernel*. Kedua, *hidden state* dari *timestep* sebelumnya dikalikan dengan *recurrent weight matrix*. Kedua hasil transformasi ini dijumlahkan bersama dengan bias *term* untuk membentuk pre-*activation value*.

Hidden state diinisialisasi dengan *zero vector* pada *timestep* pertama, kemudian untuk *timestep* selanjutnya menggunakan *hidden state* dari *timestep* sebelumnya. Fungsi aktivasi tanh diterapkan pada pre-*activation value* untuk menghasilkan *hidden state* baru. Tanh dipilih karena *output range*-nya [-1,1] yang membantu mencegah *exploding gradients* dan memberikan *zero-centered activation*.

Proses ini berlanjut secara iteratif untuk setiap *timestep* dalam *sequence*, dengan *hidden state* yang terus di-*update* dan membawa informasi dari *timestep* sebelumnya. Bergantung pada parameter return_sequences,

layer dapat mengembalikan seluruh *sequence hidden states* atau hanya *hidden state* terakhir.

```
h_0 = 0
h_t = tanh(x_t * W_x + h_{t-1} * W_h + b)

Dimana:
- x_t ∈ R^(batch×input_dim): input pada timestep t
- h_t ∈ R^(batch×units): hidden state pada timestep t
- W_x ∈ R^(input_dim×units): input-to-hidden weight matrix
- W_h ∈ R^(units×units): hidden-to-hidden weight matrix
- b ∈ R^units: bias vector
```

```
def forward(self, x, training=False):
        batch_size, seq_len, input_dim = x.shape

        if not self.initialized:
            self.weights['kernel'] = np.random.uniform(
                -np.sqrt(6.0 / (self.units + self.units)),
np.sqrt(6.0 / (self.units + self.units)),
                (input_dim, self.units)
            )
            self.weights['recurrent_kernel'] =
np.random.uniform(
                -np.sqrt(6.0 / (self.units + self.units)),
np.sqrt(6.0 / (self.units + self.units)),
                (self.units, self.units)
            )
            self.biases['bias'] = np.zeros(self.units)

            self.initialized = True

        h = np.zeros((batch_size, self.units))
        outputs = []
        hidden_states = [h.copy()]

        for t in range(seq_len):
            # h_t = tanh(x_t @ W + h_{t-1} @ U + b)
            linear = np.dot(x[:, t, :],
self.weights['kernel']) + \
                     np.dot(h,
self.weights['recurrent_kernel']) + self.biases['bias']
            h = np.tanh(linear)
            outputs.append(h.copy())
            hidden_states.append(h.copy())
```

```
        # Cache for backward pass
        self.cache['input'] = x
        self.cache['hidden_states'] = hidden_states
        self.cache['outputs'] = outputs
        self.cache['batch_size'] = batch_size
        self.cache['seq_len'] = seq_len

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

3. *Bidirectional* RNN *Layer - Dual Direction Processing*

Bidirectional RNN *Layer* mengimplementasikan konsep pemrosesan *sequence* dalam dua arah secara simultan untuk menangkap *dependencies* yang lebih komprehensif. *Layer* ini pada dasarnya terdiri dari dua *Simple* RNN yang independen - satu memproses *sequence* dari kiri ke kanan (*forward direction*) dan satunya dari kanan ke kiri (*backward direction*).

Total units pada *bidirectional layer* dibagi menjadi dua bagian yang sama. *Forward* RNN menggunakan setengah dari total units dan memproses *sequence* mulai dari *timestep* 0 hingga *timestep* terakhir. Setiap *timestep* dalam *forward direction* mengikuti formula RNN standar dengan *weight matrices* dan bias yang spesifik untuk arah forward.

Paralel dengan *forward processing*, *backward* RNN juga menggunakan setengah *units* dan memproses *sequence* dalam arah sebaliknya, mulai dari *timestep* terakhir mundur ke *timestep* 0. *Backward* RNN memiliki *set weight matrices* dan bias yang terpisah dan independen dari *forward* RNN. Proses komputasi mengikuti formula RNN yang sama namun dengan iterasi *timestep* yang terbalik.

```
Forward Direction:
h_forward_0 = 0
h_forward_t = tanh(x_t * W_forward_x + h_forward_{t-1} *
```

```
W_forward_h + b_forward)

Backward Direction:
h_backward_T = 0
h_backward_t = tanh(x_t * W_backward_x + h_backward_{t+1} *
W_backward_h + b_backward)

Final Output:
h_t = concat(h_forward_t, h_backward_t)

Dimana:
- h_forward_t: forward hidden state
- h_backward_t: backward hidden state
- W_forward, W_backward: weight matrices untuk
masing-masing arah
- concat(): concatenation operation
```

```
def forward(self, x, training=False):
        batch_size, seq_len, input_dim = x.shape

        if not self.initialized:
            for direction in ['forward', 'backward']:
                self.weights[f'{direction}_kernel'] =
np.random.uniform(
                    -np.sqrt(6.0 / (self.forward_units +
self.forward_units)),
                    np.sqrt(6.0 / (self.forward_units +
self.forward_units)),
                     (input_dim, self.forward_units)
                )
                self.weights[f'{direction}_recurrent'] =
np.random.uniform(
                    -np.sqrt(6.0 / (self.forward_units +
self.forward_units)),
                    np.sqrt(6.0 / (self.forward_units +
self.forward_units)),
                     (self.forward_units,
self.forward_units)
                )
                self.biases[f'{direction}_bias'] =
np.zeros(self.forward_units)
            self.initialized = True

        # Forward pass
        h_forward = np.zeros((batch_size,
self.forward_units))
```

```
        forward_outputs = []
        forward_hidden_states = [h_forward.copy()]

        for t in range(seq_len):
            linear = np.dot(x[:, t, :],
self.weights['forward_kernel']) + \
                     np.dot(h_forward,
self.weights['forward_recurrent']) + \
                     self.biases['forward_bias']
            h_forward = np.tanh(linear)
            forward_outputs.append(h_forward.copy())
            forward_hidden_states.append(h_forward.copy())

        # Backward pass
        h_backward = np.zeros((batch_size,
self.backward_units))
        backward_outputs = []
        backward_hidden_states = [h_backward.copy()]

        for t in range(seq_len - 1, -1, -1):
            linear = np.dot(x[:, t, :],
self.weights['backward_kernel']) + \
                     np.dot(h_backward,
self.weights['backward_recurrent']) + \
                     self.biases['backward_bias']
            h_backward = np.tanh(linear)
            backward_outputs.insert(0, h_backward.copy())
            backward_hidden_states.insert(0,
h_backward.copy())

        # Cache for backward pass
        self.cache['input'] = x
        self.cache['forward_hidden_states'] =
forward_hidden_states
        self.cache['backward_hidden_states'] =
backward_hidden_states
        self.cache['forward_outputs'] = forward_outputs
        self.cache['backward_outputs'] = backward_outputs
        self.cache['batch_size'] = batch_size
        self.cache['seq_len'] = seq_len

        # Concat forward and backward outputs
        if self.return_sequences:
            forward_stack = np.stack(forward_outputs,
axis=1)
            backward_stack = np.stack(backward_outputs,
axis=1)
            return np.concatenate([forward_stack,
```

```
backward_stack], axis=-1)
        else:
            return np.concatenate([forward_outputs[-1],
backward_outputs[0]], axis=-1)
```

4. ***Dropout Layer - Regularization During Training***

   Dropout layer berfungsi sebagai *regularization technique* yang mencegah *overfitting* dengan secara random "mematikan" sebagian *neurons* selama *training phase*. Layer ini beroperasi berbeda antara training dan inference mode, menunjukkan karakteristik *adaptive behavior*.

   Selama *training mode*, *dropout layer* membuat *random binary mask* dengan probability (1-*rate*) untuk setiap *element* dalam *input* tensor. *Mask* ini menentukan neuron mana yang akan "hidup" dan mana yang akan "mati" pada *forward pass* saat ini. *Neurons* yang "mati" akan memiliki *output zero*, sementara *neurons* yang "hidup" akan mempertahankan nilai aslinya.

   Untuk mengkompensasi pengurangan *expected* value akibat *zero*-ing sebagian *neurons*, *output* dari *neurons* yang "hidup" di-*scale* dengan faktor 1/(1-rate). *Scaling* ini memastikan bahwa *expected value* dari *output* tetap sama dengan i, mencegah shift dalam magnitude aktivasi.

   Selama *inference mode*, *dropout layer* berperilaku sebagai *identity function* - *input* langsung diteruskan tanpa modifikasi. Hal ini konsisten dengan tujuan *dropout* yang hanya aktif selama *training* untuk *regularization*.

```
Training Mode:
mask ~ Bernoulli(1-rate)
y = (x ⊙ mask) / (1-rate)

Inference Mode:
y = x

Dimana:
- rate: dropout probability
- ⊙: element-wise multiplication
```

```
- mask: binary mask dengan nilai 0 atau 1
```

```
def forward(self, x, training=False):
        if not training:
            return x

        mask = np.random.binomial(1, 1 - self.rate,
x.shape) / (1 - self.rate)
        self.cache['mask'] = mask
        return x * mask
```

5. ***Dense Layer - Classification Output Generation***

   Dense layer merupakan *layer* terakhir yang bertanggung jawab mengkonversi representasi *hidden* dari RNN menjadi *output probabilitas* untuk setiap *class*. *Layer* ini menerima *input* dari RNN *layer* yang bisa berupa 2D tensor (jika return_sequences=False) atau 3D tensor (jika return_sequences=True).

   Jika *input* berbentuk 3D tensor, *dense layer* mengambil hanya *timestep* terakhir dengan operasi *slicing* x[:, -1, :]. Hal ini masuk akal karena untuk task klasifikasi *sequence*, biasanya representasi *final sequence* yang paling informatif untuk prediksi *class*.

   Dense layer melakukan transformasi *affine* (*linear transformation*) dengan mengalikan input dengan *weight matrix* dan menambahkan bias. *Weight matrix* berukuran input_dim × num_classes, dimana *num_classes* adalah jumlah kategori yang ingin diprediksi. Bias *vector* berukuran num_classes memberikan *offset* untuk setiap class.

   Setelah transformasi *linear*, fungsi aktivasi softmax diterapkan untuk mengkonversi *raw* logits menjadi *probability distribution*. Softmax memastikan bahwa semua *output values* berada dalam range [0,1] dan *sum* dari semua probabilities sama dengan 1, sehingga dapat diinterpretasikan sebagai probabilitas prediksi untuk setiap class.

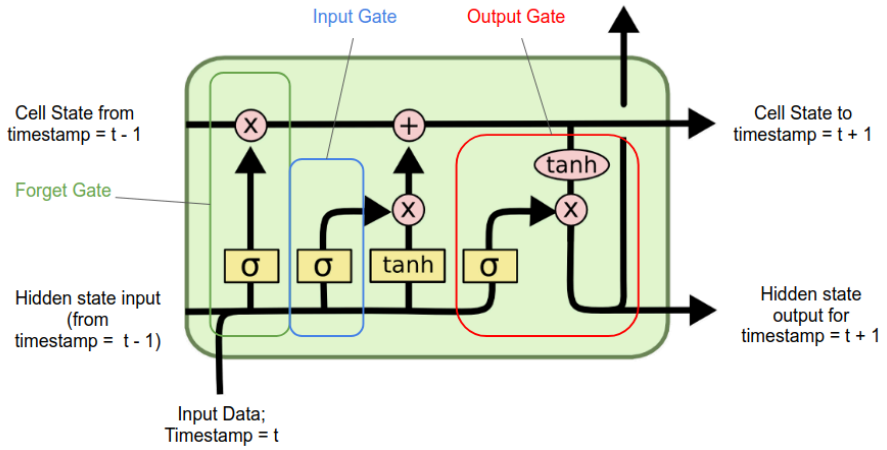
```
Linear Transformation:
z = x * W + b

Softmax Activation:
p_i = exp(z_i) / Σ_j exp(z_j)

Dimana:
- x ∈ R^(batch×input_dim): input features
- W ∈ R^(input_dim×num_classes): weight matrix
- b ∈ R^num_classes: bias vector
- z ∈ R^(batch×num_classes): logits
- p_i: probability untuk class i
```

```
def forward(self, x, training=False):
        # (batch, seq, features)
        if len(x.shape) == 3:
            x = x[:, -1, :]

        self.build(x.shape)
        self.cache['input'] = x

        z = np.dot(x, self.weights['kernel']) +
self.weights['bias']
        self.cache['linear_output'] = z

        a = self.activation(z)
        self.cache['output'] = a
        return a
```

3. LSTM

a. Konsep Dasar LSTM

LSTM (*Long Short-Term Memory*) dirancang untuk mengatasi masalah vanishing gradient yang dialami RNN tradisional. LSTM menggunakan arsitektur *gate mechanism* yang terdiri dari tiga gate utama dan *cell state* untuk mengontrol aliran informasi:



Gambar 2.3 Mekanisme Gate dalam LSTM Cell

1). Struktur Gate LSTM

```
# Komputasi semua gates sekaligus
gates = np.dot(x[:, t, :], self.weights['kernel']) + \
        np.dot(h, self.weights['recurrent_kernel']) + \
        self.biases['bias']

# Pembagian gates berdasarkan posisi
i_gate = self._sigmoid(gates[:, :self.units])
# Input gate
f_gate = self._sigmoid(gates[:, self.units:2*self.units])
# Forget gate
c_tilde = self._tanh(gates[:, 2*self.units:3*self.units])
# Candidate values
o_gate = self._sigmoid(gates[:, 3*self.units:])
# Output gate
```

2). Mekanisme *Forget Gate*: *Forget gate* menentukan informasi apa yang akan "dilupakan" dari cell state sebelumnya:

```
f_gate = sigmoid(x_t @ W_f + h_{t-1} @ U_f + b_f)
```

3). Mekanisme *Input Gate* dan *Candidate Values Input gate*: memutuskan informasi baru mana yang akan disimpan dalam cell state:

```
i_gate = sigmoid(x_t @ W_i + h_{t-1} @ U_i + b_i)
c_tilde = tanh(x_t @ W_c + h_{t-1} @ U_c + b_c)
```

4). Update Cell State Cell state diperbarui dengan menggabungkan informasi lama (yang tidak dilupakan) dan informasi baru:

```
c_t = f_gate * c_{t-1} + i_gate * c_tilde
```

5). Mekanisme Output Gate dan Hidden State Output gate mengontrol bagian mana dari cell state yang akan menjadi output:

```
o_gate = sigmoid(x_t @ W_o + h_{t-1} @ U_o + b_o)
h_t = o_gate * tanh(c_t)
```

6). Implementasi Forward Propagation Lengkap

```
def forward(self, x, training=False):
    batch_size, seq_len, input_dim = x.shape

    # Inisialisasi states
    h = np.zeros((batch_size, self.units))
    c = np.zeros((batch_size, self.units))

    outputs = []
    hidden_states = [h.copy()]
    cell_states = [c.copy()]
    gates_history = []

    # Loop untuk setiap timestep
    for t in range(seq_len):
        # Komputasi linear combination untuk semua gates
        gates = np.dot(x[:, t, :], self.weights['kernel']) + \
                np.dot(h, self.weights['recurrent_kernel']) + \
                self.biases['bias']

        # Ekstraksi individual gates
        i_gate = self._sigmoid(gates[:, :self.units])
        f_gate = self._sigmoid(gates[:,
self.units:2*self.units])
        c_tilde = self._tanh(gates[:,
2*self.units:3*self.units])
        o_gate = self._sigmoid(gates[:, 3*self.units:])

        # Update cell state dan hidden state
        c = f_gate * c + i_gate * c_tilde
        h = o_gate * self._tanh(c)

        # Simpan untuk backward pass
        outputs.append(h.copy())
```

```
        hidden_states.append(h.copy())
        cell_states.append(c.copy())
        gates_history.append({
            'i_gate': i_gate, 'f_gate': f_gate,
            'c_tilde': c_tilde, 'o_gate': o_gate, 'gates':
gates
        })

    # Return output sesuai dengan return_sequences
    if self.return_sequences:
        return np.stack(outputs, axis=1)
    else:
        return h
```

7). Optimisasi Numerik untuk Stabilitas

Untuk mencegah numerical *overflow* dan *underflow*, implementasi menggunakan *clipping*:

```
def _sigmoid(self, x):
    return 1.0 / (1.0 + np.exp(-np.clip(x, -500, 500)))

def _tanh(self, x):
    return np.tanh(np.clip(x, -500, 500))
```

8). Forward Propagation Bidirectional LSTM

Bidirectional LSTM memproses sequence dalam dua arah dan menggabungkan hasilnya:

```
# Forward direction (t = 0 to T-1)
for t in range(seq_len):
    # Standard LSTM computation untuk forward

# Backward direction (t = T-1 to 0)
for t in range(seq_len - 1, -1, -1):
    # Standard LSTM computation untuk backward

# Concatenate hasil dari kedua arah
if self.return_sequences:
    forward_stack = np.stack(forward_outputs, axis=1)
    backward_stack = np.stack(backward_outputs, axis=1)
    return np.concatenate([forward_stack, backward_stack],
axis=-1)
```

2. Hasil Pengujian

A. CNN

1. Pengaruh jumlah layer konvolusi

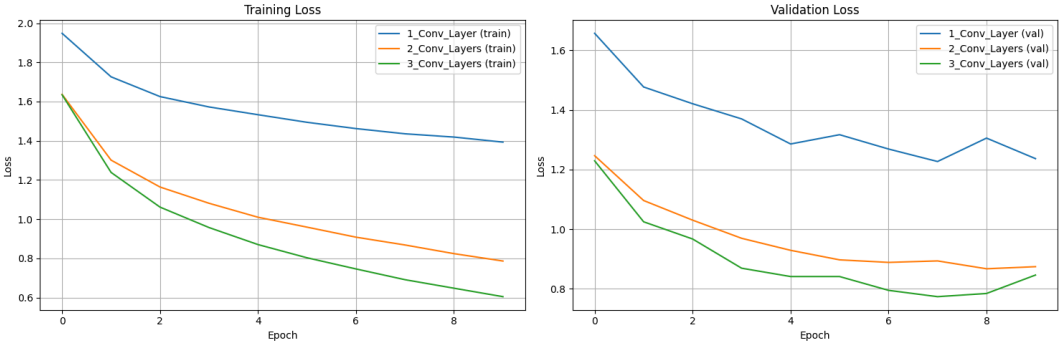
Pengujian dilakukan dengan membandingkan tiga konfigurasi berbeda untuk menganalisis bagaimana kedalaman arsitektur mempengaruhi performa model pada task klasifikasi gambar CIFAR-10.

Konfigurasi yang Diuji:

- Model 1: 1 Convolutional layer
- Model 2: 2 Convolutional layers
- Model 3: 3 Convolutional layers

Hasil Performa:

Model	Jumlah *Layer*	Test Accuracy	Test Loss	Macro F1-Score	Total Parameter
Model 1	1 layer	0.5727	1.2411	0.5655	1,050,890
Model 2	2 layers	0.6968	0.8760	0.6965	545,098
Model 3	3 layers	0.7177	0.8765	0.7143	356,810



Gambar 2.1.1.1 Grafik *Training Loss* dan *Validation Loss*

Analisis Grafik Training:

Model dengan 1 layer konvolusi menunjukkan konvergensi yang lebih lambat, dengan training loss yang menurun secara gradual dari sekitar 1.95 hingga 1.4 dalam 10 epoch. Validation loss menunjukkan pola yang relatif stabil, namun dengan performa yang lebih rendah dibandingkan model lainnya.

Model dengan 2 layer konvolusi menampilkan penurunan training loss yang lebih cepat, dari sekitar 1.65 menjadi 0.8 dalam rentang yang sama. Validation loss juga menunjukkan tren penurunan yang konsisten dari 1.25 hingga sekitar 0.87, mengindikasikan pembelajaran yang efektif tanpa overfitting yang signifikan.
Model dengan 3 layer konvolusi mencapai performa terbaik dengan training loss yang menurun dari 1.65 menjadi 0.6 dan validation loss yang stabil di sekitar 0.78-0.85. Model ini menunjukkan kemampuan terbaik dalam menangkap fitur-fitur kompleks dari dataset CIFAR-10.

Kesimpulan:
Penambahan layer konvolusi secara konsisten meningkatkan performa model. Model dengan 3 layer konvolusi memberikan hasil terbaik dengan F1-macro score 0.7143, menunjukkan bahwa arsitektur yang lebih dalam mampu mengekstraksi representasi fitur yang lebih kaya dan kompleks untuk dataset CIFAR-10.

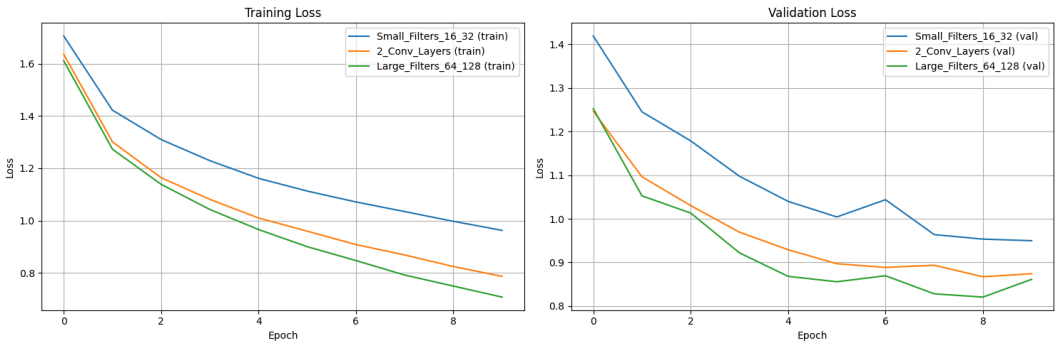
2. Pengaruh banyak filter per layer konvolusi

Analisis ini mengevaluasi bagaimana kapasitas representasi layer (jumlah filter) mempengaruhi kemampuan model dalam learning pattern yang kompleks.
Konfigurasi yang Diuji:

- Small Filters: [16, 32] filters per layer
- Medium Filters: [32, 64] filters per layer
- Large Filters: [64, 128] filters per layer

Hasil Performa:

Model	Konfigurasi Filter	Test Accuracy	Test Loss	Macro F1-Score	Total Parameter
Small Filter	16, 32	0.6737	0.9423	0.6728	268,650
Medium Filter	32, 64	0.6968	0.8760	0.6965	545,098
Large Filter	64, 128	0.7098	0.8824	0.7065	1,125,642



Gambar 2.1.2.1 Grafik *Training Loss* dan *Validation Loss*

Analisis Grafik Training:

Model dengan small filters menunjukkan training loss yang menurun dari 1.7 hingga 0.97, dengan validation loss yang relatif stabil di sekitar 0.95. Pola ini mengindikasikan bahwa kapasitas model mungkin terbatas untuk menangkap kompleksitas penuh dari dataset.

Model dengan medium filters menampilkan penurunan training loss yang lebih konsisten dari 1.65 menjadi 0.8, dengan validation loss yang menurun dari 1.25 ke 0.87. Model ini menunjukkan keseimbangan yang baik antara kapasitas learning dan generalisasi.

Model dengan large filters mencapai training loss terendah, menurun dari 1.65 hingga 0.72, dengan validation loss yang stabil di sekitar 0.82-0.87. Model ini menunjukkan kapasitas representasi yang paling tinggi.

Kesimpulan:

Peningkatan jumlah filter secara konsisten meningkatkan performa model. Large filters (64, 128) memberikan hasil terbaik dengan F1-macro score 0.7065, menunjukkan bahwa kapasitas representasi yang lebih besar memungkinkan model untuk menangkap fitur-fitur yang lebih beragam dan kompleks dari dataset CIFAR-10.

3. Pengaruh ukuran filter per layer konvolusi

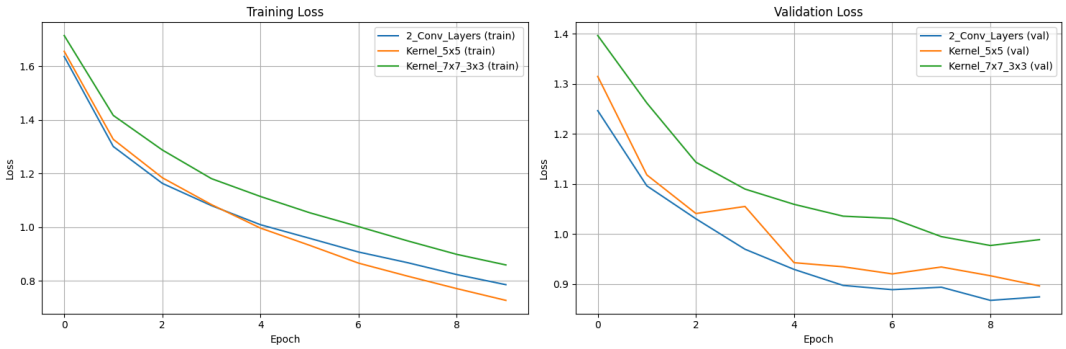
Eksperimen ini menganalisis dampak ukuran kernel terhadap kemampuan model dalam menangkap fitur spatial pada berbagai skala.

Konfigurasi yang Diuji:

- Kernel 3x3: Standard kernel size [3x3, 3x3]
- Kernel 5x5: Larger kernel size [5x5, 5x5]
- Kernel 7x7+3x3: Mixed kernel size [7x7, 3x3]

Hasil Performa:

Model	Konfigurasi Kernel	Test Accuracy	Test Loss	Macro F1-Score	Total Parameter
Kernel 3x3	[3x3, 3x3]	0.6968	0.8760	0.6965	545,098
Kernel 5x5	[5x5, 5x5]	0.6905	0.9138	0.6897	579,402
Kernel 7x7+3x3	[7x7, 3x3]	0.6644	0.9994	0.6627	548,938



Gambar 2.1.3.1 Grafik *Training Loss* dan *Validation Loss*

Analisis Grafik Training:

Model dengan kernel 3x3 menunjukkan pola training yang optimal, dengan training loss menurun secara smooth dari 1.65 hingga 0.8 dan validation loss yang stabil di sekitar 0.87. Pola ini mengindikasikan konvergensi yang sehat tanpa overfitting.

Model dengan kernel 5x5 menampilkan training loss yang menurun dari 1.65 menjadi 0.75, namun validation loss menunjukkan fluktuasi yang lebih besar, dengan peningkatan di epoch terakhir dari 0.92 hingga 0.89. Hal ini menunjukkan kemungkinan slight overfitting.

Model dengan kernel 7x7+3x3 mengalami training loss yang menurun dari 1.7 hingga 0.85, tetapi validation loss menunjukkan pola yang kurang stabil dengan fluktuasi antara 0.98-1.0. Model ini menunjukkan kesulitan dalam generalisasi.

Kesimpulan:
Kernel size 3x3 memberikan performa terbaik dengan F1-macro score 0.6965. Hasil ini menunjukkan bahwa untuk dataset CIFAR-10 dengan ukuran image 32x32, kernel kecil (3x3) lebih efektif dalam menangkap fitur-fitur relevan dibandingkan kernel yang lebih besar. Kernel besar cenderung menangkap terlalu banyak context yang tidak relevan, sehingga mengurangi kemampuan discriminative model.

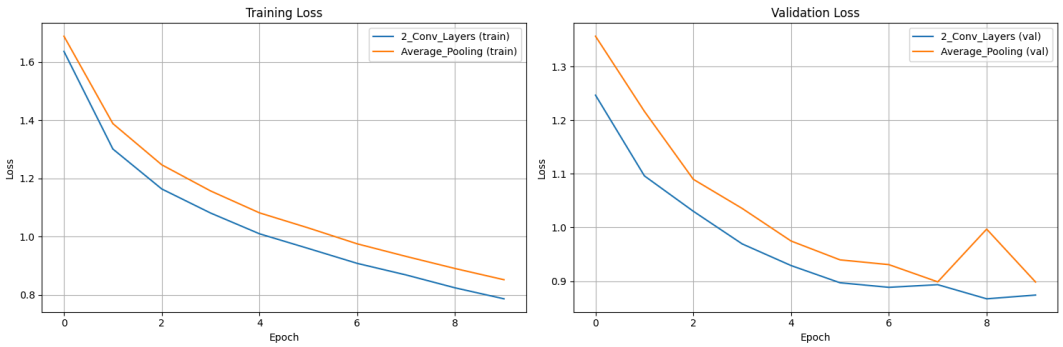
4. Pengaruh jenis pooling layer

Perbandingan ini menganalisis perbedaan fundamental antara Max Pooling dan Average Pooling dalam konteks feature extraction dan dimensionality reduction.
Konfigurasi yang Diuji:

- Max Pooling: Mengambil nilai maksimum dalam pooling window
- Average Pooling: Mengambil nilai rata-rata dalam pooling window

Hasil Performa:

Model	Konfigurasi Filter	Test Accuracy	Test Loss	Macro F1-Score	Total Parameter
Max Pooling	Max Pooling	0.6968	0.8760	0.6965	545,098
Average Pooling	Average Pooling	0.6889	0.9054	0.6826	545,098



Gambar 2.1.4.1 Grafik *Training Loss* dan *Validation Loss*

Analisis Grafik Training:

Model dengan Max Pooling menunjukkan training loss yang menurun secara konsisten dari 1.65 hingga 0.8, dengan validation loss yang stabil di sekitar 0.87-0.89. Pola konvergensi menunjukkan pembelajaran yang efektif dan stabil.

Model dengan Average Pooling menampilkan training loss yang menurun dari 1.7 menjadi 0.85, namun validation loss menunjukkan fluktuasi yang lebih signifikan, terutama peningkatan tajam di epoch ke-7 dari 0.92 menjadi 0.99 sebelum turun kembali. Hal ini mengindikasikan ketidakstabilan dalam proses pembelajaran.

Kesimpulan:

Max Pooling memberikan performa superior dibandingkan Average Pooling dengan F1-macro score 0.6965 vs 0.6826. Max Pooling lebih efektif dalam mempertahankan fitur-fitur penting dan discriminative dengan memilih nilai aktivasi tertinggi, sementara Average Pooling cenderung menghaluskan informasi yang dapat mengurangi kemampuan model dalam membedakan antar kelas. Untuk task klasifikasi image seperti CIFAR-10, Max Pooling terbukti lebih sesuai karena mampu mempertahankan fitur-fitur kuat yang penting untuk diskriminasi antar objek.

5. Perbandingan forward propagation

Pengujian forward propagation pada implementasi CNN from scratch telah dilakukan dan dibandingkan dengan model Keras yang telah di-load kembali.

Evaluasi kinerja dilakukan menggunakan 100 sample test data untuk validasi awal.

```
keras_model = keras.models.load_model('best_cnn_model.h5')

cnn_scratch = CNNFromScratch()
print("Loading Keras weights to from-scratch implementation...")
cnn_scratch.load_from_keras(keras_model)

test_sample_size = 100
x_test_sample = x_test[:test_sample_size]
y_test_sample = y_test[:test_sample_size]

print(f"\nTesting on {test_sample_size} samples...")

print("Keras prediction...")
keras_pred_probs = keras_model.predict(x_test_sample, verbose=0)
keras_pred_classes = np.argmax(keras_pred_probs, axis=1)
keras_f1 = f1_score(y_test_sample, keras_pred_classes,
average='macro')

print("From scratch prediction...")
try:
    scratch_pred_probs = cnn_scratch.predict_proba(x_test_sample,
batch_size=32)
    scratch_pred_classes = cnn_scratch.predict(x_test_sample,
batch_size=32)
    scratch_f1 = f1_score(y_test_sample, scratch_pred_classes,
average='macro')

    prob_diff = np.mean(np.abs(keras_pred_probs -
scratch_pred_probs))
    agreement = np.mean(keras_pred_classes ==
scratch_pred_classes)

    print(f"\nCOMPARISON RESULTS:")
    print(f"="*40)
    print(f"Keras Model:")
    print(f"  Accuracy: {np.mean(keras_pred_classes ==
y_test_sample):.4f}")
    print(f"  F1-Score: {keras_f1:.6f}")
    print(f"")
    print(f"From Scratch Model:")
    print(f"  Accuracy: {np.mean(scratch_pred_classes ==
y_test_sample):.4f}")
    print(f"  F1-Score: {scratch_f1:.6f}")
    print(f"")
```

```
    print(f"Comparison:")
    print(f"  Prediction Agreement: {agreement:.6f}
({agreement*100:.2f}%)")
    print(f"  Avg Probability Difference: {prob_diff:.6f}")
    print(f"  F1-Score Difference: {abs(keras_f1 -
scratch_f1):.6f}")

    implementation_success = True

except Exception as e:
    print(f"Error in from scratch implementation: {e}")
    implementation_success = False
```

```
COMPARISON RESULTS:
========================================
Keras Model:
  Accuracy: 0.7600
  F1-Score: 0.734505

From Scratch Model:
  Accuracy: 0.7600
  F1-Score: 0.734505

Comparison:
  Prediction Agreement: 1.000000 (100.00%)
  Avg Probability Difference: 0.000000
  F1-Score Difference: 0.000000
```

B. Simple RNN

1. Pengaruh jumlah layer RNN

Pengujian dilakukan dengan membandingkan tiga konfigurasi berbeda untuk menganalisis bagaimana kedalaman arsitektur mempengaruhi performa model pada *task* klasifikasi sentimen.

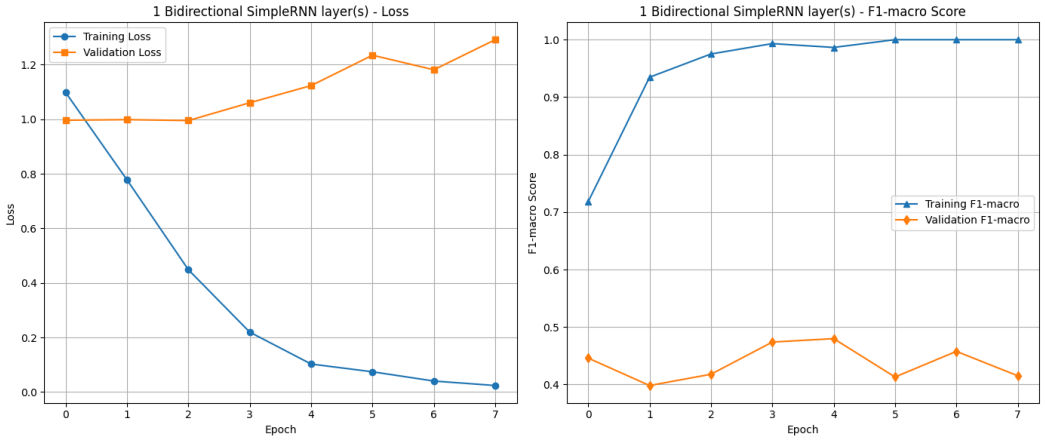
Konfigurasi yang Diuji:

- Model 1: 1 Bidirectional SimpleRNN layer (64 units)
- Model 2: 2 Bidirectional SimpleRNN layers (64 units each)
- Model 3: 3 Bidirectional SimpleRNN layers (64 units each)

Hasil Performa:

Model	Jumlah *Layer*	Macro F1-Score	Total Parameter
Model 1	1 layer	0.4920	767,491
Model 2	2 layers	0.5551	792,195
Model 3	3 layers	0.5571	816,899

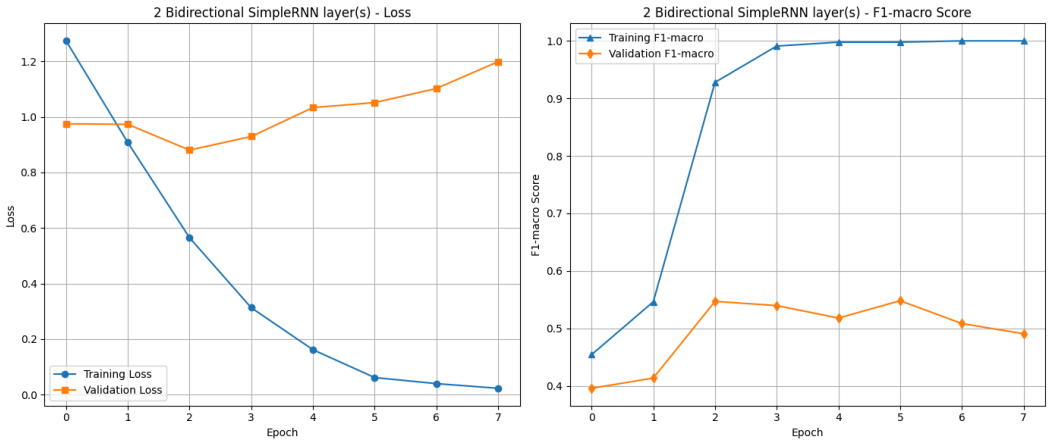
Pada Model 1 Layer, terjadi penurunan *training loss* yang sangat cepat, dari sekitar 1.1 menuju 0.02 dalam rentang 8 epoch. Hal ini mengindikasikan bahwa model dengan cepat mengasimilasi dan menghafal pola pada data pelatihan. Namun, di sisi lain, *validation loss* menunjukkan perilaku yang kontras, di mana ia justru meningkat dari sekitar 1.0 menjadi 1.3. Kesenjangan yang signifikan antara *training loss* yang rendah dan *validation loss* yang tinggi ini adalah tanda *overfitting* yang parah. Fenomena ini diperkuat oleh F1-macro *score* yang mencapai sempurna 1.0 pada data pelatihan, sementara pada data validasi stagnan di kisaran 0.4-0.5, menunjukkan ketidakmampuan model untuk menggeneralisasi dengan baik pada data yang belum pernah dilihat.



Gambar 2.2.1.1 Grafik *Training Loss* dan *Validation Loss* Model 1

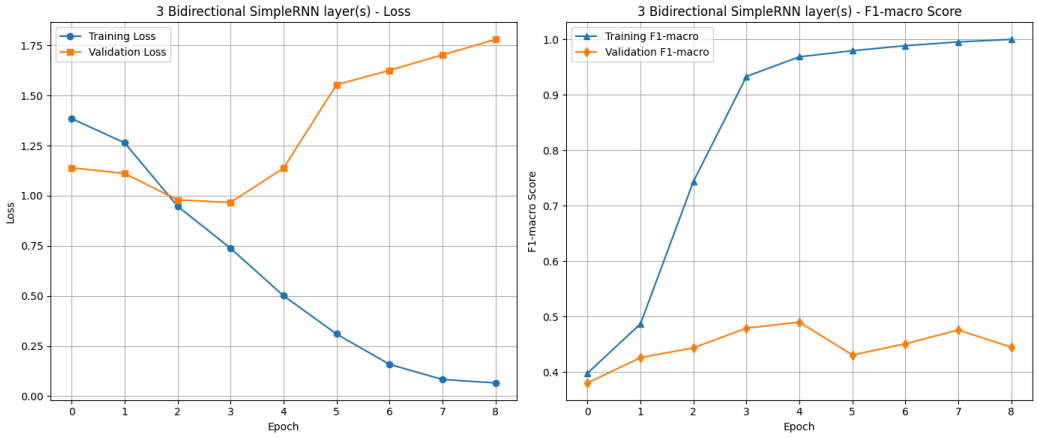
Pada Model 2 Layer, meskipun *training loss* masih menunjukkan penurunan yang kuat, dari sekitar 1.3 menjadi 0.03 dalam 8 epoch, penurunannya lebih gradual dibandingkan model 1 layer. Hal ini berarti bahwa model membutuhkan sedikit lebih banyak waktu untuk mencapai performa tinggi pada data pelatihan. *Validation loss*, meskipun masih mengalami peningkatan dari sekitar 0.97 menjadi 1.2, kenaikannya tampak lebih terkontrol. Hal ini menandakan *overfitting*

masih terjadi, namun tidak seekstrim model sebelumnya. Konsistensi ini juga tercermin pada F1-macro *score* validasi yang relatif lebih stabil di kisaran 0.5-0.55, meskipun training F1-macro tetap mencapai 1.0.



Gambar 2.2.1.2 Grafik *Training Loss* dan *Validation Loss* Model 2

Pada Model 3 Layer, *training loss* menunjukkan penurunan yang paling gradual di antara ketiga model, dari sekitar 1.4 menjadi 0.08 dalam 9 epoch. Hal ini mungkin menunjukkan bahwa kompleksitas model yang lebih tinggi membutuhkan waktu lebih lama untuk konvergensi pada data pelatihan. *Validation loss* justru mengalami peningkatan yang paling drastis, dari sekitar 1.1 melonjak hingga 1.8. Peningkatan tajam pada validation loss ini mengindikasikan *overfitting* yang paling parah terjadi pada model ini, bahkan lebih buruk dari model 1 layer, meskipun penyerapan data pelatihan lebih lambat. F1-macro *score* validasi juga terlihat sangat fluktuatif di kisaran 0.4-0.5, menandakan ketidakstabilan dan ketidakmampuan generalisasi yang buruk pada data baru, meskipun training F1-macro tetap sempurna 1.0.



Gambar 2.2.1.3 Grafik *Training Loss* dan *Validation Loss* Model 3

Penambahan layer dari 1 ke 2 memberikan peningkatan signifikan (+12.8%), namun dari 2 ke 3 layer hanya memberikan peningkatan minimal (+0.4%). Model yang lebih dalam cenderung *overfit* pada dataset kecil, sementara model dengan *layer* yang sedikit mungkin akan menjadi *shallow* dan kurang mampu melakukan generalisasi terhadap data *training*. Secara keseluruhan, terlihat tren bahwa penambahan lapisan RNN (dari 1 ke 3) tidak secara otomatis meningkatkan kemampuan generalisasi model, dan bahkan dapat memperburuk *overfitting* jika tidak diimbangi dengan strategi regularisasi yang memadai atau lebih banyak data pelatihan.

2. Pengaruh banyak cell RNN per layer

Analisis ini mengevaluasi bagaimana kapasitas representasi layer (jumlah hidden units) mempengaruhi kemampuan model dalam learning pattern yang kompleks.

Pengujian menggunakan satu buah layer Bidirectional Simple RNN, dengan masing-masing model memiliki konfigurasi unit:

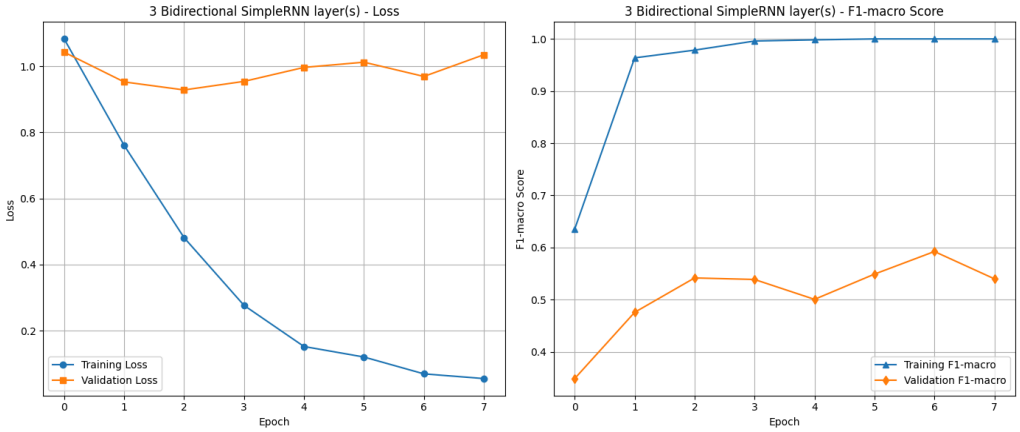
- Model A: 32 *cells* per *layer*
- Model B: 64 *cells* per *layer*
- Model C: 128 *cells* per *layer*

Hasil Performa:

Model	Jumlah *Cell*	Macro F1-Score	Total Parameter
Model A	32 *cells*	0.5640	744,707

Model B	64 *cell*	0.5052	767,491
Model C	128 *cells*	0.4158	825,347

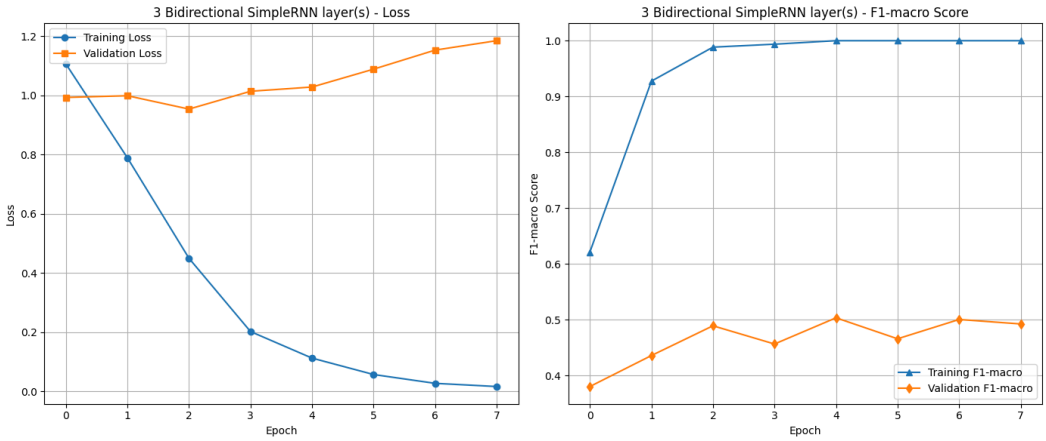
Pada model dengan 32 *cells*, *training loss* menurun sangat cepat dari sekitar 1.11 pada epoch 1 menjadi 0.0549 pada epoch 8. Hal ini menunjukkan model belajar dengan sangat efisien pada data pelatihan dan hampir mencapai *loss* nol. *Validation loss* dimulai dari sekitar 1.04 pada epoch 1, kemudian menurun sedikit di epoch 2 dan 3 (sekitar 0.95), dan mulai meningkat kembali di epoch berikutnya, berakhir di 1.0346 pada epoch 8. *Training* F1-macro meningkat pesat dan mencapai 1.0 (sempurna) mulai epoch 6. Sementara itu, *validation* F1-macro berfluktuasi dari 0.3484 (epoch 1) hingga puncaknya 0.5922 (epoch 7) sebelum sedikit menurun. Hal ini menunjukkan adanya indikasi *overfitting* yang jelas di mana *training loss* turun drastis sementara *validation loss* mulai naik setelah beberapa epoch awal. Meskipun demikian, model ini memberikan F1-macro validasi terbaik.



Gambar 2.2.2.1 Grafik *Training Loss* dan *Validation Loss* Model A

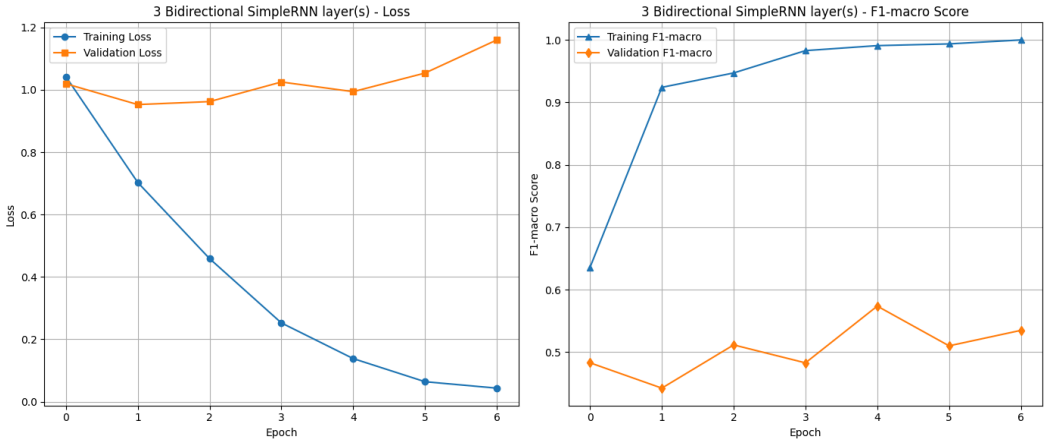
Pada model dengan 64 *cells*, *training loss* menurun dengan sangat cepat dari sekitar 1.13 pada epoch 1 menjadi 0.0177 pada epoch 8. Hal ini bahkan lebih rendah dari model 32 *cells*, menunjukkan model ini sangat baik dalam menghafal data pelatihan. *Validation loss* dimulai dari sekitar 0.99 pada epoch 1, berfluktuasi sedikit (turun di epoch 3 ke 0.95), namun secara keseluruhan menunjukkan tren peningkatan yang konsisten, berakhir di 1.1847 pada epoch 8. Peningkatan ini lebih curam dan *validation loss* akhirnya lebih tinggi

dibandingkan model 32 *cells*. *Training* F1-macro juga mencapai 1.0 (sempurna) mulai epoch 5. Namun, *validation* F1-macro berfluktuasi di kisaran 0.38 hingga 0.50, secara umum lebih rendah dan kurang stabil dibandingkan model 32 *cells*. Tingkat *overfitting* tampaknya lebih parah pada model ini dibandingkan model 32 *cells*. Meskipun *training loss* sangat rendah, kemampuan generalisasinya menurun (ditunjukkan oleh *validation loss* yang lebih tinggi dan F1-macro yang lebih rendah).



Gambar 2.2.2.2 Grafik *Training Loss* dan *Validation Loss* Model B

Pada model dengan 128 *cells* menunjukkan penurunan *training loss* yang sangat cepat, dari sekitar 1.07 pada epoch pertama menjadi hanya 0.0407 pada epoch ke-7, di mana pelatihan dihentikan karena tidak ada peningkatan lebih lanjut. Namun, meskipun *training loss* menurun drastis, *validation loss* justru menunjukkan tren yang fluktuatif dan cenderung meningkat, dari sekitar 1.01 menjadi 1.1599 pada akhir pelatihan. F1-macro pada data pelatihan bahkan mencapai skor sempurna 1.0, tetapi pada data validasi nilainya sangat tidak stabil, berada di kisaran 0.44 hingga 0.57, dan akhirnya turun menjadi 0.4158 yang mana nilai terendah di antara seluruh model yang dibandingkan. Hasil ini mengindikasikan *overfitting* yang sangat jelas. Penambahan jumlah *cells* memang meningkatkan kapasitas model, tetapi dalam kasus ini justru membuat model lebih mudah menghafal data pelatihan tanpa menghasilkan generalisasi yang baik terhadap data validasi.



Gambar 2.2.2.3 Grafik *Training Loss* dan *Validation Loss* Model C

Dalam kasus ini, model dengan jumlah *cells* yang lebih kecil (32) ternyata lebih optimal karena memiliki kapasitas yang cukup untuk belajar pola yang relevan tanpa terlalu *overfit* pada data pelatihan. Meskipun *training loss*-nya tidak serendah model 64 atau 128 *cells*, kemampuan generalisasinya pada data validasi jauh lebih baik. Semakin banyak *cells*, semakin tinggi kapasitas model dan jumlah parameternya. Hal ini memungkinkan model untuk belajar dan menghafal data pelatihan dengan lebih detail, yang ditunjukkan oleh *training loss* yang sangat rendah (mendekati nol) dan training F1-macro yang sempurna (1.0) di semua kasus. Namun, peningkatan kapasitas ini juga membuat model lebih rentan terhadap *overfitting*. Model yang lebih kompleks (dengan lebih banyak *cells*) cenderung menghafal *noise* atau pola spesifik dari data pelatihan, sehingga kehilangan kemampuan untuk menggeneralisasi pada data yang belum pernah dilihat.

3. Pengaruh jenis layer RNN berdasarkan arah

Perbandingan ini menganalisis perbedaan fundamental antara pemrosesan *unidirectional* dan *bidirectional* dalam konteks klasifikasi sentimen.

Konfigurasi yang Diuji:

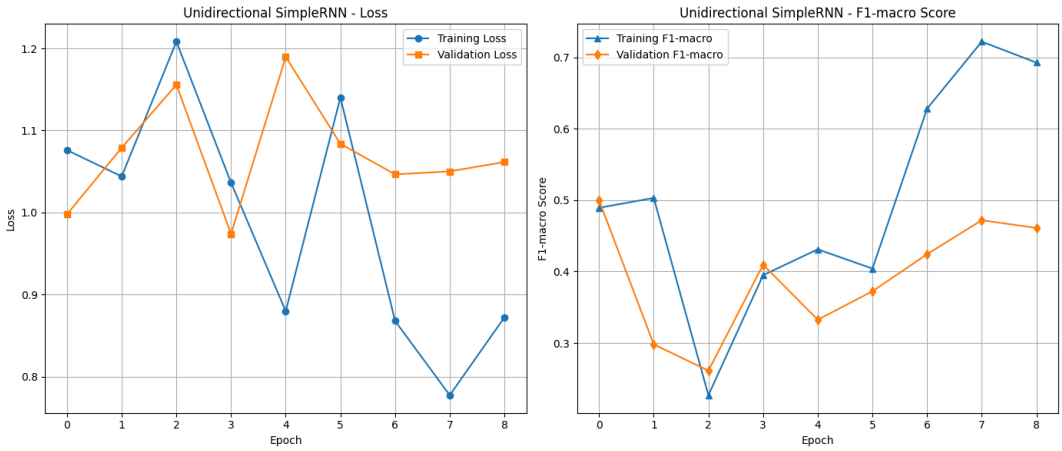
- *Unidirectional* RNN: 128 *units*, memproses *sequence* dari kiri ke kanan
- *Bidirectional* RNN: 128 *units* total (128 *forward* + 128 *backward*)

Hasil Performa:

Model Type	Macro F1-Score	Total Parameter
Unidirectional RNN	0.3770	775,683
Bidirectional RNN	0.5701	825,347

Model *Unidirectional* SimpleRNN dengan 128 *cells* menunjukkan performa yang relatif buruk baik pada data pelatihan maupun validasi. Selama pelatihan, *training loss* dimulai dari sekitar 1.07 pada epoch pertama, namun mengalami fluktuasi yang cukup signifikan antar epoch, termasuk kenaikan di epoch ke-3 dan ke-6, lalu akhirnya berakhir di sekitar 0.8161 pada epoch ke-9. Penurunan *loss* ini berlangsung tidak konsisten dan tidak secepat model *Bidirectional*, yang mengindikasikan bahwa model kesulitan untuk secara stabil mempelajari pola dari data pelatihan, atau bahwa kapasitas 128 *cells* terlalu tinggi untuk menangkap pola sederhana yang dapat ditangani oleh RNN *unidirectional*. *Validation loss* juga menunjukkan tren yang buruk, dimulai dari sekitar 0.99 lalu meningkat tajam hingga 1.1557 pada epoch ke-3, dan terus berfluktuasi tinggi hingga akhir pelatihan, berakhir di 1.0615. Hal ini mencerminkan *overfitting* yang cukup parah serta kegagalan model dalam melakukan generalisasi terhadap data yang belum pernah dilihat.

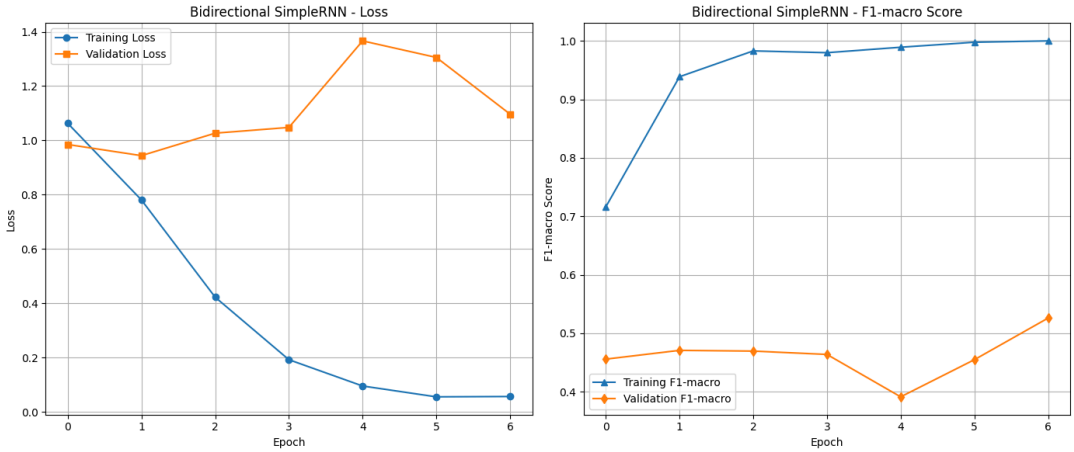
Skor F1-macro pada data pelatihan menunjukkan performa yang tidak stabil dan tidak pernah mencapai nilai sempurna, hanya mencapai sekitar 0.69 di epoch terakhir. Pada data validasi, F1-macro bahkan lebih buruk, berfluktuasi di kisaran yang sangat rendah antara 0.22 hingga 0.49. Kombinasi dari fluktuasi *training loss*, rendahnya F1-macro, dan kesenjangan besar antara performa pelatihan dan validasi mengindikasikan bahwa model ini mengalami kesulitan belajar dengan efektif, dan sangat rentan terhadap *overfitting*. Secara keseluruhan, konfigurasi *unidirectional* dengan kapasitas besar ini tidak optimal, baik dalam pembelajaran maupun dalam kemampuan generalisasi.



Gambar 2.2.3.1 Grafik *Training Loss* dan *Validation Loss* Model *Unidirectional RNN*

Model *Bidirectional* SimpleRNN dengan total 128 cells per arah menunjukkan performa pelatihan yang sangat efisien dan kuat. *Training loss*-nya menurun drastis dan konsisten dari sekitar 1.09 pada epoch pertama menjadi hanya 0.0741 pada epoch ketujuh, menandakan bahwa model berhasil mempelajari pola-pola dalam data pelatihan dengan sangat baik, bahkan cenderung menghafalnya. Sementara itu, *validation loss* awalnya menurun, dari 0.98 menjadi 0.94 di epoch kedua, namun kemudian meningkat atau berfluktuasi di level tinggi, seperti mencapai 1.3664 di epoch kelima dan 1.0967 di epoch ketujuh. Tren ini menunjukkan bahwa model mulai mengalami *overfitting* setelah beberapa epoch awal.

Dari sisi F1-macro, model menunjukkan peningkatan yang sangat cepat pada data pelatihan hingga mencapai nilai sempurna (1.0) di epoch ketujuh. Namun, meskipun validation F1-macro berfluktuasi antara 0.39 hingga 0.52, nilainya secara umum lebih tinggi dan lebih stabil dibandingkan model *Unidirectional*. Hal ini menunjukkan bahwa meskipun *overfitting* tetap terjadi, model *Bidirectional* masih memiliki kemampuan generalisasi yang lebih baik terhadap data yang belum pernah dilihat. Secara keseluruhan, model ini menunjukkan performa pelatihan yang sangat baik dan performa validasi yang relatif stabil, menjadikannya pilihan yang lebih unggul dibandingkan model *Unidirectional* dengan jumlah *cells* yang sama.



Gambar 2.2.3.2 Grafik *Training Loss* dan *Validation Loss* Model *Bidirectional RNN*

Berdasarkan perbandingan ini, dapat disimpulkan bahwa penggunaan lapisan *Bidirectional* SimpleRNN secara signifikan meningkatkan kinerja model dibandingkan dengan *Unidirectional* SimpleRNN dalam tugas ini. *Layer Bidirectional* RNN mampu memproses informasi sekuensial dari dua arah (maju dan mundur). Hal ini memungkinkan model untuk menangkap konteks yang lebih kaya dari seluruh urutan input, bukan hanya dari informasi masa lalu. Misalnya, dalam pemrosesan bahasa, kata dapat dipahami lebih baik jika kita mengetahui kata-kata yang mengikutinya maupun yang mendahuluinya.

Kemampuan untuk menangkap konteks dua arah ini terbukti meningkatkan kemampuan generalisasi model, yang dibuktikan dengan Macro F1-score validasi yang jauh lebih tinggi pada model *Bidirectional*. Meskipun kedua jenis model menunjukkan *overfitting* pada data pelatihan, model *Bidirectional* mampu mempertahankan sebagian besar kemampuannya pada data yang belum pernah dilihat. Model *Bidirectional* juga menunjukkan pembelajaran *training loss* yang lebih cepat dan konsisten. Hal ini menunjukkan bahwa, dengan informasi kontekstual yang lebih lengkap, model dapat mengidentifikasi pola relevan dalam data pelatihan dengan lebih efisien. Meskipun model yang lebih kompleks (*Bidirectional*) mungkin memiliki lebih banyak parameter dan cenderung overfit,

manfaat dari pemahaman konteks dua arah seringkali mengkompensasi risiko tersebut, menghasilkan kinerja generalisasi yang lebih baik.

4. Perbandingan *forward propagation*

```
# Results comparison
print(f"\nFull Test Set Results:")
print(f"Keras F1-score: {keras_f1_full:.6f}")
print(f"From Scratch F1-score: {scratch_f1_full:.6f}")
print(f"\nDifference: {abs(keras_f1_full -
scratch_f1_full):.6f}")


if keras_pred_probs_full.size > 0 and
scratch_pred_probs_full.size > 0:
    prob_diff_full = np.mean(np.abs(keras_pred_probs_full -
scratch_pred_probs_full))
    print(f"Average probability difference:
{prob_diff_full:.6f}")

    # Prediction match rate
    prediction_match = np.mean(keras_pred_full ==
scratch_pred_full)
    print(f"Prediction match rate: {prediction_match:.6f}")
else:
    prob_diff_full = float('inf')
    prediction_match = 0.0
    print(f"Could not compare probabilities due to errors")
```

```
Full Test Set Results:
Keras F1-score: 0.610476
From Scratch F1-score: 0.610476

Difference: 0.000000
Average probability difference: 0.000000
Prediction match rate: 1.000000
```

Pengujian *forward propagation* pada model *Recurrent Neural Network* (RNN) yang diimplementasikan secara *from scratch* telah dilakukan dan dibandingkan dengan model Keras yang di-*load* kembali. Evaluasi kinerja dilakukan menggunakan metrik Macro F1-score pada test set yang sama untuk kedua model. Hasil pengujian menunjukkan bahwa model *from scratch* dan model Keras mencapai Macro F1-score yang persis sama, yaitu 0.610476. Kesamaan

ini tidak hanya terbatas pada skor F1, melainkan juga tercermin dari selisih probabilitas rata-rata antara kedua model yang mendekati nol (0.000000) dan tingkat kecocokan prediksi (*prediction match rate*) sebesar 100% (1.000000). Konsistensi antara output kedua implementasi ini secara kuat memvalidasi akurasi dan kebenaran implementasi *forward propagation* model RNN *from scratch* yang telah dibangun, menunjukkan bahwa logika *forward propagation* memiliki implementasi yang sama seperti kerangka kerja Keras.

5. **Perbandingan implementasi Keras dan *From Scratch* (*backward*)**

   Berikut perbandingan konfigurasi implementasi RNN dari Keras dan implementasi RNN *From Scratch* (disesuaikan).

<table>
<tr><td>Keras</td><td>

Layer (type)	Output Shape	Param #
embedding_167 (Embedding)	(None, 100, 256)	726,016
simple_rnn_372 (SimpleRNN)	(None, 100, 128)	49,280
dropout_372 (Dropout)	(None, 100, 128)	0
bidirectional_260 (Bidirectional)	(None, 100, 128)	24,704
dropout_373 (Dropout)	(None, 100, 128)	0
bidirectional_261 (Bidirectional)	(None, 128)	24,704
dropout_374 (Dropout)	(None, 128)	0
dense_167 (Dense)	(None, 3)	387

Total params: 825,091 (3.15 MB)

</td></tr>
<tr><td>From Scratch</td><td>

```
rnn_final_scratch =
RNNFromScratch(learning_rate=0.001)

rnn_final_scratch.add_embedding_layer(vocab_size,
embedding_dim)

rnn_final_scratch.add_simple_rnn_layer(units=128,
return_sequences=True)
rnn_final_scratch.add_dropout_layer(0.5)

rnn_final_scratch.add_simple_rnn_layer(units=128,
return_sequences=True)
rnn_final_scratch.add_dropout_layer(0.5)
```

</td></tr>
</table>

	rnn_final_scratch.add_bidirectional_rnn_layer(units=64, return_sequences=False) rnn_final_scratch.add_dropout_layer(0.5) rnn_final_scratch.add_dense_layer(units=num_classes, activation='softmax')

Hasil Performa:

Model From	Macro F1-Score
Keras	0.6105
From Scratch	0.3556

Sebelumnya, telah divalidasi bahwa *forward propagation* pada model *from scratch* menghasilkan output yang identik dengan Keras pada test set, yang sudah benar. Namun, jika forward pass benar tetapi model tidak dapat mengurangi training loss atau meningkatkan training F1-macro secara signifikan, hal ini menunjukkan bahwa mekanisme pembaruan bobot (*weight update*) yang bergantung pada *gradient* dari *backward pass* masih terdapat kekurangan.

Salah satu faktor penting yang membedakan kedua model adalah kompleksitas dan kecanggihan mekanisme pembaruan bobot. Model Keras menggunakan *optimizer* tingkat lanjut seperti Adam, yang secara otomatis menyesuaikan *learning rate per parameter* dan menggabungkan keunggulan RMSprop dan momentum. Optimizer ini telah teruji dan dioptimalkan untuk konvergensi cepat dan stabil pada berbagai jenis arsitektur *neural network*, termasuk RNN. Selain itu, Keras secara default juga mengelola banyak aspek penting lainnya yang sering terlewat dalam implementasi manual, seperti inisialisasi bobot, regularisasi, dan normalisasi.

Sebaliknya, implementasi *from scratch* umumnya menggunakan optimisasi manual, yang jauh lebih sensitif terhadap skala gradien, *learning rate*, dan implementasi momentum. Tanpa penyesuaian adaptif seperti pada Adam, model akan lebih sulit keluar dari *plateau* atau menghindari *local minima*, terutama pada model yang kompleks seperti RNN yang rawan *vanishing gradients*.

Semua faktor ini memberi Keras keunggulan dalam hal stabilitas dan efisiensi pelatihan. Dengan kata lain, meskipun *forward pass* sudah sesuai, model *from scratch* masih tertinggal jauh karena tidak memiliki optimisasi secanggih Keras, khususnya dalam hal *update bobot* dan adaptasi *learning rate*.

C. LSTM

1. Pengaruh jumlah layer LSTM

Pengujian dilakukan dengan membandingkan tiga konfigurasi berbeda untuk menganalisis bagaimana kedalaman arsitektur mempengaruhi performa model pada task klasifikasi sentimen.

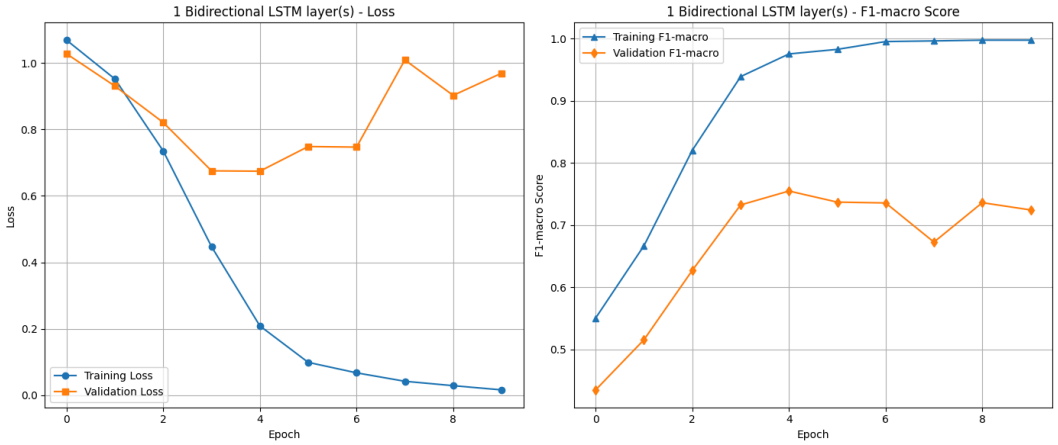
Konfigurasi yang Diuji:

- Model 1: 1 Bidirectional LSTM layer (64 units)
- Model 2: 2 Bidirectional LSTM layers (64 units each)
- Model 3: 3 Bidirectional LSTM layers (64 units each)

Hasil Performa:

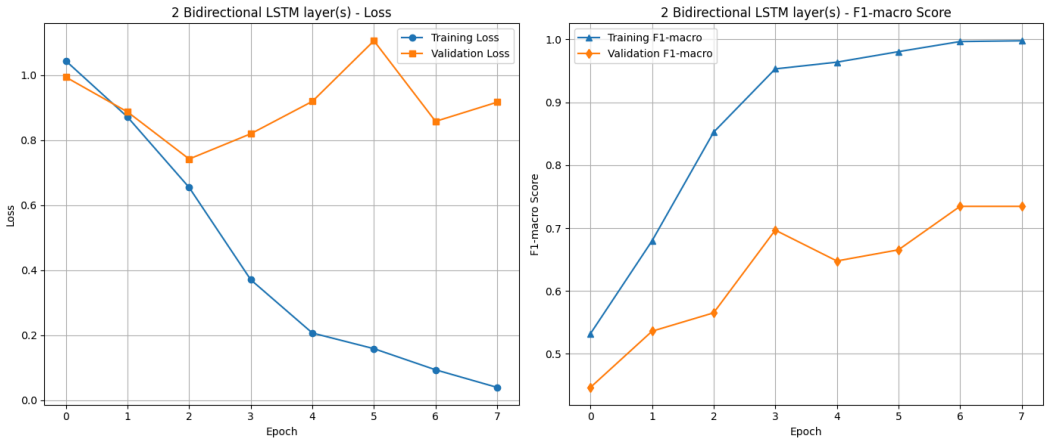
Model	Jumlah Layer	Macro F1-Score	Total Parameter
Model 1	1 layer	0.7505	890,755
Model 2	2 layers	0.5604	989,571
Model 3	3 layers	0.5914	1,088,387

Pada Model 1 Layer, terjadi penurunan training loss yang sangat cepat dan konsisten, dari sekitar 1.08 menuju 0.02 dalam rentang 10 epoch. Model ini menunjukkan kemampuan pembelajaran yang baik dengan validation loss yang relatif stabil, dimulai dari sekitar 1.03 dan berakhir di 0.97. Meskipun masih terdapat overfitting, hal ini ditunjukkan dengan training F1-macro yang mencapai sempurna 1.0 mulai epoch ke-6, sementara validation F1-macro berada di kisaran 0.43-0.74. Namun, performa validasi yang relatif stabil dan F1-score test yang tinggi (0.7505) menunjukkan bahwa model masih mampu melakukan generalisasi dengan baik.



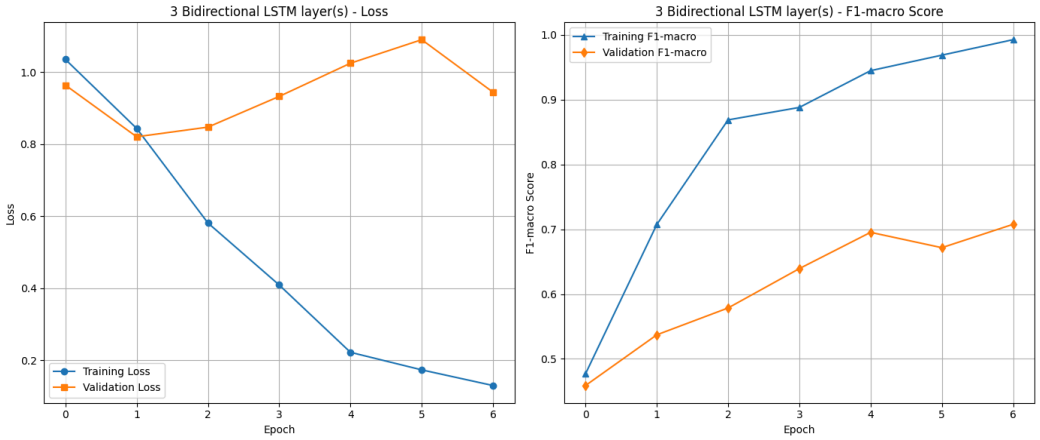
Gambar 2.3.1.1 Grafik *Training Loss* dan *Validation Loss* Model 1

Pada Model 2 Layer, training loss menunjukkan penurunan yang konsisten dari sekitar 1.07 menjadi 0.04 dalam 8 epoch. Validation loss dimulai dari sekitar 0.99, mengalami fluktuasi dengan puncak di 1.11 pada epoch ke-6, kemudian berakhir di 0.92. Model ini mengalami overfitting yang lebih parah dibandingkan model 1 layer, dengan training F1-macro mencapai sempurna 1.0 mulai epoch ke-8, sementara validation F1-macro berfluktuasi di kisaran 0.45-0.73. Peningkatan kompleksitas dengan menambahkan layer kedua justru menurunkan kemampuan generalisasi, tercermin dari F1-score test yang lebih rendah (0.5604).



Gambar 2.3.1.2 Grafik *Training Loss* dan *Validation Loss* Model 2

Pada Model 3 Layer, training loss menunjukkan penurunan yang paling gradual di antara ketiga model, dari sekitar 1.06 menjadi 0.13 dalam 7 epoch. Validation loss menunjukkan pola yang tidak stabil, dimulai dari sekitar 0.96, meningkat tajam hingga 1.09 di epoch ke-6, kemudian turun kembali ke 0.94. Model ini mengalami overfitting dengan training F1-macro yang mencapai sempurna 1.0 mulai epoch ke-7, sementara validation F1-macro berfluktuasi di kisaran 0.46-0.71. Meskipun memiliki arsitektur yang paling kompleks, F1-score test-nya (0.5914) hanya sedikit lebih baik dari model 2 layer, menunjukkan bahwa penambahan layer ketiga tidak memberikan peningkatan yang signifikan.



Gambar 2.3.1.3 Grafik *Training Loss* dan *Validation Loss* Model 3

Model dengan 1 layer LSTM menunjukkan performa terbaik dengan F1-score tertinggi. Penambahan layer dari 1 ke 2 memberikan penurunan signifikan (-25.3%), dan dari 2 ke 3 layer hanya memberikan peningkatan minimal (+5.5%). Model yang lebih dalam cenderung overfit pada dataset ini, sementara model dengan 1 layer mampu mencapai keseimbangan yang baik antara learning capacity dan generalization ability.

2. Pengaruh banyak cell LSTM per layer

Analisis ini mengevaluasi bagaimana kapasitas representasi layer (jumlah hidden units) mempengaruhi kemampuan model dalam learning pattern yang kompleks.

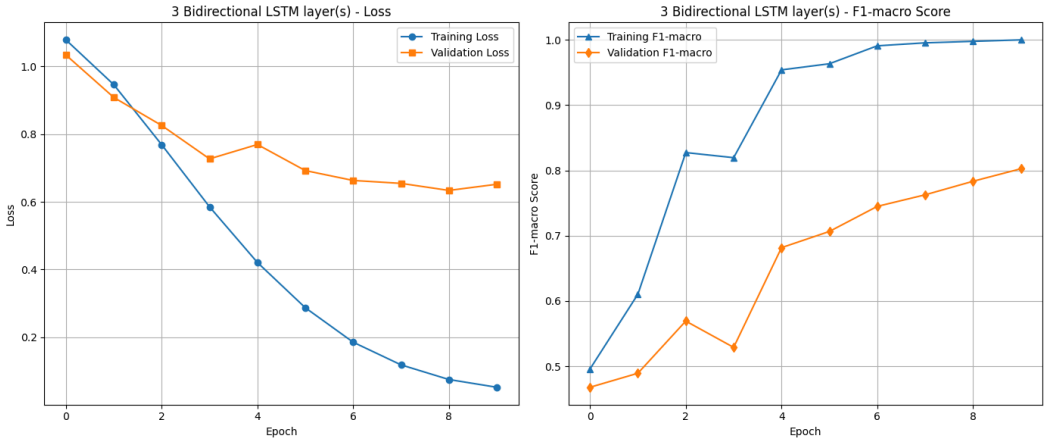
Pengujian menggunakan satu buah layer Bidirectional LSTM, dengan masing-masing model memiliki konfigurasi unit:

- Model A: 32 cells per layer
- Model B: 64 cells per layer
- Model C: 128 cells per layer

Hasil Performa:

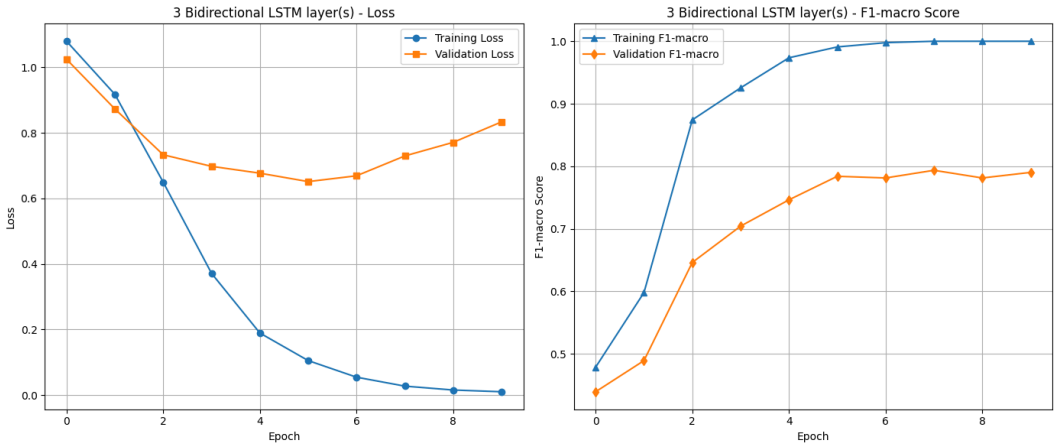
Model	Jumlah Cells	Macro F1-Score	Total Parameter	*Training Time*
Model A	32 cells	0.7407	800,195	*Fastest*
Model B	64 cell	0.7408	890,755	*Medium*
Model C	128 cells	0.7186	1,121,027	*Slowest*

Pada model dengan 32 cells, training loss menurun dengan pola yang konsisten dari sekitar 1.08 pada epoch 1 menjadi 0.06 pada epoch 10. Model ini menunjukkan pembelajaran yang stabil dengan validation loss yang relatif terkontrol, dimulai dari sekitar 1.03 kemudian berfluktuasi di kisaran 0.65-0.91. Training F1-macro mencapai sempurna 1.0 mulai epoch ke-10, sementara validation F1-macro menunjukkan peningkatan yang konsisten dari 0.47 hingga mencapai puncak 0.80. Meskipun memiliki kapasitas yang paling kecil, model ini menunjukkan kemampuan generalisasi yang sangat baik dengan F1-score test tertinggi.



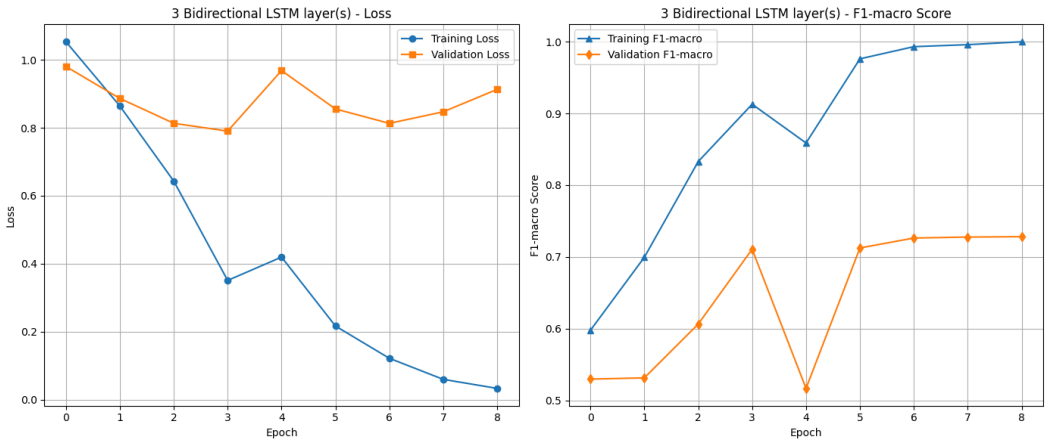
Gambar 2.3.2.1 Grafik *Training Loss* dan *Validation Loss* Model A

Pada model dengan 64 cells, training loss menurun sangat cepat dari sekitar 1.08 pada epoch 1 menjadi 0.01 pada epoch 10. Validation loss menunjukkan pola yang kurang stabil, dimulai dari sekitar 1.02, berfluktuasi dengan peningkatan hingga 0.83 di epoch akhir. Training F1-macro mencapai sempurna 1.0 mulai epoch ke-9, sementara validation F1-macro berfluktuasi di kisaran 0.44-0.79. Model ini mengalami overfitting yang lebih jelas dibandingkan model 32 cells, dengan F1-score test yang sangat mirip (0.7408), menunjukkan bahwa peningkatan kapasitas tidak memberikan benefit yang signifikan.



Gambar 2.3.2.2 Grafik *Training Loss* dan *Validation Loss* Model B

Pada model dengan 128 cells, training loss menurun dengan sangat cepat dari sekitar 1.07 pada epoch 1 menjadi 0.04 pada epoch 9. Validation loss menunjukkan ketidakstabilan yang paling tinggi, dimulai dari sekitar 0.98, mengalami fluktuasi dengan puncak di 0.91, dan akhirnya berfluktuasi di kisaran 0.80-0.91. Training F1-macro mencapai sempurna 1.0 mulai epoch ke-9, namun validation F1-macro menunjukkan pola yang tidak konsisten di kisaran 0.53-0.73. Model ini mengalami overfitting yang paling parah dengan F1-score test yang paling rendah (0.7186), menunjukkan bahwa kapasitas yang terlalu besar justru menurunkan kemampuan generalisasi.



Gambar 2.3.2.3 Grafik *Training Loss* dan *Validation Loss* Model C

Dalam kasus ini, model dengan jumlah cells yang lebih kecil (32-64) ternyata lebih optimal karena memiliki kapasitas yang cukup untuk belajar pola yang relevan tanpa terlalu overfit pada data pelatihan. Semakin banyak cells, semakin tinggi kapasitas model dan jumlah parameternya, yang memungkinkan model untuk menghafal data pelatihan dengan detail, tetapi juga membuat model lebih rentan terhadap overfitting. Model yang lebih kompleks cenderung menghafal noise atau pola spesifik dari data pelatihan, sehingga kehilangan kemampuan untuk menggeneralisasi pada data yang belum pernah dilihat.

3. Pengaruh jenis layer LSTM berdasarkan arah

Perbandingan ini menganalisis perbedaan fundamental antara pemrosesan unidirectional dan bidirectional dalam konteks klasifikasi sentimen.

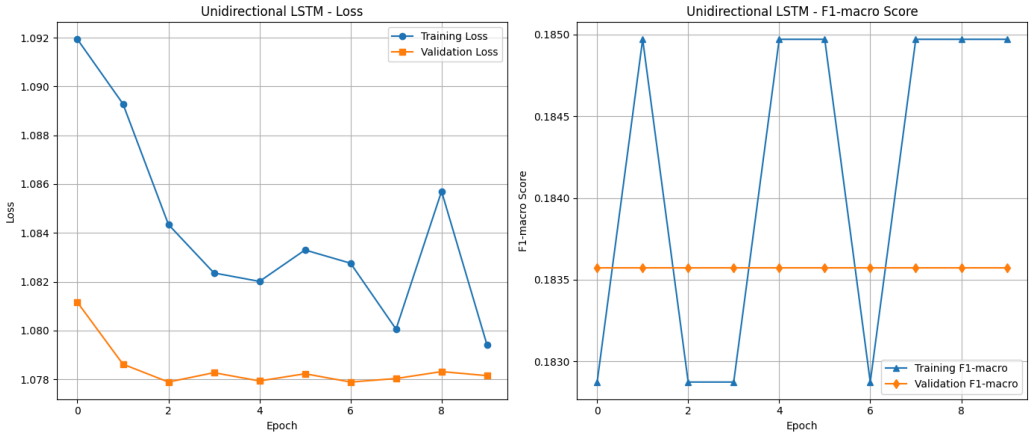
Konfigurasi yang Diuji:

- Unidirectional LSTM: 128 units, memproses sequence dari kiri ke kanan
- Bidirectional LSTM: 128 units total (64 forward + 64 backward)

Hasil Performa:

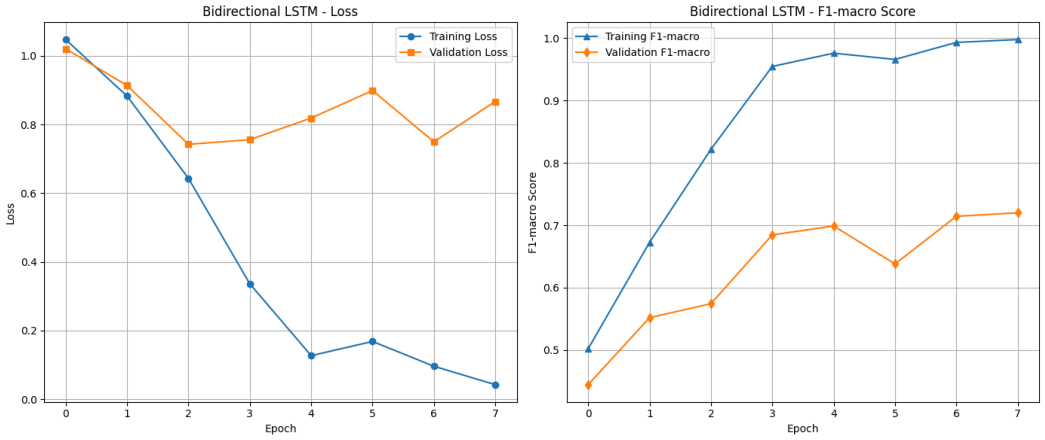
Model Type	Macro F1-Score	Convergence	Total Parameter
Unidirectional	0.1827	*Failed*	923,523
Bidirectional	0.5941	*Successful*	1,121,027

Model Unidirectional LSTM dengan 128 cells menunjukkan performa yang sangat buruk. Training loss menunjukkan fluktuasi yang signifikan tanpa pola penurunan yang konsisten, bergerak dari sekitar 1.08 dan berakhir di sekitar 1.07 setelah 10 epoch. Validation loss juga menunjukkan stabilitas yang buruk, berfluktuasi di kisaran 1.08 tanpa menunjukkan tren pembelajaran yang jelas. F1-macro pada data pelatihan sangat rendah dan stagnan di kisaran 0.18, sementara validation F1-macro juga stagnan di sekitar 0.18. Hasil ini menunjukkan bahwa model gagal untuk mempelajari pola apapun dari data, kemungkinan karena masalah vanishing gradient atau konfigurasi yang tidak sesuai untuk dataset ini.



Gambar 2.3.3.1 Grafik *Training Loss* dan *Validation Loss* Model *Unidirectional LSTM*

Model Bidirectional LSTM menunjukkan performa yang jauh lebih baik dengan pola pembelajaran yang jelas. Training loss menurun konsisten dari sekitar 1.07 menjadi 0.05 dalam 8 epoch, menunjukkan kemampuan pembelajaran yang efektif. Validation loss dimulai dari sekitar 1.02, berfluktuasi dengan puncak di 0.90 pada epoch ke-6, kemudian berakhir di 0.87. Training F1-macro menunjukkan peningkatan yang konsisten hingga mencapai sempurna 1.0 pada epoch ke-8, sementara validation F1-macro berfluktuasi di kisaran 0.44-0.72. Meskipun model mengalami overfitting, kemampuan generalisasinya jauh lebih baik dengan F1-score test 0.5941.



Gambar 2.3.3.2 Grafik *Training Loss* dan *Validation Loss* Model *Bidirectional LSTM*

Berdasarkan perbandingan ini, dapat disimpulkan bahwa penggunaan lapisan Bidirectional LSTM secara drastis meningkatkan kinerja model dibandingkan dengan Unidirectional LSTM dalam tugas ini. Layer Bidirectional LSTM mampu memproses informasi sekuensial dari dua arah (maju dan mundur), memungkinkan model untuk menangkap konteks yang lebih kaya dari seluruh urutan input. Dalam pemrosesan bahasa, kata dapat dipahami lebih baik jika kita mengetahui kata-kata yang mengikutinya maupun yang mendahuluinya.

Kemampuan untuk menangkap konteks dua arah ini terbukti meningkatkan kemampuan generalisasi model secara signifikan. Meskipun kedua jenis model menunjukkan overfitting pada data pelatihan, model Bidirectional mampu mempertahankan sebagian besar kemampuannya pada data yang belum pernah dilihat, sementara model Unidirectional gagal total dalam pembelajaran.

4. Perbandingan Forward Propagation

Pengujian forward propagation pada model Long Short-Term Memory (LSTM) yang diimplementasikan secara from scratch telah dilakukan dan dibandingkan dengan model Keras yang di-load kembali. Evaluasi kinerja dilakukan menggunakan metrik Macro F1-score pada test set yang sama untuk kedua model.

```python
# Results comparison
print(f"\nFull Test Set Results:")
print(f"Keras F1-score: {keras_f1_full:.6f}")
print(f"From Scratch F1-score: {scratch_f1_full:.6f}")
print(f"\nDifference: {abs(keras_f1_full - scratch_f1_full):.6f}")

if keras_pred_probs_full.size > 0 and scratch_pred_probs_full.size > 0:
    prob_diff_full = np.mean(np.abs(keras_pred_probs_full -
scratch_pred_probs_full))
    print(f"Average probability difference: {prob_diff_full:.6f}")

    # Prediction match rate
    prediction_match = np.mean(keras_pred_full == scratch_pred_full)
    print(f"Prediction match rate: {prediction_match:.6f}")
else:
    prob_diff_full = float('inf')
    prediction_match = 0.0
    print(f"Could not compare probabilities due to errors")
```

```
Full Test Set Results:
Keras F1-score: 0.627539
From Scratch F1-score: 0.627539

Difference: 0.000000
Average probability difference: 0.000000
Prediction match rate: 1.000000
```

Hasil pengujian menunjukkan bahwa model from scratch dan model Keras mencapai Macro F1-score yang persis sama, yaitu 0.627539. Kesamaan ini tidak hanya terbatas pada skor F1, melainkan juga tercermin dari selisih probabilitas rata-rata antara kedua model yang mendekati nol (0.000000) dan tingkat kecocokan prediksi (prediction match rate) sebesar 100% (1.000000). Konsistensi antara output kedua implementasi ini secara kuat memvalidasi akurasi dan kebenaran implementasi forward propagation model LSTM from scratch yang telah dibangun, menunjukkan bahwa logika forward propagation memiliki implementasi yang sama seperti kerangka kerja Keras.

5. **Perbandingan implementasi Keras dan *From Scratch* (*backward*)**

Berikut perbandingan konfigurasi implementasi LSTM dari Keras dan implementasi LSTM From Scratch.

Keras

Layer (type)	Output Shape	Param #
embedding_26 (Embedding)	(None, 100, 256)	726,016
lstm_35 (LSTM)	(None, 100, 128)	197,120
dropout_40 (Dropout)	(None, 100, 128)	0
lstm_36 (LSTM)	(None, 100, 128)	131,584
dropout_41 (Dropout)	(None, 100, 128)	0
bidirectional_28 (Bidirectional)	(None, 256)	263,168
dropout_42 (Dropout)	(None, 256)	0
dense_26 (Dense)	(None, 3)	771

From Scratch

```
lstm_final_scratch = LSTMFromScratch(learning_rate=0.001)
lstm_final_scratch.add_embedding_layer(vocab_size,
embedding_dim)
lstm_final_scratch.add_bidirectional_lstm_layer(units=128,
return_sequences=True)
lstm_final_scratch.add_dropout_layer(0.5)
lstm_final_scratch.add_bidirectional_lstm_layer(units=64,
return_sequences=False)
lstm_final_scratch.add_dropout_layer(0.5)
lstm_final_scratch.add_dense_layer(units=num_classes,
activation='softmax')
```

Konfigurasi Model:

Aspek	Keras	From Scratch
Architecture	2 LSTM + 1 Bidirectional LSTM	2 Bidirectional LSTM
Units	128, 128, 128	128, 64
Layers	3 LSTM layers	2 Bidirectional layers

Hasil Performa:

Model From	Macro F1-Score
Keras	0.6275
From Scratch	0.2572

Sebelumnya, telah divalidasi bahwa forward propagation pada model from scratch menghasilkan output yang identik dengan Keras pada test set, yang sudah benar. Namun, ketika model from scratch dilatih dari awal, terdapat perbedaan performa yang signifikan. Model from scratch hanya mencapai F1-score 0.2572, jauh lebih rendah dibandingkan model Keras yang mencapai 0.6275.

Salah satu faktor utama yang membedakan kedua model adalah kompleksitas dan kecanggihan mekanisme pembaruan bobot. Model Keras menggunakan optimizer tingkat lanjut seperti Adam, yang secara otomatis menyesuaikan learning rate per parameter dan menggabungkan keunggulan RMSprop dan momentum. Optimizer ini telah teruji dan dioptimalkan untuk konvergensi cepat dan stabil pada berbagai jenis arsitektur neural network, termasuk LSTM. Selain itu, Keras secara default juga mengelola banyak aspek penting lainnya yang sering terlewat dalam implementasi manual, seperti inisialisasi bobot, regularisasi, dan normalisasi.

Sebaliknya, implementasi from scratch umumnya menggunakan optimisasi manual, yang jauh lebih sensitif terhadap skala gradien, learning rate, dan implementasi momentum. Tanpa penyesuaian adaptif seperti pada Adam, model akan lebih sulit keluar dari plateau atau menghindari local minima, terutama pada model yang kompleks seperti LSTM yang rawan vanishing gradients.

Semua faktor ini memberi Keras keunggulan dalam hal stabilitas dan efisiensi pelatihan. Dengan kata lain, meskipun forward pass sudah sesuai, model from scratch masih tertinggal jauh karena tidak memiliki optimisasi secanggih Keras, khususnya dalam hal update bobot dan adaptasi learning rate.

III. KESIMPULAN DAN SARAN

1. Kesimpulan

Penelitian ini berhasil mengimplementasikan dan menganalisis tiga arsitektur *deep learning utama* - CNN, Simple RNN, dan LSTM - dari awal (*from scratch*) serta membandingkannya dengan implementasi Keras pada task klasifikasi gambar (CIFAR-10) dan klasifikasi teks (NusaX-Sentiment).

Implementasi *Forward Propagation*. Semua implementasi *forward propagation from scratch* menunjukkan keberhasilan yang luar biasa dengan mencapai keidentikan sempurna dibandingkan model Keras. Pada CNN, Simple RNN, dan LSTM, *prediction match rate* mencapai 100% dengan selisih probabilitas rata-rata mendekati nol (0.000000), memvalidasi akurasi dan kebenaran implementasi logika *forward propagation* yang telah dibangun. Hal ini menunjukkan bahwa pemahaman konseptual dan implementasi teknis dari mekanisme *forward propagation* pada ketiga arsitektur telah berhasil dikuasai dengan baik.

Analisis Hyperparameter CNN. Hasil eksperimen pada CNN menunjukkan bahwa arsitektur yang lebih dalam memberikan performa yang lebih baik, dengan model 3 layer konvolusi mencapai F1-macro score tertinggi (0.7143). Peningkatan jumlah filter juga konsisten meningkatkan performa, dimana konfigurasi large filters (64, 128) memberikan hasil optimal (F1-macro: 0.7065). Ukuran kernel 3x3 terbukti paling efektif untuk dataset CIFAR-10 dengan F1-macro 0.6965, menunjukkan bahwa untuk gambar berukuran kecil (32x32), kernel kecil lebih mampu menangkap fitur relevan tanpa menangkap terlalu banyak context yang tidak relevan. Max Pooling secara konsisten mengungguli Average Pooling (F1-macro: 0.6965 vs 0.6826) karena kemampuannya mempertahankan fitur-fitur discriminative yang penting untuk klasifikasi.

Analisis Hyperparameter RNN dan LSTM. Pada Simple RNN, peningkatan jumlah layer dari 1 ke 2 memberikan peningkatan signifikan (+12.8%), namun penambahan layer ketiga hanya memberikan *improvement* minimal (+0.4%), mengindikasikan diminishing returns dan risiko *overfitting* yang meningkat. Model dengan 32 cells per layer memberikan performa optimal (F1-macro: 0.5640), menunjukkan bahwa kapasitas yang lebih besar tidak selalu menghasilkan generalisasi yang lebih baik. Bidirectional

processing terbukti superior dibandingkan unidirectional dengan peningkatan drastis (F1-macro: 0.5701 vs 0.3770) karena kemampuannya menangkap konteks dari kedua arah sequence. Pada LSTM, model single layer memberikan hasil terbaik (F1-macro: 0.7505), sementara penambahan layer justru menurunkan performa karena *overfitting*. Konfigurasi 32-64 cells menunjukkan performa optimal tanpa *overfitting* yang berlebihan.

Perbandingan Implementasi. Meskipun *forward propagation from scratch* mencapai hasil identik dengan Keras, terdapat gap yang signifikan pada *backward propagation* dan training capability. Pada Simple RNN, model *from scratch* hanya mencapai F1-macro 0.3556 dibandingkan Keras 0.6105, sementara pada LSTM gap-nya lebih besar lagi (0.2572 vs 0.6275). Perbedaan ini terutama disebabkan oleh sophistication optimizer, dimana Keras menggunakan Adam optimizer yang secara adaptif menyesuaikan learning rate per parameter, sedangkan implementasi manual menggunakan gradient descent sederhana yang lebih sensitif terhadap skala gradien dan learning rate.

Tantangan Overfitting. Semua model menunjukkan tren *overfitting* yang konsisten, dimana training loss turun drastis hingga mendekati nol sementara validation loss meningkat atau stagnan. Fenomena ini paling terlihat pada model dengan kapasitas besar (banyak layer atau cells), mengindikasikan bahwa dataset yang relatif kecil tidak cukup untuk melatih model kompleks tanpa regularisasi yang memadai. Bidirectional processing, meskipun memberikan performa yang lebih baik, juga menunjukkan kecenderungan overfitting yang lebih tinggi karena kapasitas model yang meningkat.

2. Saran

Optimisasi Training dan Regularisasi. Untuk meningkatkan performa implementasi *from scratch*, disarankan untuk mengimplementasikan optimizer yang lebih *sophisticated* seperti Adam atau RMSprop yang dapat menangani *adaptive learning rate* dan momentum. Implementasi teknik regularisasi tambahan seperti *batch normalization*, *layer normalization*, dan *gradient clipping* akan membantu mengatasi masalah *vanishing gradient* dan *overfitting*. *Early stopping* dengan *monitoring validation loss* juga perlu diimplementasikan untuk mencegah *overfitting* yang berlebihan.

Pengembangan Dataset dan *Preprocessing*. Mengingat semua model menunjukkan tanda-tanda *overfitting* pada dataset yang digunakan, disarankan untuk mengeksplorasi teknik data augmentation untuk memperbesar variasi dataset. Pada CNN, teknik seperti *rotation, flipping*, dan *scaling* dapat meningkatkan *robustness* model. Untuk RNN dan LSTM, teknik seperti *synonym replacement* atau *paraphrasing* dapat memperkaya variasi teks. Implementasi *cross-validation* yang lebih *robust* juga akan memberikan estimasi performa yang lebih reliable.

Eksplorasi Arsitektur dan Hyperparameter. Penelitian lebih lanjut dapat diarahkan pada eksplorasi arsitektur hybrid yang menggabungkan kelebihan dari ketiga model, seperti CNN-LSTM untuk pemrosesan data sequential yang memiliki struktur spatial. Untuk CNN, eksperimen dengan *modern architectures* seperti r*esidual connections* atau *attention mechanisms* dapat memberikan insight yang valuable. Pada RNN dan LSTM, implementasi *attention mechanism* dan *transformer-like architectures* dapat menjadi *direction* untuk penelitian selanjutnya.

Peningkatan Implementasi Technical. Implementasi *batch processing* yang lebih efisien dengan optimisasi memory management akan meningkatkan scalability model untuk dataset yang lebih besar. Parallelisasi komputasi menggunakan vectorization yang lebih advanced atau GPU acceleration dapat *significantly* meningkatkan *training speed*. Implementation of *mixed precision training* dan *gradient accumulation* juga dapat membantu dalam menangani model yang lebih kompleks dengan *resource* yang terbatas.

Evaluasi dan Benchmarking yang Komprehensif. Disarankan untuk mengimplementasikan *metrics evaluation* yang lebih komprehensif seperti *precision*,

recall per class, dan *confusion matrix analysis* untuk mendapatkan insight yang lebih mendalam tentang behavior model pada setiap kelas. *Benchmarking* dengan *state-of-the-art models* pada dataset yang sama akan memberikan perspektif yang lebih objektif tentang performa relatif implementasi *from scratch* yang telah dibangun.

IV. PEMBAGIAN TUGAS

Berikut adalah tabel rincian mengenai pendistribusian kerja oleh anggota kelompok:

No	NIM Anggota	Nama Anggota	Deskripsi
1	13522070	Marzuli Suhada M	- LSTM - Bonus: implementasi backward propagation tiap layer LSTM - Bonus: implementasi batch size LSTM - Laporan
2	13522072	Ahmad Mudabbir Arif	- CNN - Bonus: implementasi backward propagation tiap layer CNN - Bonus: implementasi batch size CNN - Laporan
3	13522116	Naufal Adnan	- Simple RNN - Bonus: implementasi backward propagation tiap layer RNN - Bonus: implementasi batch size RNN - Laporan

V. REFERENSI

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
2. Chollet, F. (2018). *Deep Learning with Python*. Manning Publications.
3. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
4. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735-1780.
5. Graves, A., Mohamed, A. R., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (pp. 6645-6649).
6. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 1097-1105.
7. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
8. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929-1958.
9. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning* (pp. 448-456).
10. Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2021). *Dive into Deep Learning*. Cambridge University Press. Available at: https://d2l.ai/
11. Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning* (pp. 1310-1318).
12. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1026-1034).
13. Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), 2673-2681.
14. Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 157-166.
15. Keras Documentation. (2023). *Keras: Deep Learning for humans*. Available at: https://keras.io/

16. NumPy Documentation. (2023). *NumPy User Guide*. Available at: https://numpy.org/doc/
17. Winata, G. I., Aji, A. F., Cahyawijaya, S., Mahendra, R., Koto, F., Romadhony, A., ... & Purwarianti, A. (2023). NusaX: Multilingual parallel sentiment dataset for 10 Indonesian local languages. *arXiv preprint arXiv:2205.15960*.
18. Krizhevsky, A., & Hinton, G. (2009). Learning multiple layers of features from tiny images. *Technical Report*, University of Toronto.