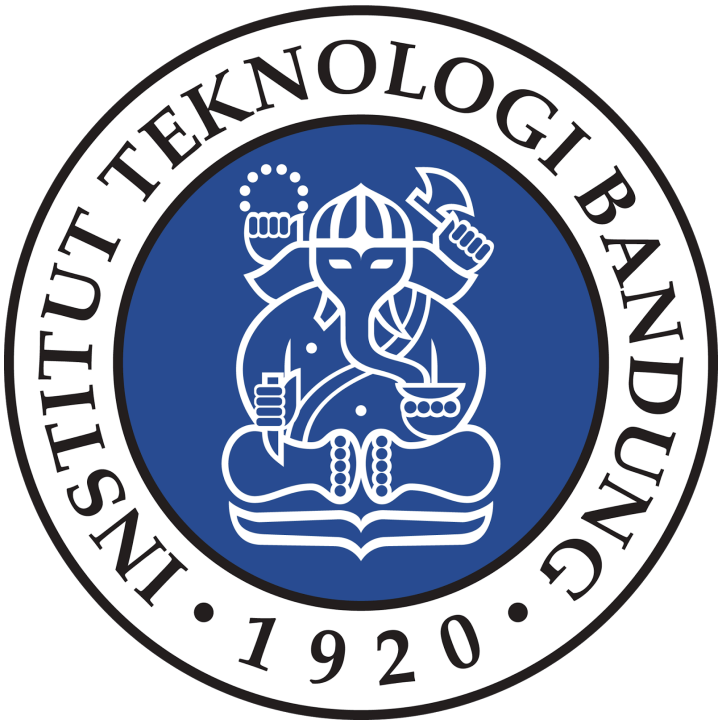
Laporan Tugas Besar 2
IF3270 Pembelajaran Mesin
Implementasi Feed Forward Propagation CNN-RNN-LSTM
Semester II Tahun 2024/2025



Disusun Oleh:
Muhammad Althariq Fairuz 13522027
Randy Verdian 13522067
Rayhan Fadhlan Azka 13522095

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025

DAFTAR ISI

BAB 1
Deskripsi Tugas

Tugas besar ini bertujuan untuk mengimplementasikan forward propagation dari tiga arsitektur neural network secara manual menggunakan NumPy, tanpa bergantung pada library machine learning seperti TensorFlow atau PyTorch. Implementasi manual ini dirancang untuk memberikan pemahaman mendalam tentang bagaimana setiap komponen neural network bekerja pada level matematika fundamental.

BAB 2
Pembahasan

2.1. CNN

2.1.1. Kelas dan Atribut

1. Kelas Conv2DLayer

Kelas yang mengimplementasikan layer konvolusi 2D untuk ekstraksi fitur hierarkis dari gambar input melalui operasi konvolusi dengan kernel yang dapat dipelajari.

Atribut:

- `filters` (int): Jumlah filter/kernel yang digunakan untuk menghasilkan feature map berbeda
- `kernel_size` (Tuple[int, int]): Dimensi kernel dalam format (tinggi, lebar) untuk menentukan area receptive field
- `stride` (Tuple[int, int]): Langkah pergeseran kernel saat sliding window, default (1,1)
- `padding` (str): Strategi padding ('valid' = tanpa padding, 'same' = padding untuk mempertahankan dimensi)
- `activation` (str): Fungsi aktivasi yang diterapkan setelah konvolusi ('relu', 'sigmoid', 'tanh', 'linear')
- `weights` (np.ndarray): Tensor bobot kernel dengan shape (kernel_h, kernel_w, input_channels, output_channels)
- `bias` (np.ndarray): Vektor bias untuk setiap filter dengan shape (output_channels,)

Method:

- `__init__()`: Konstruktor untuk menginisialisasi parameter layer konvolusi
- `load_weights(weights, bias)`: Memuat bobot dan bias dari model Keras yang telah dilatih
- `_pad_input(inputs)`: Method private untuk menerapkan padding pada input sesuai strategi yang dipilih
- `_apply_activation(x)`: Method private untuk menerapkan fungsi aktivasi pada output konvolusi
- `forward(inputs)`: Method utama untuk melakukan forward pass konvolusi pada batch input

2. Kelas PoolingLayer

Kelas yang mengimplementasikan layer pooling untuk downsampling feature map, mengurangi dimensi spasial dan computational complexity sambil mempertahankan informasi penting.

Atribut:

- `pool_size` (Tuple[int, int]): Dimensi pooling window dalam format (tinggi, lebar), default (2,2)
- `stride` (Tuple[int, int]): Langkah pergeseran pooling window, default (2,2)
- `pool_type` (str): Jenis operasi pooling ('max' untuk mengambil nilai maksimum, 'average' untuk rata-rata)

Method:

- `__init__()`: Konstruktor untuk menginisialisasi parameter pooling layer
- `forward(inputs)`: Method untuk melakukan operasi pooling pada input dengan mengiterasi setiap pooling window

3. Kelas FlattenLayer

Kelas sederhana yang mengubah tensor multidimensi menjadi vektor 1D untuk transisi dari convolutional layers ke fully connected layers.

Atribut:

Tidak memiliki atribut yang dapat dikonfigurasi.

Method:

- `forward(inputs)`: Method untuk meratakan tensor 4D (batch_size, height, width, channels) menjadi 2D (batch_size, features)

4. Kelas DenseLayer

Kelas yang mengimplementasikan fully connected layer dimana setiap neuron terhubung dengan semua neuron di layer sebelumnya, umumnya digunakan untuk klasifikasi final.

Atribut:

- `units` (int): Jumlah neuron/unit dalam layer ini
- `activation` (str): Fungsi aktivasi yang diterapkan pada output linear ('relu', 'sigmoid', 'softmax', 'tanh', 'linear')
- `weights` (np.ndarray): Matriks bobot dengan shape (input_features, units)
- `bias` (np.ndarray): Vektor bias dengan shape (units,)

Method:

- `__init__()`: Konstruktor untuk menginisialisasi parameter dense layer

- `load_weights(weights, bias)`: Memuat bobot dan bias dari model Keras
- `_apply_activation(x)`: Method private untuk menerapkan fungsi aktivasi pada output linear
- `forward(inputs)`: Method untuk melakukan transformasi linear dan aktivasi pada input

5. Kelas CNNModel

Kelas container utama yang menggabungkan semua layer dalam arsitektur CNN dan mengatur alur data dari input hingga output prediksi.

Atribut:

- `input_shape` (Tuple[int, int, int]): Dimensi input dalam format (height, width, channels)
- `num_classes` (int): Jumlah kelas target untuk klasifikasi
- `layers` (List): Daftar berurutan semua layer yang membentuk arsitektur model

Method:

- `__init__()`: Konstruktor untuk menginisialisasi model dengan input shape dan jumlah kelas
- `add_conv2d()`: Builder method untuk menambahkan layer konvolusi ke arsitektur
- `add_pooling()`: Builder method untuk menambahkan layer pooling ke arsitektur
- `add_flatten()`: Builder method untuk menambahkan layer flatten ke arsitektur
- `add_dense()`: Builder method untuk menambahkan layer dense ke arsitektur
- `load_keras_weights()`: Memuat bobot dari file model Keras (.h5) ke layer-layer yang sesuai
- `forward(inputs)`: Method utama untuk melakukan forward pass melalui seluruh arsitektur
- `predict(inputs)`: Method untuk mendapatkan prediksi kelas dengan argmax pada output probabilities
- `predict_proba(inputs)`: Method untuk mendapatkan distribusi probabilitas untuk setiap kelas
- `summary()`: Method untuk mencetak ringkasan arsitektur model termasuk parameter setiap layer

2.1.2. Tahapan *Forward Propagation*

Tahap 1: Input Layer

Forward propagation dimulai dengan menerima batch data gambar sebagai input dengan dimensi (batch_size, height, width, channels). Untuk dataset CIFAR-10, dimensi input adalah (32, 32, 32, 3) yang berarti 32 gambar dengan ukuran 32×32 pixel dan 3 channel warna RGB. Data gambar biasanya telah dinormalisasi ke dalam rentang [0,1] atau [-1,1] untuk mempercepat proses konvergensi selama training. Input layer tidak melakukan transformasi apapun, hanya meneruskan data mentah ke layer berikutnya dalam bentuk tensor yang siap diproses.

Tahap 2: Konvolusi (Conv2D Layer)

Tahap konvolusi merupakan inti dari CNN yang bertanggung jawab untuk ekstraksi fitur dari gambar input. Proses dimulai dengan penentuan padding, dimana jika menggunakan 'same' padding, input akan ditambahi border berisi nilai nol agar ukuran output sama dengan input, sedangkan 'valid' padding tidak menambahkan padding. Selanjutnya, setiap filter berukuran kernel_size bergeser melintasi seluruh area input dengan langkah stride. Pada setiap posisi, filter melakukan operasi konvolusi dengan mengalikan setiap elemen filter dengan elemen input yang bersesuaian, menjumlahkan hasilnya, dan menambahkan bias.

Konvolusi:

$$\text{Output}[i,j] = \sum(\text{Input}[i \cdot \text{stride} : (i \cdot \text{stride} + \text{kernel\_h}), j \cdot \text{stride} : (j \cdot \text{stride} + \text{kernel\_w})] * \text{Filter}) + \text{Bias}$$

Dimensi output:

$$\text{output\_height} = \frac{\text{input\_height} + 2 \cdot \text{padding} - \text{kernel\_height}}{\text{stride}} + 1$$

$$\text{output\_width} = \frac{\text{input\_width} + 2 \cdot \text{padding} - \text{kernel\_width}}{\text{stride}} + 1$$

Tahap 3: Aktivasi

Setelah operasi konvolusi, hasil linear diproses melalui fungsi aktivasi untuk menambahkan non-linearitas ke dalam jaringan. Fungsi aktivasi yang paling umum digunakan adalah ReLU (Rectified Linear Unit) yang mengubah semua nilai negatif menjadi nol sambil mempertahankan nilai positif. Proses ini penting karena tanpa fungsi aktivasi, seluruh jaringan neural hanya akan menjadi transformasi linear yang kompleks. Fungsi aktivasi memungkinkan jaringan untuk mempelajari pola-pola non-linear yang kompleks dalam data.

Fungsi ReLU:

$$f(x) = \max(0, x)$$

Fungsi Sigmoid:

$$f(x) = \frac{1}{1 + \exp(-x)}$$

Tahap 4: Pooling

Tahap pooling bertujuan untuk mengurangi dimensi spasial dari feature maps sambil mempertahankan informasi yang paling penting. Max pooling mengambil nilai maksimum dari setiap region berukuran pool_size, sedangkan average pooling mengambil nilai rata-rata. Proses ini dilakukan dengan menggeser pooling window melintasi feature maps dengan langkah stride tertentu. Pooling membantu mengurangi overfitting, mengurangi computational load, dan membuat model lebih robust terhadap translasi kecil pada gambar input.

Max Pooling:

$$\text{Output}[i,j] = \max(\text{Input}[i \cdot \text{stride} : (i \cdot \text{stride} + \text{pool\_h}), j \cdot \text{stride} : (j \cdot \text{stride} + \text{pool\_w})])$$

Average Pooling:

$$\text{Output}[i,j] = \text{mean}(\text{Input}[i \cdot \text{stride} : (i \cdot \text{stride} + \text{pool\_h}), j \cdot \text{stride} : (j \cdot \text{stride} + \text{pool\_w})])$$

Tahap 5: Flatten

Tahap flatten mengubah tensor multidimensi hasil dari convolutional layers menjadi vektor 1D yang dapat diproses oleh fully connected layers. Proses ini dilakukan dengan meratakan semua dimensi kecuali batch dimension, sehingga tensor dengan dimensi (batch_size, height, width, channels) menjadi (batch_size, height×width×channels). Flatten layer tidak memiliki parameter yang dapat dipelajari dan hanya melakukan reshaping data. Tahap ini menjadi jembatan antara bagian feature extraction (convolutional layers) dengan bagian classification (dense layers).

Tahap 6: Dense (Fully Connected)

Tahap dense layer merupakan fully connected layer dimana setiap neuron terhubung dengan semua neuron di layer sebelumnya. Input berupa vektor 1D dikalikan dengan matriks bobot dan ditambahkan dengan vektor bias untuk menghasilkan transformasi linear. Hasil transformasi kemudian diproses melalui fungsi aktivasi sesuai kebutuhan. Dense layers biasanya digunakan pada bagian akhir CNN untuk melakukan klasifikasi berdasarkan fitur-fitur yang telah diekstrak

oleh convolutional layers. Layer terakhir biasanya memiliki jumlah neuron yang sama dengan jumlah kelas target.

Transformasi linear:

$$\text{Output} = \text{Input} \times \text{Weights} + \text{Bias}$$

Dengan aktivasi:

$$\text{Activated\_Output} = \text{activation\_function}(\text{Input} \times \text{Weights} + \text{Bias})$$

Tahap 7: Output (Softmax)

Tahap terakhir adalah layer output yang menggunakan fungsi aktivasi softmax untuk mengkonversi nilai mentah (logits) menjadi distribusi probabilitas. Softmax memastikan bahwa jumlah semua probabilitas output sama dengan 1, sehingga setiap nilai output dapat diinterpretasikan sebagai probabilitas bahwa input termasuk dalam kelas tertentu. Kelas dengan probabilitas tertinggi menjadi prediksi akhir model. Softmax juga memberikan stabilitas numerik dengan mengurangi setiap elemen dengan nilai maksimum sebelum menghitung exponential.

Softmax:

$$\text{softmax}(x_i) = \frac{\exp(x_i - \max(x))}{\sum_j \exp(x_j - \max(x))}$$

dimana max(x) ditambahkan untuk stabilitas numerik dan mencegah overflow pada operasi exponential.

2.2. RNN

2.2.1. Kelas dan Atribut

Kelas EmbeddingLayer

EmbeddingLayer bertugas mengkonversi token-token integer menjadi representasi vektor dense. Setiap kata dalam vocabulary dipetakan ke dalam high-dimension vector space, kata-kata dengan makna serupa memiliki nilai vektor yang berdekatan.

Atribut Kelas:

- `vocab_size`: Integer yang menentukan ukuran vocabulary, yaitu jumlah maksimum token unik yang dapat direpresentasikan oleh layer ini

- `embedding_dim`: Integer yang menentukan dimensi dari vektor embedding untuk setiap token
- `weights`: NumPy array dengan shape (vocab_size, embedding_dim) yang menyimpan matriks embedding yang telah di train

Method yang Diimplementasikan:

- `load_weights(weights)` berfungsi untuk memuat parameter embedding yang telah dilatih sebelumnya dari model Keras.
- `forward(inputs)` merupakan core functionality yang melakukan lookup operation. Method ini menerima array 2D dengan shape (batch_size, sequence_length) berisi token indices, dan mengembalikan array 3D dengan shape (batch_size, sequence_length, embedding_dim). Implementasi menggunakan nested loop untuk memastikan setiap token dikonversi dengan benar, sambil menangani edge cases seperti out-of-vocabulary tokens.

Kelas SimpleRNNCell

SimpleRNNCell adalah implementasi dari komputasi RNN untuk satu timestep. Rumusnya adalah: $h\_t = tanh(W\_ih * x\_t + W\_hh * h\_\{t-1\} + b\_h)$, dimana transformasi linear dari input dan hidden state sebelumnya dikombinasikan dengan aktivasi tanh.

Atribut Kelas:

- `input_size`: Dimensi dari input vector pada setiap timestep
- `hidden_size`: Dimensi dari hidden state vector
- `W_ih`: Weight matrix untuk transformasi input-to-hidden dengan shape (input_size, hidden_size)
- `W_hh`: Weight matrix untuk transformasi hidden-to-hidden dengan shape (hidden_size, hidden_size)
- `b_h`: Bias vector dengan shape (hidden_size,)

Method Implementation:

- `load_weights(W_ih, W_hh, b_h)` memuat parameter yang telah dilatih dari model Keras sebelumnya.
- `forward_step(x, h_prev)` mengimplementasikan forward pass untuk satu timestep. Method ini menerima input x dengan shape (batch_size, input_size) dan hidden state sebelumnya h_prev dengan shape (batch_size, hidden_size). Komputasi dilakukan melalui matrix multiplication dan element-wise addition, diikuti dengan aplikasi fungsi aktivasi tanh.

Kelas SimpleRNNLayer

SimpleRNNLayer mengekstrak multiple SimpleRNNCell untuk memproses semua sequencenya. Layer ini bisa untuk unidirectional dan bidirectional RNN, khusus untuk bidirectional RNN, layer ini memproses sequence dari kedua arah.

Atribut Kelas:

- `input_size`: Dimensi input untuk setiap timestep
- `hidden_size`: Dimensi hidden state
- `bidirectional`: Boolean flag yang menentukan apakah layer menggunakan processing dua arah
- `forward_cell`: Instance SimpleRNNCell untuk pemrosesan forward direction
- `backward_cell`: Instance SimpleRNNCell untuk pemrosesan backward direction (hanya jika bidirectional=True)

Method Implementation:

- `load_weights(weights_dict)` memuat parameter untuk kedua cell (forward dan backward jika applicable). Dictionary weights harus berisi keys yang sesuai untuk setiap direction.
- `forward(inputs)` merupakan method kompleks yang menangani sequential processing. Untuk unidirectional RNN, method ini mengiterasi melalui timesteps secara berurutan, mempertahankan hidden state dan mengumpulkan outputs. Untuk bidirectional RNN, proses dilakukan dalam dua pass: forward pass dari awal ke akhir, dan backward pass dari akhir ke awal. Outputs dari kedua direction kemudian di concatenate.

Kelas DenseLayer

DenseLayer mengimplementasikan fully connected layer yang umum digunakan sebagai classifier atau output layer. Layer ini menerapkan transformasi linear diikuti dengan fungsi aktivasi yang dapat diatur.
Atribut Kelas:

- `input_size`: Dimensi input vector
- `output_size`: Dimensi output vector
- `activation`: Menentukan jenis fungsi aktivasi ('linear', 'softmax', 'relu', 'sigmoid')
- `weights`: Weight matrix dengan shape (input_size, output_size)
- `bias`: Bias vector dengan shape (output_size,)

Method Implementation:

- `load_weights(weights, bias)` memuat parameter yang telah dilatih untuk layer ini.

- `forward(inputs)` mengimplementasikan forward pass dengan flexibility untuk menangani input 2D maupun 3D. Untuk input 3D (batch_size, sequence_length, input_size), method ini melakukan reshaping untuk komputasi, kemudian restore shape asli. Transformasi linear dilakukan melalui matrix multiplication, diikuti dengan aplikasi fungsi aktivasi yang sesuai.
- Implementasi softmax menggunakan numerical stability trick dengan mengurangi maximum value sebelum eksponensial untuk mencegah overflow.

Kelas SimpleRNNModel

SimpleRNNModel adalah wrapper yang mengintegrasikan semua komponen menjadi end-to-end model yang dapat melakukan inference. Class ini menangani data flow dari raw input hingga final predictions.

Atribut Kelas:

- `vocab_size`: Ukuran vocabulary untuk embedding layer
- `embedding_dim`: Dimensi embedding vectors
- `hidden_sizes`: List yang menentukan arsitektur RNN (support multiple layers)
- `num_classes`: Jumlah kelas untuk classification task
- `bidirectional`: Boolean untuk bidirectional processing
- `embedding`: Instance EmbeddingLayer
- `rnn_layers`: List dari SimpleRNNLayer instances
- `dense`: Instance DenseLayer untuk final classification

Method Implementation:

- `load_keras_weights(keras_model_path)` adalah method yang mengekstrak weights dari trained Keras model dan mendistribusikannya ke komponen-komponen yang sesuai.
- `forward(inputs)` mengimplementasikan complete forward pass. Data flow dimulai dari embedding lookup, kemudian melalui sequence dari RNN layers, dan akhirnya melalui dense layer untuk classification.
- `predict(inputs)` dan `predict_proba(inputs)` mengembalikan class predictions dan predict_proba mengembalikan probability distributions.

2.2.2. Tahapan *Forward Propagation*

Tahap 1: Embedding Lookup

Proses dimulai dengan konversi token sequence menjadi dense vector representations. Input berbentuk (batch_size, sequence_length) berisi integer indices yang merepresentasikan tokens. Embedding layer melakukan lookup

operation dimana setiap token index dipetakan ke corresponding row dalam embedding matrix.

Tahap 2: Sequential Processing melalui RNN Layers

Ini adalah inti dari RNN processing dimana informasi temporal diproses. Untuk setiap RNN layer, setiap sequence akan diproses secara iteratif per time step. Pada timestep t, kita memiliki:

- Input: x_t dengan dimensi (batch_size, input_size)
- Hidden state sebelumnya: $h\_\{t - 1\}$ dengan dimensi (batch_size, hidden_size)

Komputasi RNN cell mengikuti formula:
$h\_t = tanh(W\_ih \mathbin{@} x\_t + W\_hh \mathbin{@} h\_\{t - 1\} + b\_h)$

Dimana @ mennotasikan matrix multiplication.

Tahap 3: Handling Multiple RNN Layers

Ketika kita memiliki multiple RNN layers (deep RNN), output dari layer pertama menjadi input untuk layer kedua. Ini menciptakan hierarchical representation dimana layer yang lebih tinggi dapat menangkap patterns yang lebih abstract.

Untuk layer ke-i, inputnya adalah output dari layer i-1. Dimensionalitas berubah sesuai dengan hidden_size dari masing-masing layer. Proses sequential processing diulangi untuk setiap layer, dimana setiap layer memiliki parameter W_ih, W_hh, dan b_h yang terpisah.

Tahap 4: Bidirectional Processing (jika applicable)

Dalam bidirectional RNN, kita melakukan dua pass terpisah:

Forward pass: memproses sequence dari timestep 0 hingga T-1, menghasilkan hidden states h_forward_t Backward pass: memproses sequence dari timestep T-1 hingga 0, menghasilkan hidden states h_backward_t

Final output untuk setiap timestep adalah concatenation: h_t = [h_forward_t; h_backward_t], output dari tiap timestep adalah concatenation dari forward dan backward pass. Ini menghasilkan output dengan dimensi (batch_size, sequence_length, 2*hidden_size).

Tahap 5: Sequence Classification Processing

Untuk sequence classification tasks, kita hanya menggunakan output dari timestep terakhir. Operasi ini adalah: $final\_representation = output[:, -1, :]$, dengan-1 mengindikasikan timestep terakhir.

Tahap 6: Dense Layer Classification

Final step adalah transformasi dari sequence representation ke class probabilities. Dense layer menerapkan:

$$logits = final\_representation @ W\_dense + b\_dense$$

Diikuti dengan softmax activation untuk menghasilkan probability distribution:
$probabilities = softmax(logits) = exp(logits) / sum(exp(logits))$

2.3. LSTM

2.3.1 Kelas dan Atribut

Kelas EmbeddingLayer

Kelas EmbeddingLayer memiliki fungsi yang sama dengan yang kelas EmbeddingLayer pada bagian RNN. Kelas ini mengkonversi token-token integer menjadi representasi vektor dense. Setiap kata dalam vocabulary dipetakan ke dalam high-dimension vector space, kata-kata dengan makna serupa memiliki nilai vektor yang berdekatan.

Atribut Kelas:

- vocab_size: ukuran vocabulary, yaitu jumlah maksimum token unik yang dapat direpresentasikan oleh layer ini
- embedding_dim: dimensi dari vektor embedding untuk setiap token
- weights: menyimpan matriks embedding yang telah di train

Method yang Diimplementasikan:

- Load_weights: berfungsi untuk memuat parameter embedding yang telah dilatih sebelumnya dari model Keras.
- Forward: merupakan core functionality yang melakukan lookup operation. Method ini menerima array 2D dengan shape (batch_size, sequence_length) berisi token indices, dan mengembalikan array 3D dengan shape (batch_size, sequence_length, embedding_dim). Implementasi menggunakan nested loop untuk memastikan setiap token dikonversi dengan benar, sambil menangani edge cases seperti out-of-vocabulary tokens.

Kelas LSTMCell

LSTMCell adalah kelas yang merepresentasikan satu unit LSTM untuk satu timestep. Unit LSTM menggunakan gates untuk mengatur informasi ke dalam dan keluar dari cell state, serta untuk memperbarui hidden state.

Atribut Kelas:

- input_size : Dimensi vektor input pada setiap timestep (xt).
- hidden_size : Dimensi dari hidden state vector (ht) dan cell state vector (ct).
- W_ih : Matriks bobot untuk transformasi input-ke-hidden, dengan shape (input_size, 4 * hidden_size).
- W_hh : Matriks bobot untuk transformasi hidden-ke-hidden, dengan shape (hidden_size, 4 * hidden_size).
- b_ih: Vektor bias yang terkait dengan koneksi input (gabungan bias untuk semua gerbang), dengan shape (4 * hidden_size).
- b_hh: Vektor bias yang terkait dengan koneksi hidden state. Dalam implementasi ini, saat memuat dari Keras, b_hh diatur menjadi nol karena bias Keras sudah sepenuhnya terkandung dalam b_ih.

Method yang Diimplementasikan:

- load_weights: Memuat parameter bobot dan bias yang telah dilatih.
- forward_step(x, h_prev, c_prev): Melakukan komputasi forward pass untuk satu timestep. Method ini menerima input x_t, hidden state sebelumnya h_{t-1}, dan cell state sebelumnya c_{t-1}. Kemudian menghitung nilai-nilai untuk input gate (it), forget gate (ft), cell gate/candidate state (gt), dan output gate (ot). Berdasarkan nilai-nilai gates ini, cell state baru (ct) dan hidden state baru (ht) dihitung.

Kelas LSTMLayer

LSTMLayer menggunakan satu atau lebih LSTMCell untuk memproses seluruh sekuens input. Layer ini dapat berupa unidirectional dan bidirectional.

Atribut Kelas:

- input_size : Dimensi input untuk setiap timestep.
- hidden_size : Dimensi hidden state untuk setiap LSTMCell.
- bidirectional : Flag yang menentukan apakah layer menggunakan pemrosesan dua arah. Jika True, dua LSTMCell (satu forward, satu backward) digunakan.
- forward_cell (LSTMCell): Instance LSTMCell untuk pemrosesan (forward).
- backward_cell (LSTMCell): Instance LSTMCell untuk pemrosesan arah mundur (backward), hanya jika bidirectional=True.

Method yang Diimplementasikan:

- load_weights(weights_dict): Memuat parameter untuk forward_cell dan backward_cell (jika ada).

- forward(inputs): Melakukan pemrosesan sekuensial. Untuk LSTM unidirectional, iterasi dilakukan dari awal hingga akhir sekuens. Untuk LSTM bidirectional, pemrosesan dilakukan dalam dua pass (forward dan backward), dan output dari kedua arah kemudian di *concat*.

Kelas DenseLayer

Kelas DenseLayer memiliki fungsi yang identik dengan yang dijelaskan pada bagian CNN dan RNN. Kelas ini mengimplementasikan layer fully connected.

Atribut Kelas:

- input_size : Dimensi vektor input.
- output_size: Dimensi vektor output (jumlah unit).
- activation: Fungsi aktivasi yang diterapkan (linear, softmax, relu, sigmoid).
- weights: Matriks bobot dengan shape (input_size, output_size).
- bias: Vektor bias dengan shape (output_size,).

Method yang Diimplementasikan:

- load_weights(weights, bias): Memuat bobot dan bias yang telah dilatih.
- forward(inputs): Melakukan transformasi linear diikuti oleh fungsi aktivasi.

Kelas LSTMModel

LSTMModel adalah kelas utama yang menggabungkan semua komponen (EmbeddingLayer, LSTMLayer, DenseLayer) menjadi sebuah model yang dapat melakukan inferensi pada data.

Atribut Kelas:

- vocab_size: Ukuran vocabulary untuk EmbeddingLayer.
- embedding_dim : Dimensi vektor embedding.
- hidden_sizes : jumlah unit hidden pada setiap LSTMLayer.
- num_classes : Jumlah kelas target untuk tugas klasifikasi.
- bidirectional: Menentukan apakah LSTMLayer dalam model bersifat bidireksional.
- embedding : Instance dari EmbeddingLayer.
- lstm_layers : List instance LSTMLayer.
- dense : Instance dari DenseLayer untuk klasifikasi akhir.

Method yang Diimplementasikan:

- load_keras_weights(keras_model_path): Memuat bobot dari file model Keras yang telah dilatih dan mendistribusikannya ke layer-layer yang sesuai.
- forward(inputs): Melakukan forward pass lengkap melalui seluruh arsitektur: input token diubah menjadi embedding, kemudian diproses oleh satu atau lebih LSTMLayer, output dari LSTM layer terakhir kemudian dimasukkan ke DenseLayer untuk menghasilkan probabilitas kelas.
- predict(inputs): Menghasilkan prediksi kelas (indeks kelas dengan probabilitas tertinggi.
- predict_proba(inputs): Menghasilkan distribusi probabilitas atas kelas-kelas target.

2.3.2 Tahapan Forward Propagation

Tahap 1: Embedding Lookup

Proses dimulai dengan konversi sekuens token input berbentuk (batch_size, sequence_length) menjadi representasi vektor dense. EmbeddingLayer memetakan setiap token indeks ke vektor embedding yang sesuai dari matriks bobot embedding. Output dari tahap ini adalah tensor dengan shape (batch_size, sequence_length, embedding_dim).

Tahap 2: Sequential Processing melalui LSTM Cells dalam LSTMLayer

Untuk setiap LSTMLayer, dan untuk setiap timestep t dalam sekuens (dari 0 hingga T−1):

- **Input pada timestep t**:
  - x_t: Vektor input dari hasil embedding (atau output dari layer LSTM sebelumnya) pada timestep t.
  - h_{t-1}: Hidden state dari timestep sebelumnya.
  - c_{t-1}: Cell state dari timestep sebelumnya.
- **Kalkulasi LSTM Cell**:
  - Perhitungan nilai gabungan untuk semua gates:
    $$gates\_val = (x_t W_{ih}^T + b_{ih}^T) + (h_{t-1} W_{hh}^T + b_{hh}^T)$$
    Di mana Wih adalah bobot untuk input x_t, Whh adalah bobot untuk hidden state h_{t-1}. b_{ih} adalah bias input, dan b_{hh} adalah bias hidden. gates_val adalah vektor dengan panjang 4× hidden_size.
  - Vektor gates_val dibagi menjadi empat bagian, masing-masing untuk:
    - Input Gate (it): Mengontrol seberapa banyak informasi baru dari xt dan h_{t-1} yang akan disimpan dalam cell state. $i_t = \sigma(gatesval_i)$

        - Forget Gate (ft): Mengontrol seberapa banyak informasi dari cell state sebelumnya (c_{t-1}) yang akan dilupakan atau dibuang.
          $f_t = \sigma(gates\_val_f)$
        - Cell Candidate Gate (gt): Menghasilkan nilai kandidat baru yang dapat ditambahkan ke cell state. $g_t = \tanh(gates\_val_g)$
        - Output Gate (ot): Mengontrol seberapa banyak informasi dari cell state saat ini (ct) yang akan diekspos sebagai hidden state (ht).
          $o_t = \sigma(gates\_val_o)$
    - Perhitungan Cell State: Cell state baru ct dihitung dengan menggabungkan informasi yang dilupakan dari c_{t-1} dan informasi baru dari it dan gt.
      $ct = ft \odot c_{t-1} + it \odot gt$
      Di mana ◦ melambangkan perkalian antar elemen (element-wise multiplication).
    - Perhitungan Hidden State: Hidden state baru ht dihitung berdasarkan cell state ct yang telah diperbarui dan output gate ot.
      $ht = ot \odot tanh(ct)$
      Output dari LSTMLayer pada setiap timestep adalah ht. Jika layer mengembalikan sekuens penuh, outputnya adalah kumpulan ht untuk semua timestep, dengan shape (batch_size, sequence_length, hidden_size).

Tahap 3: Handling Multiple LSTM Layers (Stacked LSTM)

Jika model memiliki lebih dari satu LSTMLayer, output hidden states (ht) dari layer LSTM ke-i menjadi input (xt) untuk layer LSTM ke-(i+1).

Tahap 4: Bidirectional Processing (jika applicable)

Untuk LSTMLayer bidirectional:

1. Forward Pass: Sekuens diproses dari timestep 0 hingga T−1, menghasilkan serangkaian hidden states forward: $h_{f,0}$, $h_{f,1}$,…,$h_{f,T-1}$.
2. Backward Pass: Sekuens diproses secara terbalik, dari timestep T−1 hingga 0, menghasilkan serangkaian hidden states backward: $h_{b,T-1}$,$h_{b,T-2}$,…,$h_{b,0}$.
3. Output akhir untuk setiap timestep t adalah *concatenation* dari hidden state forward dan backward pada timestep tersebut: [$h_{f,t}$;$h_{b,t}$]. Jadi, dimensi output dari layer bidirectional menjadi (batch_size, sequence_length, 2 * hidden_size).

Tahap 5: Sequence Classification Processing

Untuk task classification, representasi yang digunakan berbeda antara unidirectional dan bidirectional.

- Unidirectional LSTM: Hidden state dari timestep terakhir (h_{T-1}) dari layer LSTM terakhir.
- Bidirectional LSTM: *concatenation* dari hidden state terakhir dari pass forward ($h_{f,T-1}$) dan hidden state terakhir dari pass backward ($h_{b,0}$,).

Tahap 6: Dense Layer Classification

Final step adalah transformasi dari sequence representation ke class probabilities. Dense layer menerapkan:

$$logits = final\_representation \,@\, W\_dense + b\_dense$$

Diikuti dengan softmax activation untuk menghasilkan probability distribution:
$probabilities = softmax(logits) = exp(logits) \,/\, sum(exp(logits))$

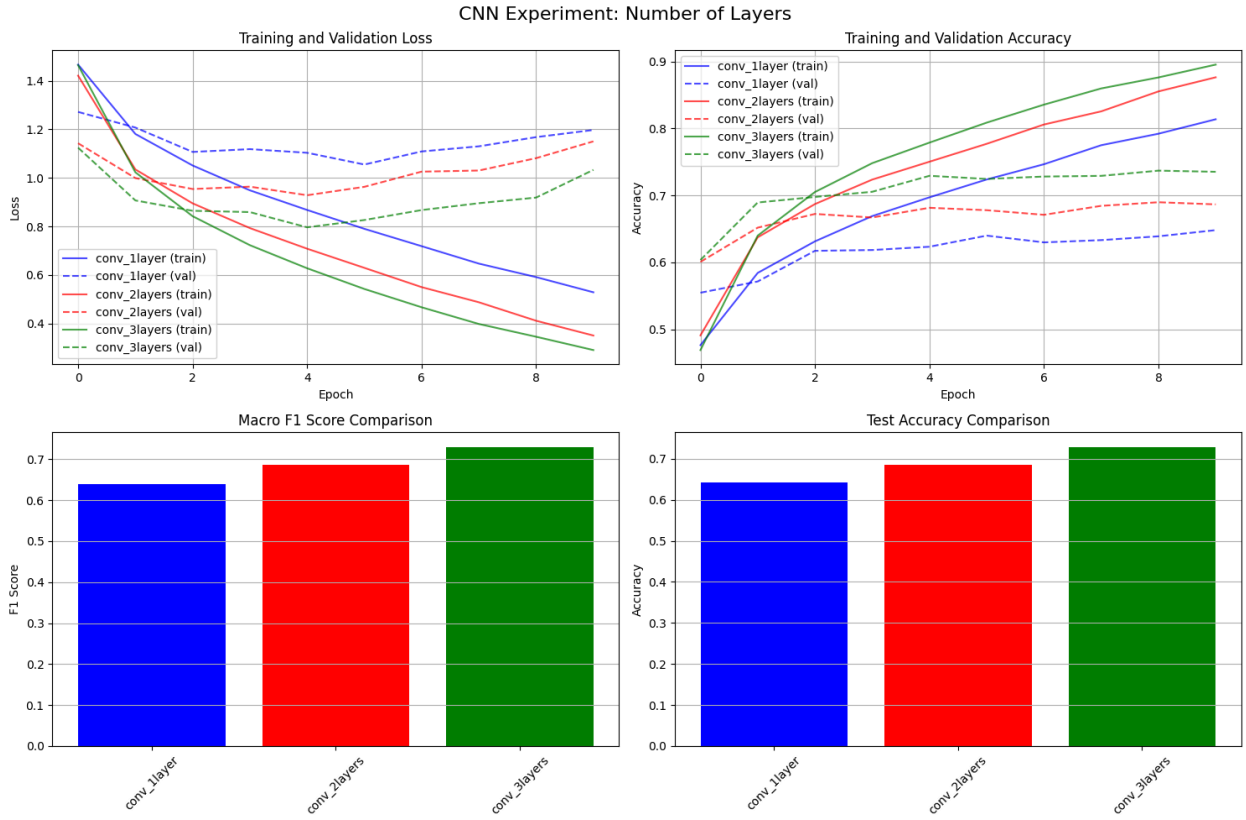
BAB 3
Hasil Pengujian

3.1. CNN

3.1.1 Pengaruh jumlah layer konvolusi

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- conv_1layer = 1 layer konvolusi (32 filter, kernel 3x3) + MaxPooling + Dense(128) + Output(10)
- conv_2layers = 2 layer konvolusi (32→64 filter, kernel 3x3) + MaxPooling setiap layer + Dense(128) + Output(10)
- conv_3layers = 3 layer konvolusi (32→64→128 filter, kernel 3x3) + MaxPooling setiap layer + Dense(128) + Output(10)

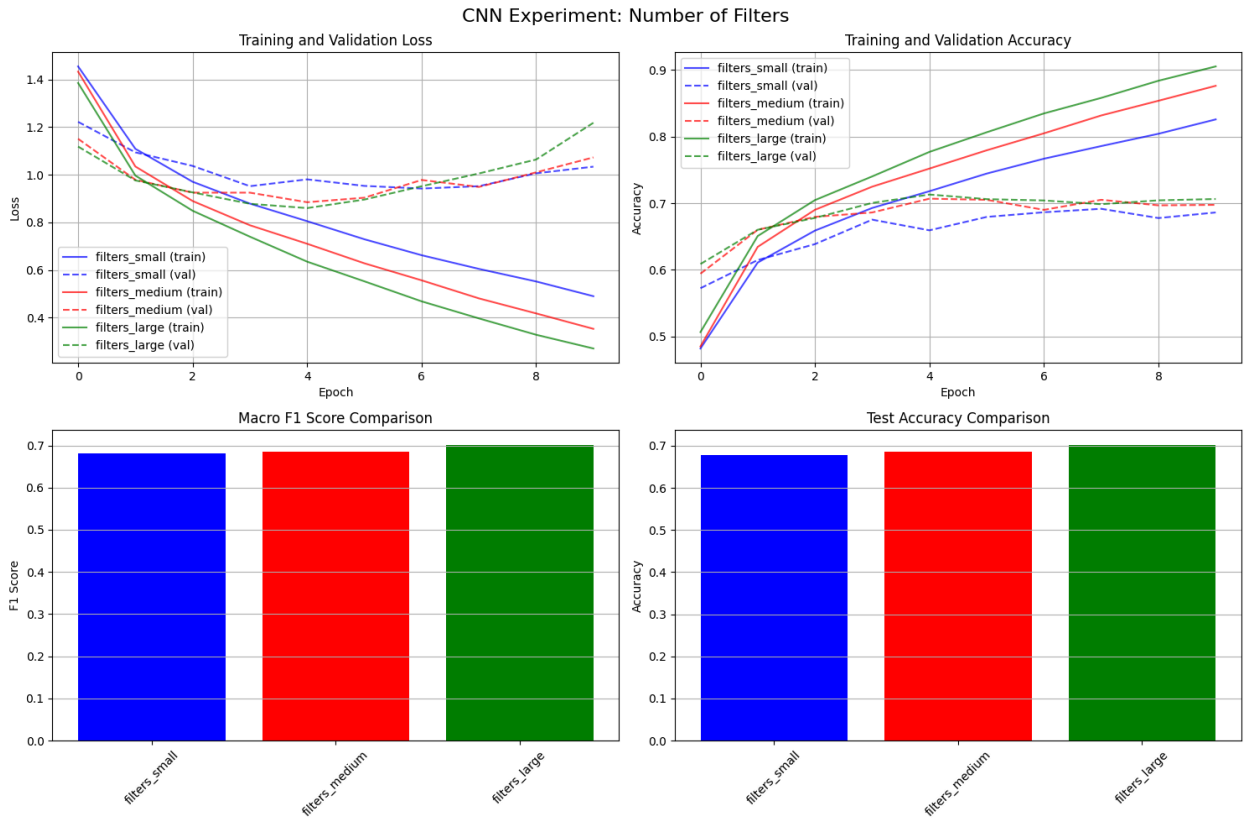


Model dengan 1 layer (conv_1layer) menunjukkan performa paling rendah dengan test accuracy 64.21% dan macro F1-score 63.94%, serta mengalami overfitting yang cukup terlihat dari gap antara training dan validation accuracy. Model dengan 2 layer (conv_2layers) memberikan peningkatan performa yang substansial menjadi 68.53% untuk test accuracy dan 68.56% untuk F1-score, dengan training yang lebih stabil. Model dengan 3 layer (conv_3layers) mencapai performa terbaik dengan test accuracy 72.87% dan macro F1-score 72.98%, menunjukkan peningkatan sekitar 8.66% dibandingkan model 1 layer.

3.1.2 Pengaruh banyak filter per layer konvolusi

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- filters_small = 2 layer konvolusi dengan filter kecil (16→32 filter, kernel 3x3) + MaxPooling + Dense(128) + Output(10)
- filters_medium = 2 layer konvolusi dengan filter sedang (32→64 filter, kernel 3x3) + MaxPooling + Dense(128) + Output(10)
- filters_large = 2 layer konvolusi dengan filter besar (64→128 filter, kernel 3x3) + MaxPooling + Dense(128) + Output(10)

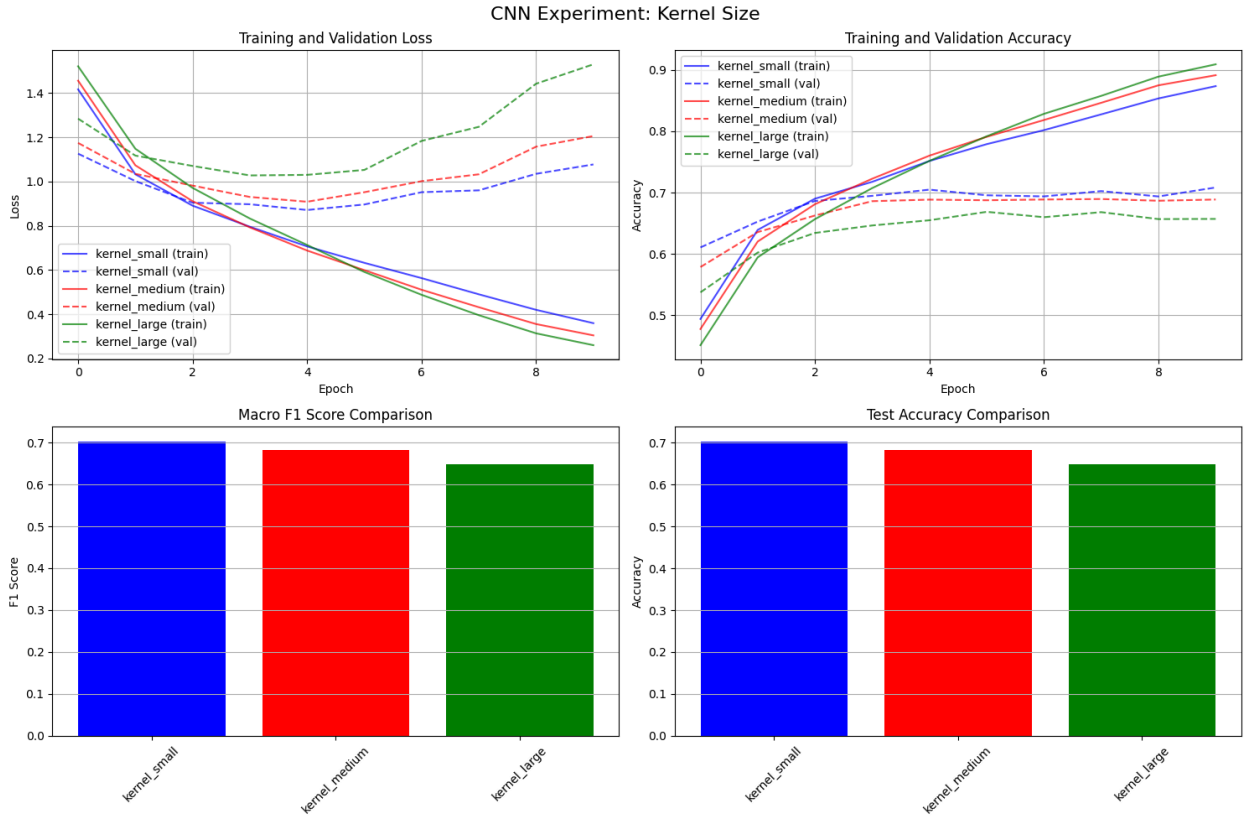


Berdasarkan hasil eksperimen pengaruh jumlah filter per layer konvolusi, terlihat bahwa peningkatan jumlah filter memberikan dampak positif yang konsisten terhadap performa model. Model dengan filter kecil (16→32) mencapai test accuracy 67.76% dan macro F1-score 68.18%, model dengan filter medium (32→64) meningkat menjadi 68.61% accuracy dan 68.51% F1-score, sedangkan model dengan filter besar (64→128) mencapai performa terbaik dengan 70.16% accuracy dan 70.09% F1-score.

3.1.3 Pengaruh ukuran filter per layer konvolusi

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- kernel_small = 2 layer konvolusi dengan kernel kecil (32→64 filter, kernel 3x3) + MaxPooling + Dense(128) + Output(10)
- kernel_medium = 2 layer konvolusi dengan kernel sedang (32→64 filter, kernel 5x5) + MaxPooling + Dense(128) + Output(10)
- kernel_large = 2 layer konvolusi dengan kernel besar (32→64 filter, kernel 7x7) + MaxPooling + Dense(128) + Output(10)

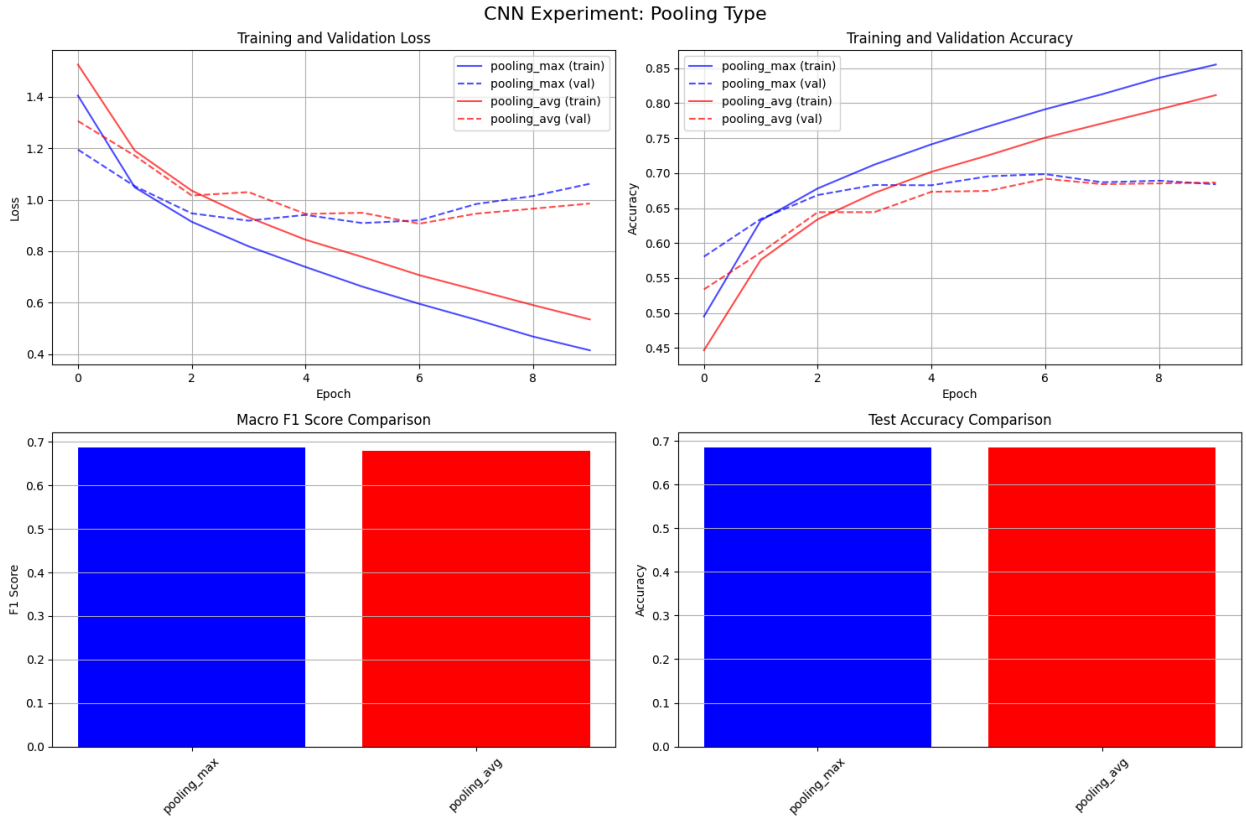


Berdasarkan hasil eksperimen pengaruh ukuran kernel, terlihat bahwa kernel yang lebih kecil memberikan performa yang lebih baik pada dataset CIFAR-10. Model dengan kernel kecil (3x3) mencapai test accuracy tertinggi sebesar 70.31% dan macro F1-score 70.24%, diikuti oleh kernel medium (5x5) dengan 68.23% accuracy dan 68.23% F1-score, sedangkan kernel besar (7x7) menunjukkan performa terendah dengan 64.84% accuracy dan 64.85% F1-score.

3.1.4 Pengaruh jenis pooling layer

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- pooling_max = 2 layer konvolusi (32→64 filter, kernel 3x3) + MaxPooling setiap layer + Dense(128) + Output(10)
- pooling_avg = 2 layer konvolusi (32→64 filter, kernel 3x3) + AveragePooling setiap layer + Dense(128) + Output(10)



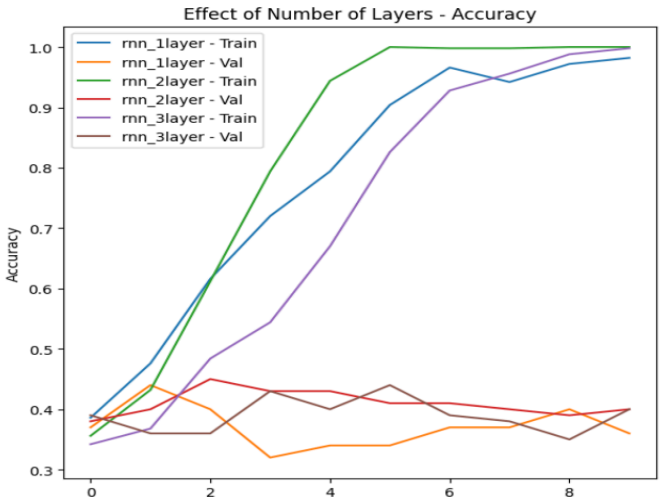
Berdasarkan hasil eksperimen pengaruh jenis pooling layer, terlihat bahwa perbedaan performa antara Max Pooling dan Average Pooling sangat minimal dan hampir tidak signifikan. Model dengan Max Pooling mencapai test accuracy 68.47% dan macro F1-score 68.79%, sedangkan model dengan Average Pooling mencapai test accuracy 68.55% dan macro F1-score 68.00%, dengan selisih hanya sekitar 0.08% untuk accuracy. Hasil ini mengindikasikan bahwa untuk dataset CIFAR-10 dengan arsitektur CNN sederhana, pilihan antara Max Pooling dan Average Pooling tidak memberikan dampak yang substansial terhadap performa model.

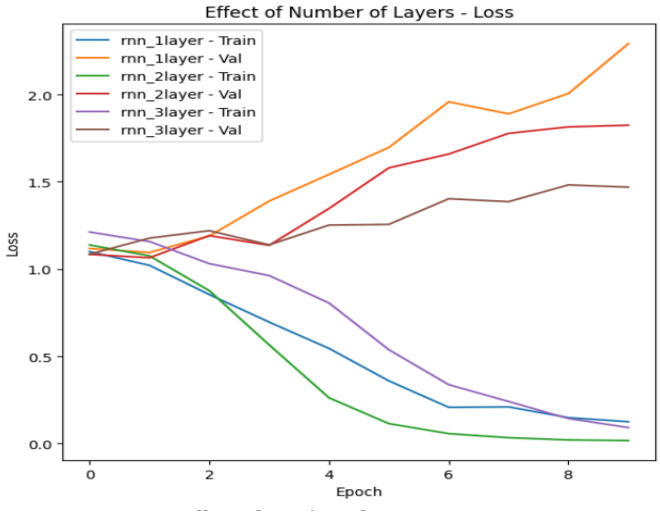
3.2. RNN

3.2.1 Pengaruh jumlah layer RNN

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- rnn_1layer = ada 1 layer dengan hidden unit 64
- rnn_2layer = ada 2 layer dengan hidden unit 64,32
- rnn_3layer = ada 3 layer dengan hidden unit 64,32,16





Berikut hasil komparasi inference process antara model Keras dengan Scratch

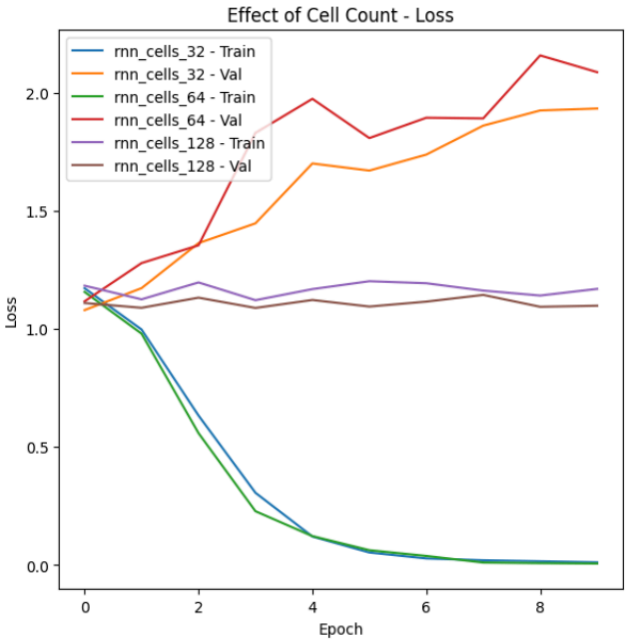
```
Comparing rnn_1layer...


Weights loaded from ../../models/rnn_1layer.h5

 Keras F1: 0.4006
  Scratch F1: 0.4006
  Max diff: 0.000005
  Agreement: 1.0000

Comparing rnn_2layer...
Weights loaded from ../../models/rnn_2layer.h5

 Keras F1: 0.3958
  Scratch F1: 0.3958
  Max diff: 0.000002
  Agreement: 1.0000

Comparing rnn_3layer...


Weights loaded from ../../models/rnn_3layer.h5
```

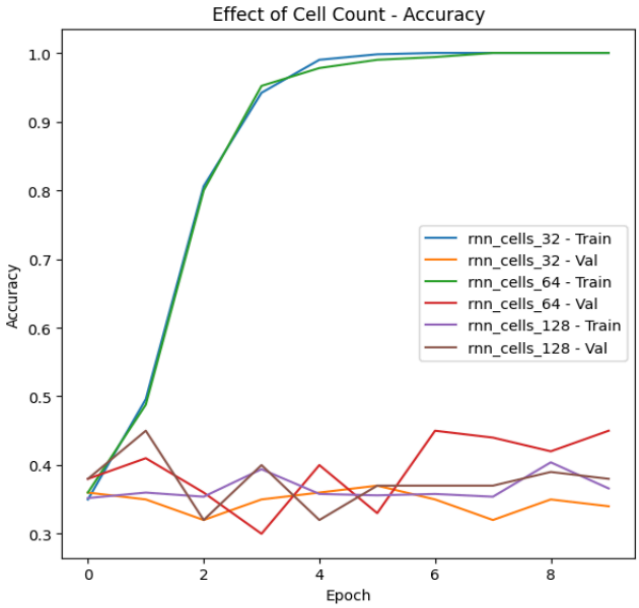
```
Keras F1: 0.3682
 Scratch F1: 0.3682
 Max diff: 0.000003
 Agreement: 1.0000
```

3.2.2 Pengaruh banyak cell pada layer RNN

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- rnn_cells_32 = ada 2 layer dengan hidden unit 32,32
- rnn_cells_64 = ada 2 layer dengan hidden unit 64,64
- rnn_cells_128 = ada 2 layer dengan hidden unit 128,128





Berikut hasil komparasi inference process antara model Keras dengan Scratch

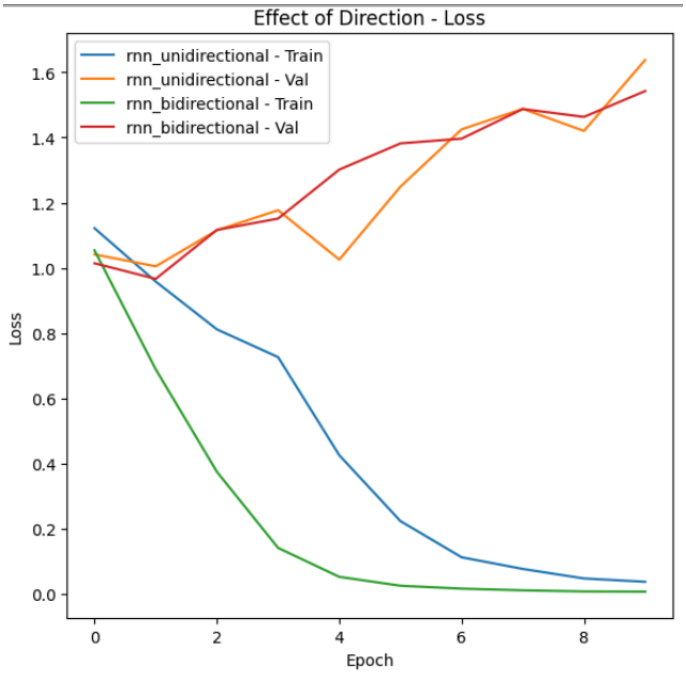
```
Comparing rnn_cells_32...
Weights loaded from ../../models/rnn_cells_32.h5


 Keras F1: 0.4298
  Scratch F1: 0.4298
  Max diff: 0.000002
  Agreement: 1.0000

Comparing rnn_cells_64...
Weights loaded from ../../models/rnn_cells_64.h5
 Keras F1: 0.3833
  Scratch F1: 0.3833
  Max diff: 0.000002
```

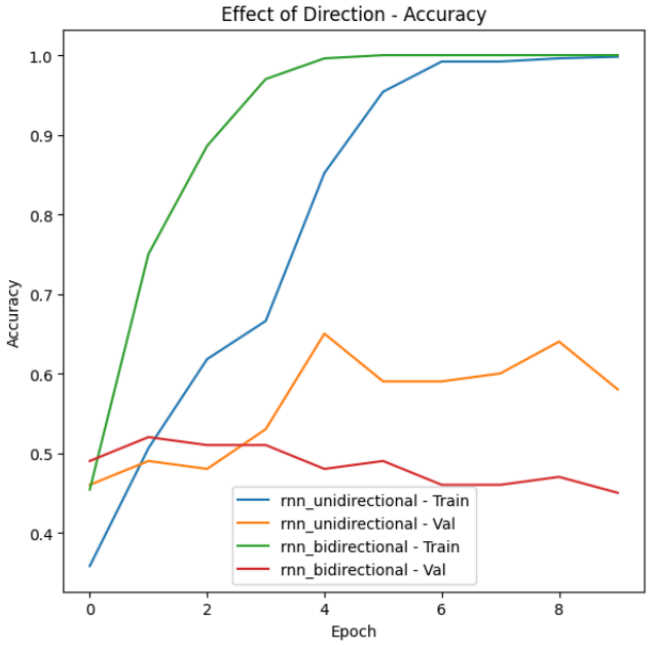
```
  Agreement: 1.0000

Comparing rnn_cells_128...
Weights loaded from ../../models/rnn_cells_128.h5

 Keras F1: 0.1844
  Scratch F1: 0.1844
  Max diff: 0.000000
  Agreement: 1.0000
```

3.2.3 Pengaruh jenis layer RNN berdasarkan arah

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- rnn_unidirectional= ada 2 layer dengan hidden unit 64,32
- rnn_bidirectional= ada 2 layer dengan hidden unit 64,32, bersifat bidirectional





Berikut hasil komparasi inference process antara model Keras dengan Scratch

```
Comparing rnn_unidirectional...
Weights loaded from ../../models/rnn_unidirectional.h5


 Keras F1: 0.5805
  Scratch F1: 0.5805
  Max diff: 0.000012
```

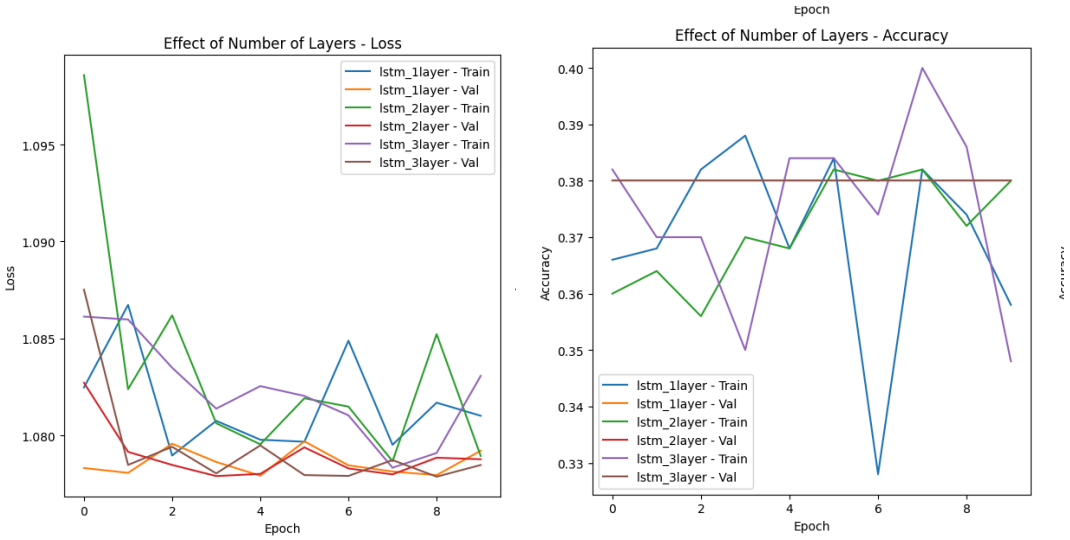
```
  Agreement: 1.0000

Comparing rnn_bidirectional...

Weights loaded from ../../models/rnn_bidirectional.h5
  Keras F1: 0.4295
  Scratch F1: 0.4295
  Max diff: 0.000003
  Agreement: 1.0000
```

3.3. LSTM

3.3.1 Pengaruh jumlah layer LSTM

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- lstm_1layer = ada 1 layer dengan hidden unit 64
- lstm_2layer = ada 2 layer dengan hidden unit 64,64
- lstm_3layer = ada 3 layer dengan hidden unit 64,64,64



Berikut hasil komparasi inference process antara model Keras dengan Scratch

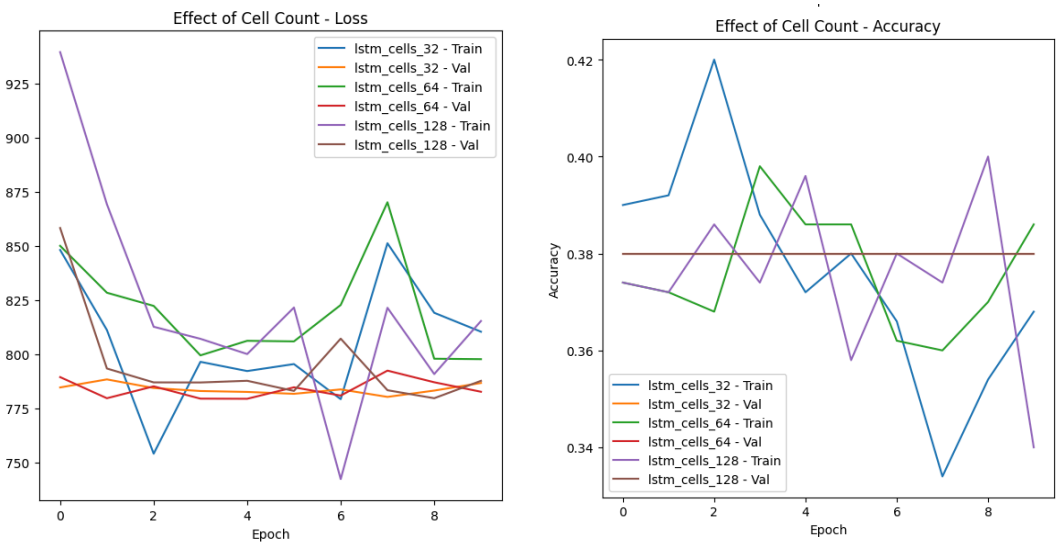
```
Comparing lstm_1layer...
Weights loaded from ../../models/lstm_1layer.keras
  Keras F1: 0.1827
  Scratch F1: 0.1827
  Max diff: 0.000000
  Agreement: 1.0000

Comparing lstm_2layer...
Weights loaded from ../../models/lstm_2layer.keras
```

```
  Keras F1: 0.1844
  Scratch F1: 0.1844
  Max diff: 0.000000
  Agreement: 1.0000

Comparing lstm_3layer...
Weights loaded from ../../models/lstm_3layer.keras
  Keras F1: 0.1827
  Scratch F1: 0.1827
  Max diff: 0.000000
  Agreement: 1.0000
```

3.3.2 Pengaruh banyak cell LSTM per layer

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- lstm_cells_16 = ada 2 layer dengan hidden unit 16,16
- lstm_cells_64 = ada 2 layer dengan hidden unit 64,64
- lstm_cells_128 = ada 2 layer dengan hidden unit 128,128



Berikut hasil komparasi inference process antara model Keras dengan Scratch

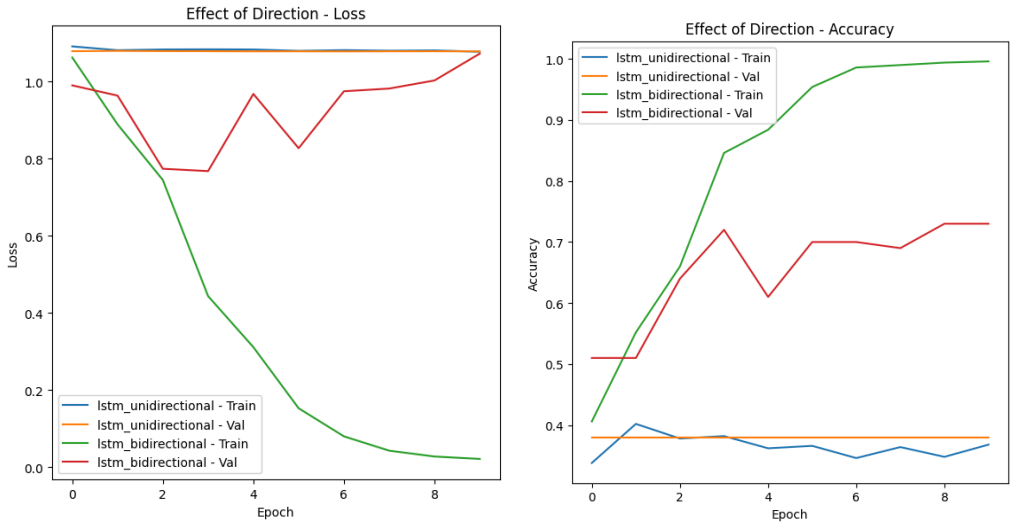
```
Comparing lstm_cells_32...
Weights loaded from ../../models/lstm_cells_32.keras
  Keras F1: 0.1844
  Scratch F1: 0.1844
  Max diff: 0.000000
  Agreement: 1.0000

Comparing lstm_cells_64...
Weights loaded from ../../models/lstm_cells_64.keras
```

```
  Keras F1: 0.1827
  Scratch F1: 0.1827
  Max diff: 0.000000
  Agreement: 1.0000

Comparing lstm_cells_128...
Weights loaded from ../../models/lstm_cells_128.keras
  Keras F1: 0.1827
  Scratch F1: 0.1827
  Max diff: 0.000000
  Agreement: 1.0000
```

3.3.3 Pengaruh jenis layer LSTM berdasarkan arah

Berikut adalah konfigurasi yang digunakan untuk tiap model:

- lstm_unidirectional= ada 2 layer dengan hidden unit 64,64
- lstm_bidirectional= ada 2 layer dengan hidden unit 64,64 bersifat bidirectional



Berikut hasil komparasi inference process antara model Keras dengan Scratch

```
Comparing lstm_unidirectional...
Weights loaded from ../../models/lstm_unidirectional.keras
  Keras F1: 0.1844
  Scratch F1: 0.1844
  Max diff: 0.000000
  Agreement: 1.0000

Comparing lstm_bidirectional...
Weights loaded from ../../models/lstm_bidirectional.keras
  Keras F1: 0.7355
  Scratch F1: 0.7355
  Max diff: 0.000001
```

```
Agreement: 1.0000
```

BAB 4
Kesimpulan dan Saran

Convolutional Neural Network merupakan arsitektur deep learning yang sangat efektif untuk tugas computer vision, khususnya klasifikasi gambar. CNN terdiri dari beberapa komponen utama yaitu convolutional layers untuk ekstraksi fitur hierarkis, pooling layers untuk downsampling dan regularisasi, serta dense layers untuk klasifikasi akhir. Setiap komponen memiliki peran spesifik dalam proses forward propagation, dimana data gambar mengalami serangkaian transformasi mulai dari deteksi fitur sederhana hingga representasi fitur kompleks yang dapat digunakan untuk klasifikasi.

Hasil eksperimen menunjukkan bahwa berbagai parameter arsitektur CNN memberikan dampak yang berbeda terhadap performa model. Penambahan jumlah layer konvolusi dari 1 hingga 3 layer memberikan peningkatan accuracy yang signifikan dari 64.21% menjadi 72.87%, mengindikasikan bahwa arsitektur yang lebih dalam mampu mengekstrak hierarki fitur yang lebih kompleks. Peningkatan jumlah filter juga memberikan dampak positif dengan model filter besar mencapai 70.16% dibandingkan filter kecil 67.76%, namun disertai dengan risiko overfitting yang lebih tinggi. Sebaliknya, pemilihan jenis pooling (max vs average) tidak memberikan perbedaan signifikan, dan kernel yang lebih kecil (3x3) justru memberikan performa terbaik dibandingkan kernel besar (7x7) pada dataset CIFAR-10.

Implementasi CNN dari scratch memberikan pemahaman mendalam tentang operasi matematis yang terjadi di setiap layer, mulai dari konvolusi, aktivasi, pooling, hingga transformasi linear pada dense layers. Pendekatan modular dengan kelas-kelas terpisah untuk setiap jenis layer memungkinkan fleksibilitas dalam membangun berbagai arsitektur CNN dan memuat bobot dari model Keras yang telah dilatih. Pemahaman ini penting untuk pengembangan model CNN yang lebih optimal, pemilihan hyperparameter yang tepat, dan identifikasi masalah seperti overfitting yang dapat diatasi dengan teknik regularisasi yang sesuai.

Berdasarkan analisis grafik training RNN, dapat disimpulkan beberapa hal penting mengenai pengaruh hyperparameter terhadap performa model. Dalam hal jumlah layer, model RNN dengan 1 layer menunjukkan performa terbaik dengan validation loss yang stabil dan accuracy sekitar 50%. Sebaliknya, model dengan 2-3 layer mengalami overfitting yang parah dimana training loss turun drastis namun validation loss justru meningkat, menunjukkan bahwa kompleksitas tambahan tidak memberikan manfaat untuk dataset ini.

Pengaruh jumlah cell per layer juga menunjukkan pola yang menarik. Model dengan 32 cells memberikan performa optimal dengan validation loss yang stabil, sementara 64 cells menyebabkan overfitting berat dengan validation loss yang melonjak drastis. Model dengan 128 cells bahkan menunjukkan performa yang buruk dengan accuracy rendah sekitar 35%, mengindikasikan bahwa model tidak dapat belajar dengan efektif ketika terlalu banyak parameter.

Dari segi arah processing, RNN bidirectional menunjukkan keunggulan yang signifikan dibanding unidirectional dengan validation accuracy tertinggi mencapai 56%. Hal ini dapat

dijelaskan karena bidirectional RNN mampu menangkap konteks dari kedua arah sequence, memberikan representasi yang lebih kaya untuk tugas sentiment analysis.

Namun, berdasarkan evaluasi F1-score pada data test, hasil menunjukkan pola yang berbeda dari analisis pada training curves. Model RNN unidirectional mencapai performa terbaik dengan F1-score 0.5805, mengalahkan bidirectional (0.4295) yang sebelumnya tampak superior dari grafik training. Hal ini mengindikasikan bahwa untuk dataset sentiment analysis berukuran kecil ini, kompleksitas tambahan dari bidirectional processing justru memperburuk generalisasi model.

Dalam hal jumlah layer, arsitektur sederhana lebih efektif dengan 1 layer (F1: 0.4006) dibanding 2 layer (0.3958) dan 3 layer (0.3682). Penurunan performa seiring bertambahnya layer menunjukkan overfitting dan vanishing gradient problem yang umum terjadi pada RNN. Untuk jumlah cell, 32 cells memberikan hasil optimal (F1: 0.4298) dibanding 64 cells (0.3833), sementara 128 cells mengalami collapse total dengan F1-score hanya 0.1844, hal ini membuktikan bahwa parameter berlebihan sangat merugikan pada dataset yang kecil.

Validasi implementasi menunjukkan keberhasilan dengan agreement 100% antara Keras dan model from scratch. Hal ini menunjukkan bahwa implementasi RNN forward propagation from scratch sudah benar karena menghasilkan hasil yang sama dengan model Keras.

Berdasarkan grafik training , model-model LSTM dengan 1, 2, atau 3 layer, serta dengan jumlah cell 16, 64, atau 128 menunjukkan performa *validation* yang kurang memuaskan. Kurva *loss* untuk validasi cenderung fluktuatif dan tidak menunjukkan penurunan yang konsisten, sementara kurva akurasi validasi juga menunjukkan volatilitas tinggi dan stagnan pada tingkat yang rendah (sekitar 33%-40%).

Secara keseluruhan, LSTM unidirectional kesulitan untuk belajar secara efektif dari data training, hal ini bisa saja terjadi karena sedikitnya data training.

Perbedaan paling signifikan dalam performa LSTM terlihat ketika membandingkan model unidirectional dengan bidirectional. Grafik training dan loss menunjukkan LSTM bidirectional jauh lebih baik daripada LSTM unidirectional. Begitu pula dengan akurasi validasi, LSTM bidirectional mencapai tingkat akurasi yang jauh lebih tinggi dibandingkan LSTM unidirectional.

Hasil F1-score juga terlihat bahwa model LSTM unidirectional hanya mencapai F1-score 0.1844 dan model LSTM bidirectional mencapai F1-score sebesar 0.7355.

Secara keseluruhan, baik model RNN maupun LSTM menunjukkan performa yang jauh lebih baik ketika menggunakan jenis bidirectional pada dataset ini. Hal ini karena pada *task* sentiment analysis kemampuan untuk menangkap konteks dari kedua arah sekuens (informasi sebelum dan sesudah token tertentu) sangat krusial. LSTM juga lebih baik daripada RNN dari segi F1-score karena LSTM mampu mampu mengatasi masalah *vanishing gradient* dengan mekanisme *gates* dan cell state-nya. Hal ini memungkinkan LSTM untuk mengingat memory

dalam sekuens data dengan lebih baik dibandingkan RNN yang seringkali kesulitan mempertahankan informasi untuk timestep yang jauh.

Terakhir, ada beberapa hal yang bisa ditingkatkan pada pengerjaan tugas besar ini, seperti melakukan hyperparameter tuning untuk tiap model agar bisa meraih skor f1 yang lebih baik.

Pembagian Tugas

NIM	Pembagian Tugas
13522027	RNN
13522067	CNN
13522095	LSTM

Referensi

CNN Explainer
Introduction to Recurrent Neural Networks | GeeksforGeeks
Understanding RNN and LSTM. What is Neural Network? | by Aditi Mittal | Medium
Understanding of LSTM Networks | GeeksforGeeks
LSTMs Explained: A Complete, Technically Accurate, Conceptual Guide with Keras | by Ryan T. J. J. | Analytics Vidhya | Medium