Laporan Tugas Besar 2
IF3270 Pembelajaran Mesin
Convolutional Neural Network dan Recurrent Neural Network

Disusun oleh:

Kelompok 20

Muhammad Zakkiy	(10122074)
Ghaisan Zaki Pratama	(10122078)
Fardhan Indrayesa	(12821046)

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

2025

DAFTAR ISI

BAB I
DESKRIPSI PERSOALAN

Tugas Besar II mata kuliah IF3270 Pembelajaran Mesin bertujuan agar mahasiswa memperoleh pemahaman mengenai implementasi Convolutional Neural Network (CNN) dan Recurrent Neural Network (RNN). Secara spesifik, mahasiswa ditugaskan untuk mengimplementasikan modul *forward propagation* untuk CNN, Simple RNN, dan LSTM *from scratch*.

Untuk bagian CNN, tugasnya meliputi:

- Melatih model CNN untuk klasifikasi gambar menggunakan *library* Keras pada dataset CIFAR-10.
- Model CNN yang dibangun minimal harus memiliki layer Conv2D, Pooling, Flatten/Global Pooling, dan Dense.
- Menggunakan *loss function* Sparse Categorical Crossentropy dan *optimizer* Adam.
- Melakukan pembagian dataset CIFAR-10 menjadi data latih, validasi, dan uji dengan rasio 40.000:10.000:10.000.
- Melakukan analisis pengaruh beberapa *hyperparameter* seperti jumlah *layer* konvolusi, banyak filter per *layer* konvolusi, ukuran filter per *layer* konvolusi, dan jenis *pooling layer* (max pooling atau average pooling) terhadap kinerja model dengan metrik F1-score macro.
- Menyimpan bobot dari model yang telah dilatih.
- Membuat modul *forward propagation from scratch* yang dapat membaca model hasil pelatihan Keras. Implementasi ini direkomendasikan bersifat modular per *layer*.
- Menguji implementasi *forward propagation from scratch* dengan membandingkan hasilnya dengan Keras menggunakan data uji dan metrik F1-score macro. Implementasi *Dense layer* dapat menggunakan kode dari Tugas Besar 1.

Untuk bagian Simple RNN dan LSTM, tugasnya meliputi:

- Melakukan *preprocessing* data teks (dataset NusaX-Sentiment Bahasa Indonesia) menjadi representasi numerik melalui tahap *tokenization* (menggunakan TextVectorization Layer Keras) dan *embedding* (menggunakan Embedding Layer Keras).
- Melatih model RNN/LSTM untuk klasifikasi teks menggunakan Keras.
- Model RNN/LSTM minimal harus memiliki *layer* Embedding, Bidirectional dan/atau Unidirectional RNN/LSTM, Dropout, dan Dense.
- Menggunakan *loss function* Sparse Categorical Crossentropy dan *optimizer* Adam.
- Melakukan analisis pengaruh beberapa *hyperparameter* seperti jumlah *layer* RNN/LSTM, banyak *cell* RNN/LSTM per *layer*, dan jenis *layer* RNN/LSTM berdasarkan arah (*bidirectional* atau *unidirectional*) terhadap kinerja model dengan metrik F1-score macro.
- Menyimpan bobot dari model yang telah dilatih.
- Membuat modul *forward propagation from scratch* yang dapat membaca model hasil pelatihan Keras. Implementasi ini direkomendasikan bersifat modular per *layer*.

- Menguji implementasi *forward propagation from scratch* dengan membandingkan hasilnya dengan Keras menggunakan data uji dan metrik F1-score macro. Implementasi *Dense layer* dapat menggunakan kode dari Tugas Besar 1.

Secara umum, implementasi *forward propagation from scratch* hanya boleh menggunakan *library* untuk perhitungan matematika seperti NumPy.

BAB II
PEMBAHASAN

2.1 Penjelasan Implementasi

2.1.1 Deskripsi Kelas

2.1.2.1 Convolutional Neural Network (CNN)

Berikut adalah deskripsi untuk setiap kelas beserta atribut dan metodenya pada implementasi forward propagation CNN from *scratch*.

1. Fungsi Aktivasi

Fungsi aktivasi memperkenalkan non-linearitas ke dalam jaringan, memungkinkan model menangkap pola kompleks:

- **ReLU(x)**
    - Deskripsi: Mengembalikan `max(0, x)` pada setiap elemen.
    - Parameter: `x` (np.ndarray)
    - Mengembalikan: (np.ndarray) output ReLU.
- **Softmax(x, axis=-1)**
    - Deskripsi: Mengubah logits menjadi distribusi probabilitas.
    - Parameter: `x` (np.ndarray), `axis` (int, default: -1)
    - Mengembalikan: (np.ndarray) distribusi softmax.

2. Fungsi Utilitas Pembuatan Bobot

Membaca struktur HDF5 dan mengekstrak bobot untuk setiap layer:

- **load_weights_from_hdf5(filepath)**
    - Deskripsi: Membuka file `.h5`, menavigasi grup `layers`, dan memanggil method `set_weights` pada setiap layer yang ditambahkan ke model.
    - Parameter: `filepath (str)`
    - Mengembalikan: None (layer diisi bobot).

3. Layer-layer

Setiap layer modul forward_prop scratch diimplementasikan sebagai kelas:

3.1 Conv2D_Scratch

- **Deskripsi**: Konvolusi 2D dari scratch, meniru `Conv2D` Keras.
- **Atribut**:
  - `W` (np.ndarray): Tensor bobot shape `(kh, kw, in_ch, out_ch)`.
  - `b` (np.ndarray): Vektor bias shape `(out_ch,)`.
  - `padding` (str): `'same'` atau `'valid'`.
  - `stride` (int).
- **Metode**:
  - `__init__(self, W, b, padding='same', stride=1)`
  - `forward(self, x)`:
    1. Tambahkan zero-padding jika `padding=='same'`.
    2. Hitung dimensi output: `H_out = (H - kh)//stride +1`.
    3. Iterasi patch dan lakukan `sum(patch * W) + b`.
    4. Mengembalikan output `(batch, H_out, W_out, out_ch)`.

3.2 Pooling_Scratch

- **Deskripsi**: Max atau Average pooling 2D.
- **Atribut**:
  - `mode` (str): `'max'` atau `'avg'`.
  - `pool_size` (tuple): `(ph, pw)`.
  - `stride` (int).
- **Metode**:
  - `__init__(self, mode='max', pool_size=(2,2), stride=2)`
  - `forward(self, x)`:
    1. Hitung `H_out`, `W_out`.
    2. Untuk setiap patch `(ph×pw)`, ambil `max` atau `mean`.
    3. Kembalikan tensor `(batch, H_out, W_out, C)`.

3.3 Flatten_Scratch

- **Deskripsi**: Meratakan tensor spasial menjadi vektor.
- **Metode**:
  - `forward(self, x)`: return `x.reshape(batch, -1)`.

3.4 GlobalAvgPool2D_Scratch

- **Deskripsi**: Rata-rata seluruh dimensi spasial.
- **Metode**:

    - `forward(self, x)`: return `x.mean(axis=(1,2))` → shape `(batch, channels)`.

3.5 Dense_Scratch

- **Deskripsi**: Lapisan fully-connected.
- **Atribut**:
    - `W` (np.ndarray): shape `(in_features, out_units)`.
    - `b` (np.ndarray): shape `(out_units,)`.
- **Metode**:
    - `__init__(self, W, b)`
    - `forward(self, x)`: return `x.dot(W) + b`.

3.6 ReLU_Scratch

- **Deskripsi**: Penerapan fungsi ReLU.
- **Metode**:
    - `forward(self, x)`: return `np.maximum(0, x)`.

3.7 Softmax_Scratch

- **Deskripsi**: Penerapan fungsi softmax.
- **Metode**:
    - `forward(self, x)`:
        1. `e = np.exp(x - x.max(axis=1, keepdims=True))`
        2. return `e / e.sum(axis=1, keepdims=True)`.

3.8 Kelas Model (CNNFromScratch)

Merepresentasikan model sekuensial scratch:

- **Atribut**:
    - `layers` (list): Daftar instansi layer scratch dalam urutan.
- **Metode**:
    - `__init__(self, h5_filepath, config)`:
        - Buka file HDF5 `layers/conv2d*`.
        - Untuk setiap `i in range(conv_layers)`, baca bobot `W, b` → `Conv2D_Scratch` → `ReLU_Scratch`.
        - Tambah `Pooling_Scratch` setelah setiap block sesuai `config['pooling']`.

        - Untuk head: jika `use_global_avg_pooling`, baca `conv2d_{n}` untuk 1×1 conv → `GlobalAvgPool2D_Scratch` → `Softmax_Scratch`; else `Flatten_Scratch` + `Dense_Scratch` + `ReLU_Scratch` + `Softmax_Scratch`.
    - `forward(self, x)`:
        - Cast `x` ke `float32`.
        - Iterasi `for layer in self.layers: x = layer.forward(x)`.
        - Return output akhir.
    - `forward_debug(self, x)`:
        - Menyimpan setiap `activation` di list untuk debugging.

2.1.2.2 Recurrent Neural Network (RNN)

Berikut adalah deskripsi untuk setiap kelas beserta atribut dan metodenya pada implementasi forward propagation RNN from *scratch*.

1. Fungsi Aktivasi (Activation Functions)

Fungsi-fungsi ini adalah komponen standar dalam neural network yang memperkenalkan non-linearitas ke dalam model, memungkinkan pembelajaran pola yang kompleks.

- `sigmoid(x)`
    - Deskripsi: Menghitung fungsi aktivasi sigmoid. Umumnya digunakan pada layer output untuk masalah klasifikasi biner.
    - Parameter:
        - `x` (np.ndarray): Input array numerik.
    - Mengembalikan: (np.ndarray) Array dengan fungsi sigmoid diterapkan pada setiap elemen.
- `tanh(x)`
    - Deskripsi: Menghitung fungsi aktivasi hyperbolic tangent (tanh). Mirip sigmoid, tetapi outputnya berkisar antara -1 dan 1. Sering digunakan pada layer tersembunyi RNN.
    - Parameter:
        - `x` (np.ndarray): Input array numerik.
    - Mengembalikan: (np.ndarray) Array dengan fungsi tanh diterapkan pada setiap elemen.
- `relu(x)`
    - Deskripsi: Menghitung fungsi aktivasi Rectified Linear Unit (ReLU). Mengembalikan input jika positif, dan nol jika negatif. Populer untuk layer tersembunyi karena efisiensi komputasinya.

    - Parameter:
        - `x` (np.ndarray): Input array numerik.
    - Mengembalikan: (np.ndarray) Array dengan fungsi ReLU diterapkan pada setiap elemen.
- `softmax(x, axis=-1)`
    - Deskripsi: Menghitung fungsi aktivasi softmax. Mengubah vektor skor numerik menjadi distribusi probabilitas atas beberapa kelas. Umumnya digunakan pada layer output untuk masalah klasifikasi multi-kelas.
    - Parameter:
        - `x` (np.ndarray): Input array numerik (skor mentah/logits).
        - `axis` (int, optional): Sumbu di mana softmax akan dihitung (default: sumbu terakhir).
    - Mengembalikan: (np.ndarray) Array dengan distribusi probabilitas softmax.

2. Fungsi Utilitas Pemuatan Bobot

Fungsi-fungsi ini bertanggung jawab untuk membaca data bobot dari file HDF5.

- `_load_weights_recursive(h5_item)`
    - Deskripsi: Fungsi helper rekursif yang dipanggil oleh `load_weights_from_hdf5`. Fungsi ini menavigasi struktur grup dan dataset dalam objek HDF5 dan membangun dictionary Python yang mencerminkan hierarki tersebut.
    - Parameter:
        - `h5_item` (h5py.Group atau h5py.Dataset): Item HDF5 (grup atau dataset) yang akan diproses.
    - Mengembalikan: (dict atau np.ndarray) Dictionary bobot untuk grup, atau array NumPy untuk dataset.
- `load_weights_from_hdf5(filepath)`
    - Deskripsi: Fungsi utama untuk memuat semua bobot dari file HDF5 yang ditentukan. Menggunakan `_load_weights_recursive` untuk membaca seluruh struktur file.
    - Parameter:
        - `filepath` (str): Path lengkap ke file `.weights.h5`.
    - Mengembalikan: (dict) Dictionary Python yang berisi semua bobot dari file, dengan kunci dan struktur yang sesuai dengan grup dan dataset di file HDF5.

3. Kelas-Kelas Layer

Setiap kelas layer di bawah ini mengimplementasikan fungsionalitas spesifik dari sebuah layer dalam neural network.

3.1. `EmbeddingLayer`

- Deskripsi Kelas: Layer ini bertugas mengubah sekuens indeks integer (yang mewakili kata atau token) menjadi sekuens vektor numerik padat (dense vectors) yang disebut embedding. Embedding menangkap informasi semantik dari token.
- Atribut:
    - `vocab_size` (int): Ukuran total vocabulary (jumlah token unik yang diketahui).
    - `embedding_dim` (int): Dimensi dari setiap vektor embedding yang dihasilkan.
    - `name` (str): Nama unik untuk layer ini dalam model.
    - `embedding_matrix` (np.ndarray): Matriks bobot aktual dari layer embedding dengan shape `(vocab_size, embedding_dim)`. Diinisialisasi menjadi `None` dan diisi saat metode `set_weights` dipanggil.
    - `_is_built` (bool): Flag internal yang menandakan apakah bobot layer telah berhasil dimuat dan layer siap digunakan.
- Metode:
    - `__init__(self, vocab_size, embedding_dim, name=None)`
        - Deskripsi: Konstruktor untuk `EmbeddingLayer`.
        - Parameter: `vocab_size` (int), `embedding_dim` (int), `name` (str, optional).
    - `set_weights(self, weights_data)`
        - Deskripsi: Mengatur matriks embedding dari `weights_data`. Metode ini mampu mem-parsing berbagai format dictionary `weights_data` untuk menemukan matriks bobot yang sebenarnya (misalnya, dari kunci `'embeddings:0'`, `'weight'`, atau struktur bersarang seperti `{'vars': {'0': ...}}`).
        - Parameter: `weights_data` (dict atau np.ndarray).
    - `forward(self, inputs)`
        - Deskripsi: Melakukan operasi forward pass, yaitu lookup embedding. Mengambil array indeks token dan menggantinya dengan vektor embedding yang sesuai.
        - Parameter: `inputs` (np.ndarray) - Array 2D `(batch_size, sequence_length)` berisi indeks integer token.
        - Mengembalikan: (np.ndarray) - Array 3D `(batch_size, sequence_length, embedding_dim)` berisi vektor embedding.

3.2. `SimpleRNNLayer`

- Deskripsi Kelas: Mengimplementasikan layer Recurrent Neural Network (RNN) sederhana unidirectional. Layer ini memproses sekuens data dengan mempertahankan state tersembunyi (hidden state) yang diperbarui pada setiap timestep.
- Atribut:
    - `units` (int): Jumlah unit dalam layer RNN, yang juga menentukan dimensi dari state tersembunyi dan output (jika `return_sequences=False`).
    - `name` (str): Nama unik layer.

    - `activation_fn` (function): Fungsi aktivasi (misalnya, `tanh` atau `sigmoid`) yang diterapkan pada state tersembunyi.
    - `return_sequences` (bool): Jika `True`, layer mengembalikan output (state tersembunyi) dari setiap timestep dalam sekuens input. Jika `False`, hanya output dari timestep terakhir yang dikembalikan.
    - `kernel` (np.ndarray): Matriks bobot untuk transformasi input (Wih) dengan shape `(input_dim, units)`.
    - `recurrent_kernel` (np.ndarray): Matriks bobot untuk transformasi state tersembunyi dari timestep sebelumnya (Whh) dengan shape `(units, units)`.
    - `bias` (np.ndarray): Vektor bias (bh) dengan shape `(units,)`.
    - `_is_built` (bool): Flag status pemuatan bobot.
- Metode:
    - `__init__(self, units, activation='tanh', return_sequences=False, name=None)`
        - Deskripsi: Konstruktor untuk `SimpleRNNLayer`.
        - Parameter: `units` (int), `activation` (str atau callable, optional), `return_sequences` (bool, optional), `name` (str, optional).
    - `_parse_activation(self, activation_input)`
        - Deskripsi: Metode helper internal untuk mengonversi string nama aktivasi menjadi objek fungsi yang sesuai.
        - Parameter: `activation_input` (str atau callable).
    - `set_weights(self, weights_data)`
        - Deskripsi: Mengatur bobot `kernel`, `recurrent_kernel`, dan `bias` dari `weights_data`. Mampu mem-parsing format Keras standar maupun format TensorFlow internal (misalnya, `{'cell': {'vars': {'0': k, '1': rk, '2': b}}}`).
        - Parameter: `weights_data` (dict).
    - `forward(self, inputs, initial_state=None)`
        - Deskripsi: Melakukan forward pass RNN. Menghitung state tersembunyi pada setiap timestep.
        - Parameter: `inputs` (np.ndarray) - Array 3D `(batch_size, timesteps, input_dim)`. `initial_state` (np.ndarray, optional) - State tersembunyi awal.
        - Mengembalikan: (np.ndarray) - Output RNN. Shape `(batch_size, timesteps, units)` jika `return_sequences=True`, atau `(batch_size, units)` jika `return_sequences=False`.

3.3. `BidirectionalSimpleRNNLayer`

- Deskripsi Kelas: Layer RNN Bidirectional yang memproses sekuens input dalam dua arah: dari depan ke belakang (forward) dan dari belakang ke depan (backward). Ini dilakukan dengan menggunakan dua instance `SimpleRNNLayer` internal. Output dari kedua arah kemudian digabungkan.
- Atribut:
    - `units` (int): Jumlah unit untuk *setiap* instance `SimpleRNNLayer` internal (forward dan backward).
    - `name` (str): Nama unik layer.
    - `return_sequences` (bool): Menentukan apakah output gabungan dari setiap timestep dikembalikan.
    - `merge_mode` (str): Metode untuk menggabungkan output dari RNN forward dan backward (misalnya, `'concat'` untuk konkatenasi, `'sum'` untuk penjumlahan).
    - `forward_rnn` (`SimpleRNNLayer`): Instance `SimpleRNNLayer` untuk memproses sekuens arah forward.
    - `backward_rnn` (`SimpleRNNLayer`): Instance `SimpleRNNLayer` untuk memproses sekuens arah backward.
    - `_is_built` (bool): Flag status pemuatan bobot (menunjukkan apakah bobot sub-layer sudah dimuat).
- Metode:
    - `__init__(self, units, activation='tanh', return_sequences=False, merge_mode='concat', name=None)`
        - Deskripsi: Konstruktor untuk `BidirectionalSimpleRNNLayer`.
        - Parameter: `units` (int), `activation` (str atau callable, optional), `return_sequences` (bool, optional), `merge_mode` (str, optional), `name` (str, optional).
    - `set_weights(self, weights_data)`
        - Deskripsi: Mengatur bobot untuk kedua `SimpleRNNLayer` internal (forward dan backward). `weights_data` diharapkan berupa dictionary yang berisi sub-dictionary untuk bobot masing-masing arah (misalnya, dengan kunci yang mengandung `'forward'` dan `'backward'`).
        - Parameter: `weights_data` (dict).
    - `forward(self, inputs)`
        - Deskripsi: Melakukan forward pass Bidirectional RNN. Memproses input menggunakan RNN forward dan backward, lalu menggabungkan hasilnya.
        - Parameter: `inputs` (np.ndarray) - Array 3D `(batch_size, timesteps, input_dim)`.
        - Mengembalikan: (np.ndarray) - Output gabungan. Jika `merge_mode='concat'` dan `return_sequences=True`, shape-nya `(batch_size, timesteps, 2*units)`. Jika `return_sequences=False`, shape-nya `(batch_size, 2*units)`.

3.4. `DenseLayer`

- Deskripsi Kelas: Layer fully-connected (terhubung penuh) standar, juga dikenal sebagai layer linear atau Multi-Layer Perceptron (MLP) layer. Menghitung transformasi linear dari input (`dot(input, kernel) + bias`) diikuti oleh fungsi aktivasi (opsional).
- Atribut:
    - `units` (int): Jumlah neuron atau dimensi output dari layer.
    - `name` (str): Nama unik layer.
    - `activation_fn` (function): Fungsi aktivasi yang diterapkan pada output layer. Jika `None`, tidak ada aktivasi yang diterapkan (aktivasi linear).
    - `kernel` (np.ndarray): Matriks bobot (W) dari layer dengan shape `(input_features, units)`.
    - `bias` (np.ndarray): Vektor bias (b) dari layer dengan shape `(units,)`.
    - `_is_built` (bool): Flag status pemuatan bobot.
- Metode:
    - `__init__(self, units, activation=None, name=None)`
        - Deskripsi: Konstruktor untuk `DenseLayer`.
        - Parameter: `units` (int), `activation` (str atau callable, optional), `name` (str, optional).
    - `_parse_activation(self, activation_input)`
        - Deskripsi: Metode helper internal untuk mengonversi string nama aktivasi menjadi objek fungsi.
        - Parameter: `activation_input` (str atau callable).
    - `set_weights(self, weights_data)`
        - Deskripsi: Mengatur bobot `kernel` dan `bias` dari `weights_data`. Mampu mem-parsing format Keras standar maupun format TensorFlow internal (misalnya, `{'vars': {'0': kernel, '1': bias}}`).
        - Parameter: `weights_data` (dict).
    - `forward(self, inputs)`
        - Deskripsi: Melakukan operasi forward pass: `output = activation(dot(inputs, kernel) + bias)`. Dapat menangani input 2D atau 3D (menerapkan operasi Dense ke setiap timestep pada input 3D).
        - Parameter: `inputs` (np.ndarray) - Array 2D `(batch_size, input_features)` atau 3D `(batch_size, timesteps, input_features)`.
        - Mengembalikan: (np.ndarray) - Output layer. Jika input 2D, outputnya `(batch_size, units)`. Jika input 3D, outputnya `(batch_size, timesteps, units)`.

3.5. `DropoutLayer`

- Deskripsi Kelas: Layer Dropout. Selama proses inferensi (prediksi), layer ini tidak melakukan operasi apa pun pada input (identitas). Logika dropout yang sebenarnya (menonaktifkan neuron secara acak) hanya relevan selama fase training model untuk mencegah overfitting.
- Atribut:
    - `rate` (float): Tingkat dropout, yaitu probabilitas sebuah unit output dari layer sebelumnya akan di-nol-kan selama training. Nilainya antara 0 dan 1.
    - `name` (str): Nama unik layer.
    - `_is_built` (bool): Selalu `True` karena layer ini tidak memiliki bobot yang perlu dimuat dari file untuk berfungsi saat inferensi.
- Metode:
    - `__init__(self, rate, name=None)`
        - Deskripsi: Konstruktor untuk `DropoutLayer`.
        - Parameter: `rate` (float), `name` (str, optional).
    - `set_weights(self, weights_data)`
        - Deskripsi: Tidak melakukan apa-apa karena `DropoutLayer` tidak memiliki bobot yang dapat dimuat dari file.
        - Parameter: `weights_data` (dict, diabaikan).
    - `forward(self, inputs, training=False)`
        - Deskripsi: Melakukan forward pass. Jika `training=False` (default untuk inferensi), mengembalikan input tanpa perubahan. Jika `training=True`, menerapkan operasi dropout.
        - Parameter: `inputs` (np.ndarray) - Data input. `training` (bool, optional) - Menandakan mode training.
        - Mengembalikan: (np.ndarray) - `inputs` itu sendiri jika `training=False`.

4. Kelas `Model`

- Deskripsi Kelas: Kelas ini merepresentasikan model neural network sekuensial, yang meniru sebagian API dari Keras Sequential model. Memungkinkan pengguna untuk menambahkan layer-layer secara berurutan, memuat bobot yang sudah dilatih sebelumnya dari file HDF5, dan melakukan prediksi (forward pass).
- Atribut:
    - `layers` (list): Sebuah list yang menyimpan semua instance layer yang telah ditambahkan ke model, dalam urutan penambahannya.
    - `name` (str): Nama untuk keseluruhan model.
    - `auto_name_counts` (dict): Dictionary internal yang digunakan untuk melacak dan menghasilkan nama unik secara otomatis untuk layer yang ditambahkan tanpa nama eksplisit.
- Metode:
    - `__init__(self, name=None)`
        - Deskripsi: Konstruktor untuk kelas `Model`.

        - Parameter: `name` (str, optional) - Nama yang akan diberikan untuk model.
    - `add(self, layer)`
        - Deskripsi: Menambahkan sebuah instance layer ke dalam model. Jika layer yang ditambahkan belum memiliki atribut `name` (atau `name` adalah `None`), metode ini akan memberikan nama otomatis berdasarkan tipe layer dan jumlah layer sejenis yang sudah ada (misalnya, "dense", "dense_1", dst.).
        - Parameter: `layer` (object) - Instance dari salah satu kelas layer (misalnya, `DenseLayer(...)`, `EmbeddingLayer(...)`).
    - `load_weights(self, filepath)`
        - Deskripsi: Memuat bobot dari file HDF5 dan mendistribusikannya ke masing-masing layer yang sesuai dalam model. Pencocokan dilakukan berdasarkan atribut `name` dari setiap layer dengan nama grup di file HDF5. Setiap layer yang relevan kemudian memanggil metode `set_weights()`-nya sendiri.
        - Parameter: `filepath` (str) - Path lengkap ke file `.weights.h5`.
    - `forward(self, inputs)`
        - Deskripsi: Melakukan proses forward pass data input melalui semua layer dalam model secara berurutan. Selama proses ini, sebuah progress bar ditampilkan di konsol.
        - Parameter: `inputs` (np.ndarray) - Data input untuk layer pertama model.
        - Mengembalikan: (np.ndarray) - Output dari layer terakhir model setelah diproses oleh seluruh sekuens layer.
    - `summary(self)`
        - Deskripsi: Mencetak ringkasan arsitektur model ke konsol. Ringkasan ini mencakup nama dan tipe setiap layer, placeholder untuk bentuk outputnya (karena bentuk output aktual bergantung pada bentuk input), dan estimasi jumlah parameter yang dapat dilatih untuk setiap layer serta total parameter model.

2.1.2.1 Long Short-Term Memory (LSTM)

Berikut adalah deskripsi untuk setiap kelas beserta atribut dan metodenya pada implementasi forward propagation LSTM from *scratch*.

1. Fungsi Utilitas

Fungsi ini sebagai alat bantu untuk mendukung proses utama dalam pemodelan.

- `batch_array(x, batch_size)`
    - Deskripsi: Fungsi untuk batch inference. Menginferensi data input sesuai dengan banyaknya batch.
    - Parameter:
        - x (np.ndarray) sebagai data input.
        - batch_size (int) menentukan banyak sampel yang diprediksi dalam satu batch (default: 32).

- Mengembalikan: (np.ndarray) Array dengan banyak sampel sesuai ukuran batch.

2. Kelas-Kelas Layer

Setiap kelas layer di bawah ini mengimplementasikan fungsionalitas spesifik dari sebuah layer dalam neural network.

2.1 `lstm`

- Deskripsi kelas: Mengimplementasikan layer Long Short-Term Memory (LSTM) sederhana unidirectional. Layer ini memproses sekuens data dengan mempertahankan dan memperbarui hidden state dan cell state pada setiap timestep.
- Atribut:
  - Units (int): Jumlah unit dalam layer LSTM, yang juga menentukan dimensi dari hidden state dan output.
  - return_seq (bool): Jika `True`, layer mengembalikan output (state tersembunyi) dari setiap timestep dalam sekuens input. Jika `False`, hanya output dari timestep terakhir yang dikembalikan.
  - return_state (bool): Jika True, layer mengembalikan output (hidden state timestep terakhir), hidden state di seluruh sekuens, dan cell state. Jika False, output yang dikembalikan hanya out saja, yang merupakan hidden state di timestep terakhir.
  - weights (list of np.ndarray): Untuk menyimpan bobot 1 layer LSTM.
  - name (string): Nama layer yang didefinisikan pada saat penambahan layer ini. Apabila pada model sudah ada layer yang sama, namanya akan diganti dan disesuaikan dengan jumlah layer yang sama.
- Metode:
  - __init__(units, return_seq=False, return_state=False)
    - Deskripsi: Konstruktor untuk lstm.
    - Parameter: units (int), return_seq (bool, optional), return_state (bool, optional).
  - set_weights(x):
    - Deskripsi: Metode untuk load bobot layer lstm.
    - Parameter: x (np.ndarray)
  - input(x, h):
    - Deskripsi: Metode untuk perhitungan input gate.
    - Parameter: x, h (np.ndarray)
  - forget(x, h):
    - Deskripsi: Metode untuk perhitungan forget gaate.
    - Parameter: x, h (np.ndarray)
  - c_tilde(x, h):
    - Deskripsi: Metode untuk perhitungan cell state awal.
    - Parameter: x, h (np.ndarray)
  - output(x, h):
    - Deskirpsi: Metode untuk perhitungan output gate.
    - Parameter: x,h (np.ndarray)
  - c_state(f, c, i, c_tilde):

        - Deskripsi: Metode untuk perhitungan cell state akhir.
        - Parameter: c, i, c_tilde (np.ndarray)
    - h_state(o, c):
        - Deskripsi: Metode untuk perhitungan hidden state akhir.
        - Parameter: o, c (np.ndarray)
    - get_initial_state(batch_size):
        - Deskripsi: Method untuk mendapatkan nilai cell dan hidden state awal (urutan sekuens 0).
        - Parameter: batch_size (int)
    - forward(x):
        - Deskripsi: Method untuk prediksi layer LSTM, mulai dari perhitungan masing-masing gate hingga menghasilkan output berupa hidden state di timestep terakhir (default).

2.2 dropout

- Deskripsi kelas: Layer Dropout. Selama proses inferensi (prediksi), layer ini tidak melakukan operasi apa pun pada input (identitas). Logika dropout yang sebenarnya (menonaktifkan neuron secara acak) hanya relevan selama fase training model untuk mencegah overfitting.
- Atribut:
    - rate (float): Tingkat dropout, yaitu probabilitas sebuah unit output dari layer sebelumnya akan di-nol-kan selama training. Nilainya antara 0 dan 1.
    - training (bool): Sebagai penanda apakah untuk training atau inferensi.
    - r (list of int): masking neuron yang ditetapkan berdasarkan probabilitas binomial.
    - name (str): Nama unik layer.
- Metode:
    - __init__(rate, training=False):
        - Deskripsi: Konstruktor untuk dropout layer.
        - Parameter: rate (float), training (bool, optional)
    - forward(x):
        - Deskripsi: Melakukan forward pass. Jika `training=False` (default untuk inferensi), mengembalikan input tanpa perubahan. Jika `training=True`, menerapkan operasi dropout.
        - Parameter: x (nd.array)
        - Mengembalikan: (np.ndarray) hasil dropout atau data input jika training=False.

2.3 dense

- Deskripsi kelas: Layer Dropout. Layer fully-connected (terhubung penuh) standar, juga dikenal sebagai layer linear atau Multi-Layer Perceptron (MLP) layer. Menghitung transformasi linear dari input (`dot(input, kernel) + bias`) diikuti oleh fungsi aktivasi (opsional).
- Atribut:
    - units (int): Jumlah neuron atau dimensi output dari layer.
    - activation (str): Fungsi aktivasi yang diterapkan pada output layer.

    - weights (list of np.ndarray): Atribut untuk menyimpan bobot layer
    - name (str): Nama unik layer.
- Metode:
    - __init__(units, activation=”linear”):
        - Deskripsi: Konstruktor untuk dense layer.
        - Parameter: units (int), activation (str)
    - set_weights(x):
        - Deskripsi: Metode untuk load bobot layer dense.
        - Parameter: x (np.ndarray)
    - forward(x):
        - Deskripsi: Melakukan forward pass untuk dense layer.
        - Parameter: x (nd.array)
        - Mengembalikan: (np.ndarray) hasil neuron dense layer yang sudah dihitung menggunakan fungsi aktivasi.

2.4 embedding

- Deskripsi kelas: bertugas mengubah sekuens indeks integer (yang mewakili kata atau token) menjadi sekuens vektor numerik padat (dense vectors) yang disebut embedding. Embedding menangkap informasi semantik dari token.
- Atribut:
    - input_dim (int): Ukuran total vocabulary.
    - output_dim (int): Dimensi output untuk setiap vektor embedding yang dihasilkan.
    - weights (list of np.ndarray): Atribut untuk menyimpan bobot layer.
    - name (str): Nama unik layer.
- Metode:
    - __init__(input_dim, output_dim, weights):
        - Deskripsi: Konstruktor untuk dense layer.
        - Parameter: input_dim (int), output_dim (int), weights (list of np.ndarray)
    - forward(x):
        - Deskripsi: Melakukan forward pass untuk embedding layer, yaitu mengambil indeks dari matriks bobot berdasarkan indeks setiap kata pada data input.
        - Parameter: x (nd.array)
        - Mengembalikan: (np.ndarray) vektor hasil embedding

2.5 bidirectional

- Deskripsi kelas: Layer LSTM Bidirectional yang memproses sekuens input dalam dua arah: dari depan ke belakang (forward) dan dari belakang ke depan (backward). Ini dilakukan dengan menggunakan dua instance `lstm` internal. Output dari kedua arah kemudian digabungkan.
- Atribut:
    - layer (class lstm): Berisi object layer LSTM
    - merge_mode (str): Mode penggabungan yang digunakan (default: concat).
    - backward_layer (class lstm): Berisi object layer LSTM (default).
    - weights (list of np.ndarray): Atribut untuk menyimpan bobot layer.

    - name (str): Nama unik layer.
- Metode:
    - __init__(layer, merge_mode, backward_layer):
        - Deskripsi: Konstruktor untuk dense layer.
        - Parameter: layer (class lstm), merge_mode (str), backward_layer (class lstm)
    - set_weights(x):
        - Deskripsi: Metode untuk load bobot layer bidirectional.
        - Parameter: x (np.ndarray)
    - forward(x):
        - Deskripsi: Melakukan forward pass untuk bidirectional layer LSTM. Memproses input menggunakan LSTM forward dan backward, lalu menggabungkan hasilnya.
        - Parameter: x (nd.array)
        - Mengembalikan: (np.ndarray) Output hasil gabungan kedua layer LSTM.

3. Kelas Model (Sequential)

- Deskripsi kelas: Kelas ini merepresentasikan model neural network sekuensial. Memungkinkan pengguna untuk menambahkan layer-layer secara berurutan, memuat bobot yang sudah dilatih sebelumnya dari file HDF5, dan melakukan prediksi (forward pass).
- Atribut:
    - seq (list of layer class): List yang menyimpan semua layer yang telah ditambahkan ke model, dengan urutan sesuai dengan penambahannya.
    - layer_names (list of str): Nama untuk masing-masing layer
    - weights (list of np.ndarray): Atribut untuk menyimpan bobot masing-masing layer.
- Metode:
    - __init__():
        - Deskripsi: Konstruktor untuk sequential class.
    - get_weights(fname):
        - Deskripsi: Metode untuk load bobot setiap layer.
        - Parameter: fname (str) - path dan nama file
    - add(layer):
        - Deskripsi: Metode untuk menambahkan layer ke model sequential.
        - Parameter: layer (class of <layer>).
    - predict(x, batch_size):
        - Deskripsi: method dengan argumen data input (x) dan ukuran batch yang berfungsi menghitung model secara forward propagation untuk setiap batch data. Nilai default batch adalah 32, tetapi apabila jumlah sampel data input kurang dari 32, jumlahnya akan disesuaikan dengan banyak sampel pada data input tersebut.
        - Parameter: x (nd.array), batch_size (int)
        - Mengembalikan: (np.ndarray) Output hasil forward propagation seluruh layer.

2.1.2 Penjelasan Forward Propagation

2.1.2.1 Convolutional Neural Network (CNN)

Arsitektur CNN untuk model Keras sebagai berikut:

```
import os
from tensorflow.keras import layers, models, initializers
from tensorflow.keras.callbacks import CSVLogger, ModelCheckpoint

os.makedirs('checkpoints', exist_ok=True)

def build_cnn(conv_layers=3,
              filters=[96,192,192],
              kernel_size=(3,3),
              pooling='max',
              use_global_avg_pooling=True):
  assert conv_layers == len(filters)
  inp = layers.Input(shape=(32,32,3))
  x = inp
  for i in range(conv_layers) :
    x = layers.Conv2D(filters=filters[i],
                      kernel_size=kernel_size,
                      padding='same',
                      activation='relu')(x)
    if (i + 1) % 3 == 0:
      if pooling == 'max' :
        x = layers.MaxPooling2D(pool_size=(3,3),strides=2)(x)
      else :
        x = layers.AveragePooling2D(pool_size=(3,3),strides=2)(x)
  if use_global_avg_pooling :
    x = layers.Conv2D(10,(1,1),padding='valid')(x)
    x = layers.GlobalAveragePooling2D()(x)
  else :
    x = layers.Flatten()(x)
                                                          x               =
layers.Dense(200,activation='relu',kernel_initializer=initializers.glorot_
normal())(x)
    x = layers.Dense(100,activation='relu')(x)
    x = layers.Dense(10)(x)
  out = layers.Activation('softmax',dtype='float32')(x)

  model = models.Model(inputs=inp, outputs=out)
  model.compile(
```

```
      loss = 'sparse_categorical_crossentropy',
      optimizer = 'adam',
      metrics = []
  )
  return model
variant_name = "conv3_filters_96-192-192_kernel3_pool-max"
model = build_cnn(conv_layers=3,
              filters=[96,192,192],
              kernel_size=(3,3),
              pooling='max',
              use_global_avg_pooling=True)

csv_logger = CSVLogger(f'history_{variant_name}.csv')
chkpt = ModelCheckpoint(
    filepath=f'checkpoints/{variant_name}.weights.h5',
    save_weights_only=True,
    monitor='val_loss',
    save_best_only=True
)

history = model.fit(
    ds_train,
    epochs=20,
    validation_data=ds_val,
    callbacks=[csv_logger, chkpt]
)
from sklearn.metrics import f1_score

weights_path = f'checkpoints/{variant_name}.weights.h5'
model.load_weights(weights_path)

y_pred_probs = model.predict(ds_test)
y_pred = y_pred_probs.argmax(axis=-1)

y_true = y_test.flatten()

f1 = f1_score(y_true, y_pred, average='macro')
print(f"[{variant_name}] Keras macro-F1 on test set: {f1:.4f}")
```

Nilai Macro F1 untuk model Keras sebagai berikut

`[conv3_filters_96-192-192_kernel3_pool-max] Keras macro-F1 on test set: 0.7142`

Arsitektur CNN untuk model from scratch sebagai berikut:

Forward propagation "from scratch" untuk CNN, kami mengorganisasikan setiap komponen ke dalam kelas-kelas Python terpisah, sesuai prinsip pemrograman berorientasi objek (*Object Oriented Programming*). Setiap kelas merepresentasikan satu jenis layer, mulai dari konvolusi (Conv2D_Scratch), fungsi aktivasi (ReLU_Scratch), pooling (Pooling_Scratch), flattening (Flatten_Scratch), global average pooling (GlobalAvgPool2D_Scratch), dense fully-connected (Dense_Scratch), dan softmax (Softmax_Scratch). Dengan memecah menjadi bagian-bagian kecil, fungsi `CNNFromScratch` menjadi sangat ringkas, di dalam konstruktor (`__init__`) perlu membuka file bobot Keras (`.weights.h5`), membaca bobot dan bias masing-masing layer, lalu menambahkan instance-instance layer scratch ke dalam list `self.layers` sesuai urutan arsitektur. Semua detail tentang bagaimana bobot disimpan, di grup HDF5 `layers/conv2d, conv2d_1,` dst, dibaca sekaligus di-map ke objek layer yang sesuai. Setiap kelas layer memiliki atribut bobot dan method `forward(x)` untuk menjalankan perhitungan numerik.

```
class Conv2D_Scratch:
    def __init__(self, W, b, padding='same', stride=1):
        self.W, self.b = W, b
        self.padding, self.stride = padding, stride

    def forward(self, x):
        batch, H, W, in_ch = x.shape
        kh, kw, _, out_ch = self.W.shape
        s = self.stride
        if self.padding == 'same':
            pad_h = (kh - 1) // 2
            pad_w = (kw - 1) // 2
                 x = np.pad(x, ((0,0),(pad_h,pad_h),(pad_w,pad_w),(0,0)),
mode='constant')
        H_out = (x.shape[1] - kh) // s + 1
        W_out = (x.shape[2] - kw) // s + 1
        out = np.zeros((batch, H_out, W_out, out_ch), dtype=np.float32)
        for n in range(batch):
            for i in range(H_out):
                for j in range(W_out):
                    for c in range(out_ch):
                        v0 = i * s
                        h0 = j * s
                        patch = x[n, v0:v0+kh, h0:h0+kw, :]
```

```
                        out[n, i, j, c] = np.sum(patch * self.W[..., c]) +
self.b[c]
        return out
```

Class `Conv2D_Scratch` mereplikasi operasi konvolusi 2D dalam Keras. Konstruktornya menyimpan kernel `W` (dimensi (`kh, kw, in_ch, out_ch`)) dan bias `b` (panjang `out_ch`), juga parameter **padding** dan **stride**. Pada method `forward`, masukan `x` berdimensi (`batch, H, W, in_ch`) di-pad jika padding '`same`', lalu loop nested menghasilkan tensor keluaran (`batch, H_out, W_out, out_ch`). Setiap elemen dihitung dengan dot-product patch input dan kernel, lalu ditambah bias.

```
class ReLU_Scratch:
    def forward(self, x):
        return np.maximum(0, x)
```

Layer `ReLU_Scratch` menyediakan fungsi aktivasi **ReLU** (*Rectified Linear Unit*). Mengubah setiap nilai negatif di tensor masukan menjadi nol, mempertahankan nilai positif apa adanya (linear). Ini sama dengan Keras `activation='relu'` pada setiap lapisan konvolusi.

```
class Pooling_Scratch:
    def __init__(self, mode='max', pool_size=(2,2), stride=2):
        self.mode = mode
        self.ph, self.pw = pool_size
        self.stride = stride

    def forward(self, x):
        batch, H, W, C = x.shape
        ph, pw, s = self.ph, self.pw, self.stride
        H_out = (H - ph) // s + 1
        W_out = (W - pw) // s + 1
        out = np.zeros((batch, H_out, W_out, C), dtype=x.dtype)
        for n in range(batch):
            for i in range(H_out):
                for j in range(W_out):
                    v0 = i * s
                    h0 = j * s
                    patch = x[n, v0:v0+ph, h0:h0+pw, :]
                    if self.mode == 'max':
                        out[n, i, j, :] = patch.reshape(-1, C).max(axis=0)
                    else:
                                        out[n, i, j, :] = patch.reshape(-1,
C).mean(axis=0)
        return out
```

`Pooling_Scratch` meniru `MaxPooling2D` atau `AveragePooling2D` di Keras. Konstruktor menerima `mode`, `pool_size`, dan `stride`. Method `forward` melakukan slide window berukuran (`ph, pw`) pada tensor (`batch, H, W, C`), kemudian memilih nilai tertinggi pada tiap channel ketika mode `'max'` atau menghitung rata-rata nilai ketika mode `'avg'`. Pooling mengurangi resolusi spasial dan membantu ekstraksi fitur lebih kompak.

```
class Flatten_Scratch:
    def forward(self, x):
        return x.reshape(x.shape[0], -1)

class GlobalAvgPool2D_Scratch:
    def forward(self, x):
        return x.mean(axis=(1,2))
```

`Flatten_Scratch` mengubah tensor 4D (`batch, H, W, C`) menjadi vektor 2D (`batch, H*W*C`) agar dapat dilanjutkan ke dense layer. `GlobalAvgPooling2D_Scratch` menghitung rata-rata nilai di dimensi spasial (`H, W`) untuk setiap channel, menghasilkan tensor (`batch, C`). Ini identik dengan Keras `GlovalAveragePooling2D`, memungkinkan model merangkum fitur spasial secara efisien tanpa jumlah parameter baru.

```
class Dense_Scratch:
    def __init__(self, W, b):
        self.W, self.b = W, b
    def forward(self, x):
        return x.dot(self.W) + self.b
```

`Dense_Scratch` menggunakan bobot dan bias yang sama dengan lapisan Dense di Keras. Method `forward` melakukan transformasi linear `xW+b`, diikuti aktivasi ReLU atau softmax.

```
class Softmax_Scratch:
    def forward(self, x):
        e = np.exp(x - x.max(axis=1, keepdims=True))
        return e / e.sum(axis=1, keepdims=True)
```

`Softmax_Scratch` mengubah **logits** menjadi distribusi probabilitas, stabil dengan pengurangan `x.max(...)`. Ini sama dengan Keras `Activation('softmax')`, memastikan persis nilai output di seluruh kelas sesuai.

Berikut merupakan wrapper yang merangkai semua layer scratch sesuai arsitektur yang digunakan dengan Keras:

```
class CNNFromScratch:
    def __init__(self, h5_filepath, config):
        self.layers = []
        f = h5py.File(h5_filepath, 'r')
        layers_group = f['layers']
        # conv blocks
```

```
        for i in range(config['conv_layers']):
            layer_name = 'conv2d' if i == 0 else f'conv2d_{i}'
            vars_group = layers_group[layer_name]['vars']
            W = vars_group['0'][:]
            b = vars_group['1'][:]
                  self.layers.append(Conv2D_Scratch(W, b, padding='same',
stride=1))
            self.layers.append(ReLU_Scratch())
            if (i+1) % 3 == 0:
                self.layers.append(Pooling_Scratch(mode=config['pooling'],
pool_size=(3,3), stride=2))
        # output head
        if config['use_global_avg_pooling']:
            out_name = f'conv2d_{config["conv_layers"]}'
            vars_group = layers_group[out_name]['vars']
            Wf = vars_group['0'][:]
            bf = vars_group['1'][:]
                self.layers.append(Conv2D_Scratch(Wf, bf, padding='valid',
stride=1))
            self.layers.append(GlobalAvgPool2D_Scratch())
            self.layers.append(Softmax_Scratch())
        else:
            self.layers.append(Flatten_Scratch())
            # dense layers sequential
                      dense_names = [name for name in layers_group if
name.startswith('dense')]
            for dn in dense_names:
                vars_group = layers_group[dn]['vars']
                Wd = vars_group['0'][:]
                bd = vars_group['1'][:]
                self.layers.append(Dense_Scratch(Wd, bd))
                self.layers.append(ReLU_Scratch())
            self.layers.append(Softmax_Scratch())
        f.close()

    def forward(self, x):
        out = x.astype(np.float32)
        for layer in self.layers:
            out = layer.forward(out)
        return out
```

Konstruktor membuka file bobot, membaca grup `layers`, lalu menambahkan instance-instance layer secara berurutan di list `self.layers`. Method `forward` menjalankan tensor input melalui seluruh layer scratch, sehingga mengembalikan output probabilitas akhir.
Method forward pada CNNFromScratch menggabungkan semua model layer ini menjadi satu alur komputasi linier. Pertama-tama mengonversi input ke tipe float32 dan mengikuti skema pelatihan Keras tanpa normalisasi, memanggil layer.forward(out) berulang pada setiap objek di self.layers. Urutan ini mereplikasi sejumlah layer berikut, beberapa blok Conv2D ke ReLU diikuti setiap tiga konvolusi oleh Pooling, kemudian di jalur global-average-pooling diakhiri dengan Conv2D 1x1 dan GlovalAvgPool2D, sebelum Softmax. Jalur alternatif Flatten ke Dense lalu ke Dense kemudian ke Dense lagi sampai ke Softmax dapat digunakan jika use_global_avg_pooling=False. Dengan cara ini, semua transformasi numerik dan non-linear dalam Keras dijalankan ulang di scratch hanya dalam mode inference, tanpa backpropagation atau update bobot.

```
import numpy as np
from sklearn.metrics import f1_score
for x_batch, y_batch in ds_test.take(1):
    x_batch_np = x_batch.numpy()  # shape (16,32,32,3)
    break
variant_name = "conv3_filters_96-192-192_kernel3_pool-max"
keras_model = build_cnn(
    conv_layers=3,
    filters=[96,192,192],
    kernel_size=(3,3),
    pooling='max',
    use_global_avg_pooling=True
)
keras_model.load_weights(f'checkpoints/{variant_name}.weights.h5')
y_keras = keras_model.predict(x_batch_np).argmax(axis=-1)

config = {
    'conv_layers': 3,
    'filters': [96,192,192],
    'kernel_size': (3,3),
    'pooling': 'max',
    'use_global_avg_pooling': True
}
scratch = CNNFromScratch(f'checkpoints/{variant_name}.weights.h5', config)
y_scratch_probs = scratch.forward(x_batch_np)
y_scratch = y_scratch_probs.argmax(axis=-1)
```

```
print("Keras preds:   ", y_keras)
print("Scratch preds: ", y_scratch)
print("Same?          ", np.all(y_keras == y_scratch))

matches = (y_keras == y_scratch)
print(f"Batch         accuracy         match:         {matches.mean():.4f}
({matches.sum()}/{len(matches)})")

mismatch_idx = np.where(~matches)[0]
print("Example mismatches at positions:", mismatch_idx[:10])

f1_batch = f1_score(y_batch.numpy().flatten(), y_scratch, average='macro')
print(f"Batch macro-F1: {f1_batch:.4f}")
```

```
Keras preds:    [3 8 8 8 6 6 1 4 3 1 0 9 5 7 9 8 5 7 8 6 7 0 8 9 4 5 3 0 9
6 6 5 4 3 9 8 4
 1 9 5 4 6 3 6 0 9 3 9 7 4 9 8 7 3 8 8 5 3 5 3 7 3 6 3 6 2 1 2 3 9 2 4 8 8
 0 2 9 5 3 8 8 1 1 7 2 2 2 4 8 9 0 3 8 6 4 6 6 2 0 7 4 2 5 3 1 1 8 6 8 7 4
 2 2 4 1 0 0 9 2 5 8 3 1 2 8 1 8 3]
Scratch preds:  [3 8 8 8 6 6 1 4 3 1 0 9 5 7 9 8 5 7 8 6 7 0 8 9 4 5 3 0 9
6 6 5 4 3 9 8 4
 1 9 5 4 6 3 6 0 9 3 9 7 4 9 8 7 3 8 8 5 3 5 3 7 3 6 3 6 2 1 2 3 9 2 4 8 8
 0 2 9 5 3 8 8 1 1 7 2 2 2 4 8 9 0 3 8 6 4 6 6 2 0 7 4 2 5 3 1 1 8 6 8 7 4
 2 2 4 1 0 0 9 2 5 8 3 1 2 8 1 8 3]
Same?           True
Batch accuracy match: 1.0000 (128/128)
Example mismatches at positions: []
Batch macro-F1: 0.7414
```

Pada bagian tersebut, kami melakukan verifikasi ent-to-end bahwa forward propagation from scratch benar-benar sama dengan prediksi model Keras hingga ke label terakhir. Prosesnya dapat dirangkum seperti berikut:

1. Persiapan Batch Test
   Kami ambil satu batch dari `ds_test` (ukuran 128 citra 32x32) dan ubah ke NumPy array (`x_batch_np`) agar bisa diproses. Kami tidak melakukan normalisasi karena hasil macro F1 yang didapat lebih bagus tanpa normalisasi.
2. Prediksi dengan Keras
   Model Keras yang sama dengan yang dilatih, dengan 3 layer konvolusi `[96,192,192]`, kernel 3x3, dan max pooling dimuat bobotnya, kemudian kami panggil `keras_model.predict(x_batch_np).argmax(axis=-1)` untuk mendapatkan label prediksi..
3. Prediksi dengan Forward Propagation from Scratch

Objek `CNNFromScratch` diinisialisasi dengan file bobot `.h5` dan `config` identik. Memanggil `scratch.forward(x_batch_np)` menjalankan seluruh rangkaian layers secara manual menggunakan NumPy. Argmax dari probabilitas hasil ini adalah `y_scratch`.

4. Perbandingan Prediksi
   Keluaran `y_keras` dan `y_scratch` dibandingkan dengan `np.all(y_keras==y_scratch)`, kami mendapat True, artinya semua 128 prediksi identik di kedua model, dengan accuracy match nya 100%. Kami tidak menggunakan seluruh data test karena komputasi yang sangat besar dan lama sehingga google colab yang sudah menggunakan GPU T4 tetap tidak kuat untuk menampilkan hasilnya.
5. Analisis Mismatches
   Karena tidak ada mismatch, daftar indeks mismatch kosong, menunjukkan tidak ada perbedaan apapun di sepanjang batch.
6. Evaluasi Macro F1 pada Batch
   Hasl Macro F1 yang didapat sama baik menggunakan Keras ataupun from scratch.

Dengan demikian, bagian ini memberikan bukti bahwa implementasi forward propagation from scratch berhasil memberikan hasil yang sama dengan Keras.

2.1.2.2 Simple Recurrent Neural Network (RNN)

Berikut merupakan class dan method yang digunakan dalam RNN from scratch:

```
import h5py
import numpy as np
import os
import sys


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

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

def relu(x):
    return np.maximum(0, x)

def softmax(x, axis=-1):
    e_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
    return e_x / np.sum(e_x, axis=axis, keepdims=True)

def _load_weights_recursive(h5_item):
    weights = {}
    if isinstance(h5_item, h5py.Group):
        for name_key in h5_item.keys():
            item = h5_item[name_key]
            if isinstance(item, h5py.Dataset):
                weights[name_key] = item[()]
```

```
            elif isinstance(item, h5py.Group):
                weights[name_key] = _load_weights_recursive(item)
    elif isinstance(h5_item, h5py.Dataset):
        return h5_item[()]
    return weights

def load_weights_from_hdf5(filepath):
    if not os.path.exists(filepath):
        raise FileNotFoundError(f"File bobot tidak ditemukan di: {filepath}")
    with h5py.File(filepath, 'r') as hf:
        return _load_weights_recursive(hf)

class EmbeddingLayer:
    def __init__(self, vocab_size, embedding_dim, name=None):
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.name = name
        self.embedding_matrix = None
        self._is_built = False

    def set_weights(self, weights_data):
        actual_matrix = None
        if isinstance(weights_data, np.ndarray):
            actual_matrix = weights_data
        elif isinstance(weights_data, dict):
              potential_key_direct = next((k for k in weights_data if 'embeddings' in k or k ==
'weight'), None)
                if potential_key_direct and isinstance(weights_data.get(potential_key_direct),
np.ndarray):
                actual_matrix = weights_data[potential_key_direct]
            elif 'vars' in weights_data and isinstance(weights_data.get('vars'), dict):
                vars_dict = weights_data['vars']
                if '0' in vars_dict and isinstance(vars_dict.get('0'), np.ndarray):
                    actual_matrix = vars_dict['0']
                else:
                     potential_key_in_vars = next((k for k in vars_dict if 'embeddings' in k or
k == 'weight'), None)
                                                                    if   potential_key_in_vars   and
isinstance(vars_dict.get(potential_key_in_vars), np.ndarray):
                        actual_matrix = vars_dict[potential_key_in_vars]
            elif len(weights_data) == 1:
                first_value = next(iter(weights_data.values()))
                if isinstance(first_value, np.ndarray):
                    actual_matrix = first_value

        if actual_matrix is None:
                raise ValueError(f"Tidak dapat mengekstrak matriks embedding untuk layer
'{self.name}' dari data: {list(weights_data.keys() if isinstance(weights_data,dict) else
'N/A')}")

        # Validasi shape
```

```
                if actual_matrix.shape[0] != self.vocab_size or actual_matrix.shape[1] != self.embedding_dim:
                    raise ValueError(f"Shape matriks embedding ({actual_matrix.shape}) untuk layer '{self.name}' tidak cocok "
                                     f"dengan konfigurasi ({self.vocab_size}, {self.embedding_dim}).")
            self.embedding_matrix = actual_matrix
            self._is_built = True

        def forward(self, inputs):
            """Melakukan forward pass."""
            if not self._is_built or self.embedding_matrix is None:
                raise RuntimeError(f"Bobot untuk EmbeddingLayer '{self.name}' belum dimuat. Panggil load_weights pada model.")

            if not isinstance(inputs, np.ndarray): inputs = np.array(inputs)
            if inputs.ndim == 0: inputs = np.expand_dims(inputs, axis=0)

            if np.any(inputs >= self.embedding_matrix.shape[0]) or np.any(inputs < 0):
                    raise ValueError(f"Indeks input ({np.min(inputs)}-{np.max(inputs)}) di luar jangkauan embedding matrix untuk '{self.name}' (0-{self.embedding_matrix.shape[0]-1}). Shape matriks: {self.embedding_matrix.shape}")

            if inputs.ndim == 1: return self.embedding_matrix[inputs]
            elif inputs.ndim == 2:
                batch_size, seq_len = inputs.shape
                output = np.zeros((batch_size, seq_len, self.embedding_dim))
                for i in range(batch_size): output[i] = self.embedding_matrix[inputs[i]]
                return output
            else: raise ValueError("Input Embedding harus 1D atau 2D.")


    class SimpleRNNLayer:
        def __init__(self, units, activation='tanh', return_sequences=False, name=None):
            self.units = units
            self.name = name
            if isinstance(activation, str):
                if activation == 'tanh': self.activation_fn = tanh
                elif activation == 'sigmoid': self.activation_fn = sigmoid
                elif activation == 'relu': self.activation_fn = relu
                else: raise ValueError(f"Aktivasi string tidak dikenal: {activation}")
            else: self.activation_fn = activation
            self.return_sequences = return_sequences
            self.kernel, self.recurrent_kernel, self.bias = None, None, None
            self._is_built = False

        def set_weights(self, weights_data):
            k, rk, b = None, None, None
            k = weights_data.get('kernel') or weights_data.get('kernel:0')
            rk = weights_data.get('recurrent_kernel') or weights_data.get('recurrent_kernel:0')
            b = weights_data.get('bias') or weights_data.get('bias:0')
```

```
        if k is None and 'cell' in weights_data and isinstance(weights_data.get('cell'), dict) and \
                'vars' in weights_data['cell'] and isinstance(weights_data['cell'].get('vars'), dict):
            cell_vars = weights_data['cell']['vars']
            k, rk, b = cell_vars.get('0'), cell_vars.get('1'), cell_vars.get('2')

        if k is None or rk is None or b is None:
                raise ValueError(f"Bobot kernel/recurrent_kernel/bias untuk layer '{self.name}' tidak ditemukan atau tidak lengkap di data: {list(weights_data.keys()) if isinstance(weights_data,dict) else 'N/A'}")

        if not (k.shape[1] == self.units and rk.shape[0] == self.units and \
                rk.shape[1] == self.units and b.shape[0] == self.units):
                    raise ValueError(f"Shape bobot untuk '{self.name}' tidak cocok dengan units={self.units}. K:{k.shape}, RK:{rk.shape}, B:{b.shape}")

        self.kernel, self.recurrent_kernel, self.bias = k, rk, b
        self._is_built = True

    def forward(self, inputs, initial_state=None):
        """Melakukan forward pass."""
          if not self._is_built: raise RuntimeError(f"Bobot untuk SimpleRNNLayer '{self.name}' belum dimuat.")

            if inputs.ndim != 3: raise ValueError(f"Input SimpleRNN '{self.name}' harus 3D (batch_size, timesteps, input_dim). Diterima: {inputs.shape}")
        batch_size, timesteps, input_dim = inputs.shape

             if self.kernel.shape[0] != input_dim: raise ValueError(f"Dimensi input kernel ({self.kernel.shape[0]}) di '{self.name}' tidak cocok dengan input_dim ({input_dim}).")

        h_t = np.zeros((batch_size, self.units)) if initial_state is None else initial_state
        outputs_sequence = []
        for t in range(timesteps):
            x_t = inputs[:, t, :]
                        h_t = self.activation_fn(np.dot(x_t, self.kernel) + np.dot(h_t, self.recurrent_kernel) + self.bias)
            if self.return_sequences: outputs_sequence.append(h_t)

        return np.stack(outputs_sequence, axis=1) if self.return_sequences else h_t



class BidirectionalSimpleRNNLayer:
    def __init__(self, units, activation='tanh', return_sequences=False, merge_mode='concat', name=None):
        self.units = units
        self.name = name
        self.return_sequences = return_sequences
        self.merge_mode = merge_mode
```

```
                    self.forward_rnn = SimpleRNNLayer(units, activation, return_sequences,
name=f"{name}_forward_rnn_internal")
                    self.backward_rnn = SimpleRNNLayer(units, activation, return_sequences,
name=f"{name}_backward_rnn_internal")
        self._is_built = False

    def set_weights(self, weights_data):
        fwd_key, bwd_key = None, None
        for k_dict in weights_data:
            if 'forward' in k_dict.lower(): fwd_key = k_dict
            elif 'backward' in k_dict.lower(): bwd_key = k_dict

        if not fwd_key or not bwd_key:
              raise ValueError(f"Kunci forward/backward layer tidak ditemukan di '{self.name}'
dari data: {list(weights_data.keys())}")

        self.forward_rnn.set_weights(weights_data[fwd_key])
        self.backward_rnn.set_weights(weights_data[bwd_key])
        self._is_built = True

    def forward(self, inputs):
        """Melakukan forward pass."""
          if not self._is_built: raise RuntimeError(f"Bobot untuk BidirectionalSimpleRNNLayer
'{self.name}' belum dimuat.")

        output_forward = self.forward_rnn.forward(inputs)
        inputs_reversed = np.flip(inputs, axis=1)
        output_backward_reversed = self.backward_rnn.forward(inputs_reversed)

          output_backward = np.flip(output_backward_reversed, axis=1) if self.return_sequences
else output_backward_reversed

                    if self.merge_mode == 'concat': return np.concatenate((output_forward,
output_backward), axis=-1)
        elif self.merge_mode == 'sum': return output_forward + output_backward
            else: raise ValueError(f"Mode penggabungan '{self.merge_mode}' tidak didukung di
'{self.name}'.")


class DenseLayer:
    def __init__(self, units, activation=None, name=None):
        self.units = units
        self.name = name
        if isinstance(activation, str):
            if activation == 'tanh': self.activation_fn = tanh
            elif activation == 'sigmoid': self.activation_fn = sigmoid
            elif activation == 'relu': self.activation_fn = relu
            elif activation == 'softmax': self.activation_fn = softmax
            elif activation is None or activation == 'linear': self.activation_fn = None
            else: raise ValueError(f"Aktivasi string tidak dikenal: {activation}")
        elif callable(activation): self.activation_fn = activation
```

```
            elif activation is not None: raise ValueError(f"Tipe aktivasi tidak valid: {type(activation)}")
        else: self.activation_fn = None
        self.kernel, self.bias = None, None
        self._is_built = False

    def set_weights(self, weights_data):
        k, b = None, None
        k = weights_data.get('kernel') or weights_data.get('kernel:0')
        b = weights_data.get('bias') or weights_data.get('bias:0')
        if k is None and 'vars' in weights_data and isinstance(weights_data.get('vars'), dict):
            layer_vars = weights_data['vars']
            k, b = layer_vars.get('0'), layer_vars.get('1')

        if k is None or b is None:
            raise ValueError(f"Bobot kernel/bias untuk layer '{self.name}' tidak ditemukan atau tidak lengkap di data: {list(weights_data.keys()) if isinstance(weights_data,dict) else 'N/A'}")

        if k.shape[1] != self.units or b.shape[0] != self.units:
             raise ValueError(f"Shape bobot untuk '{self.name}' tidak cocok units={self.units}. K:{k.shape}, B:{b.shape}")

        self.kernel, self.bias = k, b
        self._is_built = True

    def forward(self, inputs):
        """Melakukan forward pass."""
         if not self._is_built: raise RuntimeError(f"Bobot untuk DenseLayer '{self.name}' belum dimuat.")

          if inputs.shape[-1] != self.kernel.shape[0]: raise ValueError(f"Dimensi input Dense ({inputs.shape[-1]}) di '{self.name}' tidak cocok dengan dimensi input kernel ({self.kernel.shape[0]}).")

        if inputs.ndim == 3:
            output = np.einsum('btf,fu->btu', inputs, self.kernel) + self.bias
        elif inputs.ndim == 2:
            output = np.dot(inputs, self.kernel) + self.bias
         else: raise ValueError(f"Input Dense '{self.name}' harus 2D atau 3D. Diterima shape: {inputs.shape}")

        return self.activation_fn(output) if self.activation_fn else output


class DropoutLayer:
    def __init__(self, rate, name=None):
        self.rate = rate
        self.name = name if name else f"dropout_{np.random.randint(1000)}"

    def set_weights(self, weights_data):
        pass
```

```
    def forward(self, inputs, training=False):
        if training:
            mask = np.random.binomial(1, 1 - self.rate, size=inputs.shape) / (1 - self.rate)
            return inputs * mask
        return inputs


class Model:
    def __init__(self, name=None):
        self.layers = []
        self.name = name if name else "MySequentialModel"
        self.auto_name_counts = {}

    def add(self, layer):
        user_provided_name = getattr(layer, 'name', None)
        final_name = user_provided_name

        if final_name is None:
            layer_class_name = layer.__class__.__name__
            base_name = ""

            if isinstance(layer, EmbeddingLayer):
                base_name = "embedding"
                        elif isinstance(layer, SimpleRNNLayer) and not isinstance(layer,
BidirectionalSimpleRNNLayer):
                base_name = "simple_rnn"
            elif isinstance(layer, BidirectionalSimpleRNNLayer):
                base_name = "bidirectional"
            elif isinstance(layer, DenseLayer):
                base_name = "dense"
            elif isinstance(layer, DropoutLayer):
                base_name = "dropout"
            else:
                base_name = layer_class_name.lower().replace("layer", "")

            current_count = self.auto_name_counts.get(base_name, 0)

            if current_count == 0:
                final_name = base_name
            else:
                final_name = f"{base_name}_{current_count}"

            layer.name = final_name
            self.auto_name_counts[base_name] = current_count + 1

        self.layers.append(layer)

    def load_weights(self, filepath):
        all_h5_weights = load_weights_from_hdf5(filepath)

        for layer in self.layers:
```

```
            if not hasattr(layer, 'set_weights'):
                        print(f"   Layer '{getattr(layer, 'name', layer.__class__.__name__)}'
({layer.__class__.__name__}) tidak memiliki metode 'set_weights', dilewati.")
                continue

            if isinstance(layer, DropoutLayer):
                if hasattr(layer, 'set_weights'): layer.set_weights(None)
                continue

            if not layer.name:
                        raise ValueError(f"Layer {layer} tidak memiliki atribut 'name' setelah
ditambahkan, tidak bisa memuat bobot.")

            weights_data_for_this_layer = all_h5_weights["layers"].get(layer.name)

            if weights_data_for_this_layer is None:
                raise ValueError(
                     f"Bobot untuk layer '{layer.name}' tidak ditemukan sebagai kunci top-level
"
                    f"di file HDF5 ('{filepath}'). "
                    f"Kunci top-level yang tersedia: {list(all_h5_weights.keys())}"
                )

            try:
                layer.set_weights(weights_data_for_this_layer)
            except Exception as e:
                print(f"ERROR saat memuat bobot untuk layer '{layer.name}': {e}")
                raise

    def forward(self, inputs):
        x = inputs
        total_layers = len(self.layers)

        for i, layer in enumerate(self.layers):
            x = layer.forward(x)

            progress_percentage = (i + 1) * 100 / total_layers
            bar_length = 40
            filled_length = int(bar_length * (i + 1) // total_layers)
            bar_display = '' * filled_length + '-' * (bar_length - filled_length)

                                    sys.stdout.write(f'\rProcessing   Layers:   |{bar_display}|
{progress_percentage:.2f}%')
            sys.stdout.flush()

        sys.stdout.write('\n')
        return x

    def summary(self):
        print(f"\n--- Ringkasan Model: '{self.name}' ---")
        total_params = 0
        print("_________________________________________________________________")
```

```
        print("Layer (type)                 Output Shape              Param #   ")
        print("=================================================================")
        for layer in self.layers:
                    layer_name_str = layer.name if hasattr(layer, 'name') and layer.name else
layer.__class__.__name__
            layer_type_str = layer.__class__.__name__
            output_shape_str = "(Variable)"

            params_count = 0
                  if hasattr(layer, 'kernel') and layer.kernel is not None: params_count +=
np.prod(layer.kernel.shape)
                    if hasattr(layer, 'bias') and layer.bias is not None: params_count +=
np.prod(layer.bias.shape)
                if hasattr(layer, 'recurrent_kernel') and layer.recurrent_kernel is not None:
params_count += np.prod(layer.recurrent_kernel.shape)
                if hasattr(layer, 'embedding_matrix') and layer.embedding_matrix is not None:
params_count += np.prod(layer.embedding_matrix.shape)

            if isinstance(layer, BidirectionalSimpleRNNLayer):
                params_count = 0
                   if hasattr(layer.forward_rnn, 'kernel') and layer.forward_rnn.kernel is not
None: params_count += np.prod(layer.forward_rnn.kernel.shape)
                  if hasattr(layer.forward_rnn, 'bias') and layer.forward_rnn.bias is not None:
params_count += np.prod(layer.forward_rnn.bias.shape)
                                     if hasattr(layer.forward_rnn, 'recurrent_kernel') and
layer.forward_rnn.recurrent_kernel is not None: params_count +=
np.prod(layer.forward_rnn.recurrent_kernel.shape)
                  if hasattr(layer.backward_rnn, 'kernel') and layer.backward_rnn.kernel is not
None: params_count += np.prod(layer.backward_rnn.kernel.shape)
                 if hasattr(layer.backward_rnn, 'bias') and layer.backward_rnn.bias is not None:
params_count += np.prod(layer.backward_rnn.bias.shape)
                                    if hasattr(layer.backward_rnn, 'recurrent_kernel') and
layer.backward_rnn.recurrent_kernel is not None: params_count +=
np.prod(layer.backward_rnn.recurrent_kernel.shape)

            total_params += params_count
            params_str = str(params_count) if params_count > 0 else "0"

            print(f"{layer_name_str[:28]:<29} {output_shape_str:<26} {params_str:<10}")
        print("=================================================================")
        print(f"Total params: {total_params}")
        print("_________________________________________________________________")
```

1. Mekanisme Pemuatan Bobot (Weight Loading)

Kemampuan untuk memuat bobot yang sudah ada adalah fitur krusial. Proses ini diimplementasikan dalam beberapa tahap:

1. Pembacaan File HDF5 (`load_weights_from_hdf5` dan `_load_weights_recursive`):

- Fungsi `load_weights_from_hdf5` bertindak sebagai titik masuk utama, menerima path ke file `.weights.h5`.
- Secara internal, fungsi ini memanggil `_load_weights_recursive` yang bertugas menelusuri (traverse) struktur hierarkis file HDF5. File HDF5 menyimpan data dalam grup (mirip folder) dan dataset (mirip file berisi array).
- `_load_weights_recursive` secara rekursif membaca setiap grup dan dataset. Jika menemukan dataset, datanya (array NumPy) akan dibaca dan disimpan. Jika menemukan grup, fungsi akan memanggil dirinya sendiri untuk memproses grup tersebut.
- Hasilnya adalah sebuah dictionary Python (`all_h5_weights`) yang strukturnya mencerminkan secara persis hierarki grup dan dataset dalam file HDF5. Kunci-kunci dalam dictionary ini adalah nama grup atau dataset, dan nilainya bisa berupa sub-dictionary (untuk grup) atau array NumPy (untuk dataset).

2. Orkestrasi oleh Kelas `Model` (`Model.load_weights(filepath)`):
   - Metode ini pada kelas `Model` mengorkestrasi keseluruhan proses pemuatan bobot ke dalam layer-layer yang telah didefinisikan dalam model.
   - Pertama, ia memanggil `load_weights_from_hdf5(filepath)` untuk mendapatkan dictionary `all_h5_weights`.
   - Kemudian, ia mengiterasi melalui setiap `layer` yang telah ditambahkan ke model (disimpan dalam `self.layers`).
   - Untuk setiap `layer`, ia menggunakan atribut `layer.name` (yang bisa di-set oleh pengguna atau di-generate otomatis saat `model.add()`) sebagai kunci untuk mencari data bobot yang relevan di dalam `all_h5_weights`. Diasumsikan bahwa nama layer dalam model akan cocok dengan nama grup top-level di file HDF5 (ini adalah konvensi yang mirip dengan Keras saat memuat bobot `by_name`).
   - Jika dictionary bobot untuk layer tersebut ditemukan (`weights_data_for_this_layer`), dictionary ini akan diteruskan ke metode `set_weights()` milik layer yang bersangkutan.
3. Pengaturan Bobot pada Layer Individual (`Layer.set_weights(weights_data)`):
   - Setiap kelas layer (misalnya, `EmbeddingLayer`, `SimpleRNNLayer`, `DenseLayer`) memiliki metode `set_weights(self, weights_data)` sendiri.
   - Metode ini bertanggung jawab untuk mem-parsing `weights_data` (dictionary bobot yang spesifik untuk layer tersebut) dan mengekstrak array NumPy yang sesuai untuk kernel, bias, matriks embedding, dll.
   - Implementasi `set_weights` pada setiap layer dirancang untuk fleksibel terhadap berbagai format penyimpanan bobot yang mungkin ditemui di file HDF5. Misalnya, ia akan mencoba mencari nama bobot standar Keras (seperti `'kernel:0'`, `'bias:0'`) dan juga struktur yang lebih internal dari TensorFlow (seperti `{'vars': {'0': array_bobot}}` atau `{'cell': {'vars': {'0': array_bobot_kernel}}}`).

- Setelah array bobot diekstrak, dilakukan validasi untuk memastikan shape (bentuk) dari array tersebut sesuai dengan hyperparameter yang telah didefinisikan saat layer diinisialisasi (misalnya, `units` pada `DenseLayer` atau `vocab_size` dan `embedding_dim` pada `EmbeddingLayer`).
- Jika valid, array bobot tersebut akan ditetapkan ke atribut internal layer (misalnya, `self.kernel`, `self.bias`, `self.embedding_matrix`) dan flag `self._is_built` di-set menjadi `True`.

2. Implementasi Kelas Layer Individual

Setiap layer memiliki struktur dan logika internalnya sendiri:

1. Konstruktor (`__init__(...)`):
   - Bertugas utama untuk menyimpan hyperparameter yang spesifik untuk layer tersebut (misalnya, `units` untuk `DenseLayer`, `vocab_size` dan `embedding_dim` untuk `EmbeddingLayer`, `rate` untuk `DropoutLayer`) dan `name` layer.
   - Atribut yang akan menyimpan bobot aktual (seperti `self.kernel`, `self.bias`) diinisialisasi ke `None`. Bobot ini baru akan diisi oleh metode `set_weights()`.
   - Sebuah flag `self._is_built` diinisialisasi ke `False` untuk menandakan bahwa bobot belum dimuat.
2. Pengelolaan Fungsi Aktivasi (`_parse_activation`):
   - Beberapa layer seperti `DenseLayer` dan `SimpleRNNLayer` memiliki metode helper (misalnya, `_parse_activation`) untuk mengonversi nama fungsi aktivasi yang diberikan sebagai string (contoh: `"relu"`, `"sigmoid"`) menjadi objek fungsi Python yang dapat dipanggil (misalnya, fungsi `relu` atau `sigmoid` yang telah didefinisikan). Ini memberikan fleksibilitas bagi pengguna untuk menentukan aktivasi.
3. Metode `forward(inputs, ...)`:
   - Ini adalah inti dari setiap layer, di mana komputasi *forward propagation* dilakukan.
   - Sebelum melakukan komputasi, metode ini biasanya memeriksa flag `self._is_built` untuk memastikan bahwa bobot layer sudah dimuat. Jika belum, sebuah `RuntimeError` akan dimunculkan.
   - Semua operasi numerik dalam metode `forward` diimplementasikan menggunakan NumPy.
   - `EmbeddingLayer`: Metode `forward`-nya melakukan operasi *lookup*. Input berupa array indeks integer akan digunakan untuk mengambil vektor embedding yang sesuai dari `self.embedding_matrix`.
   - `SimpleRNNLayer`: Mengimplementasikan perhitungan rekuren standar: ht=aktivasi(Wihxt+Whhht−1+bh). Perhitungan ini dilakukan secara iteratif untuk setiap *timestep* dalam sekuens input. Metode ini juga menangani parameter `return_sequences` untuk menentukan apakah output dari setiap timestep atau hanya timestep terakhir yang dikembalikan.

- `BidirectionalSimpleRNNLayer`: Mengelola dua instance `SimpleRNNLayer` internal (satu untuk arah *forward* dan satu untuk arah *backward*). Untuk pass *backward*, sekuens input akan dibalik urutannya terlebih dahulu. Output dari kedua arah kemudian digabungkan sesuai dengan `merge_mode` yang ditentukan (misalnya, konkatenasi atau penjumlahan).
- `DenseLayer`: Melakukan operasi matriks standar: `output = aktivasi(dot(inputs, kernel) + bias)`. Dapat menangani input 2D (batch data tabular) maupun 3D (batch sekuens, di mana operasi dense diterapkan pada setiap timestep secara independen – sering disebut *time-distributed dense*).
- `DropoutLayer`: Selama inferensi (ketika `training=False`, yang merupakan default pada model ini), metode `forward`-nya hanya mengembalikan input tanpa perubahan (fungsi identitas). Logika dropout yang sebenarnya hanya relevan saat training.

3. Implementasi Kelas `Model` Sekuensial

Kelas `Model` menyediakan API yang mirip Keras Sequential untuk membangun dan menggunakan model:

1. Metode `add(layer)`:
   - Memungkinkan pengguna untuk menambahkan instance layer ke model secara berurutan. Layer-layer ini disimpan dalam list internal `self.layers`.
   - Implementasi ini juga mencakup logika penamaan otomatis. Jika pengguna menambahkan layer tanpa memberikan atribut `name` secara eksplisit saat membuat instance layer, metode `add()` akan secara otomatis menghasilkan nama unik untuk layer tersebut. Penamaan otomatis ini mengikuti pola seperti `"embedding"`, `"dense"`, `"dense_1"`, `"simple_rnn"`, `"simple_rnn_1"`, dan seterusnya, berdasarkan tipe layer dan jumlah instance dari tipe tersebut yang sudah ditambahkan. Penghitungan ini dilacak menggunakan dictionary `self.auto_name_counts`.
2. Metode `forward(inputs)`:
   - Mengimplementasikan alur *forward propagation* untuk keseluruhan model. Metode ini mengiterasi melalui `self.layers` secara berurutan.
   - Output dari satu layer menjadi input untuk layer berikutnya (`x = layer.forward(x)`).
   - Untuk memberikan feedback visual kepada pengguna selama proses yang mungkin lama, metode ini menampilkan progress bar berbasis teks di konsol. Progress bar ini diimplementasikan secara manual menggunakan `sys.stdout.write('\r')` untuk menimpa baris yang sama dan `sys.stdout.flush()` untuk memastikan pembaruan langsung terlihat.
3. Metode `summary()`:

- Menyediakan ringkasan arsitektur model yang telah dibangun, mirip dengan `model.summary()` di Keras.
- Ringkasan ini menampilkan daftar layer secara berurutan, beserta nama layer (yang telah ditetapkan oleh pengguna atau di-generate otomatis), tipe kelasnya, placeholder untuk bentuk outputnya ("(Variable)", karena bentuk output aktual bergantung pada bentuk input yang belum tentu diketahui saat `summary()` dipanggil), dan estimasi jumlah parameter untuk setiap layer.
- Jumlah parameter diestimasi dengan mengakses atribut bobot (seperti `self.kernel`, `self.bias`, `self.embedding_matrix`) dari setiap layer dan menghitung jumlah total elemen dalam array bobot tersebut menggunakan `np.prod(shape)`. Untuk `BidirectionalSimpleRNNLayer`, parameternya dihitung sebagai jumlah parameter dari dua sub-layer RNN internalnya.

4. Alur Data Keseluruhan (Contoh Kontekstual)

Dalam konteks aplikasi pemrosesan teks, alur data tipikal akan melibatkan langkah-langkah berikut (meskipun `TextVectorization` adalah komponen eksternal dari Keras, bukan bagian dari model *scratch* ini, penting untuk memahami bagaimana input disiapkan):

1. Input Teks Mentah: Data berupa kalimat atau dokumen.
2. Vektorisasi Teks (Eksternal): Menggunakan layer seperti `tensorflow.keras.layers.TextVectorization` untuk mengubah teks mentah menjadi sekuens integer. Setiap integer mewakili sebuah token (kata atau sub-kata) dalam vocabulary. Sekuens ini juga biasanya di-padding atau di-truncate agar memiliki panjang yang seragam.
3. Input ke Model `Model.forward()`: Sekuens integer hasil vektorisasi ini kemudian menjadi input untuk metode `forward()` dari instance `Model` kustom kita.
4. Proses Internal Model:
    - `EmbeddingLayer` mengubah sekuens integer menjadi sekuens vektor embedding.
    - Output dari `EmbeddingLayer` kemudian bisa dimasukkan ke `SimpleRNNLayer` atau `BidirectionalSimpleRNNLayer` untuk menangkap dependensi temporal atau kontekstual dalam sekuens.
    - Output dari layer rekuren (atau layer sebelumnya) kemudian dapat dimasukkan ke satu atau lebih `DenseLayer` untuk melakukan transformasi lebih lanjut atau untuk menghasilkan output klasifikasi akhir (misalnya, dengan aktivasi `softmax` atau `sigmoid`).
    - `DropoutLayer` bisa disisipkan di antara layer-layer lain untuk regularisasi (meskipun efeknya hanya saat training).

Berikut merupakan konfigurasi yang digunakan pada perbandingan model RNN from scratch:
- Embedding: (5000, 128)
- Bidirectional: SimpleRNN, 64 units

- Dropout: 0.2
- Dense: 128 unit, ReLU
- Dropout: 0.2
- Dense: 3 units, softmax

Didapatkan hasil berikut.

Model	F1-Score (Macro)	F1-Score Per Class [negative, neutral, positive]	Agreement Rate	Mismatch Data
Keras	0.64	[0.56, 0.58, 0.77]	100%	0
From scratch	0.64	[0.56, 0.58, 0.77]		

Dari tabel tersebut, terlihat bahwa model from scratch dan model keras menghasilkan nilai f1-score macro yang sama. Selain itu, nilai f1-score untuk masing-masing kelas yang dihasilkan juga sama. Lalu, prediksi yang dihasilkan oleh model from scratch sama dengan prediksi yang dihasilkan model keras. Dengan demikian, implementasi forward propagation RNN sudah memberikan performa yang sama persis dengan model keras.

2.1.2.3 Long Short-Term Memory (LSTM)

Berikut merupakan class dan method yang digunakan dalam LSTM from scratch:
Fungsi utilitas dan fungsi aktivasi

```
def sigmoid(x):
  s = 1/(1 + np.exp(-(x)))
  return s

def softmax(x):
  if len(x.shape) == 1:
    x = np.array([x])
  elif len(x.shape) != 2:
     raise ValueError(f"Input harus berupa array 1D atau 2D (sample, classes), didapat {len(x.shape)}D")

  exp_x = np.exp(x)
  sum_exp_x = np.sum(exp_x, axis=1)

  p = np.array([exp_x[i,:]/sum_exp_x[i] for i in range(len(x))])
  return p

def relu(x):
  return np.maximum(0, x)
```

```
def batch_array(x, batch_size):
  for i in range(0, len(x), batch_size):
    yield x[i:i+batch_size]
```

Class model sequential untuk menyimpan setiap layer dan load bobot dari file

```
import re
import keras
import h5py

# class lembut: (global)
class sequential:
  def __init__(self):
    self.seq = []
    self.layer_names = []
    self.weights = []

    def get_weights(self, fname):      # baca file .weights.h5, ambil bobot masing-masing layer
    weights = {}
    names = []

    with h5py.File(fname, "r") as f:
      layers = f["layers"]
      name = layers.keys()

      for n in name:
        names.append(n)
        w = layers[n]["vars"]

        if "dense" == re.sub(r'[^a-zA-Z\s]', '', n):
          w = layers[n]["vars"]
          weight_layer = [w["0"][:], w["1"][:]]

        elif "lstm" == re.sub(r'[^a-zA-Z\s]', '', n):
          w = layers[n]["cell"]["vars"]
          units = w["1"].shape[0]
          weight_layer = [w["0"][:], w["1"][:], w["2"][:]]

        elif "embedding" == re.sub(r'[^a-zA-Z\s]', '', n):
          w = layers[n]["vars"]
          weight_layer = w["0"][:]
```

```
        elif "dropout" == re.sub(r'[^a-zA-Z\s]', '', n):
          continue

        elif "bidirectional" == re.sub(r'[^a-zA-Z\s]', '', n):
          w = layers[n]["forward_layer"]["cell"]["vars"]
          units = w["1"].shape[0]
          weight_layer = [w["0"][:], w["1"][:], w["2"][:]]

          w = layers[n]["backward_layer"]["cell"]["vars"]
          units = w["1"].shape[0]
          weight_layer.append(w["0"][:])
          weight_layer.append(w["1"][:])
          weight_layer.append(w["2"][:])

        weights[n] = weight_layer

    sorted_seq = [lay.name for lay in self.seq]
    sorted_seq = sorted(sorted_seq)

    for lay1, lay2 in zip(names, sorted_seq):
      if lay1 != lay2:
            raise ValueError(f"Arsitektur tidak sama dengan file bobot. Layer
{lay1} tidak sama dengan {lay2}.")

    for key, val in weights.items():
      # masukkan weight ke masing-masing layer scratch
      for layer in self.seq:
        if key == layer.name:
          if "embedding" == re.sub(r'[^a-zA-Z\s]', '', key):
            layer.weights = [val]
            continue
          layer.set_weights(val)

    for lay in self.seq:
      if re.sub(r'[^a-zA-Z\s]', '', lay.name) != "dropout":
        self.weights = self.weights + lay.weights

    return self

  def add(self, layer):
    # apabila sudah ada model yang sama, tambahkan _count
```

```
    # add layer to seq
    count = 0
    for lay in self.seq:
      if layer.name == re.sub(r'[^a-zA-Z\s]', '', lay.name):
        count = count + 1

    if count >= 1:
      layer.name = layer.name + "_" + str(count)

    self.layer_names.append(layer.name)

    self.seq.append(layer)
    return self

  def predict(self, x, batch_size=32):
    # predict through each layer
    out = []
    batch_size = batch_size if len(x) >= batch_size else len(x)

    for batch in batch_array(x, batch_size=batch_size):
      x_batch = batch
      for lay in self.seq:
        x_batch = lay.forward(x_batch)
      out.append(x_batch)

    out = np.concatenate(out, axis=0)
    return out
```

Class LSTM

```
class lstm:
  def __init__(self, units, return_seq=False, return_state=False):
    self.units = units
    self.return_seq = return_seq
    self.return_state = return_state
    self.weights = None
    self.name = "lstm"

  def set_weights(self, x):
    self.weights = x
    self.w = self.weights[0]  # bobot input
    self.u = self.weights[1]  # bobot recurrent
    self.b = self.weights[2]  # bobot bias
```

```
    # bobot kernel (bobot input)
    self.w_i = self.w[:, :self.units]
    self.w_f = self.w[:, self.units: self.units * 2]
    self.w_c = self.w[:, self.units * 2: self.units * 3]
    self.w_o = self.w[:, self.units * 3:]

    # bobot recurrent kernel
    self.u_i = self.u[:, :self.units]
    self.u_f = self.u[:, self.units: self.units * 2]
    self.u_c = self.u[:, self.units * 2: self.units * 3]
    self.u_o = self.u[:, self.units * 3:]

    # bobot bias
    self.b_i = self.b[:self.units]
    self.b_f = self.b[self.units: self.units * 2]
    self.b_c = self.b[self.units * 2: self.units * 3]
    self.b_o = self.b[self.units * 3:]
    return self

def input(self, x, h):
    out = np.dot(x, self.w_i) + np.dot(h, self.u_i) + self.b_i
    i = sigmoid(out)
    return i

def forget(self, x, h):
    out = np.dot(x, self.w_f) + np.dot(h, self.u_f) + self.b_f
    f = sigmoid(out)
    return f

def c_tilde(self, x, h):
    out = np.dot(x, self.w_c) + np.dot(h, self.u_c) + self.b_c
    c_ti = np.tanh(out)
    return c_ti

def output(self, x, h):
    out = np.dot(x, self.w_o) + np.dot(h, self.u_o) + self.b_o
    o = sigmoid(out)
    return o

def c_state(self, f, c, i, c_tilde):
    c = f*c + i*c_tilde
```

```
    return c

  def h_state(self, o, c):
    h = o*np.tanh(c)
    return h

  def get_initial_state(self, batch_size):
    h = np.zeros((batch_size, self.units))
    c = np.zeros((batch_size, self.units))
    return h, c

  def forward(self, x):
    h, c = self.get_initial_state(batch_size=x.shape[0])

    c_t = np.zeros((x.shape[0], x.shape[1], self.units))
    h_t = np.zeros((x.shape[0], x.shape[1], self.units))

    seq_len = x.shape[1]

    for t in range(seq_len):
      i = self.input(x[:, t], h)
      f = self.forget(x[:, t], h)
      c_ti = self.c_tilde(x[:, t], h)
      o = self.output(x[:, t], h)

      c = self.c_state(f, c, i, c_ti)
      h = self.h_state(o, c)

      c_t[:, t, :] = c # if return_state, return c_t dan h_t
      h_t[:, t, :] = h  # if return_sequence, return h_t semua timestep

    out = h_t[:,-1,:]

    if self.return_seq == True and self.return_state == False:
      return h_t
    elif self.return_seq == True and self.return_state == True:
      return h_t, out, c_t[:,-1,:]
    elif self.return_seq == False and self.return_state == True:
      return out, out, c_t[:,-1,:]
    elif self.return_seq == False and self.return_state == False:
      return out
```

Class Dense

```
class dense:
  def __init__(self, units, activation="linear"):
    self.units = units
    self.activation = "linear" if activation is None else activation
    self.weights = None
    self.name = "dense"

  def set_weights(self, x):
    self.weights = x
    self.w = self.weights[0]
    self.b = self.weights[1]
    return self

  def forward(self, x):
    z = np.dot(x,self.w) + self.b.reshape((1,len(self.b)))

    if self.activation == "linear":
      out = z
    elif self.activation == "sigmoid":
      out = sigmoid(z)
    elif self.activation == "softmax":
      out = softmax(z)
    elif self.activation == "relu":
      out = relu(z)

    return out
```

Class Dropout

```
# class dropout:
class dropout:
  def __init__(self, rate, training=False):
    self.rate = rate
    self.training = training
    self.r = None
    self.name = "dropout"

  def forward(self, x):  # inverted dropout
    if self.training == True:
      q = 1 - self.rate
      self.r = np.random.binomial(1, self.rate, x.shape)
```

```
      out = (1/q)*self.r*x
      return out
    else:
      return x
```

Class Embedding

```
# class embedding:
class embedding:
  def __init__(self, input_dim, output_dim, weights=None):
    self.input_dim = input_dim  # max token
    self.output_dim = output_dim  # number of output features
    self.weights = [weights]  # (input_dim, output_dim)
    self.name = "embedding"

  def forward(self, x):
    # x: (batch, seq)
    x = np.array(x)

    if len(x.shape) == 1:
      x = np.array([x])
    elif len(x.shape) != 2:
          raise ValueError(f"x harus berupa array 1D atau 2D, didapat array
{len(x.shape)}D: {x.shape}")

    if x.max() >= self.input_dim:
       raise ValueError(f"Maksimum ID yang tersedia adalah {self.input_dim-1}.
ID maksimum yang ditemukan adalah {x.max()}")

    embed_out = np.array([[self.weights[0][x[i][j]] for j in range(len(x[0]))]
for i in range(len(x))])
    return embed_out
```

Class Bidirectional

```
class bidirectional:
  def __init__(self, layer, merge_mode="concat", backward_layer=None):
    self.layer = layer  # RNN, LSTM
    self.merge_mode = merge_mode
     self.backward_layer = copy.deepcopy(self.layer) if backward_layer is None
else backward_layer # RNN/LSTM
    self.name = "bidirectional"
```

```
    self.weights = None

  def set_weights(self, x):
    self.layer.set_weights(x[0:3])
    self.backward_layer.set_weights(x[3:6])
    self.weights = self.layer.weights + self.backward_layer.weights

  def forward(self, x):
    if self.layer.return_seq == True:
      fwd = self.layer.forward(x)
      bwd = self.backward_layer.forward(x[:,::-1,:])

      if self.merge_mode == "concat":
        out = np.concatenate((fwd, bwd[:,::-1]), axis=-1)
      elif self.merge_mode == "sum":
        out = np.sum([fwd, bwd[:,::-1]], axis=0)
      elif self.merge_mode == "ave":
        out = np.mean([fwd, bwd[:,::-1]], axis=0)

      return out

    else:  # return_sequence = False
      fwd = self.layer.forward(x)
      bwd = self.backward_layer.forward(x[:,::-1,:])

      if self.merge_mode == "concat":
        out = np.concatenate((fwd, bwd), axis=1)
      elif self.merge_mode == "sum":
        out = np.sum([fwd, bwd], axis=0)
      elif self.merge_mode == "ave":
        out = np.mean([fwd, bwd], axis=0)

      return out
```

Implementasi model neural network dari *scratch* ini bertujuan untuk mereplikasi fungsionalitas dasar dari model sekuensial seperti yang ditemukan pada library Keras, dengan fokus utama pada proses *forward propagation* dan kemampuan untuk memuat bobot (weights) yang sudah dilatih sebelumnya dari file berformat HDF5. Seluruh komputasi numerik dilakukan menggunakan library NumPy.

1. Mekanisme Pemuatan Bobot

- Definisikan model sequential ke dalam suatu variabel baru.

- Buat arsitektur model dengan menambahkan layer ke model sequential dengan method add. Model yang dibuat, harus memiliki arsitektur yang sesuai dengan model yang telah dibuat menggunakan Keras.
- Muat bobot dengan method get_weights, dengan argumen path atau nama file bobot model hasil training.
- Baca file bobot, lalu masukkan bobot setiap layer pada file tersebut ke dalam dictionary.
- Pastikan layernya memiliki arsitektur yang sama dengan model yang sudah dibangun dengan Keras.
- Masukkan masing-masing (dictionary) bobot yang telah didapat dari file ke masing-masing layer.

2. Implementasi Kelas Layer Indvidual

Umumnya, setiap layer yang diimplementasikan from scratch, memiliki struktur umum sebagai berikut

1. Konstruktor (__init__(. . .)): Bertugas untuk menyimpan hyperparameter yang spesifik untuk layer tersebut dan menyimpan name pada layer.
2. Set bobot (set_weights(. . .)): Bagian ini yang bertugas untuk load bobot dari nama file.
3. Forward(forward(. . .)):
   - embedding: Metode `forward`-nya melakukan operasi *lookup*. Input berupa array indeks integer akan digunakan untuk mengambil vektor embedding yang sesuai dari self.weights.
   - lstm: Mengimplementasikan perhitungan rekurens LSTM. Perhitungan ini dilakukan secara iteratif untuk setiap *timestep* dalam sekuens input. Metode ini menangani parameter return_seq untuk menentukan apakah output yang dikembalikan hanya *timestep* terakhir atau seluruh *timestep*.
   - bidirectional: Mengelola dua object class lstm, masing-masing untuk forward pass dan backward pass. Untuk backward pass, sekuens input akan dibalik terlebih dahulu. Output dari kedua object class lstm ini akan digabungkan sesuai dengan merge_mode.
   - dense: Melakukan operasi matriks neuron standar.
   - dropout: Selama inferensi, layer ini hanya mengembalikan seluruh output layer sebelumnya.

3. Implementasi Kelas Model Sekuensial

1. Metode add(layer): Memungkinkan pengguna untuk menambahkan instance layer ke model secara berurutan. Layer-layer ini disimpan dalam list self.seq.
2. Metode predict(inputs): Mengimplementasikan forward propagation untuk keseluruhan model. Metode ini mengiterasi seluruh layer dalam self.seq secara berurutan.

4. Alur Data Keseluruhan

1. Input Teks Mentah: Data berupa kalimat atau dokumen.
2. Vektorisasi Teks (Eksternal): Menggunakan layer seperti `tensorflow.keras.layers.TextVectorization` untuk mengubah teks mentah menjadi sekuens integer. Setiap integer mewakili sebuah token (kata atau sub-kata) dalam vocabulary. Sekuens ini juga biasanya di-padding atau di-truncate agar memiliki panjang yang seragam.

3. Input ke Model `Model.forward()`: Sekuens integer hasil vektorisasi ini kemudian menjadi input untuk metode `forward()` dari instance `Model` kustom kita.
4. Proses Internal Model:
    a. `embedding` mengubah sekuens integer menjadi sekuens vektor embedding.
    b. Output dari `embedding` kemudian bisa dimasukkan ke `lstm` atau `bidirectional` untuk menangkap dependensi temporal atau kontekstual dalam sekuens.
    c. Output dari layer rekuren (atau layer sebelumnya) kemudian dapat dimasukkan ke satu atau lebih `DenseLayer` untuk melakukan transformasi lebih lanjut atau untuk menghasilkan output klasifikasi akhir (misalnya, dengan aktivasi `softmax` atau `sigmoid`).
    d. `dropout` bisa disisipkan di antara layer-layer lain untuk regularisasi (meskipun efeknya hanya saat training).

Berikut merupakan konfigurasi yang digunakan pada perbandingan model LSTM from scratch:
- Batch size: 24
- Units: 128
- Susunan layer:
    - Embedding: (3000, 64)
    - Bidirectional: LSTM, 128 units
    - Bidirectional: LSTM, 89 units
    - LSTM: 89 units
    - Dropout: 0.4
    - Bidirectional: LSTM, 64 units
    - LSTM: 64 units
    - Dropout: 0.4
    - Dense: 3 units, softmax

Tabel 2.1.1.3.1 Daftar nilai metrik untuk masing-masing model Keras dan from scratch pada data testing.

Model	F1-Score (Macro)	F1-Score Per Class [negative, neutral, positive]	Agreement Rate	Mismatch Data
Keras	0.6603	[0.62, 0.6, 0.76]	100%	0
From scratch	0.6603	[0.62, 0.6, 0.76]		

Berdasarkan Tabel 2.1.1.3.1, terlihat bahwa kedua model menghasilkan performa yang sama pada saat inferensi. Selain itu, kedua model juga memiliki hasil yang sama, diperlihatkan oleh agreement rate sebesar 100%. Karena memiliki nilai yang sama dengan implementasi Keras, inferensi hasil model from scratch tidak memiliki hasil yang berbeda (mismatch) dengan yang diimplementasikan oleh Keras. Dengan demikian, hasil forward propagation from scratch yang diimplementasikan sudah berhasil memberikan hasil yang sama dengan Keras.

2.2 Hasil Pengujian

2.2.1 Convolutional Neural Network (CNN)

Analisis pengaruh hyperparameter dalam CNN dilakukan pada dataset CIFAR-10

2.2.1.1 Pengaruh jumlah layer konvolusi

Untuk menganalisis pengaruh banyaknya layer, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah input layer
- n buah Conv2D + ReLU (dengan n = jumlah layer konvolusi yang diuji)
- Setelah setiap 3 layer konvolusi: 1 buah MaxPooling2D (pool size 3x3, stride 2)
- GlobalAveragePooling2D
- Dense dengan 10 neuron + Softmax

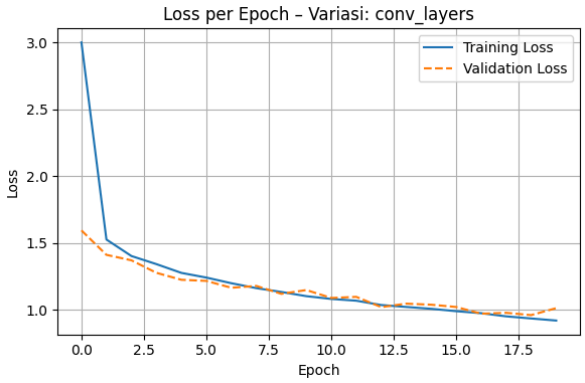
Detail Konfigurasi Layer:

- Conv2D
  - Filter per layer: sesuai konfigurasi (misalkan [96,192] untuk 2 layer, [96,192,192] untuk 3 layer, dst.)
  - Kernel size: (3x3)
  - Padding: same
  - Aktivasi: ReLU
- MaxPooling2D
  - Pool size: (3x3)
  - Stride: 2
- GlobalAveragePooling2D
  - Merata-ratakan fitur spasial di tiap channel
- Dense (output)
  - Jumlah neuron: 10
  - Aktivasi: Softmax

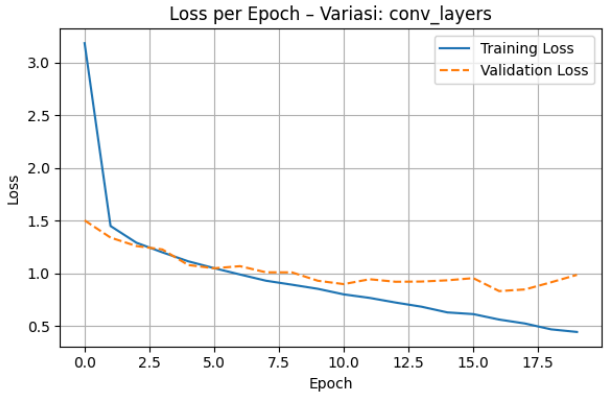
Fungsi Loss & Optimizer:

- Loss: sparse categorical crossentropy
- Optimizer: Adam

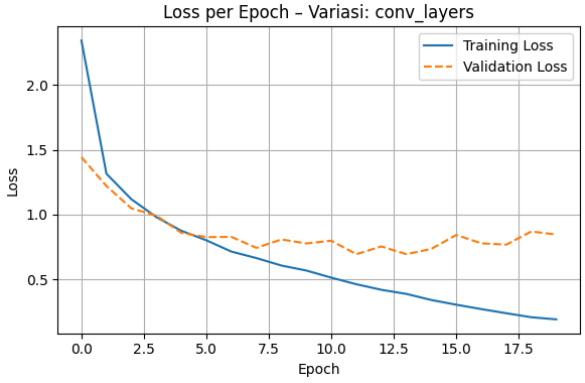
Didapatkan hasil sebagai berikut.



Gambar 2.2.1.1.1 Plot loss untuk model dengan 2 layer konvolusi



Gambar 2.2.1.1.2 Plot loss untuk model dengan 3 layer konvolusi



Gambar 2.2.1.1.3 Plot loss untuk model dengan 4 layer konvolusi

```
=== Eksperimen conv_layers -> c2_f96-192_k3_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step
conv_layers -> Macro-F1: 0.6665
```

```
=== Eksperimen conv_layers -> c3_f96-192-192_k3_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step
conv_layers -> Macro-F1: 0.7203
=== Eksperimen conv_layers -> c4_f96-192-192-192_k3_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 18ms/step
conv_layers -> Macro-F1: 0.7732
```

Gambar 2.2.1.1.1 dapat terlihat dengan hanya dua layer konvolusi, model ini menurunkan training loss cukup cepat, dari sekitar 3.0 di epoch awal menjadi kurang lebih 0.9 di epoch akhir. Validation loss juga menurun di awal dari kurang lebih 1.6 ke kurang lebih 1.1, tetapi mulai sedikit meningkat setelah epoch tiga per empat akhir. Selain itu, macro F1 pada test set, model 2-layer mencapai 0.665. Ini menunjukkan dua layer sudah cukup menangkap pola dasar, tetapi kapasitasnya terbatas untuk detail lebih kompleks.

Gambar 2.2.1.1.2 dapat terlihat dengan tiga layer konvolusi, model ini menurunkan training loss menjadi lebih cepat dan mencapai kurang lebih 0.45 di epoch akhir, menandakan kemampuan representasi yang lebih baik dibandingkan dua layer. Validation loss juga menunjukkan penurunan stabil hingga epoch setengah awal, kemudian sedikit fluktuasi di kisaran 0.8-0.9, tetapi tidak mengalami lonjakan yang tajam.Selain itu, macro F1 pada test set, model 3-layer mencapai 0.7203. Ini menunjukkan tiga layer memberikan keseimbangan yang sangat baik antara kapasitas fitur dan regularisasi (global average pooling).

Gambar 2.2.1.1.3 dapat terlihat dengan empat layer konvolusi, model ini menurunkan training loss paling agresif yang akhirnya mendekati 0.2. Validation loss mengalami fluktuasi sedikit lebih besar di beberapa epoch, umumnya konsisten di bawah 1.0, menandakan bahwa model masih mampu menggeneralisasi dengan baik. Selain itu, macro F1 pada test set, model 4-layer mencapai 0.7732, tercapai oleh konfigurasi terdalam ini menandakan bahwa dalam eksperimen banyak layer ini, menambah kedalaman hingga empat layer konvolusi terus meningkatkan akurasi klasifikasi tanpa menimbulkan overfitting serius.

Penambahan layer konvolusi dari 2 ke 4 secara konsisten memperbaiki performa test set eksperimen ini. Model 2-layer sudah cukup mempelajari fitur dasar, tetapi dengan 3 layer performa meningkat tajam. Menambah layer ke-4 masih memberikan manfaat yang cukup signifikan, dengan training loss yang sangat rendah namun validation loss tetap terkendali. Ini menunjukkan bahwa, untuk dataset CIFAR-10 dan arsitektur GlobalAvgPool yang digunakan, menambah kedalaman hingga empat lapis konvolusi masih berada di bawah ambang overfitting yang merugikan, dan sebaiknya dipilih ketika sumber daya komputasi memungkinkan.

2.2.1.2 Pengaruh banyak filter per layer konvolusi

Untuk menganalisis pengaruh banyaknya filter per layer konvolusi, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah input layer
- 3 buah Conv2D + ReLU
- 3 buah MaxPooling2D (pool size 3x3, stride 2)

- GlobalAveragePooling2D
- Dense dengan 10 neuron + Softmax

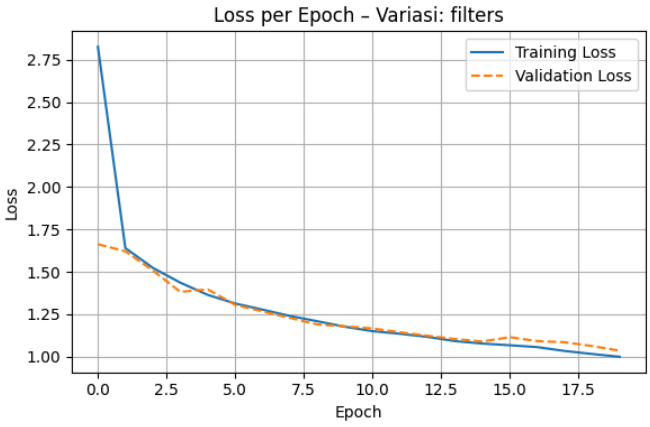
Detail Konfigurasi Layer:

- Conv2D
  - ❖ Filter per layer: sesuai konfigurasi ([32,32,32], [64,64,64], dan [128,128,128])
  - ❖ Kernel size: (3x3)
  - ❖ Padding: same
  - ❖ Aktivasi: ReLU
- MaxPooling2D
  - ❖ Pool size: (3x3)
  - ❖ Stride: 2
- GlobalAveragePooling2D
  - ❖ Merata-ratakan fitur spasial di tiap channel
- Dense (output)
  - ❖ Jumlah neuron: 10
  - ❖ Aktivasi: Softmax

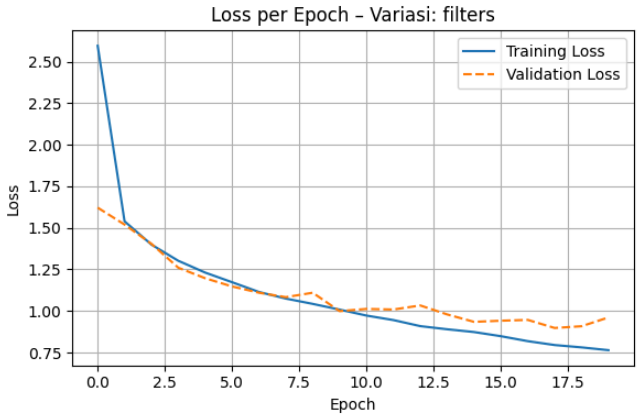
Fungsi Loss & Optimizer:

- Loss: sparse categorical crossentropy
- Optimizer: Adam

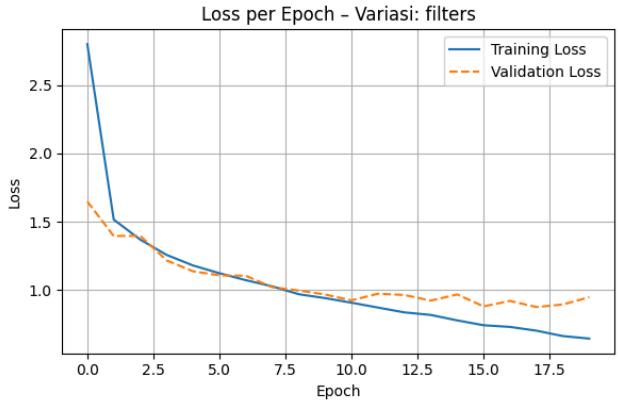
Didapatkan hasil sebagai berikut.



Gambar 2.2.1.2.1 Plot loss untuk model dengan banyak filter per layer [32,32,32]



Gambar 2.2.1.2.2 Plot loss untuk model dengan banyak filter per layer [64,64,64]



Gambar 2.2.1.2.3 Plot loss untuk model dengan banyak filter per layer [128,128,128]

```
=== Eksperimen filters -> c3_f32-32-32_k3_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
filters -> Macro-F1: 0.6283
=== Eksperimen filters -> c3_f64-64-64_k3_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step
filters -> Macro-F1: 0.6847
=== Eksperimen filters -> c3_f128-128-128_k3_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step
filters -> Macro-F1: 0.6964
```

Gambar 2.2.1.2.1 dengan 32 filter, training loss menurun secara moderat dari kurang lebih 2.8 ke kurang lebih 1.0 pada epoch terakhir. Validation loss turun meniru garis training loss, tetapi sedikit meningkat di akhir. Selain itu, macro F1 yang didapat ialah 0.6283.

Gambar 2.2.1.2.2 dengan 64 filter, training loss menurun lebih cepat menjadi kurang lebih 0.8 pada epoch terakhir. Validation loss turun lebih stabil sampai kurang lebih 0.9 sebelum fluktuasi kecil di epoch akhir. Selain itu, macro F1 yang didapat ialah 0.6847.

Gambar 2.2.1.2.3 dengan 128 filter, training loss menurun paling agresif kurang lebih sampai 0.65 di akhir. Validation loss menahan lebih rendah kurang lebih 0.85-0.95 tetapi menunjukkan fluktuasi sedikit lebih besar dibandingkan 64 filter. Selain itu, macro F1 yang didapat ialah 0.6964.
Menambah jumlah filter per layer meningkatkan kapasitas representasi sehingga treining loss menurun lebih cepat dan macro F1 pada test set naik dari 0.628 ke 0.6847 ke 0.6964. Namun, peningkatan F1 dari 64 ke 128 filter lebih kecil, sedangkan overfitting mulai lebih terlihat. Dari segi efisiensi, konfigurasi 64 filter memberikan hampir performa puncak dengan overhead komputasi lebih rendah dibandingkan 128 filter.

2.2.1.3 Pengaruh ukuran filter per layer konvolusi

Untuk menganalisis pengaruh ukuran filter per layer konvolusi, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah input layer
- 3 buah Conv2D + ReLU
- Setelah setiap 3 layer konvolusi: 1 buah MaxPooling2D
- GlobalAveragePooling2D
- Dense dengan 10 neuron + Softmax

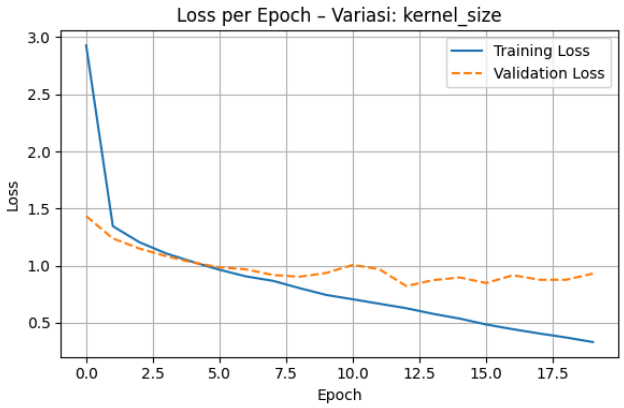
Detail Konfigurasi Layer:

- Conv2D
  - Filter per layer: [96,192,192]
  - Kernel size: variasi (3x3), (5x5), (7x7)
  - Padding: same
  - Aktivasi: ReLU
- MaxPooling2D
  - Pool size: (3x3)
  - Stride: 2
- GlobalAveragePooling2D
  - Merata-ratakan fitur spasial di tiap channel
- Dense (output)
  - Jumlah neuron: 10
  - Aktivasi: Softmax

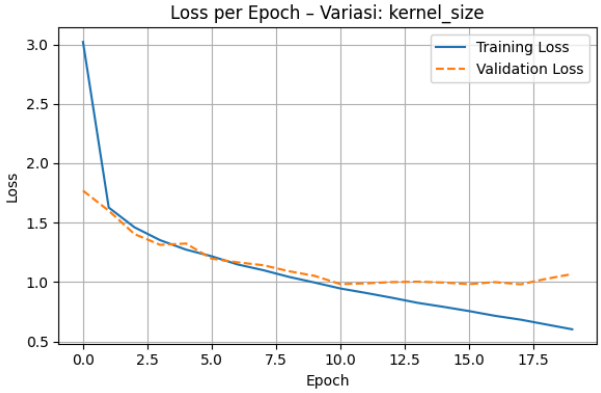
Fungsi Loss & Optimizer:

- Loss: sparse categorical crossentropy
- Optimizer: Adam

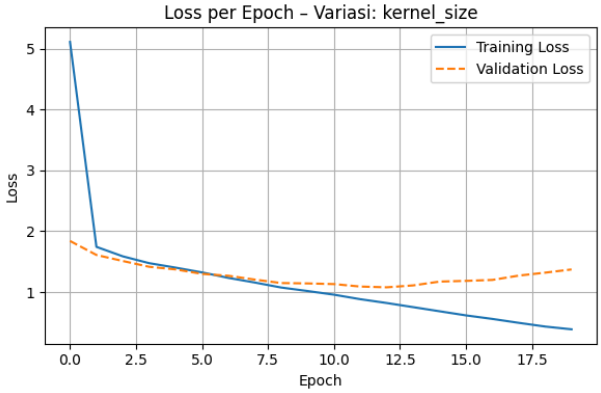
Didapatkan hasil sebagai berikut.



Gambar 2.2.1.3.1 Plot loss untuk model dengan ukuran filter per layer 3x3



Gambar 2.2.1.3.2 Plot loss untuk model dengan ukuran filter per layer 5x5



Gambar 2.2.1.3.3 Plot loss untuk model dengan ukuran filter per layer 7x7

```
=== Eksperimen kernel_size -> c3_f96-192-192_k3_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step
kernel_size -> Macro-F1: 0.7239
=== Eksperimen kernel_size -> c3_f96-192-192_k5_max ===
```

```
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 26ms/step
kernel_size -> Macro-F1: 0.6655
=== Eksperimen kernel_size -> c3_f96-192-192_k7_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 46ms/step
kernel_size -> Macro-F1: 0.6359
```

Gambar 2.2.1.3.1 dengan filter 3x3, training loss menurun cepat dari kurang lebih 2.9 ke kurang lebih 0.35 pada epoch terakhir menandakan konvergensi stabil. Validation loss turun konsisten hingga kurang lebih 0.9 lalu relatif stabil tanpa lonjakan besar, mengindikasikan generalisasi yang baik. Selain itu, macro F1 yang didapat ialah 0.7239.

Gambar 2.2.1.3.2 dengan filter 5x5, training loss menurun lebih lambat dibandingkan filter 3x3, dari kurang lebih 3.1 ke kurang lebih 0.6. Validation loss cenderung mirip yaitu turun ke kurang lebih 1.0, tetapi mengalami sedikit fluktuasi naik turun sekitar setengah epoch akhir. Selain itu, macro F1 yang didapat ialah 0.6655.

Gambar 2.2.1.3.3 dengan filter 7x7, training loss menurun lebih lambat dan berakhir menuju kurang lebih 0.35. Validation loss hanya turun ke kurang lebih 1.0, lalu naik lagi menjadi kurang lebih 1.3, menunjukkan overfitting dan underfitting bersamaan karena banyak fitur spasial yang tidak terhitung. Selain itu, macro F1 yang didapat ialah 0.6359.

Ukuran filter 3x3 memberikan receptive field yang cukup untuk mendeteksi pola tepi, sudut, dan tekstur lokal pada citra 32x32 piksel CIFAR-10. Dengan filter yang lebih besar (5x5 atau 7x7), jumlah parameter per filter meningkat drastis, sehingga model memerlukan lebih banyak data untuk menghindari overfitting pada fitur spasial. Filter 5x5 menunjukkan konvergensi lebih lambat dan sedikit fluktuasi validation loss, sedangkan kernel 7x7 paling rentan terhadap overfit dan kesulitan belajar pola halus. Oleh karena itu, kernel size 3x3 adalah pilihan optimal yang memberikan keseimbangan paling baik antara kapasitas representasi lokal dan kemampuan generalisasi.

2.2.1.4 Pengaruh jenis pooling layer

Untuk menganalisis pengaruh jenis pooling layer, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah input layer
- 3 buah Conv2D + ReLU
- Setelah setiap 3 layer konvolusi: 1 buah MaxPooling2D (pool size 3x3, stride 2)
- GlobalAveragePooling2D
- Dense dengan 10 neuron + Softmax

Detail Konfigurasi Layer:

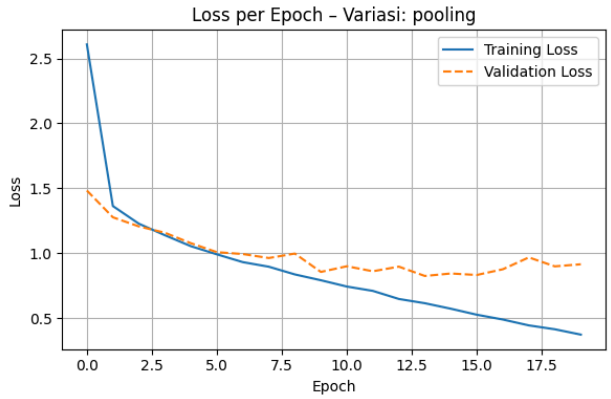
- Conv2D
  - Filter per layer: [96,192,192]
  - Kernel size: (3x3)
  - Padding: same
  - Aktivasi: ReLU

- MaxPooling2D atau AveragePooling2D
  - ❖ Pool size: (3x3)
  - ❖ Stride: 2
- GlobalAveragePooling2D
  - ❖ Merata-ratakan fitur spasial di tiap channel
- Dense (output)
  - ❖ Jumlah neuron: 10
  - ❖ Aktivasi: Softmax

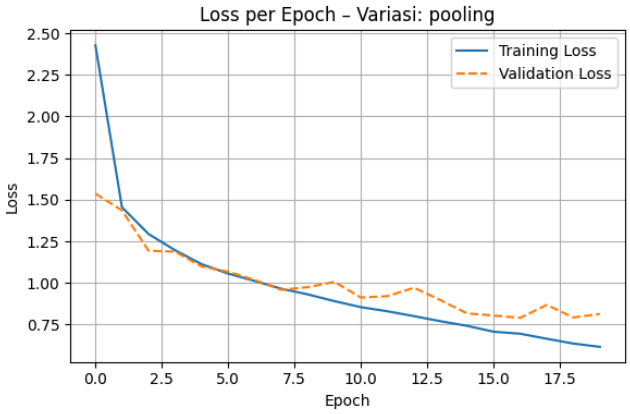
Fungsi Loss & Optimizer:

- Loss: sparse categorical crossentropy
- Optimizer: Adam

Didapatkan hasil sebagai berikut.



Gambar 2.2.1.4.1 Plot loss untuk model dengan max pooling



Gambar 2.2.1.4.2 Plot loss untuk model dengan average pooling

```
=== Eksperimen pooling -> c3_f96-192-192_k3_max ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step
pooling -> Macro-F1: 0.7093
=== Eksperimen pooling -> c3_f96-192-192_k3_avg ===
79/79 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step
pooling -> Macro-F1: 0.7171
```

Gambar 2.2.1.4.1 dengan max pooling, training loss yang didapat cukup bagus karena kurang lebih 0.4. Validation loss yang didapat pun sangat baik kurang dari 1.0. Selain itu, macro F1 yang didapat ialah 0.7093.

Gambar 2.2.1.4.2 dengan average pooling, training loss yang didapat tidak lebih baik yaitu kurang lebih 0.7. Validation loss yang didapat lebih baik yaitu sekitar 0.7. Selain itu, macro F1 yang didapat ialah 0.7171.

Untuk arsitektur dan dataset CIFAR-10, AveragePooling2D memberikan generalisasi yang sedikit lebih baik tetapi tidak signifikan perbedaannya.

2.2.2 Simple Recurrent Neural Network (RNN)

Analisis pengaruh beberapa hyperparameter dalam RNN dilakukan pada dataset NusaX-Sentiment (Bahasa Indonesia)

2.2.2.1 Pengaruh jumlah layer RNN

Untuk menganalisis pengaruh banyaknya layer, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah **Layer Embedding**
- n buah **Layer Bidirectional RNN** (nilai n disesuaikan dengan pengujian)
- 2 buah **Layer Dense**
- 4 buah **Layer Dropout**

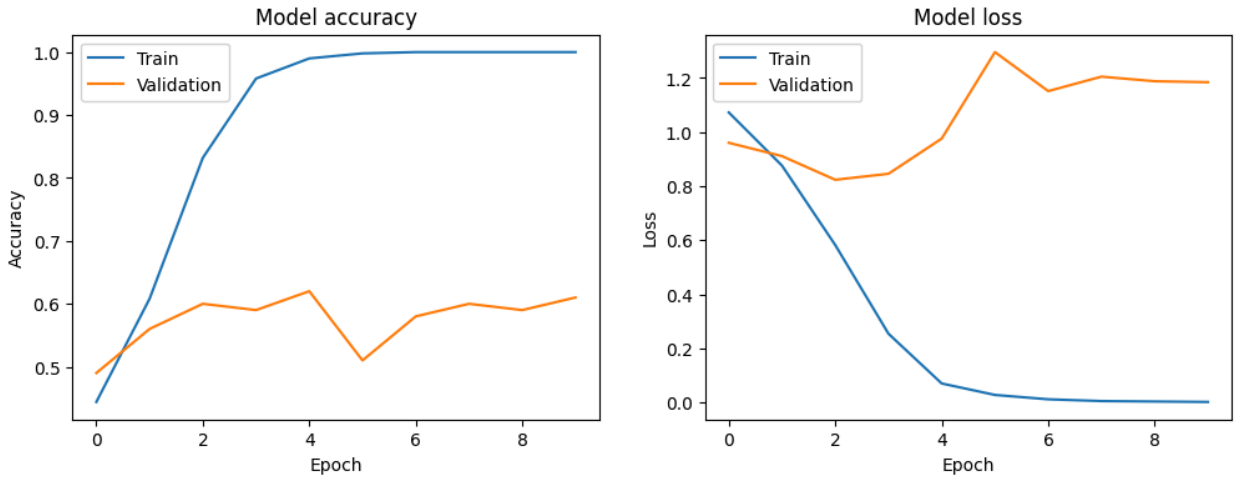
Detail Konfigurasi Layer:

1. **Layer Embedding:**
    - input_dim: 5000
    - output_dim: 128
2. **Layer Bidirectional RNN:**
    - Jumlah neuron: 64
3. **Layer Dense Pertama:**
    - Jumlah neuron: 128
    - Fungsi aktivasi: ReLU
4. **Layer Dense Kedua (Output Layer):**
    - Jumlah neuron: 3
    - Fungsi aktivasi: Softmax

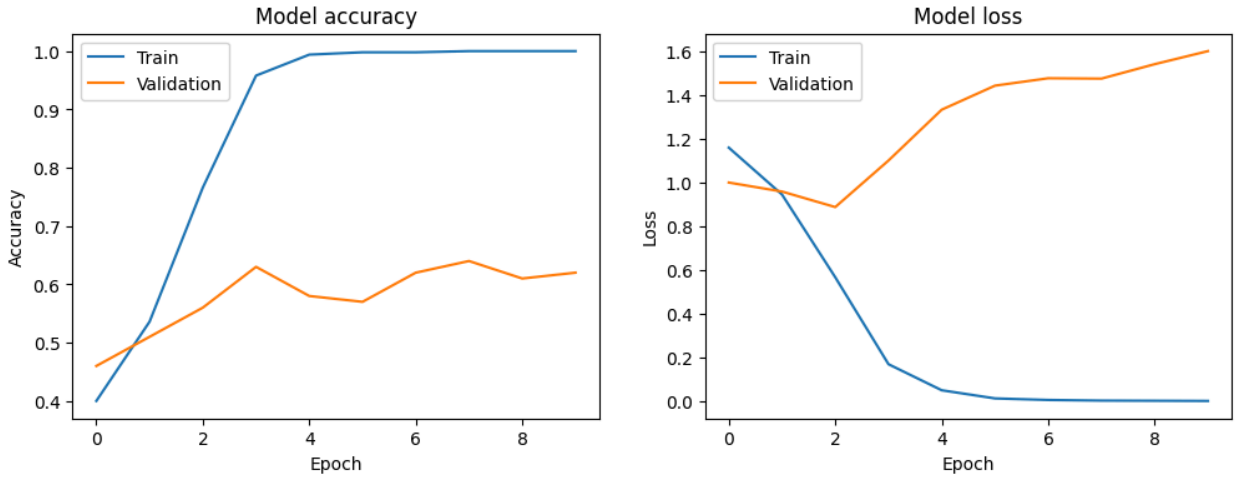
Fungsi Loss:

- Fungsi loss yang akan digunakan adalah sparse categorical crossentropy.

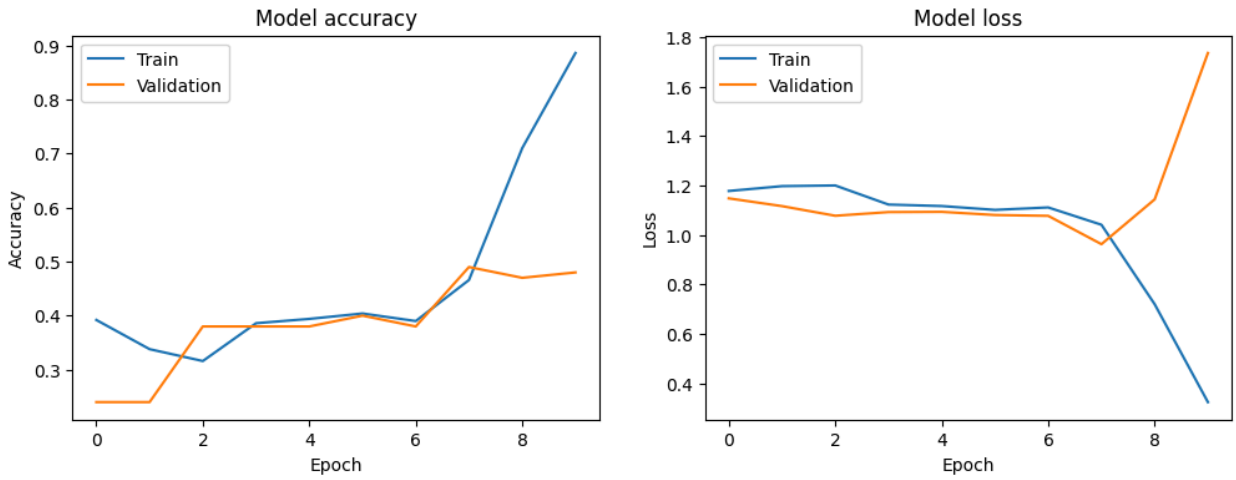
Didapatkan hasil sebagai berikut.



Gambar 2.2.2.1.1 Plot loss dan accuracy untuk model dengan 1 layer RNN



Gambar 2.2.2.1.2 Plot loss dan accuracy untuk model dengan 3 layer RNN



Gambar 2.2.2.1.3 Plot loss dan accuracy untuk model dengan 5 layer RNN

Analisis plot akurasi dan loss untuk model dengan jumlah layer RNN yang berbeda menunjukkan pengaruh signifikan terhadap performa model. Model dengan 1 layer RNN menunjukkan kemampuan belajar yang cepat pada data training dengan akurasi training yang mendekati 100% dan loss mendekati nol setelah hanya beberapa epoch. Namun, akurasi validasinya stagnan di sekitar 60% dan loss validasi mulai meningkat setelah epoch ke-3 atau ke-4. Hal tersebut mengindikasikan terjadinya overfitting yang jelas dimana model gagal menggeneralisasi pengetahuannya ke data yang belum pernah dilihat.

Ketika jumlah layer RNN ditingkatkan menjadi 3, model tetap menunjukkan kemampuan menghafal data training dengan sangat baik, mirip dengan model 1 layer. Akan tetapi, masalah overfitting tampak lebih parah. Meskipun akurasi validasi juga berada di kisaran 60%, loss validasi pada model 3 layer meningkat lebih tajam dan mencapai nilai yang lebih tinggi dibandingkan model 1 layer pada akhir epoch. Hal ini menyiratkan bahwa penambahan layer dalam kasus ini tidak meningkatkan kemampuan generalisasi, melainkan memperburuk kecenderungan model untuk terlalu spesifik pada data training.

Selanjutnya, model dengan 5 layer RNN menunjukkan performa yang paling buruk di antara ketiganya. Akurasi training meningkat lebih lambat dan tidak mencapai level setinggi model dengan layer lebih sedikit, sementara loss training juga tidak turun serendah model lainnya. Lebih lanjut, akurasi validasi sangat rendah dan sangat fluktuatif, seringkali di bawah 50%, dengan loss validasi yang tinggi dan tidak stabil. Ini mengindikasikan bahwa model dengan 5 layer RNN terlalu kompleks untuk dataset atau jumlah epoch yang diberikan, sehingga mengalami kesulitan dalam proses training, menunjukkan tanda-tanda underfitting pada data training relatif terhadap model yang lebih simpel, sekaligus overfitting yang parah dan ketidakstabilan pada data validasi.

Secara keseluruhan, untuk dataset dan konfigurasi yang diuji, peningkatan jumlah layer RNN tidak selalu menghasilkan model yang lebih baik. Model dengan 1 layer sudah mengalami overfitting. Penambahan menjadi 3 layer cenderung memperburuk overfitting tersebut tanpa adanya peningkatan performa generalisasi. Peningkatan lebih lanjut menjadi 5 layer justru merusak kemampuan model untuk belajar secara efektif dan melakukan generalisasi, menunjukkan bahwa kompleksitas model yang berlebihan dapat menjadi kontraproduktif.

Kemudian, berikut merupakan *Classification report* untuk masing-masing model.

	precision	recall	f1-score	support
negative	0.59	0.52	0.56	153
neutral	0.56	0.60	0.58	96
positive	0.74	0.79	0.77	151
accuracy			0.65	400
macro avg	0.63	0.64	0.64	400
weighted avg	0.64	0.65	0.64	400

Gambar 2.2.2.1.4 *Classification report* untuk model dengan 1 layer RNN

	precision	recall	f1-score	support
negative	0.46	0.48	0.47	153
neutral	0.39	0.39	0.39	96
positive	0.65	0.64	0.64	151
accuracy			0.52	400
macro avg	0.50	0.50	0.50	400
weighted avg	0.52	0.52	0.52	400

Gambar 2.2.2.1.5 *Classification report* untuk model dengan 3 layer RNN

	precision	recall	f1-score	support
negative	0.37	0.18	0.25	153
neutral	0.39	0.81	0.53	96
positive	0.69	0.58	0.63	151
accuracy			0.48	400
macro avg	0.49	0.52	0.47	400
weighted avg	0.50	0.48	0.46	400

Gambar 2.2.2.1.6 *Classification report* untuk model dengan 5 layer RNN

Model dengan 1 layer RNN menunjukkan performa terbaik di antara ketiga konfigurasi. Model ini berhasil mencapai akurasi keseluruhan sebesar 0.65, dengan *macro average F1-score* 0.64 dan *weighted*

average F1-score yang juga 0.64. Performa per kelas juga paling unggul, dengan F1-score 0.77 untuk kelas 'positive', 0.58 untuk 'neutral', dan 0.56 untuk 'negative'. Hal ini mengindikasikan bahwa model yang lebih sederhana ini paling efektif dalam melakukan klasifikasi pada dataset yang diuji.

Ketika jumlah layer ditingkatkan menjadi 3 layer RNN terjadi penurunan performa yang cukup signifikan dibandingkan model 1 layer. Akurasi model turun menjadi 0.52, dengan *macro average F1-score* 0.50 dan *weighted average F1-score* 0.52. Dilihat dari F1-score per kelas, kelas 'positive' mencapai 0.64, kelas 'negative' 0.47, dan kelas 'neutral' 0.39. Meskipun masih memberikan hasil yang lumayan, penurunan ini jelas menunjukkan dampak negatif dari penambahan layer.

Penurunan performa menjadi semakin jelas pada model dengan 5 layer RNN. Model ini mencatatkan akurasi terendah, yaitu 0.48, dengan *macro average F1-score* 0.47 dan *weighted average F1-score* 0.46. F1-score untuk kelas 'negative' sangat rendah (0.25), sementara untuk kelas 'positive' adalah 0.63, dan 'neutral' 0.53. Kinerja ini adalah yang paling rendah di antara ketiga model. Hal tersebut mengindikasikan bahwa penambahan layer hingga lima justru semakin memperburuk kemampuan generalisasi model.

Secara keseluruhan, terdapat tren yang konsisten: peningkatan jumlah layer RNN dari satu, ke tiga, hingga lima layer secara progresif menurunkan kualitas klasifikasi model. Model dengan 1 layer RNN menunjukkan kinerja optimal, sementara penambahan layer selanjutnya tampaknya menyebabkan overfitting atau kesulitan optimasi yang berdampak negatif pada hasil akhir. Hal ini menggarisbawahi pentingnya menemukan kompleksitas model yang tepat karena model yang lebih kompleks tidak selalu menghasilkan performa yang lebih baik.

2.2.2.2 Pengaruh Banyak *cell* RNN Per Layer

Untuk menganalisis pengaruh banyaknya *cell* RNN per layer, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah **Layer Embedding**
- 1 buah **Layer Bidirectional RNN**
- 2 buah **Layer Dense**
- 4 buah **Layer Dropout**

Detail Konfigurasi Layer:

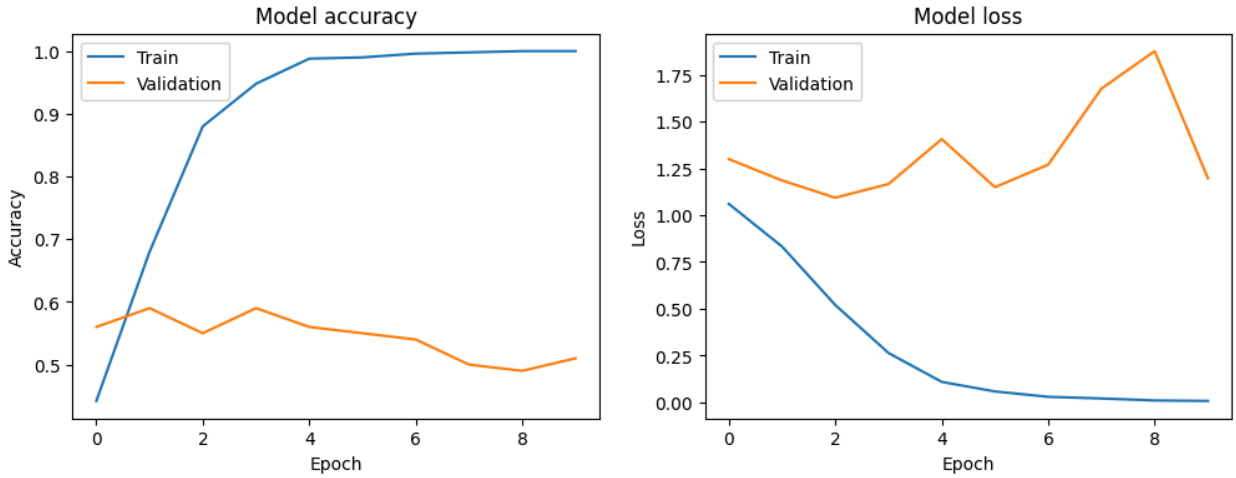
5. **Layer Embedding:**
    - input_dim: 5000
    - output_dim: 128
6. **Layer Bidirectional RNN:**
    - Jumlah neuron disesuaikan dengan pengujian
7. **Layer Dense Pertama:**
    - Jumlah neuron: 128
    - Fungsi aktivasi: ReLU

8. **Layer Dense Kedua (Output Layer):**
    - Jumlah neuron: 3
    - Fungsi aktivasi: Softmax

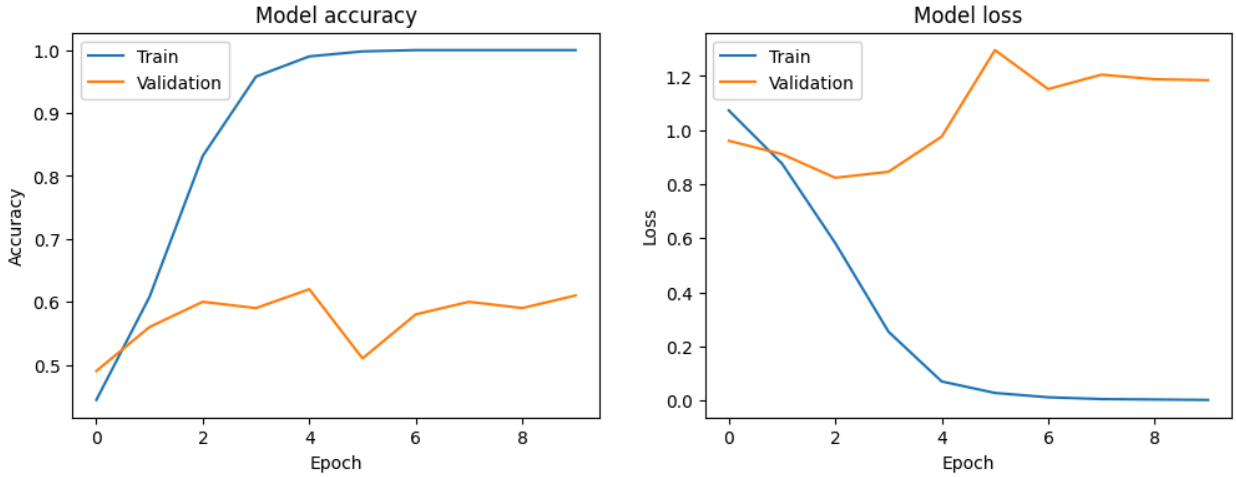
Fungsi Loss:

- Fungsi loss yang akan digunakan adalah sparse categorical crossentropy.

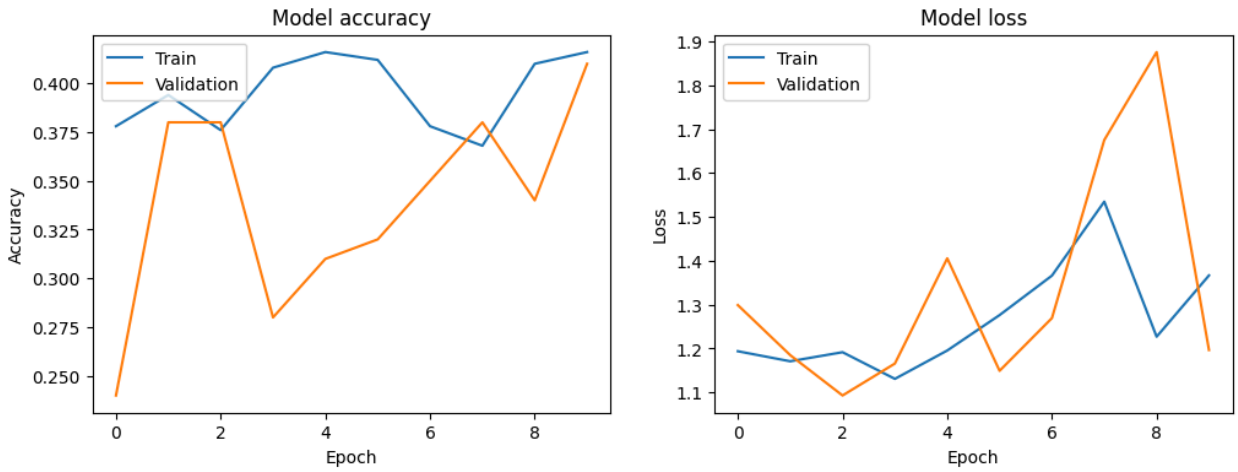
Didapatkan hasil sebagai berikut.



Gambar 2.2.2.2.1 Plot loss dan accuracy untuk model dengan 16 cell



Gambar 2.2.2.2.2 Plot loss dan accuracy untuk model dengan 64 *cell*



Gambar 2.2.2.2.3 Plot loss dan accuracy untuk model dengan 256 *cell*

Analisis plot akurasi dan loss untuk model RNN dengan jumlah *cell* yang berbeda menunjukkan dampak signifikan terhadap kemampuan model dalam mempelajari data training dan melakukan generalisasi pada data validasi.

Model dengan 16 *cell* menunjukkan kemampuan belajar yang baik pada data training di mana akurasi training meningkat pesat dan mencapai nilai mendekati 1.0 setelah sekitar 3 epoch dengan loss training yang juga turun mendekati nol. Namun, pada data validasi, akurasi hanya mencapai puncaknya di sekitar 0.6 dan kemudian cenderung stagnan atau sedikit menurun, sementara loss validasi menurun hingga epoch ke-4 lalu mulai meningkat kembali. Peningkatan loss validasi setelah titik tertentu ini menandakan adanya overfitting di mana model terlalu menghafal data training namun kurang baik dalam generalisasi.

Ketika jumlah *cell* ditingkatkan menjadi 64 *cell*, pola yang serupa dengan model 16 *cell* teramati. Akurasi training juga mencapai nilai yang sangat tinggi dengan cepat, dan loss training menurun drastis. Akurasi validasi juga mencapai sekitar 0.6, namun fluktuasinya tampak sedikit lebih besar, dan loss validasi juga menunjukkan tren peningkatan setelah mencapai titik minimum, mengindikasikan overfitting yang serupa atau bahkan sedikit lebih jelas dibandingkan model dengan 16 *cell*. Gap antara performa training dan validasi tetap signifikan.

Perubahan drastis terlihat pada model dengan 256 *cell*. Model ini menunjukkan kesulitan dalam proses training dan performa yang tidak stabil. Akurasi training hanya mencapai sekitar 0.4 dan sangat berfluktuasi, tidak pernah mencapai tingkat konvergensi yang baik seperti pada model dengan *cell* lebih sedikit. Akurasi validasi bahkan lebih rendah dan lebih tidak stabil berada di bawah akurasi training. Kurva loss, baik training maupun validasi, juga sangat tidak stabil dengan nilai yang tinggi dan fluktuasi besar. Loss validasi bahkan sempat melonjak sangat tinggi. Ini mengindikasikan bahwa model dengan 256 *cell* terlalu kompleks untuk dataset yang digunakan menyebabkan kesulitan optimasi, ketidakstabilan, dan performa yang buruk baik pada data training maupun validasi. Model ini gagal mempelajari pola yang berguna.

Secara keseluruhan, peningkatan jumlah *cell* dari 16 ke 64 tidak memberikan perbaikan signifikan pada performa validasi dan keduanya menunjukkan tanda-tanda overfitting. Namun, peningkatan jumlah *cell* secara drastis menjadi 256 justru merusak kemampuan model untuk belajar secara efektif, menghasilkan model yang tidak stabil dan performa yang jauh lebih buruk. Hal ini menunjukkan bahwa ada batas optimal untuk jumlah *cell*, dan melebihi batas tersebut dapat menyebabkan masalah training yang serius dan penurunan performa generalisasi.

Kemudian, berikut merupakan *classification report* untuk masing-masing model.

	precision	recall	f1-score	support
negative	0.52	0.54	0.53	153
neutral	0.48	0.60	0.54	96
positive	0.79	0.64	0.71	151
accuracy			0.59	400
macro avg	0.60	0.59	0.59	400
weighted avg	0.61	0.59	0.60	400

Gambar 2.2.2.2.4 *classification report* untuk model dengan 16 *cell*

	precision	recall	f1-score	support
negative	0.59	0.52	0.56	153
neutral	0.56	0.60	0.58	96
positive	0.74	0.79	0.77	151
accuracy			0.65	400
macro avg	0.63	0.64	0.64	400
weighted avg	0.64	0.65	0.64	400

Gambar 2.2.2.2.5 *classification report* untuk model dengan 64 *cell*

	precision	recall	f1-score	support
negative	0.00	0.00	0.00	153
neutral	0.30	0.35	0.32	96
positive	0.42	0.79	0.55	151
accuracy			0.39	400
macro avg	0.24	0.38	0.29	400
weighted avg	0.23	0.39	0.29	400

Gambar 2.2.2.2.6 *classification report* untuk model dengan 256 *cell*

Analisis *classification report* untuk model RNN dengan jumlah *cell* yang berbeda menunjukkan bahwa jumlah *cell* memiliki dampak yang signifikan dan tidak selalu linear terhadap performa klasifikasi.

Model dengan 16 *cell* mencapai akurasi keseluruhan sebesar 0.59 dengan *macro average F1-score* 0.59 dan *weighted average F1-score* 0.60. Dilihat dari performa per kelas, F1-score untuk kelas 'positive' adalah 0.71, kelas 'neutral' 0.54, dan kelas 'negative' 0.53. Ini menunjukkan performa awal yang cukup seimbang dengan kemampuan klasifikasi yang lebih baik untuk sentimen positif.

Ketika jumlah *cell* ditingkatkan menjadi 64 *cell*, terlihat peningkatan performa yang cukup baik. Akurasi model naik menjadi 0.65 dengan *macro average F1-score* 0.64 dan *weighted average F1-score* juga 0.64. Peningkatan ini juga terlihat pada F1-score masing-masing kelas: kelas 'positive' meningkat menjadi 0.77, kelas 'neutral' menjadi 0.58, dan kelas 'negative' menjadi 0.56. Ini mengindikasikan bahwa penambahan *cell* hingga 64 unit membantu model untuk menangkap pola data dengan lebih baik dan meningkatkan kemampuan generalisasinya.

Namun, peningkatan jumlah *cell* lebih lanjut menjadi 256 *cell* justru mengakibatkan penurunan performa yang sangat drastis. Akurasi model anjlok menjadi hanya 0.39 dengan *macro average F1-score* 0.29 dan *weighted average F1-score* juga 0.29. Yang paling mengkhawatirkan adalah performa pada kelas 'negative' di mana nilai *precision*, *recall*, dan *F1-score* semuanya 0.00, yang berarti model sama sekali gagal mengidentifikasi kelas ini. Kelas 'neutral' juga memiliki F1-score yang rendah (0.32), sementara kelas 'positive' turun menjadi 0.55. Penurunan drastis ini menunjukkan bahwa model dengan 256 *cell* menjadi terlalu kompleks dan mengalami kesulitan dalam optimasi atau overfitting yang cukup parah sehingga tidak mampu menghasilkan klasifikasi yang berguna.

Secara ringkas, berdasarkan *classification report* ini, peningkatan jumlah *cell* dari 16 ke 64 memberikan dampak positif terhadap performa klasifikasi. Akan tetapi, peningkatan lebih lanjut hingga 256 *cell* justru merusak performa model secara signifikan bahkan menyebabkan kegagalan total dalam mengklasifikasikan salah satu kelas. Hal ini menekankan pentingnya menemukan jumlah *cell* yang optimal karena terlalu sedikit atau terlalu banyak *cell* dapat berdampak negatif pada hasil akhir.

2.2.1.3 Pengaruh jenis layer RNN berdasarkan arah

Untuk menganalisis pengaruh jenis layer RNN berdasarkan arah, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah **Layer Embedding**
- 1 buah **Layer Bidirectional RNN atau Layer Unidirectional RNN**
- 2 buah **Layer Dense**
- 4 buah **Layer Dropout**

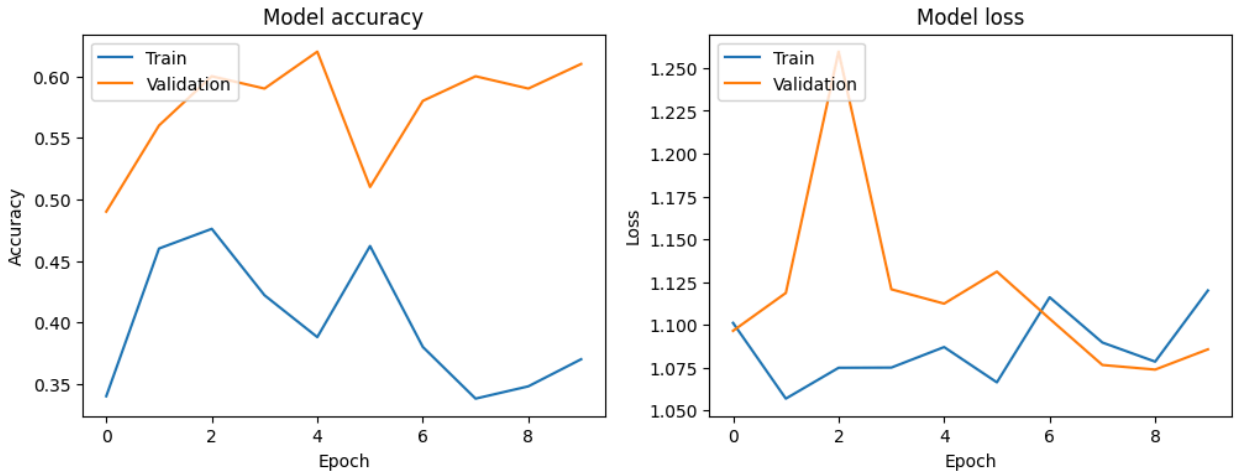
Detail Konfigurasi Layer:

9. **Layer Embedding:**
    - input_dim: 5000
    - output_dim: 128
10. **Layer Bidirectional RNN:**
    - Jumlah neuron disesuaikan dengan pengujian
11. **Layer Dense Pertama:**
    - Jumlah neuron: 128
    - Fungsi aktivasi: ReLU
12. **Layer Dense Kedua (Output Layer):**
    - Jumlah neuron: 3
    - Fungsi aktivasi: Softmax

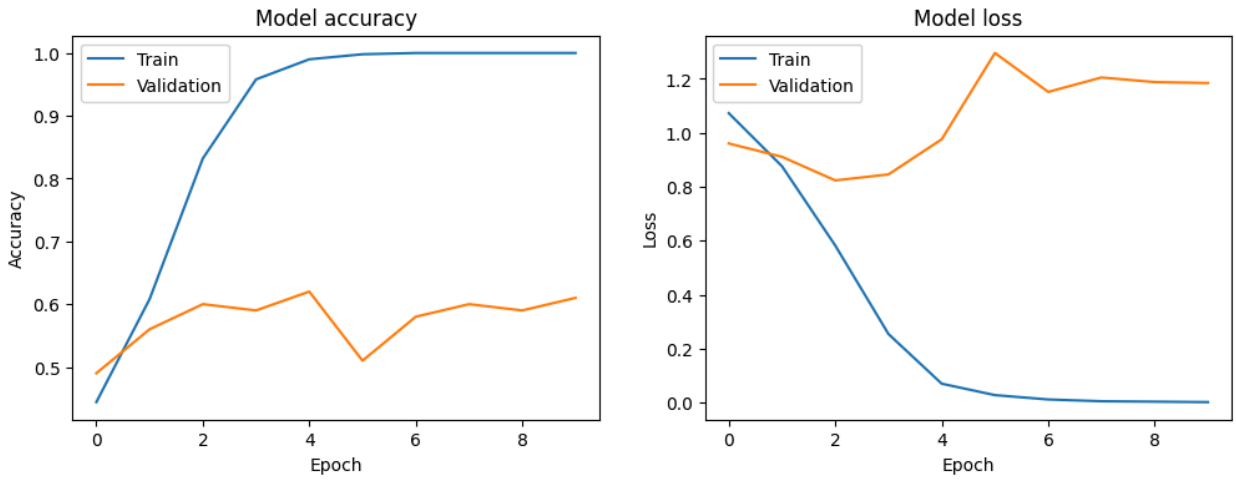
Fungsi Loss:

- Fungsi loss yang akan digunakan adalah sparse categorical crossentropy.

Didapatkan hasil sebagai berikut.



Gambar 2.2.2.3.1 Plot loss dan accuracy untuk model unidirectional



Gambar 2.2.2.3.2 Plot loss dan accuracy untuk model bidirectional

Model RNN unidirectional tampak kesulitan dalam mempelajari pola data secara efektif. Akurasi trainingnya berfluktuasi dan hanya mencapai puncaknya di bawah 0.5 sebelum akhirnya menurun. Akurasi validasi juga menunjukkan pola yang tidak stabil dengan puncaknya sekitar 0.6 namun dengan variabilitas yang tinggi dan tidak menunjukkan konvergensi yang baik. Kurva loss untuk model unidirectional juga tidak ideal; loss training berfluktuasi dan tidak turun secara konsisten ke nilai yang rendah, sementara loss validasi juga tinggi dan sangat tidak stabil bahkan sempat melonjak drastis. Ini mengindikasikan bahwa model unidirectional kesulitan untuk belajar dan cenderung tidak stabil serta tidak mampu melakukan generalisasi dengan baik pada data validasi.

Sebaliknya, model RNN bidirectional menunjukkan kemampuan belajar yang jauh lebih superior dan stabil. Akurasi training meningkat pesat dan konsisten, mencapai nilai mendekati 1.0 setelah sekitar 3-4 epoch dengan loss training yang juga turun drastis mendekati nol. Ini menandakan model bidirectional mampu mempelajari data training dengan sangat baik. Akurasi validasi mencapai puncaknya di sekitar 0.6 dan kemudian menunjukkan sedikit fluktuasi atau stagnasi, loss validasi pada awalnya menurun secara signifikan bersamaan dengan loss training sebelum mulai sedikit meningkat. Performa loss validasi yang awalnya menurun ini menunjukkan bahwa model bidirectional lebih mampu menangkap konteks dan dependensi dalam data karena memproses sekuens dari kedua arah (maju dan mundur). Walaupun overfitting tetap menjadi perhatian setelah beberapa epoch, kemampuan awalnya untuk belajar dan menggeneralisasi jauh lebih baik.

Secara keseluruhan, penggunaan arsitektur bidirectional pada RNN memberikan keuntungan signifikan dalam performa model dibandingkan arsitektur unidirectional untuk kasus ini. Model bidirectional menunjukkan kemampuan belajar yang lebih cepat, lebih stabil, dan mencapai akurasi training yang jauh lebih tinggi, serta menunjukkan tanda-tanda awal generalisasi yang lebih baik pada data validasi sebelum overfitting mulai mendominasi.

Kemudian, berikut merupakan *classification report* untuk masing-masing model.

	precision	recall	f1-score	support
negative	0.39	0.33	0.36	153
neutral	0.00	0.00	0.00	96
positive	0.38	0.68	0.48	151
accuracy			0.38	400
macro avg	0.26	0.34	0.28	400
weighted avg	0.29	0.38	0.32	400

Gambar 2.2.2.3.3 *Classification report* untuk model unidirectional

	precision	recall	f1-score	support
negative	0.59	0.52	0.56	153
neutral	0.56	0.60	0.58	96
positive	0.74	0.79	0.77	151
accuracy			0.65	400
macro avg	0.63	0.64	0.64	400
weighted avg	0.64	0.65	0.64	400

Gambar 2.2.2.3.4 *Classification report* untuk model bidirectional

Model RNN unidirectional menunjukkan performa yang tidak terlalu baik. Akurasi keseluruhan yang dicapai hanya 0.38 dengan *macro average F1-score* yang lebih rendah lagi, yaitu 0.28, dan *weighted average F1-score* 0.32. Kemudian, model ini tidak mampu untuk mengklasifikasikan kelas 'neutral' sama sekali yang terlihat dari nilai *precision*, *recall*, dan *F1-score* sebesar 0.00 untuk kelas tersebut. Kelas 'negative' dan 'positive' juga memiliki F1-score yang rendah, masing-masing 0.36 dan 0.48. Hasil ini mengindikasikan bahwa model unidirectional kesulitan dalam mempelajari pola yang relevan dari data dan gagal melakukan klasifikasi yang akurat.

Sebaliknya, model RNN bidirectional menunjukkan peningkatan performa yang dramatis di semua metrik. Akurasi keseluruhan melonjak menjadi 0.65 dengan *macro average F1-score* 0.64 dan *weighted average F1-score* juga 0.64. Peningkatan ini konsisten di semua kelas. Kelas 'negative' mencapai F1-score 0.56, kelas 'neutral' yang sebelumnya gagal total kini mencapai F1-score 0.58, dan kelas 'positive' meningkat pesat hingga F1-score 0.77. Kemampuan model bidirectional untuk memproses informasi sekuens dari kedua arah (maju dan mundur) tampaknya memberikan pemahaman kontekstual yang jauh lebih baik sehingga menghasilkan klasifikasi yang jauh lebih akurat dan seimbang antar kelas.

Secara keseluruhan, penggunaan arsitektur bidirectional pada RNN terbukti memberikan peningkatan performa yang sangat baik dibandingkan dengan arsitektur unidirectional untuk tugas klasifikasi ini.

Model bidirectional tidak hanya meningkatkan akurasi secara keseluruhan tetapi juga secara signifikan memperbaiki kemampuan model untuk mengidentifikasi semua kelas target, termasuk kelas yang sama sekali tidak terdeteksi oleh model unidirectional. Hal ini dengan kuat menunjukkan bahwa informasi dari konteks masa lalu dan masa depan dalam sekuens sangat krusial untuk mencapai klasifikasi yang efektif dalam kasus ini.

2.2.3 Long Short-Term Memory (LSTM)

Analisis pengaruh beberapa hyperparameter dalam LSTM dilakukan pada dataset NusaX-Sentiment (Bahasa Indonesia)

2.2.3.1 Pengaruh jumlah layer LSTM

Untuk menganalisis pengaruh banyaknya layer, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah **Layer Embedding**
- n buah **Layer Unidirectional LSTM** (nilai n disesuaikan dengan pengujian)
- 1 buah **Layer Dense**
- 1-5 buah **Layer Dropout**

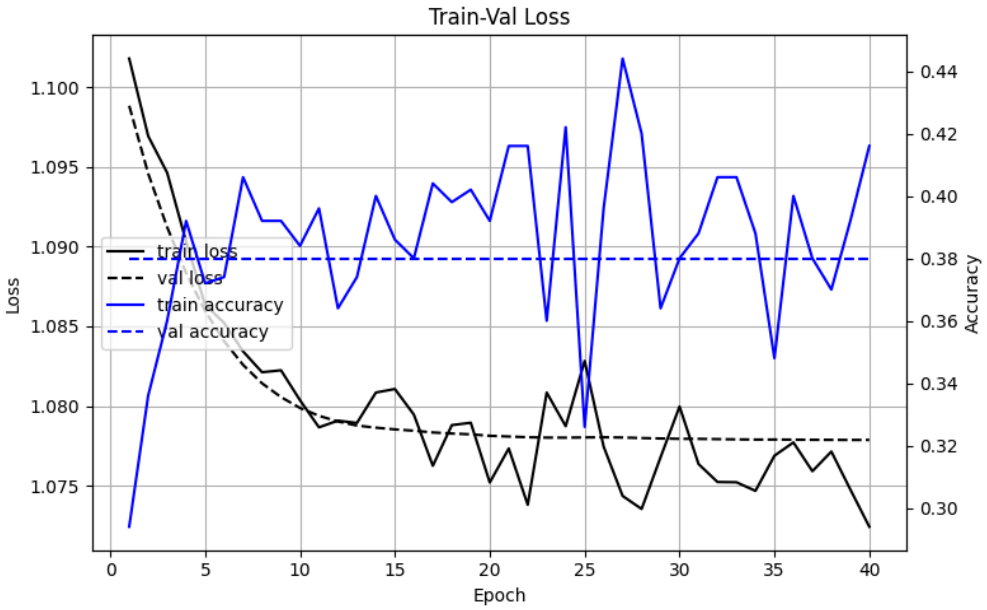
Detail Konfigurasi Layer:

13. **Layer Embedding:**
    - input_dim: 3000
    - output_dim: 64
14. **Layer Unidirectional LSTM:**
    - Jumlah neuron: 32
15. **Layer Dense (Output Layer):**
    - Jumlah neuron: 3
    - Fungsi aktivasi: Softmax

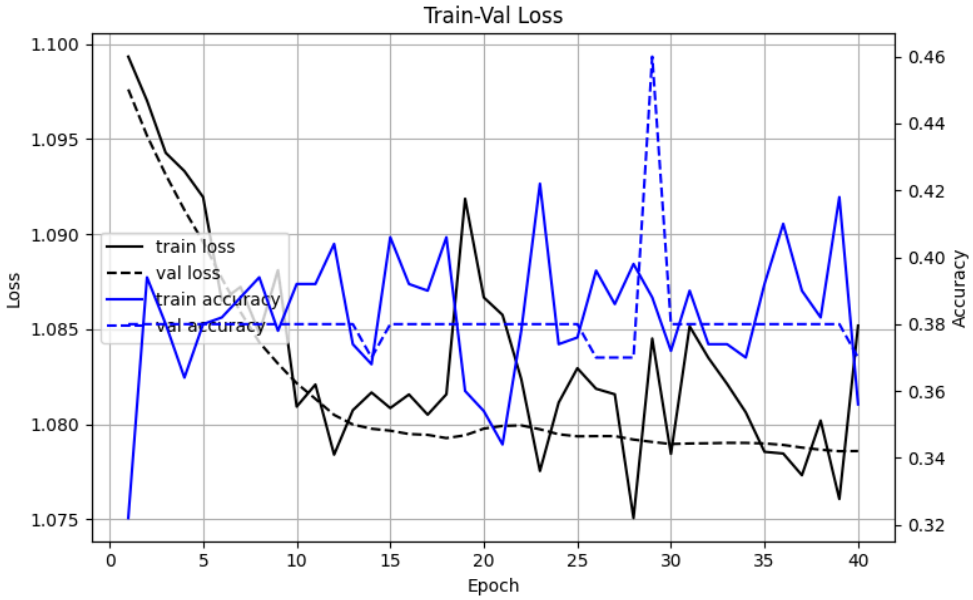
Fungsi Loss:

- Fungsi loss yang akan digunakan adalah sparse categorical crossentropy.

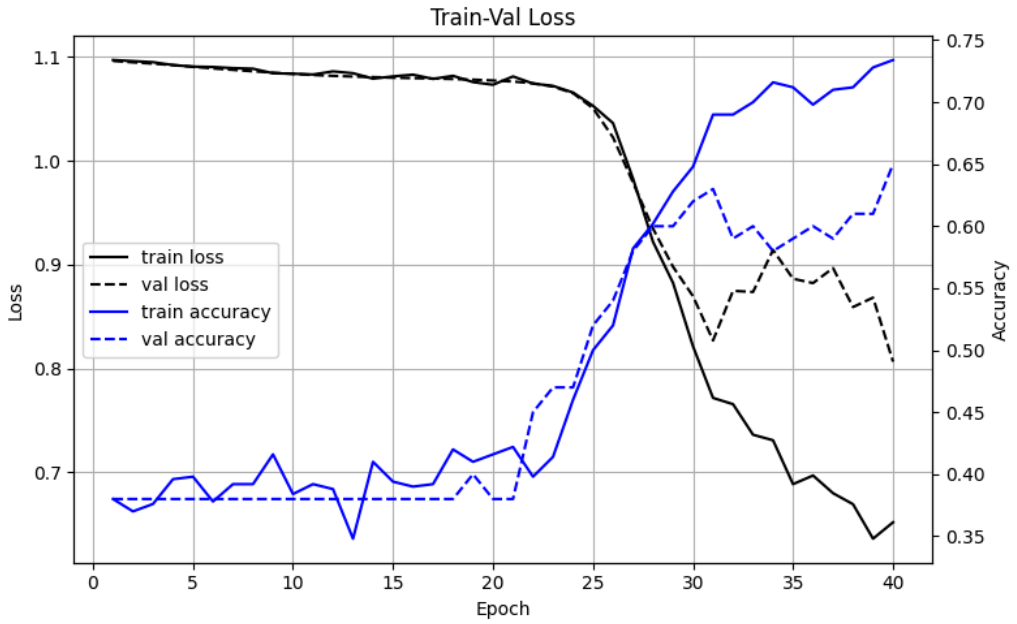
Didapatkan hasil sebagai berikut.



Gambar 2.2.3.1.1 Plot loss dan accuracy untuk model dengan 1 layer LSTM. Garis hitam merupakan grafik loss, garis biru merupakan grafik akurasi. Garis solid merupakan grafik hasil data training, grafik putus-putus merupakan grafik hasil data validasi.



Gambar 2.2.3.1.2 Plot loss dan accuracy untuk model dengan 3 layer LSTM. Garis hitam merupakan grafik loss, garis biru merupakan grafik akurasi. Garis solid merupakan grafik hasil data training, grafik putus-putus merupakan grafik hasil data validasi.



Gambar 2.2.3.1.3 Plot loss dan accuracy untuk model dengan 5 layer LSTM. Garis hitam merupakan grafik loss, garis biru merupakan grafik akurasi. Garis solid merupakan grafik hasil data training, grafik putus-putus merupakan grafik hasil data validasi.

Berdasarkan Gambar 2.2.3.1.1, 2.2.3.1.2, dan 2.2.3.1.3, menunjukkan bahwa jumlah layer LSTM memberikan pengaruh yang cukup signifikan pada performa model. Model dengan 1 layer LSTM yang ditunjukkan oleh Gambar 2.2.3.1.1, memperlihatkan bahwa nilai loss training yang menurun dan berfluktuasi dan nilai loss validasi yang menurun secara stabil dan konvergen. Hal ini mengindikasikan bahwa model belajar cukup baik terhadap data yang belum pernah dilihat. Nilai loss yang berfluktuasi pada data training menunjukkan bahwa layer dropout cukup berpengaruh pada generalisasi model. Selain itu, nilai akurasi pada data latih menunjukkan performa yang meningkat di awal epoch, namun mulai berfluktuasi pada epoch ke-4, serta nilai akurasi yang stagnan pada akurasi untuk data validasi. Hal ini menunjukkan bahwa walaupun model mengalami penurunan nilai loss pada data validasi, akurasi model tetap tidak meningkat, yang dapat disebabkan oleh model yang hanya mempelajari sebagian pola tanpa generalisasi yang baik terhadap data baru.

Pada gambar ketika menggunakan 3 layer LSTM, yang diperlihatkan oleh Gambar 2.2.3.1.2, model mengalami hal yang serupa dengan yang dialami oleh model 1 layer LSTM, yaitu loss pada data train menurun walaupun berfluktuasi dan loss pada data validasi cenderung menurun dengan stabil dan konvergen. Hal ini menunjukkan bahwa model cukup belajar dari data training, walaupun nilai loss pada data training cukup berfluktuasi. Fluktuasi ini disebabkan oleh penggunaan layer dropout pada layer LSTM. Selain itu, nilai akurasi pada data training menunjukkan peningkatan pada awal epoch, tetapi mulai berfluktuasi pada epoch 3 dan akurasi data validasi yang cenderung stagnan, walaupun mengalami peningkatan sekali pada epoch 28, serta penurunan akurasi pada epoch terakhir. Kondisi ini mencerminkan bahwa pengaruh performa model tidak hanya dipengaruhi oleh hasil kurva loss saja, melainkan terdapat faktor lain, seperti penggunaan hyperparameter selain jumlah layer yang belum optimal.

Selanjutnya, pada pengaruh jumlah 5 layer LSTM, yang ditunjukkan oleh Gambar 2.2.3.1.3, menggambarkan bahwa nilai loss data training dan validasi cukup stabil pada epoch 1 hingga 25, lalu menurun setelahnya. Nilai loss validasi mulai mengalami fluktuasi pada epoch 32 dan nilai loss training cenderung menurun pada epoch 26, serta sedikit berfluktuasi pada akhir epoch. Hal ini mengindikasikan bahwa model mengalami *overfitting* terhadap data pelatihan, yakni terlalu menyesuaikan diri dengan data tersebut sehingga performanya menurun pada data baru. Gejala overfitting ini juga tercermin pada grafik akurasi, yang menunjukkan penurunan akurasi validasi pada epoch ke-32, sementara akurasi data pelatihan yang terus meningkat.

Secara keseluruhan, pengaruh peningkatan jumlah layer LSTM terhadap performa model cukup signifikan, walaupun tidak selalu menghasilkan performa terbaik. Model dengan 1 dan 3 layer LSTM, memperlihatkan kurva loss yang cukup baik, tetapi model tidak menunjukkan peningkatan performa, yang diperlihatkan oleh nilai akurasi yang cenderung stagnan di sekitar nilai 0.38. Sedangkan model dengan 5 layer LSTM, menunjukkan kurva loss yang cenderung overfitting, tetapi menghasilkan peningkatan akurasi pada data validasi, walaupun pada beberapa epoch terakhir, akurasinya berfluktuasi.

Tabel 2.2.3.1.1 Daftar nilai metrik untuk masing-masing model dengan layer yang berbeda-beda pada data testing. Test Loss yang digunakan adalah sparse categorical crossentropy.

Jumlah Layer	Test Loss	Test Accuracy	Macro F1-Score	F1-Score Per Class [negative, neutral, positive]
1 layer	1.0778	0.3850	0.1892	[0.55, 0, 0.01]
3 layer	1.0783	0.4175	0.2611	[0.56, 0, 0.22]
5 layer	0.8685	0.6150	0.4674	[0.61, 0, 0.79]

Tabel 2.2.3.1.1 memperlihatkan nilai metrik untuk masing-masing model dengan layer yang berbeda. Berdasarkan tabel tersebut, terlihat bahwa model dengan 5 layer LSTM lebih unggul di semua metrik dalam mengklasifikasikan sentimen suatu teks. Hal ini terlihat pada nilai akurasi dan F1-Score (macro)-nya yang memiliki nilai paling tinggi, yaitu 0.6150 dan 0.4674. Nilai akurasi menunjukkan seberapa baik model dalam mengklasifikasikan teks ke kategori sentimen yang benar. Hal ini menunjukkan bahwa sekitar 61.5% data teks (testing) diklasifikasikan dengan benar oleh model. Selain itu, nilai macro F1-Score menunjukkan rata-rata nilai keseimbangan antara presisi model dan recall model (yaitu, memiliki nilai true positive sebanyak mungkin, serta nilai false negative dan false positive sesedikit mungkin). Semakin mendekati 1 nilai F1-Score, maka model memiliki akurasi yang baik dalam mengklasifikasikan setiap teks ke dalam kategori yang benar. Walaupun nilainya tergolong kecil, tetapi model dengan 5 layer LSTM memiliki nilai F1-score yang paling besar dibandingkan dengan model yang memiliki lebih sedikit layer LSTM. Hal ini menunjukkan bahwa semakin banyak layer LSTM, maka ketepatan model untuk mengklasifikasikan teks sentimen ke dalam kategori yang benar semakin meningkat. Kondisi ini juga diperlihatkan pada nilai F1-Score per kelas, nilai F1-score meningkat pada kelas negative dan positive. Namun, seluruh model masih gagal dalam mengklasifikasikan teks ke dalam sentimen netral yang benar. Hal ini dapat disebabkan oleh ketidakseimbangan data yang memiliki klasifikasi netral. Selain itu, pengaruh hyperparameter atau arsitektur model juga dapat mempengaruhi akurasi model dalam mengklasifikasikan teks ke dalam sentimen tertentu.

2.2.3.2 Pengaruh banyak cell LSTM per layer

Untuk menganalisis pengaruh banyaknya *cell* LSTM per layer, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah **Layer Embedding**
- 5 buah **Layer Unidirectional LSTM**
- 1 buah **Layer Dense**
- 5 buah **Layer Dropout**

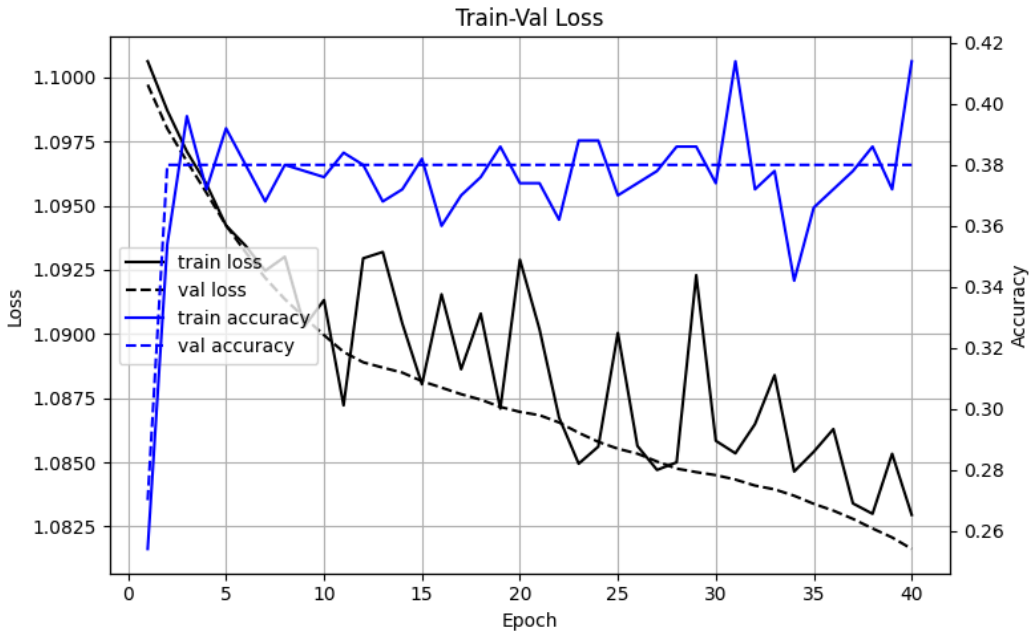
Detail Konfigurasi Layer:

16. **Layer Embedding:**
    - input_dim: 3000
    - output_dim: 64
17. **Layer Unidirectional LSTM:**
    - Jumlah neuron disesuaikan dengan pengujian
18. **Layer Dense (Output Layer):**
    - Jumlah neuron: 3
    - Fungsi aktivasi: Softmax

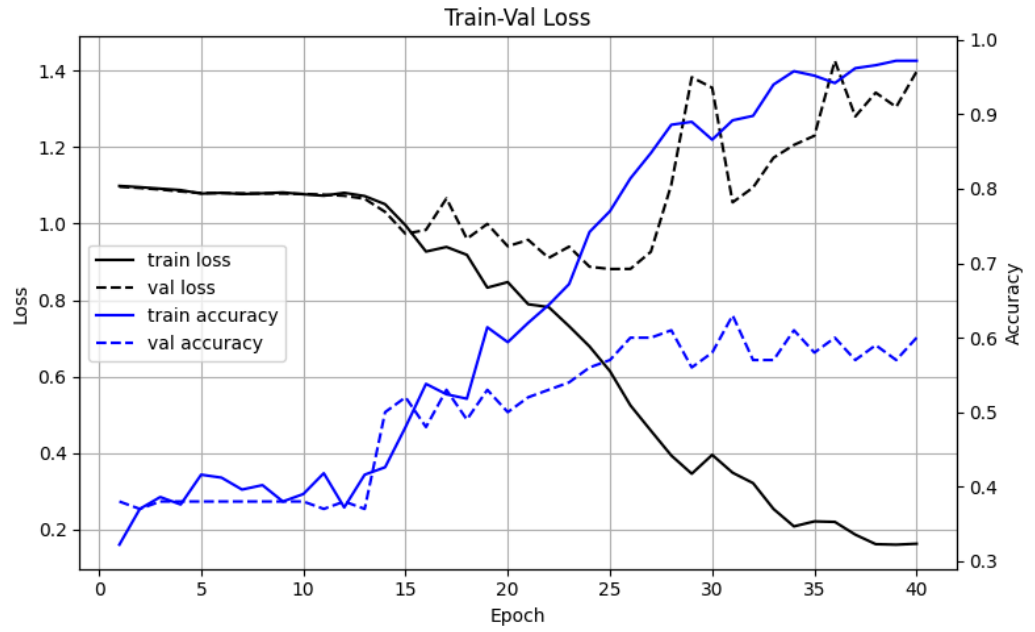
Fungsi Loss:

- Fungsi loss yang akan digunakan adalah sparse categorical crossentropy.

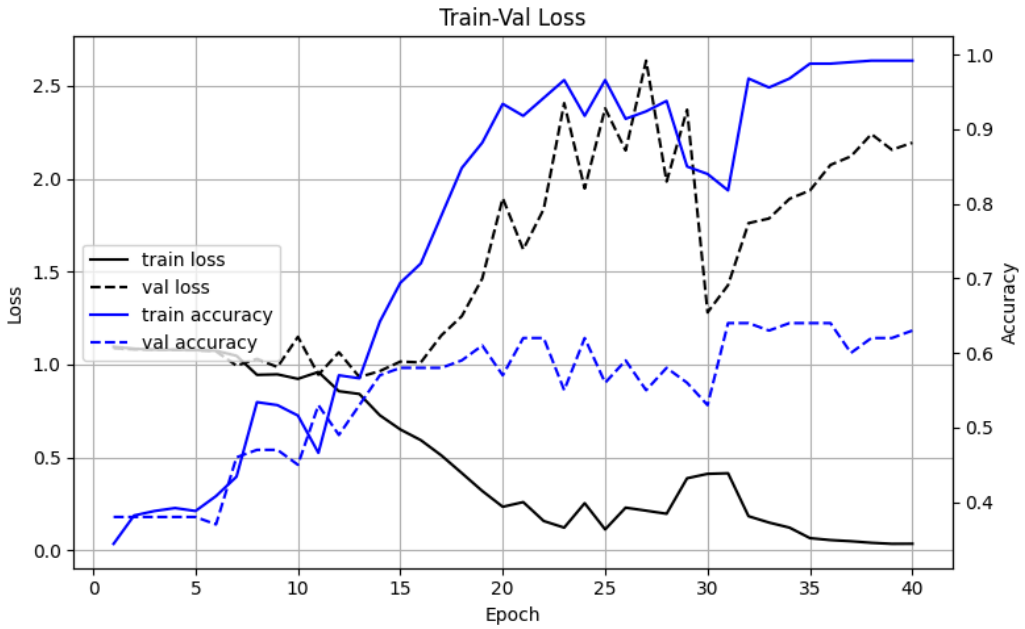
Didapatkan hasil sebagai berikut.



Gambar 2.2.3.2.1 Plot loss dan accuracy untuk model dengan 16 neuron layer LSTM. Garis hitam merupakan grafik loss, garis biru merupakan grafik akurasi. Garis solid merupakan grafik hasil data training, grafik putus-putus merupakan grafik hasil data validasi.



Gambar 2.2.3.2.1 Plot loss dan accuracy untuk model dengan 128 neuron layer LSTM. Garis hitam merupakan grafik loss, garis biru merupakan grafik akurasi. Garis solid merupakan grafik hasil data training, grafik putus-putus merupakan grafik hasil data validasi.



Gambar 2.2.3.2.3 Plot loss dan accuracy untuk model dengan 512 neuron layer LSTM. Garis hitam merupakan grafik loss, garis biru merupakan grafik akurasi. Garis solid merupakan grafik hasil data training, grafik putus-putus merupakan grafik hasil data validasi.

Gambar 2.2.3.2.1, 2.2.3.2.2, dan 2.2.3.2.3 menunjukkan grafik loss training dan validasi, serta grafik akurasi loss dan validasi pada masing-masing model yang memiliki jumlah neuron pada layer LSTM yang berbeda. Berdasarkan Gambar 2.2.3.2.1, diperlihatkan bahwa model dengan jumlah 16 neuron mengalami penurunan loss pada data validasi maupun training, tetapi pada loss training, nilainya berfluktuasi. Selain itu, nilai akurasinya meningkat pada awal epoch, namun pada data training, akurasinya berfluktuasi di sekitar 0.38 dan akurasi pada data validasi cenderung konstan pada nilai 0.38. Hal ini mengindikasikan bahwa walaupun grafik loss menurun, model dengan 16 neuron tidak mengalami peningkatan performa. Kondisi ini mirip seperti pada Gambar 2.2.3.1.1, yang memperlihatkan model hanya mempelajari sebagian pola tanpa generalisasi yang baik terhadap data baru.

Berdasarkan Gambar 2.2.3.2.2 dan 2.2.23.2.3, nilai loss training pada kedua gambar menurun, tetapi nilai loss validasinya mulai meningkat pada beberapa epoch setelah epoch awal. Selain itu, nilai akurasi training pada kedua gambar cenderung meningkat, tetapi nilai akurasi validasinya mulai stagnan pada beberapa epoch di pertengahan epoch. Hal ini menunjukkan bahwa kedua model mengalami overfitting, yaitu model terlalu mempelajari data training, sehingga apabila diterapkan pada data baru, performanya cenderung menurun.

Secara keseluruhan peningkatan jumlah neuron tidak selalu meningkatkan performa model. Hal ini dapat dilihat pada grafik loss validasi yang meningkat cukup signifikan pada jumlah neuron 512 (Gambar 2.2.3.2.3). Sedangkan pada jumlah neuron 128 (Gambar 2.2.3.2.2), walaupun mengalami peningkatan loss validasi, nilainya tidak terlalu signifikan dibandingkan pada model dengan jumlah neuron 512. Selain itu, kondisi ini juga menunjukkan bahwa model yang terlalu kompleks, belum tentu menghasilkan performa yang lebih baik.

Tabel 2.2.3.2.1 Daftar nilai metrik untuk masing-masing model dengan jumlah neuron yang berbeda-beda. Test Loss yang digunakan adalah sparse categorical crossentropy.

Jumlah Neuron	Test Loss	Test Accuracy	Macro F1-Score	F1-Score Per Class [negative, neutral, positive]
16 neuron	1.0795	0.3775	0.1827	[0, 0, 0.55]
128 neuron	1.1160	0.6625	0.6354	[0.64, 0.48, 0.78]
512 neuron	2.2762	0.5975	0.5543	[0.55, 0.39, 0.73]

Tabel 2.2.3.2.1 menunjukkan nilai metrik untuk masing-masing model LSTM dengan jumlah neuron yang berbeda-beda. Berdasarkan tabel tersebut, terlihat bahwa model mengalami penurunan performa akurasi dan F1-Score (macro) pada jumlah neuron yang lebih besar, terutama pada jumlah 512 neuron. Pada nilai F1-Score per kelas, terlihat bahwa model dengan 16 neuron memiliki performa yang paling buruk dalam mengklasifikasikan teks dengan sentimen negatif dan netral. Pada model dengan 128 neuron terjadi peningkatan performa yang signifikan pada masing-masing hasil klasifikasi. Selanjutnya, pada model dengan 512 neuron, akurasinya menurun lagi, walaupun masih lebih baik daripada model dengan 16 neuron. Hal ini mengindikasikan bahwa model yang kompleks tidak menjamin terjadinya peningkatan performa model.

2.2.3.3 Pengaruh jenis layer LSTM berdasarkan arah

Untuk menganalisis pengaruh jenis layer LSTM berdasarkan arah, akan dilakukan pengujian menggunakan sebuah model dengan arsitektur spesifik sebagai berikut:

Komponen Model:

- 1 buah **Layer Embedding**
- 1 buah **Layer Bidirectional LSTM atau Layer Unidirectional LSTM**
- 1 buah **Layer Dense**
- 5 buah **Layer Dropout**

Detail Konfigurasi Layer:

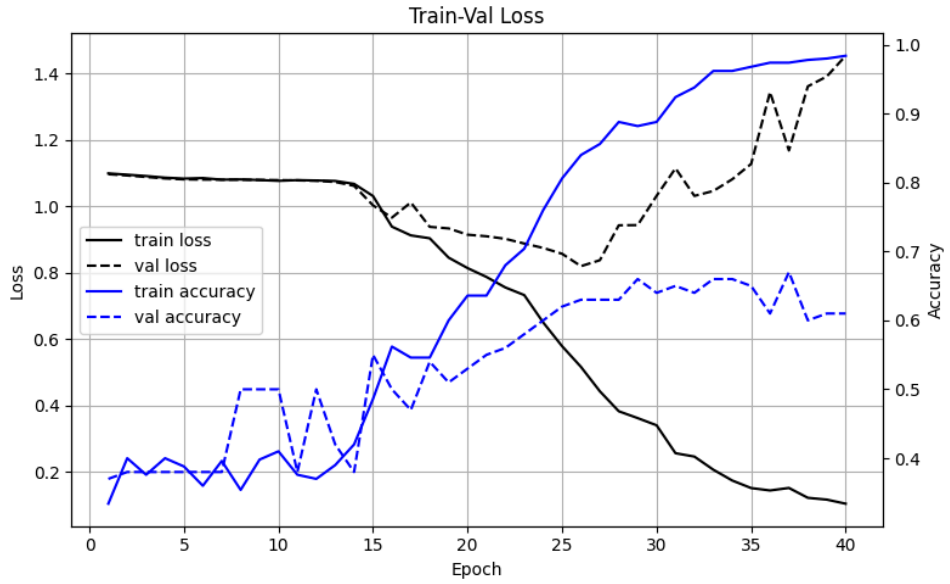
19. **Layer Embedding:**
    - input_dim: 3000
    - output_dim: 64
20. **Layer Bidirectional LSTM:**
    - Jumlah neuron: 128
21. **Layer Unidirectional LSTM:**
    - Jumlah neuron: 128
22. **Layer Dense (Output Layer):**
    - Jumlah neuron: 3

- Fungsi aktivasi: Softmax

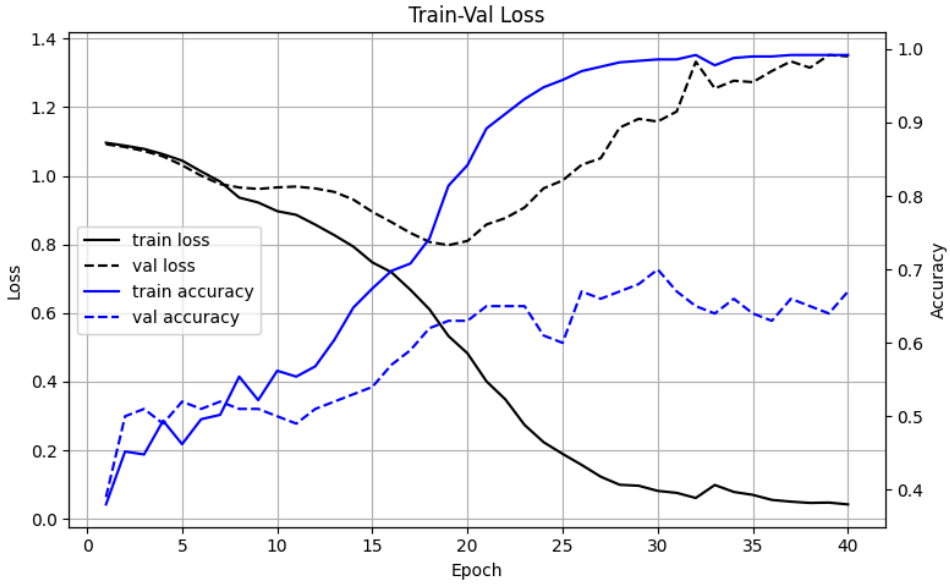
Fungsi Loss:

- Fungsi loss yang akan digunakan adalah sparse categorical crossentropy.

Didapatkan hasil sebagai berikut.



Gambar 2.2.3.3.1 Plot loss dan accuracy untuk model dengan unidirectional LSTM. Garis hitam merupakan grafik loss, garis biru merupakan grafik akurasi. Garis solid merupakan grafik hasil data training, grafik putus-putus merupakan grafik hasil data validasi.



Gambar 2.2.3.3.2 Plot loss dan accuracy untuk model dengan bidirectional LSTM. Garis hitam merupakan grafik loss, garis biru merupakan grafik akurasi. Garis solid merupakan grafik hasil data training, grafik putus-putus merupakan grafik hasil data validasi.

Gambar 2.2.3.3.1 dan 2.2.3.3.2 menunjukkan grafik loss dan akurasi untuk data training dan validasi pada model LSTM unidirectional dan bidirectional. Berdasarkan kedua gambar tersebut, terlihat bahwa kedua model mengalami overfitting, yang ditunjukkan oleh grafik loss training yang menurun dan grafik loss validasi yang meningkat pada pertengahan epoch. Namun, pada model dengan layer LSTM bidirectional (Gambar 2.2.3.3.2), peningkatan loss validasi terjadi lebih dahulu pada epoch awal (epoch 12) dibandingkan dengan model LSTM unidirectional (epoch 17). Namun, pada akhir epoch, nilai loss validasi pada LSTM bidirectional lebih kecil daripada LSTM unidirectional, serta nilai loss akurasi pada layer bidirectional lebih besar dari layer unidirectional. Selain itu, grafik akurasi pada unidirectional menunjukkan penurunan pada akhir epoch, sedangkan pada grafik akurasi bidirectional, grafiknya cenderung konstan. Hal ini mengindikasikan bahwa, walaupun loss validasi pada bidirectional meningkat lebih dahulu, nilai akurasinya cenderung lebih stabil dibandingkan pada unidirectional.

Tabel 2.2.3.1.1 Daftar nilai metrik untuk masing-masing model dengan arah layer unidirectional dan bidirectional. Test Loss yang digunakan adalah sparse categorical crossentropy.

Arah Layer	Test Loss	Test Accuracy	Macro F1-Score	F1-Score Per Class [negative, neutral, positive]
Unidirectional	1.1715	0.6675	0.6630	[0.57, 0.62, 0.8]
Bidirectional	1.3140	0.6800	0.6668	[0.62, 0.57, 0.81]

Tabel 2.2.3.1.1 menunjukkan nilai metrik pada model LSTM unidirectional dan bidirectional. Berdasarkan tabel tersebut, terlihat bahwa nilai akurasi dan F1-Score pada kedua model hampir sama, dengan bidirectional memiliki akurasi dan F1-Score (macro) yang paling tinggi. Hal ini mengindikasikan bahwa jenis layer yang digunakan pada model memberikan pengaruh yang tidak terlalu signifikan pada performa model. Selain itu, pada hasil F1-Score masing-masing kelas, akurasi pada kelas negative dan positif mengalami peningkatan pada layer bidirectional, tetapi akurasinya menurun pada kategori netral. Hal ini menandakan bahwa penggunaan layer bidirectional membuat model lebih baik dalam mengenali kelas negatif dan positif, tetapi justru menurunkan kemampuannya dalam mengenali kelas netral. Artinya peningkatan akurasi pada beberapa kelas, bisa berdampak pada penurunan performa di kelas lainnya.

BAB III
KESIMPULAN DAN SARAN

3.1 Kesimpulan

Forward Propagation untuk CNN memberikan hasil yang sama dengan menggunakan Keras ataupun dengan menggunakan CNN from scratch. Terbukti dari pengujian batch dari data test memberikan hasil inferensi yang sama baik menggunakan Keras maupun CNN from scratch. Berdasarkan hasil pengujian variasi hyperparameter CNN pada dataset CIFAR-10, terbukti bahwa arsitektur dengan empat layer konvolusi berfilter sedang (sekitar 96–128), kernel 3×3, dan max pooling—diakhiri global average pooling—memberikan keseimbangan terbaik antara kapasitas representasi dan kemampuan generalisasi, mencapai macro F1 hingga 0.77, sedangkan model yang terlalu dangkal atau menggunakan kernel/filter terlalu besar menunjukkan kecenderungan overfitting atau underfitting. Dengan rekap Macro F1 untuk berbagai variasi hyperparameter sebagai berikut

```
=== Ringkasan Eksperimen ===
```

	variant	type	macro_f1
2	c4_f96-192-192-192_k3_max	conv_layers	0.773205
6	c3_f96-192-192_k3_max	kernel_size	0.723859
1	c3_f96-192-192_k3_max	conv_layers	0.720315
10	c3_f96-192-192_k3_avg	pooling	0.717090
9	c3_f96-192-192_k3_max	pooling	0.709337
5	c3_f128-128-128_k3_max	filters	0.696367
4	c3_f64-64-64_k3_max	filters	0.684670
0	c2_f96-192_k3_max	conv_layers	0.666513
7	c3_f96-192-192_k5_max	kernel_size	0.665467
8	c3_f96-192-192_k7_max	kernel_size	0.635923
3	c3_f32-32-32_k3_max	filters	0.628291

Berdasarkan keseluruhan analisis yang telah dilakukan terhadap berbagai konfigurasi model RNN, dapat disimpulkan bahwa pemilihan arsitektur dan hiperparameter memiliki dampak yang sangat signifikan terhadap performa klasifikasi. Secara konsisten, model RNN dengan arsitektur bidirectional menunjukkan keunggulan performa yang jauh melampaui model unidirectional, baik dari segi stabilitas proses pembelajaran maupun metrik evaluasi akhir seperti akurasi dan F1-score. Terkait kedalaman model, konfigurasi dengan satu layer RNN terbukti memberikan hasil klasifikasi paling optimal di mana penambahan jumlah layer menjadi tiga atau bahkan lima layer cenderung menurunkan kinerja model. Sementara itu, jumlah *cell* dalam layer RNN juga menunjukkan adanya nilai optimal; penggunaan 64 *cell* menghasilkan performa klasifikasi terbaik, sedangkan jumlah *cell* yang lebih sedikit (16) memberikan hasil yang moderat, dan jumlah *cell* yang terlalu banyak (256) secara drastis menurunkan kemampuan klasifikasi model hingga menyebabkan kegagalan dalam mengidentifikasi kelas tertentu. Temuan ini secara kolektif menggarisbawahi bahwa arsitektur yang lebih kompleks tidak selalu menjamin hasil yang lebih baik, dan optimasi yang cermat terhadap arah pemrosesan sekuens, jumlah layer, serta jumlah *cell* merupakan langkah krusial untuk mencapai kinerja model RNN yang maksimal dalam tugas yang dihadapi.

Forward propagation untuk LSTM memberikan hasil yang sama antara model dari Keras ataupun from scratch. Hal ini terlihat pada hasil pengujian yang dilakukan pada data test untuk kedua model menghasilkan inferensi yang sama. Berdasarkan hasil pengujian jumlah layer LSTM, model yang memiliki layer lebih banyak (5 layer) menghasilkan performa yang paling baik daripada model dengan layer LSTM lebih sedikit. Selain itu, jumlah neuron yang diimplementasikan menunjukkan bahwa layer dengan 128 neuron memiliki performa yang lebih baik dibandingkan layer LSTM dengan 512 neuron. Hal ini menunjukkan bahwa penambahan neuron pada setiap layer LSTM belum tentu memberikan performa yang lebih baik pada model. Penerapan bidirectional pada layer LSTM menghasilkan performa yang sedikit lebih baik daripada unidirectional, walaupun performa pada beberapa kelas harus dikorbankan (performa beberapa kelas yang berhasil diprediksi ada yang meningkat dan menurun).

3.2 Saran

Untuk meningkatkan performa Forward Propagation CNN lebih jauh, tambahkan teknik regularisasi seperti batch normalization dan dropout ringan di antara layer konvolusi dan dense head, terapkan data augmentation (rotasi, flipping, cropping) selama pelatihan, serta optimasi forward propagation from scratch dengan pendekatan im2col atau JIT (Numba) untuk mempercepat inferensi pada eksperimen hyperparameter skala besar.

Untuk pengujian RNN, pengembangan model selanjutnya dapat memprioritaskan penggunaan arsitektur RNN bidirectional karena keunggulannya yang signifikan dan konsisten atas model unidirectional. Sebaiknya dimulai dengan konfigurasi satu layer RNN yang terbukti paling efektif dan jika eksplorasi penambahan layer dilakukan, harus disertai dengan pemantauan ketat

terhadap potensi overfitting dan penurunan performa. Jumlah *cell* pada layer RNN juga perlu dioptimalkan dengan cermat dengan fokus pada rentang menengah (misalnya, sekitar 64 *cell* yang menunjukkan hasil baik) dan menghindari jumlah *cell* yang terlalu sedikit atau terlalu banyak yang terbukti merugikan. Mengingat adanya indikasi overfitting pada beberapa konfigurasi, penerapan atau penguatan teknik regularisasi seperti dropout pada berbagai layer, L1/L2 regularization, atau penggunaan *early stopping* berdasarkan performa validasi sangat dianjurkan untuk meningkatkan kemampuan generalisasi model.

Dalam implementasi Keras ataupun from scratch pada forward propagation untuk LSTM, dapat diterapkan teknik regularisasi seperti L2 regularization atau early stopping agar model tidak overfitting. Selain itu, untuk optimasi forward propagation, dapat memanfaatkan penggunaan GPU untuk proses pehitungan paralel agar lebih efisien.

LAMPIRAN

Link Repository: https://github.com/UburUburLembur/TugasBesar2IF3270_Kelompok20

Pembagian tugas tiap anggota kelompok:

Nama	NIM	Tugas
Muhamamd Zakkiy	10122074	Mengerjakan bagian RNN
Ghaisan Zaki Pratama	10122078	Mengerjakan bagian CNN
Fardhan Indrayesa	12821046	Mengerjakan bagian LSTM

DAFTAR PUSTAKA

https://towardsdatascience.com/dropout-in-neural-networks-47a162d621d9/

https://stackoverflow.com/questions/42861460/how-to-interpret-weights-in-a-lstm-layer-in-keras

https://d2l.ai/chapter_recurrent-modern/lstm.html