

IF3270 Pembelajaran Mesin
Convolutional Neural Network dan Recurrent Neural Network

Tugas Besar 2

Disusun untuk memenuhi tugas mata kuliah Pembelajaran Mesin
pada Semester 2 (dua) Tahun Akademik 2024/2025

Tugas Besar IF3270 Pembelajaran Mesin



Oleh

Shabrina Maharani **13522134**

Muhammad Davis Adhipramana **13522157**

Valentino Chryslie Triadi **13522164**

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2025

DAFTAR ISI

DAFTAR ISI	2
BAB 1 DESKRIPSI persoalan	5
BAB 2 PEMBAHASAN	6
2.1 Penjelasan Implementasi	6
2.1.1 Deskripsi Kelas, Atribut, dan Method	6
2.1.1.1 CNN	6
2.1.1.2 RNN	17
2.1.1.3 LSTM	25
2.1.2 CNN	35
2.1.3 RNN	36
2.1.4 LSTM	38
2.2 Hasil Pengujian	41
2.2.1 CNN	41
2.2.1.1 Pengaruh Jumlah Konvolusi Layer	41
2.2.1.1.2 Grafik Training Loss dan Validation Loss	42
2.2.1.1.2.1 Layers = 1	42
2.2.1.1.2.2 Layers = 2	42
2.2.1.1.2.2 Layers = 3	42
2.2.1.1.3 Analisis Pengaruh Jumlah Konvolusi Layer	42
2.2.1.2 Pengaruh Banyak Filter per Layer Konvolusi	43
2.2.1.2.2 Grafik Training Loss dan Validation Loss	44
2.2.1.2.2.1 16 filter dan 32 filter	44
2.2.1.2.2.2 32 filter dan 64 filter	44
2.2.1.2.2.3 64 filter dan 128 filter	44
2.2.1.2.3 Analisis Pengaruh Banyak Filter per Layer Konvolusi	45
2.2.1.3 Pengaruh Ukuran Filter per Layer Konvolusi	45
2.2.1.3.2.1 Kernel dengan layer 1 (2,2) dan layer 2 (2,2)	46
2.2.1.3.2.1 Kernel dengan layer 1 (3,3) dan layer 2 (3,3)	46
2.2.1.3.2.3 Kernel dengan layer 1 (5,5) dan layer 2 (5,5)	46
2.2.1.4 Pengaruh Jenis Pooling Layer yang Digunakan	47
2.2.1.4.2.1 Max Pooling	47
2.2.1.4.2.1 Average Pooling	48
2.2.2.4 Perbandingan From Scratch dengan Keras Model	49
2.2.2.4.1 Hasil Akhir Prediksi	49
2.2.2.4.2 Analisis Perbandingan	49
2.2.2 RNN	50
2.2.2.1 Pengaruh Jumlah Layer RNN	50
2.2.2.1.1 Hasil Akhir Prediksi	50

2.2.2.1.2 Grafik Training Loss dan Validation Loss	50
2.2.2.1.2.1 Layers = 25	50
2.2.2.1.2.2 Layers = 50	50
2.2.2.1.2.3 Layers = 100	50
2.2.2.1.2.4 Grafik Gabungan	51
2.2.2.1.3 Analisis Pengaruh Jumlah Layer RNN	51
2.2.2.2 Pengaruh Banyak Cell RNN per Layer	52
2.2.2.2.1 Hasil Akhir Prediksi	52
2.2.2.2.2 Grafik Training Loss dan Validation Loss	52
2.2.2.2.2.1 Cells = 32	52
2.2.2.2.2.2 Cells = 64	52
2.2.2.2.2.3 Cells = 128	52
2.2.2.2.2.4 Grafik Gabungan	53
2.2.2.2.3 Analisis Pengaruh Banyak Cell RNN per Layer	53
2.2.2.3 Pengaruh Jenis Layer RNN berdasarkan Arah	54
2.2.2.3.1 Hasil Akhir Prediksi	54
2.2.2.3.2 Grafik Training Loss dan Validation Loss	54
2.2.2.3.2.1 Bidirectional	54
2.2.2.3.2.2 Unidirectional	54
2.2.2.3.2.3 Grafik Gabungan	54
2.2.2.3.3 Analisis Pengaruh Jenis Layer RNN berdasarkan Arah	54
2.2.2.4 Perbandingan From Scratch dengan Keras Model	56
2.2.2.4.1 Hasil Akhir Prediksi	56
2.2.2.4.2 Analisis Perbandingan	56
2.2.3 LSTM	58
2.2.3.1 Pengaruh Jumlah Layer LSTM	58
2.2.3.1.1 Hasil Akhir Prediksi	58
2.2.3.1.1.1 Satu Layer Bi-Directional LSTM	58
2.2.3.1.1.2 Dua Layer Bi-Directional LSTM	58
2.2.3.1.1.3 Tiga Layer Bi-Directional LSTM	60
2.2.3.1.2 Grafik Training Loss dan Validation Loss	60
2.2.3.1.2.1 Satu Layer Bi-Directional LSTM	60
2.2.3.1.2.2 Dua Layer Bi-Directional LSTM	61
2.2.3.1.2.3 Tiga Layer Bi-Directional LSTM	61
2.2.3.1.3 Analisis Perbandingan	61
2.2.3.2 Pengaruh Banyak Cell LSTM per Layer	62
2.2.3.2.1 Hasil Akhir Prediksi	62
2.2.3.2.1.1 Cells = 32	62
2.2.3.2.1.2 Cells = 64	63
2.2.3.2.1.3 Cells = 128	64
2.2.3.2.2 Grafik Training Loss dan Validation Loss	65

2.2.3.2.2.1 Cells = 32	65
2.2.3.2.2.1 Cells = 64	65
2.2.3.2.2.1 Cells = 128	65
2.2.3.2.3 Analisis Pengaruh Banyak Cell LSTM per Layer	65
2.2.3.3 Pengaruh Jenis Layer LSTM berdasarkan Arah	66
2.2.3.3.1 Hasil Akhir Prediksi	66
2.2.3.3.1.1 Bidirectional	66
2.2.3.3.1.2 Unidirectional	67
2.2.3.3.2 Grafik Training Loss dan Validation Loss	68
2.2.3.3.2.1 Bidirectional	68
2.2.3.3.2.1 Unidirectional	68
2.2.3.3.3 Analisis Pengaruh Jenis Layer RNN berdasarkan Arah	68
2.2.3.4 Perbandingan From Scratch dengan Keras Model	69
2.2.3.4.1 Hasil Akhir Prediksi	69
2.2.3.4.2 Analisis Perbandingan	69
BAB 3 KESIMPULAN DAN SARAN	70
3.1 Kesimpulan	70
3.1 Saran	70
Pembagian Tugas	71
Referensi	72

BAB 1 DESKRIPSI PERSOALAN

Convolutional Neural Network (CNN) dan *Recurrent Neural Network* (RNN) merupakan dua arsitektur *neural network* yang umum digunakan dalam bidang pembelajaran mesin. CNN dirancang untuk mengenali pola spasial dalam data, seperti dalam klasifikasi gambar, sedangkan RNN dan variannya seperti LSTM digunakan untuk memproses data berurutan seperti teks atau sinyal waktu. Kedua arsitektur ini memiliki keunggulan dalam menangani konteks data yang berbeda sehingga banyak digunakan dalam berbagai aplikasi nyata.

Pada tugas ini, implementasi berfokus pada *forward propagation* dari CNN, Simple RNN, dan LSTM *from scratch*, yaitu tanpa bantuan library deep learning seperti TensorFlow atau PyTorch. Proses pelatihan model tetap dilakukan menggunakan library Keras, dengan tujuan agar hasil dari forward propagation yang diimplementasikan *from scratch* dapat divalidasi melalui perbandingan dengan hasil keluaran dari Keras. Validasi dilakukan dengan menggunakan data uji serta metrik evaluasi berupa macro F1-score.

Implementasi CNN dilakukan untuk menyelesaikan permasalahan klasifikasi citra dengan menggunakan dataset CIFAR-10. Arsitektur model harus mencakup beberapa jenis lapisan utama seperti Conv2D, pooling (maksimum atau rata-rata), flatten atau global pooling, dan dense. Model dilatih menggunakan fungsi loss *Sparse Categorical Crossentropy* dan *optimizer Adam*. Selain itu, dataset perlu dibagi menjadi tiga bagian, yaitu train, validation, dan test. Hasil pelatihan disimpan dan digunakan kembali dalam proses *forward propagation from scratch*.

Sementara itu, RNN dan LSTM diterapkan pada permasalahan klasifikasi teks dengan memanfaatkan dataset NusaX-Sentiment yang berisi data dalam Bahasa Indonesia. Proses awal mencakup tokenisasi teks menggunakan TextVectorization dan konversi token ke vektor menggunakan Embedding layer. Model kemudian dibangun dengan menggabungkan embedding layer, RNN atau LSTM (dengan variasi arah *unidirectional* dan *bidirectional*), *dropout layer*, dan *dense layer*. Hasil pelatihan model Keras digunakan untuk menguji implementasi *forward propagation from scratch*.

Evaluasi model dilakukan dengan membandingkan hasil keluaran antara model Keras dan model hasil implementasi mandiri. Selain itu, dilakukan juga eksperimen terhadap pengaruh berbagai hyperparameter, seperti jumlah lapisan konvolusi atau rekuren, jumlah filter atau *neuron* per lapisan, ukuran kernel, serta jenis arah lapisan rekuren. Seluruh hasil eksperimen dievaluasi menggunakan metrik macro F1-score dan visualisasi grafik training loss serta validation loss untuk setiap epoch guna memahami pengaruh konfigurasi terhadap performa model.

BAB 2 PEMBAHASAN

2.1 Penjelasan Implementasi

2.1.1 Deskripsi Kelas, Atribut, dan Method

Berikut adalah penjelasan dari kelas, atribut dari masing-masing kelas, dan method yang ada dalam masing-masing kelas.

2.1.1.1 CNN

Class CNN(cnn.py)

Kelas CNN Bertanggung jawab untuk melakukan prediction dan forward propagation dibantu dengan Weight yang sudah dihasilkan melalui Keras Weight Training yang dibuat sebelumnya. Class ini akan mengiterasi seluruh layer yang ada pada model keras dan melakukan perlakuan berdasarkan layer tersebut.

Atribut



```
1 def __init__(self):
2     self.layers = []
```

Nama Atribut	Penjelasan
self.layers	Menyimpan layer-layer yang digunakan oleh keras akan digunakan untuk prediction nantinya
Konstruktor	
Fungsi __init__ di atas hanya menginisiasi layers kosong yang akan diisi pada forward propagation dan digunakan ketika prediction.	
Fungsi/Prosedur	Penjelasan
<pre>def add_layer(self, layer_instance):</pre>	Metode add layer digunakan untuk menambahkan layer instance baru pada self.layers

<pre>def load_keras_model(self, keras_model_instance):</pre>	<p>Load keras model berfungsi menyimpan seluruh layer yang ada pada keras_model_instance pada self.layers, yang nantinya akan digunakan untuk prediction</p>
<pre>def predict_single(self, input_sample, verbose=False):</pre>	<p>Melakukan Forward propagation untuk memprediksi hasil sebuah input sample. Pada fungsi ini seluruh layers yang di-isi sebelumnya akan diapply pada input_sample dan dihasilkan sebuah hasil akhir</p>
<pre>def predict_batch(self, input_batch, model_name_tag="Scratch Model"):</pre>	<p>Melakukan Forward propagation untuk banyak input sekaligus.</p>
Class KerasModel (keras_model.py)	
<p>Kelas KerasModel adalah sebuah class yang digunakan untuk menghasilkan Weight dan Model Keras yang ada</p>	
Atribut	

```

● ● ●

1 def __init__(self, num_class, input_shape, random_seed = 42):
2     self.input_shape = input_shape
3     self.num_classes = num_class
4     self.random_seed = random_seed
5     self.model = None
6     self.history = None
7     self._set_seeds()

```

Nama Atribut	Penjelasan
input_shape	Shape atau bentuk dari input yang diberikan
num_classes	Digunakan secara spesifik untuk menangani kasus softmax dimana dibutuhkan jumlah class didalamnya
random_seed	Digunakan untuk set random seed
model	Nantinya digunakan untuk menyimpan model keras yang sudah dibuat
history	History menyimpan hasil modell yang sudah di train
Konstruktor	
Fungsi __init__ di atas juga sekaligus konstruktor yang menginisialisasi variables yang akan digunakan untuk menghasilkan model keras yang baik	
Fungsi/Prosedur	Penjelasan
<code>def _set_seeds(self):</code>	Set seed untuk randomisasi np dan tensorflow

<pre>def preprocess_data(self, x_train_full, y_train_full, x_test_orig, y_test_orig, val_split_ratio=0.2):</pre>	<p>Melakukan splitting data train menjadi data train dan data val</p>
<pre>def define_model(self, model_name, conv_blocks_config, global_pooling_type, dense_layers_config):</pre>	<p>Menghasilkan model berdasarkan konfigurasi yang sudah ditentukan sebelumnya dan menyimpannya pada self.model.</p>
<pre>def compile_model(self, optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy']):</pre>	<p>Melakukan compile model keras yang didefinisikan sebelumnya</p>
<pre>def train_model(self, epochs, batch_size, verbose=1):</pre>	<p>Melakukan training model keras dan menyimpannya pada history</p>
<pre>def evaluate_model(self, verbose=0):</pre>	<p>Melakukan prediksi pada model yang sudah dibuat dalam bentuk loss, accuracy, dan f1 score</p>
<pre>def save_model_weights(self, filepath):</pre>	<p>Menyimpan weight model pada filepath dalam extension h5</p>

```

def
plot_training_history(s
elf, experiment_name,
model_name,
base_save_path="experim
ent_results"):

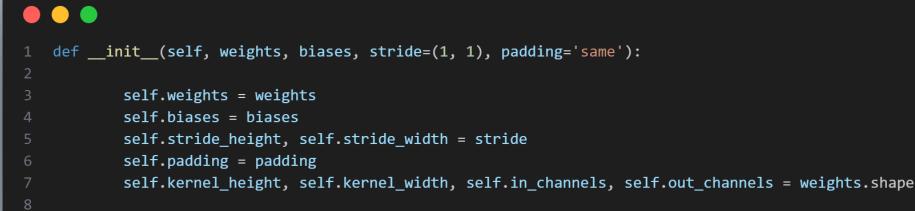
```

Menghasilkan gambar plot training yang berisikan loss dan accuracy pada tiap epochnya

Class Conv2DLayer (file : layers.py)

Sebuah class yang akan menghasilkan sebuah convolution layer.

Atribut



```

● ● ●
1 def __init__(self, weights, biases, stride=(1, 1), padding='same'):
2
3     self.weights = weights
4     self.biases = biases
5     self.stride_height, self.stride_width = stride
6     self.padding = padding
7     self.kernel_height, self.kernel_width, self.in_channels, self.out_channels = weights.shape
8

```

Nama Atribut	Penjelasan
weights	Kernel yang digunakan spesifik untuk Convolusi layer ini
biases	Menyimpan bias
stride_height	Menyimpan tinggi ukuran stride. Tidak menggunakan single value stride untuk menghandle kasus gambar yang tidak kotak
stride_width	Menyimpan panjang dari stride
kernel_height	Menyimpan tinggi kernel
kernel_width	Menyimpan panjang kernel

in_channels	Menyimpan ukuran channel dari input
out_channels	Menyimpan ukuran filter

Konstruktor

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi variables yang akan digunakan untuk forward propagation pada layer ini

Fungsi/Prosedur	Penjelasan
<pre>def forward(self, input_data):</pre>	<p>Melakukan forward untuk input data. Pada intinya, dilayer konvolusi ini akan dihasilkan terlebih dahulu output_data (V) berdasarkan ukuran padding dan stride yang ada.</p> <p>Selanjutnya akan dilakukan perhitungan untuk masing masing kernel, dihasilkan nilai yang kemudian digabungkan menghasilkan sebuah output data</p>

Class ReLULayer (file : layers.py)

Sebuah class berisikan 1 fungsi yang akan menghasilkan layer yang diberi aktivasi function ReLU

Fungsi/Prosedur	Penjelasan
<pre>def forward(self, input_data):</pre>	Menghasilkan output_data berupa input_data tanpa nilai negatif

Class MaxPooling2DLayer(file : layers.py)

Sebuah class yang akan mengecilkan ukuran layer dengan nilai maksimal pada tiap feature map nya

Atribut

```
1 class MaxPooling2DLayer:  
2     def __init__(self, pool_size=(2, 2), stride=None):  
3         self.pool_height, self.pool_width = pool_size  
4         self.stride_height, self.stride_width = stride if stride is not None else pool_size
```

Nama Atribut	Penjelasan
Pool_height, pool_width	Menyimpan ukuran pooling
stride_height, stride_width	Menyimpan ukuran stride

Konstruktor

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi variables yang akan digunakan untuk forward layers pada class ini

Fungsi/Prosedur	Penjelasan
<code>def forward(self, input_data):</code>	Melakukan forward untuk input data. Pada intinya akan dihasilkan sebuah feature map oleh kernel dengan ukuran pooling. Setelah melakukan dot product antara pool dengan input, selanjutnya didapatkan nilai maksimal dari feature map nya yang akan digabungkan menjadi 1 output

Class AveragePooling2DLayer(file : layers.py)

Sebuah class yang akan mengecilkan ukuran layer dengan nilai rata pada tiap feature map nya

Atribut

```
1 class MaxPooling2DLayer:  
2     def __init__(self, pool_size=(2, 2), stride=None):  
3         self.pool_height, self.pool_width = pool_size  
4         self.stride_height, self.stride_width = stride if stride is not None else pool_size
```

Nama Atribut	Penjelasan
Pool_height, pool_width	Menyimpan ukuran pooling
stride_height, stride_width	Menyimpan ukuran stride

Konstruktor

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi variables yang akan digunakan untuk forward layers pada class ini

Fungsi/Prosedur	Penjelasan
<code>def forward(self, input_data):</code>	Melakukan forward untuk input data. Pada intinya akan dihasilkan sebuah feature map oleh kernel dengan ukuran pooling. Setelah melakukan dot product antara pool dengan input, selanjutnya didapatkan nilai rata-rata dari feature map nya yang akan digabungkan menjadi 1 output

Class FlattenLayer(file : layers.py)

Sebuah class yang akan memetakan input menjadi sebuah vektor 1 dimensi

Fungsi/Prosedur	Penjelasan
-----------------	------------

<pre>def forward(self, input_data):</pre>	Melakukan flatten pada input_data
Class GlobalAveragePooling2DLayer(file : layers.py)	
Sebuah class yang akan menghasilkan nilai average untuk vector 1 dim	
Fungsi/Prosedur	Penjelasan
<pre>def forward(self, input_data):</pre>	Menghasilkan nilai rata rata dari tiap elemen pada input_data
Class DenseLayer(file : layers.py)	
Sebuah class yang akan mengecilkan ukuran layer dengan nilai rata pada tiap feature map nya	
Atribut	
 <pre>1 class DenseLayer: 2 def __init__(self, weights, biases): 3 self.weights = weights 4 self.biases = biases</pre>	
Nama Atribut	Penjelasan

weights	Menyimpan bobot
biases	Menyimpan bias

Konstruktor

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi variables yang akan digunakan untuk forward layers pada class ini

Fungsi/Prosedur	Penjelasan
<pre>def forward(self, input_data):</pre>	Melakukan forward dengan melakukan dot product antara input-data dengan weights + bias

Class SoftmaxLayer(file : layers.py)

Sebuah class yang akan melakukan activation function softmax pada `input_data`

Fungsi/Prosedur	Penjelasan
<pre>def forward(self, input_data):</pre>	Melakukan softmax activation function pada <code>input_data</code>

Class TanhLayer(file : layers.py)

Sebuah class yang akan melakukan activation function yaitu Tanh pada `input_data`

Fungsi/Prosedur	Penjelasan
<pre>def forward(self, input_data):</pre>	Melakukan tanh activation function pada <code>input_data</code>

Class SigmoidLayer(file : layers.py)	
Sebuah class yang akan melakukan activation function yaitu sigmoid pada input_Data	
Fungsi/Prosedur	Penjelasan
<pre>def forward(self, input_data):</pre>	Melakukan sigmoid activation function pada input_data

2.1.1.2 RNN

Class TextClassifier (text_classification.py)	
<p>Kelas TextClassifier bertanggung jawab untuk menangani seluruh alur klasifikasi teks menggunakan model Recurrent Neural Network (RNN) yang dibangun dengan library Keras, mulai dari pengunduhan dan pemrosesan dataset, vektorisasi teks, pembangunan dan pelatihan model, hingga evaluasi performa. Kelas ini juga dirancang untuk melakukan eksperimen dengan membandingkan berbagai konfigurasi hyperparameter, seperti jumlah layer RNN, jumlah unit per layer, dan arah propagasi (bidirectional atau tidak), guna menemukan model dengan performa terbaik dalam tugas klasifikasi sentimen teks.</p>	
Atribut	

```

1  def __init__(self, data_dir="data"):
2      self.data_dir = data_dir
3
4      os.makedirs(data_dir, exist_ok=True)
5
6      self.train_file = os.path.join(data_dir, "train.csv")
7      self.valid_file = os.path.join(data_dir, "valid.csv")
8      self.test_file = os.path.join(data_dir, "test.csv")
9
10     self.train_df = None
11     self.valid_df = None
12     self.test_df = None
13     self.label_map = None
14
15     self.layer_vectorize = None
16     self.model = None
17     self.history = None

```

Nama Atribut	Penjelasan
self.data_dir	Direktori tempat file data disimpan dan diakses
self.train_file	Path lengkap ke file train.csv di dalam data_dir
self.valid_file	Path lengkap ke file valid.csv di dalam data_dir
self.test_file	Path lengkap ke file test.csv di dalam data_dir
self.train_df	DataFrame yang menyimpan data pelatihan setelah dibaca dari CSV
self.valid_df	DataFrame yang menyimpan data validasi setelah dibaca dari CSV
self.test_df	DataFrame yang menyimpan data pengujian setelah dibaca dari CSV
self.label_map	Dictionary yang memetakan label string menjadi nilai numerik

self.layer_vectorize	Layer TextVectorization dari TensorFlow untuk memproses teks input
self.model	Objek model neural network yang akan dibangun dan dilatih
self.history	Objek histori hasil pelatihan model, berisi loss dan akurasi per epoch
Konstruktor	
Fungsi <code>__init__</code> di atas juga sekaligus merupakan konstruktor yang menginisialisasi berbagai atribut penting seperti direktori data, lokasi file dataset, dan struktur dasar untuk menyimpan data, model, serta histori pelatihan, sehingga model siap digunakan untuk proses pemuatan data, pelatihan, evaluasi, hingga penyimpanan hasil klasifikasi teks secara terstruktur dan otomatis.	
Fungsi/Prosedur	Penjelasan
<code>def __init__(self, data_dir="data"):</code>	Metode <code>__init__</code> bertanggung jawab untuk menginisialisasi atribut-atribut utama seperti direktori data, path file dataset, serta objek model dan data, memastikan bahwa objek TextClassifier siap digunakan dalam proses klasifikasi teks.
<code>def download_data(self):</code>	Metode <code>download_data</code> bertanggung jawab untuk mengunduh dataset pelatihan, validasi, dan pengujian dari sumber online, memastikan file yang dibutuhkan tersedia secara lokal sebelum digunakan.

<pre><code>def load_data(self):</code></pre>	<p>Metode load_data bertanggung jawab untuk me-load file CSV ke dalam DataFrame dan memetakan label string ke integer, memastikan data siap digunakan untuk pelatihan dan evaluasi.</p>
<pre><code>def preprocess_data(self):</code></pre>	<p>Metode preprocess_data bertanggung jawab untuk membuat dan melatih layer TextVectorization, serta menyimpan vocabulary dan mengonversi label menjadi ID numerik, memastikan teks dapat direpresentasikan dalam format numerik yang sesuai.</p>
<pre><code>def create_datasets(self):</code></pre>	<p>Metode create_datasets bertanggung jawab untuk mengonversi DataFrame menjadi objek tf.data.Dataset yang telah dibatch dan diprefetch, memastikan data dapat diakses secara efisien selama proses pelatihan dan evaluasi.</p>
<pre><code>def build_model(self, rnn_layers=1, rnn_units=64, bidirectional=True):</code></pre>	<p>Metode build_model bertanggung jawab untuk membangun arsitektur model RNN berdasarkan parameter konfigurasi seperti jumlah layer dan arah jaringan, memastikan model siap dilatih dengan struktur yang sesuai.</p>

```
def train_model(self,
model_name="rnn_model"):
:
```

Metode train_model bertanggung jawab untuk melatih model menggunakan dataset yang telah diproses, dengan bantuan callback seperti early stopping dan checkpoint, menghasilkan model yang optimal tanpa overfitting.

```
def
evaluate_model(self):
:
```

Metode evaluate_model bertanggung jawab untuk melakukan berbagai eksperimen kemudian membandingkan dan memilih model terbaik dari keseluruhan eksperimen.

```
def
plot_loss_comparison(hi
stories, model_names,
save_path='images/loss_
comparison.png'):
:
```

Metode plot_loss_comparison bertanggung jawab untuk memvisualisasikan dan membandingkan nilai loss selama proses pelatihan dan validasi dari berbagai model, memastikan bahwa performa relatif antar model dapat dilihat secara visual dan objektif.

```
def plot_history(self,
model_name="rnn_model"):
:
```

Metode plot_history bertanggung jawab untuk memvisualisasikan riwayat akurasi dan loss selama proses pelatihan untuk satu model tertentu, menghasilkan grafik yang memberikan gambaran perkembangan performa model dari epoch ke epoch.

```
def run_experiments():
```

Metode `run_experiments` bertanggung jawab untuk menjalankan serangkaian eksperimen pelatihan dan evaluasi model dengan berbagai konfigurasi RNN, memastikan perbandingan kinerja antar model dilakukan secara sistematis dan hasilnya disimpan serta ditampilkan dengan jelas.

Class RNNFromScratch (forward_propagation.py)

Kelas `RNNFromScratch` bertanggung jawab untuk melakukan proses forward propagation dari model Recurrent Neural Network (RNN) secara *from scratch* berdasarkan bobot yang diekstraksi dari model Keras yang sudah dilatih. Kelas ini juga yang bertanggungjawab melakukan perbandingan antara prediksi dari model Keras dan implementasi *from scratch*.

Atribut

```

1 def __init__(self, model_path, hidden_activation='tanh', output_activation='tanh'):
2     self.model_path = model_path
3     self.hidden_activation = hidden_activation
4     self.output_activation = output_activation
5
6     # Load Keras model
7     self.keras_model = load_model(model_path)
8
9     # Mengekstrak bobot dan config dari model Keras
10    self.weights = {}
11    self.configs = {}
12    self.extract_weights()
13
14    # Inisialisasi fungsi aktivasi
15    self.activation_functions = {
16        'tanh': lambda x: np.tanh(x),
17        'sigmoid': lambda x: 1 / (1 + np.exp(-x)),
18        'relu': lambda x: np.maximum(0, x),
19        'softmax': lambda x: np.exp(x) / np.sum(np.exp(x), axis=-1, keepdims=True)
20    }

```

Nama Atribut	Penjelasan
model_path	Path ke file .h5 model Keras yang akan diekstraksi
hidden_activation	Nama fungsi aktivasi yang digunakan pada layer RNN (misalnya 'tanh', 'sigmoid', 'relu')
output_activation	Nama fungsi aktivasi untuk layer output (umumnya 'softmax' atau 'tanh')
keras_model	Objek model Keras yang dimuat dari model_path
weights	Dictionary yang menyimpan bobot layer dari model Keras (kernel, recurrent_kernel, dan bias)
configs	Dictionary berisi konfigurasi setiap layer, seperti jumlah unit, jenis aktivasi, dan apakah return_sequences
activation_functions	Dictionary berisi fungsi-fungsi aktivasi yang tersedia dan dapat dipanggil secara dinamis
Konstruktor	

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi objek, memuat model Keras dari file, mengekstrak bobot dan konfigurasi layer, serta menyiapkan fungsi aktivasi yang akan digunakan..

Fungsi/Prosedur	Penjelasan
<pre data-bbox="393 523 796 777"><code>def __init__(self, model_path, hidden_activation='tanh', output_activation='tanh'): </code></pre>	<p>Metode ini bertanggung jawab untuk menginisialisasi objek, memuat model Keras dari file, mengekstrak bobot dan konfigurasi layer, serta menyiapkan fungsi aktivasi yang akan digunakan. Konstruktor ini memastikan objek siap melakukan forward pass dari input mentah.</p>
<pre data-bbox="393 975 768 1060"><code>def extract_weights(self): </code></pre>	<p>Metode ini bertanggung jawab untuk mengekstrak bobot (kernel, recurrent_kernel, dan bias) serta konfigurasi dari layer-layer SimpleRNN dan Dense dalam model Keras, memastikan seluruh parameter siap digunakan untuk komputasi manual.</p>
<pre data-bbox="393 1229 768 1313"><code>def forward_pass(self, inputs): </code></pre>	<p>Metode ini bertanggung jawab untuk melakukan forward propagation secara manual terhadap input tensor, lapis demi lapis sesuai bobot yang telah diekstrak, menghasilkan output akhir dari jaringan seperti prediksi probabilistik kelas.</p>
<pre data-bbox="393 1482 796 1607"><code>def load_and_preprocess_test_data(data_path): </code></pre>	<p>Metode ini bertanggung jawab untuk memuat data uji dari file CSV dan mengubah label teks menjadi ID numerik, memastikan data uji siap digunakan untuk evaluasi model.</p>
<pre data-bbox="393 1700 796 1875"><code>def preprocess_text(texts, max_sequence_length=100): </code></pre>	<p>Metode ini bertanggung jawab untuk melakukan tokenisasi sederhana pada teks, memetakan token ke indeks, dan memastikan setiap teks direpresentasikan sebagai urutan angka dengan panjang seragam.</p>

<pre>def compare_predictions(keras_model, scratch_model, test_data):</pre>	Metode ini bertanggung jawab untuk menghasilkan prediksi dari model Keras dan model RNNFromScratch, membandingkan hasilnya, serta menghitung metrik evaluasi seperti akurasi dan F1-score, memastikan kualitas prediksi dari kedua pendekatan dapat diukur dan dianalisis.
<pre>def visualize_comparison(metrics):</pre>	Metode ini bertanggung jawab untuk memvisualisasikan perbandingan metrik performa (seperti akurasi dan F1-score) antara model Keras dan model manual, menghasilkan grafik yang memperjelas perbedaan performa antar model.
<pre>def main():</pre>	Metode ini bertanggung jawab untuk menjalankan seluruh pipeline dari pencarian model terbaik, pemuatan model, pemrosesan data, perbandingan prediksi, hingga visualisasi hasil, memastikan seluruh proses eksperimen dan analisis dapat dilakukan secara otomatis dan menyeluruh.

2.1.1.3 LSTM

Class LSTMPreprocess (preprocessing.py)
Kelas LSTMPreprocess bertanggung jawab untuk melakukan preprocessing terhadap data, seperti untuk melakukan tokenisasi atau embedding. Kelas ini memungkinkan pengguna untuk merubah input berupa susunan kata menjadi bentuk array of integer yang kemudian dapat dilakukan embedding supaya dapat menyesuaikan ukuran input pada model keras.
Atribut



```
1 def __init__(self):
2     self.embedding_layer = keras.Sequential()
3     self.vectorize_layer = None
```

Nama Atribut	Penjelasan
self.embedding_layer	Atribut ini memiliki fungsi untuk menyimpan model keras sequential yang nantinya akan diisi dengan embedding model dari keras
self.vectorize_layer	Atribut ini memiliki fungsi untuk menyimpan Text Vectorization Model yang nantinya hasil train dari model ini akan berguna untuk mendapatkan ukuran vocabulary
Konstruktor	
Fungsi <code>__init__</code> di atas juga sekaligus konstruktor yang menginisialisasi layer embedding dengan model keras sequential kosong yang nantinya akan ditambahkan	

model embedding didalamnya dan layer vectorization sebagai variabel kosong untuk nantinya digunakan sebagai tempat penyimpanan model yang telah di train.

Fungsi/Prosedur	Penjelasan
<pre data-bbox="393 494 796 544"><code>def vectorization(...):</code></pre>	<p>Metode ini bertanggung jawab untuk membuat model text vectorization yang akan disimpan pada self.vectorization_layer dan juga melakukan vectorization kepada string input</p>
<pre data-bbox="393 817 796 868"><code>def embedding(...):</code></pre>	<p>Metode ini bertanggung jawab untuk menambahkan layer embedding kedalam model sequential dan melakukan embedding kepada token yang telah dihasilkan pada vectorization</p>
Class BidirectionalScratch (bidirectional.py)	
<p>Kelas BidirectionalScratch bertanggung jawab untuk menjadi kelas wrapper untuk model LSTM didalamnya. Kelas ini berfungsi untuk menggandakan proses foward LSTM dengan salah satunya dari arah terbalik.</p>	
Atribut	

```

● ● ●

1 def __init__(self, units, all_weights):
2     # all_weights adalah list [W_f, U_f, b_f, W_b, U_b, b_b]
3     self.forward_lstm = LSTMLayerScratch(units, all_weights[:3])
4     self.backward_lstm = LSTMLayerScratch(units, all_weights[3:])

```

Nama Atribut	Penjelasan
forward_lstm	Model untuk LSTM yang akan melakukan forward dari depan ke belakang
backward_lstm	Model untuk LSTM yang akan melakukan forward dari belakang ke depan
Konstruktor	
Fungsi __init__ di atas juga sekaligus konstruktor yang menginisialisasi model LSTM untuk maju dan LSTM untuk mundur digunakan dalam proses <i>forward</i> .	
Fungsi/Prosedur	Penjelasan
<code>def forward(self, x):</code>	Metode ini bertanggung jawab melakukan forward propagation kepada dua jenis arah model.
Class DenseScratch (dense.py)	
Kelas DenseScratch bertanggung jawab sebagai layer dense / fully connected layer.	
Atribut	



```

1 def __init__(self, weights, activation_name=None):
2     # weights adalah list [W, b]
3     self.W, self.b = weights
4     self.activation_name = activation_name

```

Nama Atribut	Penjelasan
W	Bobot dari masing-masing neuron
b	Bias dari masing-masing neuron
activation_name	Fungsi aktivasi yang akan digunakan untuk mencegah non-linearity

Konstruktor	
Fungsi/Prosedur	Penjelasan
<code>def forward(self, x):</code>	Metode ini bertanggung jawab untuk melakukan forward propagation seperti pada layer FFNN dan menentukan fungsi aktivasi yang akan digunakan.

Class EmbeddingScratch (dense.py)
--

Kelas EmbeddingScratch bertanggung jawab untuk melakukan embedding terhadap token.

Atribut

```
● ● ●
1 def __init__(self, weights):
2     self.embedding_matrix = weights[0]
3     # self.embedding_matrix[0] = np.zeros_like(self.embedding_matrix[0])
4
```

Nama Atribut	Penjelasan
embedding_matrix	Embedding Value dari masing-masing token

Konstruktor

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi matrix embedding yang digunakan dalam proses *embedding*.

Fungsi/Prosedur	Penjelasan
<code>def forward(self, x):</code>	Metode ini bertanggung jawab untuk melakukan embedding masukan token.

File fungsi_aktivasi.py

Fungsi/Prosedur	Penjelasan

<code>def sigmoid(x) :</code>	Fungsi ini bertanggung jawab untuk melakukan perhitungan fungsi aktivasi sigmoid
<code>def tanh(x) :</code>	Fungsi ini bertanggung jawab untuk melakukan perhitungan fungsi aktivasi tanh
<code>def softmax(x) :</code>	Fungsi ini bertanggung jawab untuk melakukan perhitungan fungsi aktivasi softmax
<code>def relu(x) :</code>	Fungsi ini bertanggung jawab untuk melakukan perhitungan fungsi aktivasi relu
Kelas LSTMLayerScratch (lstm.py)	
Kelas LSTMLayerScratch bertanggung jawab sebagai layer LSTM yang melakukan perhitungan pada 4 gate, h, dan output.	
Atribut	

```

● ● ●

1 def __init__(self, units, weights):
2     # weights adalah list [W, U, b]
3     W, U, b = weights
4     self.units = units
5
6     # Membagi matriks bobot menjadi 4 gate (input, forget, cell, output)
7     self.W_i = W[:, :units]
8     self.W_f = W[:, units:units*2]
9     self.W_c = W[:, units*2:units*3]
10    self.W_o = W[:, units*3:]
11
12    self.U_i = U[:, :units]
13    self.U_f = U[:, units:units*2]
14    self.U_c = U[:, units*2:units*3]
15    self.U_o = U[:, units*3:]
16
17    self.b_i = b[:units]
18    self.b_f = b[units:units*2]
19    self.b_c = b[units*2:units*3]
20    self.b_o = b[units*3:]

```

Nama Atribut	Penjelasan
W	Matrix Bobot recurrent dari Trained LSTM Model
U	Matrix Bobot input dari Trained LSTM Model
b	Matrix Bias dari Trained LSTM Model
units	Jumlah LSTM unit yang digunakan
W_i	Bobot recurrent gate input
W_f	Bobot recurrent gate forget
W_c	Bobot recurrent cell state
W_o	Bobot recurrent gate output
U_i	Bobot input gate input
U_f	Bobot input gate forget
U_c	Bobot input cell state
U_o	Bobot input gate output

b_i	Bias gate input
b_f	Bias gate forget
b_c	Bias cell state
b_o	Bias gate output

Konstruktor

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi bobot dan bias pada model LSTM yang akan digunakan dalam proses *forward*.

Fungsi/Prosedur	Penjelasan
<code>def forward(self, x):</code>	Metode ini bertanggung jawab untuk melakukan perhitungan masing-masing gate, menghitung cell state terbaru, net, and output
Kelas ModelScratch (sequential.py)	
Kelas ModelScratch bertanggung jawab sebagai model sequential yang akan digunakan untuk menjalankan masing-masing layer satu per satu secara berurutan.	
Atribut	



```
1 def __init__(self, layers):  
2     self.layers = layers
```

Nama Atribut	Penjelasan
layers	Layer-layer yang terurut yang akan dijalankan secara berurutan nantinya.

Konstruktor

Fungsi `__init__` di atas juga sekaligus konstruktor yang menginisialisasi layer yang akan dijalankan secara berurutan nantinya digunakan dalam proses *forward*.

Fungsi/Prosedur	Penjelasan
<code>def predict(self, x):</code>	Metode ini bertanggung jawab untuk menjalankan masing-masing layer satu per satu secara berurutan

2.1.2 CNN

Convolutional Neural Network (CNN) merupakan arsitektur neural network yang dirancang untuk menangani data berbentuk gambar atau data yang memiliki bentuk diatas 1 dimensi, hal ini karena CNN mampu menangkap sebuah pattern yang ada pada sebuah gambar. Berbeda dengan FFNN dimana pada FFNN sebuah gambar biasanya akan di mapping menjadi sebuah vektor yang sangat panjang. Pada CNN ini sebuah input akan diproses secara langsung dengan menggunakan konsep local connectivity menggunakan kernel dengan ukuran filter tertentu. Kelebihan dari metode ini, selain dari kecepatan dan pengurangan jumlah kalkulasi yang perlu dilakukan. Metode ini juga dapat menangkap pattern sehingga mampu mengklasifikasikan gambar dengan baik.

Pada penggerjaanya, setelah dilakukan embedding weight oleh keras. Untuk masing-masing layer yang ada pada keras, akan di proses pada class CNN untuk kemudian dihasilkan output akhir yang digunakan untuk prediction. Untuk memahami ini lebih lanjut, kita perlu memahami beberapa layer yang ada pada CNN, diantaranya

1. Convolution Layer

- Ini adalah inti dari CNN. Lapisan ini menggunakan kernel untuk melakukan operasi konvolusi pada gambar input.
- Untuk input data yang ada akan dilakukan pengambilan feature atau feature extracting dengan kernel, stride, dan padding yang sudah ditentukan. Nantinya feature tersebut akan ditambahkan dengan bias dan menghasilkan sebuah output terbaru
- Setiap kernel menghasilkan sebuah "feature map" yang menunjukkan di mana fitur tertentu terdeteksi.

2. Detector Layer

- Detector layer ini merupakan layer yang hanya mengaplikasikan sebuah activation function tertentu pada input data
- Terdapat beberapa activation function yang digunakan seperti softmax, ReLu, tanh, dan sigmoid
- Pada layer ini tidak terjadi pengurangan atau penambahan ukuran input

3. Pooling Layer

- Lapisan ini bertugas mengurangi ukuran fitur. Tujuannya untuk mengurangi kompleksitas komputasi dan membuat model lebih invariant pada perubahan kecil pada posisi objek
- Terdapat 2 tipe pooling yang digunakan yaitu Max Pooling dengan mengambil nilai maksimal dari feature map dan Average Pooling yang mengambil nilai rata-rata dari feature map yang ada.

4. Dense Layer

- Ini merupakan layer yang biasanya berada di akhir. Dense layer ini akan menghubungkan neuron saat ini dengan semua neuron di layer sebelumnya
- Sebelum dense layer ini, akan dilakukan flattening terlebih dahulu dimana data yang masih berbentuk 3d ataupun 2d akan di flatten menjadi vektor dengan ukuran 1 dimensi
- Dense layer ini berfungsi untuk melakukan kualifikasi berdasarkan fitur-fitur yang sudah diekstraksi oleh lapisan-lapisan konvolusi dan pooling

2.1.3 RNN

Simple Recurrent Neural Network (Simple RNN) merupakan arsitektur *neural network* yang dirancang untuk menangani data sekuensial, di mana urutan data mempengaruhi hasil prediksi. Berbeda dengan feedforward neural network (FFNN) yang mengasumsikan input tidak memiliki urutan, RNN mempertimbangkan urutan dan konteks dari setiap elemen dalam *sequence* melalui mekanisme *feedback* pada *hidden state*. Hal ini menjadikannya sangat cocok untuk tugas-tugas seperti analisis sentimen, pemodelan bahasa, dan prediksi *time-series*.

Sebelum masuk ke proses forward propagation, input teks terlebih dahulu diproses menggunakan fungsi `preprocess_text(...)`. Pada fungsi ini, teks diubah menjadi token, lalu setiap token dikonversi menjadi indeks integer sesuai dengan `vocab`, dan hasilnya adalah tensor NumPy berdimensi (`batch_size`, `sequence_length`).

```
sequence = [vocab.get(token, vocab["<UNK>"]) for token in tokens]
```

Selanjutnya, hasil tokenisasi ini digunakan sebagai input untuk layer embedding.

Embedding layer adalah representasi awal dari setiap token dalam bentuk vektor berdimensi tetap. Dalam versi *from scratch*, bobot *embedding* diproses dengan cara sebagai berikut:

```
embedding_weights = embedding_layer.get_weights()[0]
embedded_input = embedding_weights[vectorized_texts]
```

Hasilnya berupa tensor (`batch_size`, `sequence_length`, `embedding_dim`), yang menjadi input untuk layer berikutnya.

Setelah embedding, input akan masuk ke layer RNN. Hidden state diinisialisasi terlebih dahulu:

```
h = np.zeros((batch_size, units))
```

Dalam implementasi forward propagation *from scratch*, proses dimulai dengan memuat model hasil pelatihan menggunakan Keras. Model ini digunakan untuk mengekstrak bobot dari layer Simple RNN dan Dense agar dapat digunakan kembali dalam implementasi manual. Setiap bobot yang diambil terdiri dari:

- kernel atau U : bobot untuk input terhadap hidden layer,
- recurrent_kernel atau W_h : bobot untuk hidden state terhadap hidden state berikutnya,
- bias atau b : bias untuk hidden state

Semua bobot ini direpresentasikan dalam bentuk array NumPy dan disimpan dalam struktur self.weights. Selain itu, informasi konfigurasi layer seperti jumlah unit (neurons), jenis aktivasi, dan apakah layer mengembalikan seluruh sequence atau hanya output akhir (return_sequences) juga disimpan dalam self.configs.

Proses *forward propagation* dimulai dengan mengambil input berukuran (batch_size, sequence_length, embedding_dim). Setiap input pada timestep t, yaitu x_t , diproses secara berurutan. Pada setiap langkah waktu, hidden state h dihitung berdasarkan hidden state sebelumnya dan input saat ini x_t , melalui persamaan:

$$h_t = f(Ux_t + (Wh_{t-1} + b_{xh}))$$

dengan

- U = bobot untuk input terhadap hidden layer,
- W = bobot untuk hidden state terhadap hidden state,
- X_t = input
- h_{t-1} = hidden state timestamp sebelumnya
- b_{xh} = bias untuk input terhadap hidden state

Dilakukan iterasi per timestep untuk menghitung hidden state berdasarkan input saat ini dan hidden state sebelumnya. Dalam kode direpresentasikan dengan:

```
for t in range(sequence_length):
    x_t = inputs[:, t, :]
    h = self.apply_activation(
```

```
    np.dot(x_t, W_x) + np.dot(h, W_h) + b,  
    self.hidden_activation  
)
```

Hidden state awal h_0 biasanya diinisialisasi dengan nol. Kemudian pada setiap timestep, hidden state baru dihitung dan jika `return_sequences=True`, seluruh hidden state dari setiap timestep disimpan sebagai output sequence berdimensi tiga. Jika `return_sequences=False`, maka hanya h_t (hidden state terakhir) yang digunakan sebagai output dan diteruskan ke layer berikutnya.

Dalam kode direpresentasikan dengan:

```
if layer_config['return_sequences']:  
    outputs.append(h)
```

Setelah hidden state terakhir diperoleh dari layer RNN, hasilnya dimasukkan ke layer Dense yang melakukan klasifikasi. Fungsi aktivasi yang digunakan fungsi aktivasi softmax.

```
dense_net = np.dot(inputs, W) + b  
predictions = self.apply_activation(dense_net, 'softmax')
```

Setelah melalui seluruh tahapan, mulai dari embedding hingga perhitungan output pada Dense layer, fungsi `forward_pass()` akan menghasilkan prediksi akhir berupa probabilitas untuk setiap kelas. Nilai ini merupakan hasil dari fungsi aktivasi softmax, yang memastikan bahwa setiap elemen dalam vektor output berada pada rentang 0 hingga 1, dan totalnya berjumlah 1.

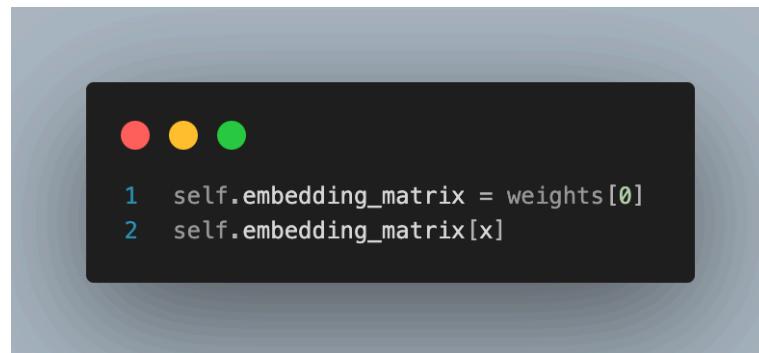
2.1.4 LSTM

Long Short-Term Memory (LSTM) merupakan arsitektur *recurrent neural network* (RNN) yang lebih canggih, dirancang khusus untuk mengatasi masalah *vanishing* dan *exploding gradient* yang sering terjadi pada Simple RNN. Ini memungkinkan LSTM untuk mempelajari dependensi jangka panjang dalam data sekuensial secara lebih efektif. Seperti Simple RNN, LSTM mempertimbangkan urutan dan konteks dari setiap elemen dalam *sequence* melalui mekanisme *feedback*, namun dengan struktur internal yang lebih kompleks yang melibatkan **cell state** dan beberapa **gates** (lupa/forget, masuk/input,

keluar/output). Hal ini menjadikannya sangat cocok untuk tugas-tugas seperti terjemahan mesin, pemodelan bahasa yang kompleks, dan prediksi *time-series* jangka panjang.

Sebelum masuk ke proses *forward propagation*, input teks terlebih dahulu diproses menggunakan fungsi tokenization(...). Pada fungsi ini, teks diubah menjadi token, lalu setiap token dikonversi menjadi indeks integer sesuai dengan **vocab**, dan hasilnya adalah tensor berdimensi (batch_size, max_len).

Selanjutnya, hasil tokenisasi ini digunakan sebagai input untuk layer embedding. Embedding layer adalah representasi awal dari setiap token dalam bentuk vektor berdimensi tetap. Dalam versi *from scratch*, bobot *embedding* diproses dengan cara sebagai berikut:



Hasilnya berupa tensor (batch_size, sequence_length, output_length), yang menjadi input untuk layer berikutnya.

Setelah embedding, input akan masuk ke beberapa layer sebagai berikut:

1. Layer Bidirectional

- Pada layer ini, input data digandakan dan salah satunya dilakukan *reverse* untuk mensimulasikan proses forward dengan arah yang terbalik. Hal ini dilakukan dengan cara sebagai berikut:

```
x = x[:, ::-1, :]
```

- Bobot dari model kedua (arah berlawanan) juga dibalik dengan cara sebagai berikut:

```
all_weights[3:]
```

- Kemudian kedua model LSTM (yang berbeda arah) dijalankan.
- Hasil dari kedua model kemudian di concat/gabungkan sehingga outputnya memiliki dimensi (**batch_size, 2 * units**)

2. Layer LSTM

- Pada layer ini, dilakukan perhitungan untuk masing-masing gate, cell state, dan output.
- Perhitungan tersebut dilakukan dengan cara berikut:

```

● ● ●

1 # Input Gate (i_t), Forget Gate (f_t), Output Gate (o_t), dan Cell Gate (c_tilde_t)
2 i_t = sigmoid(np.dot(x_t, self.W_i) + np.dot(h_t, self.U_i) + self.b_i)
3 f_t = sigmoid(np.dot(x_t, self.W_f) + np.dot(h_t, self.U_f) + self.b_f)
4 o_t = sigmoid(np.dot(x_t, self.W_o) + np.dot(h_t, self.U_o) + self.b_o)
5 c_tilde_t = tanh(np.dot(x_t, self.W_c) + np.dot(h_t, self.U_c) + self.b_c)
6
7 # Hitung cell state dan hidden state baru
8 c_t = f_t * c_tilde_t + i_t * c_tilde_t
9 h_t = o_t * tanh(c_t)

```

Rumus yang digunakan adalah sebagai berikut:

$$\begin{aligned}
ft &= \sigma(U_f \cdot x_t + W_f \cdot h(t-1) + b_f) \\
it &= \sigma(U_i \cdot x_t + W_i \cdot h(t-1) + b_i) \\
ot &= \sigma(U_o \cdot x_t + W_o \cdot h(t-1) + b_o) \\
c_{\sim t} &= \tanh(U_c \cdot x_t + W_c \cdot h(t-1) + b_c) \\
\\
ct &= ft * ct + it * c_{\sim t} \\
ht &= ot * \tanh(ct)
\end{aligned}$$

- Perhitungan ini dilakukan per time step sehingga memerlukan loop sebagai berikut:

```

● ● ●

1 # Iterasi melalui setiap timestep
2 for t in range(seq_len):
3     x_t = x[:, t, :]
4
5     # Perhitungan gate
6     # ...

```

3. Layer Dense

- Pada layer ini, behavior layer kurang lebih sama seperti layer FFNN, yaitu melakukan perhitungan sebagai berikut:

```
z = np.dot(x, self.W) + self.b
```

- Kemudian z (net) akan dilakukan aktivasi sesuai dengan fungsi aktivasi yang digunakan.

```
if self.activation_name == 'relu':  
    return relu(z)  
elif self.activation_name == 'softmax':  
    return softmax(z)  
else:  
    return z
```

2.2 Hasil Pengujian

2.2.1 CNN

2.2.1.1 Pengaruh Jumlah Konvolusi Layer

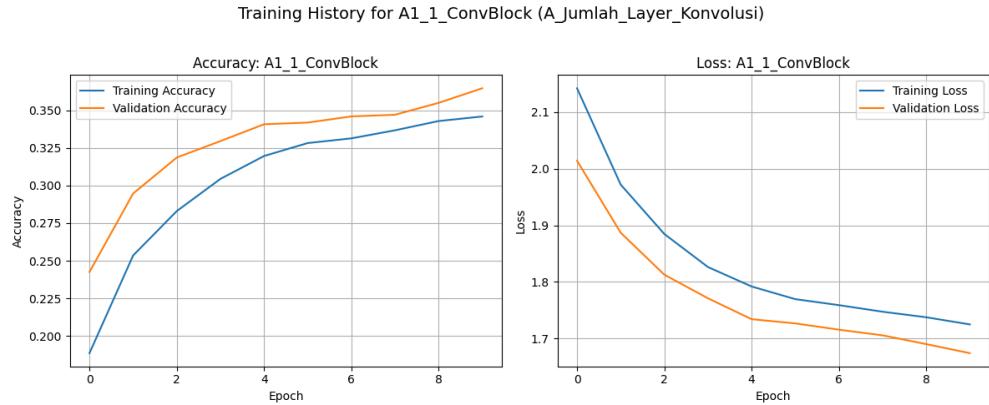
2.2.1.1.1 Hasil Akhir Prediksi

Variasi jumlah konvolusi layer yang digunakan adalah konvolusi 1 layer, 2 layer, dan 3 layer.

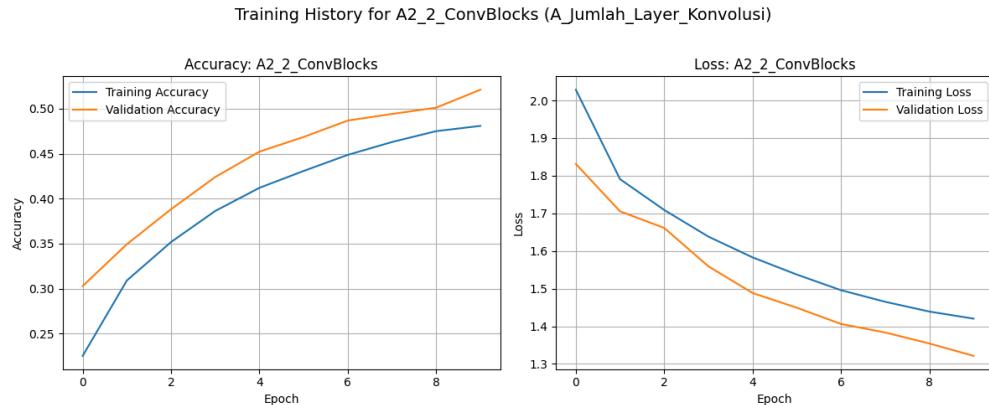
	model_name	f1_keras	f1_scratch
0	A1_1_ConvBlock	0.440476	0.440476
1	A2_2_ConvBlocks	0.476779	0.476779
2	A3_3_ConvBlocks	0.691220	0.691220

2.2.1.1.2 Grafik Training Loss dan Validation Loss

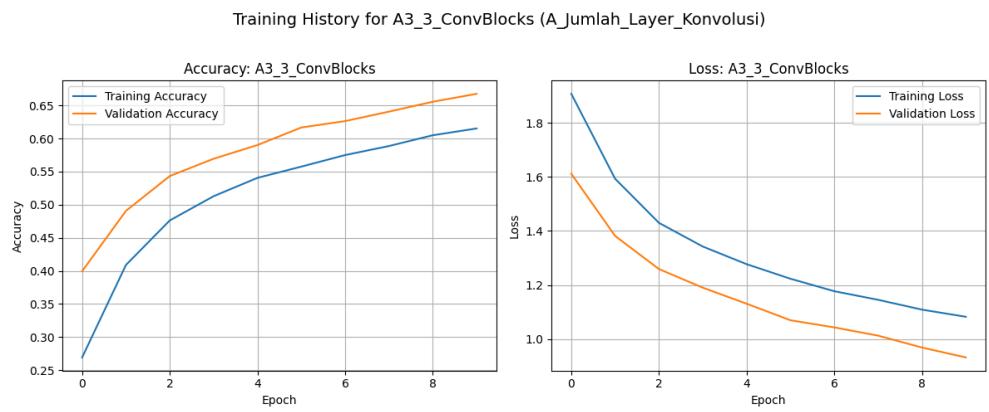
2.2.1.1.2.1 Layers = 1



2.2.1.1.2.2 Layers = 2



2.2.1.1.2.2 Layers = 3



2.2.1.1.3 Analisis Pengaruh Jumlah Konvolusi Layer

Pengaruh jumlah layer konvolusi pada arsitektur CNN menunjukkan bahwa model dengan 3 layer memberikan hasil prediksi terbaik

dibandingkan model dengan 2 layer maupun 1 layer. Hal ini terlihat dari nilai F1-Macro sebesar 0.6912 dan akurasi 0.36, yang lebih tinggi dari nilai pada konfigurasi layer lainnya. Model dengan 3 layer cenderung underfitting, ditunjukkan oleh performa F1-Macro yang lebih rendah (0.44), kemungkinan karena hanya dilakukan filtering sekali sehingga tidak dapat menangkap pattern dengan baik. Sementara itu, model dengan 2 layer menunjukkan adanya kenaikan performa (F1-Macro 0.47 dan akurasi 0.54), Namun, terjadi kenaikan berikutnya yang mencapai F1 sebesar 0.69. Hal ini dapat berarti 2 hal yaitu antara Model dengan 3 layer adalah model terbaik atau terjadi overfitting untuk model ke-3.

Namun demikian, berdasarkan tren loss ataupun accuracy yang terjadi ke-3 model memiliki tingkat perkembangan yang cukup stabil dengan jarak antara accuracy maupun loss antara training data dengan validation yang tidak terlalu jauh. Ini berarti ke-3 model untuk saat ini kemungkinan besar belum mengalami terjadinya overfitting.

2.2.1.2 Pengaruh Banyak Filter per Layer Konvolusi

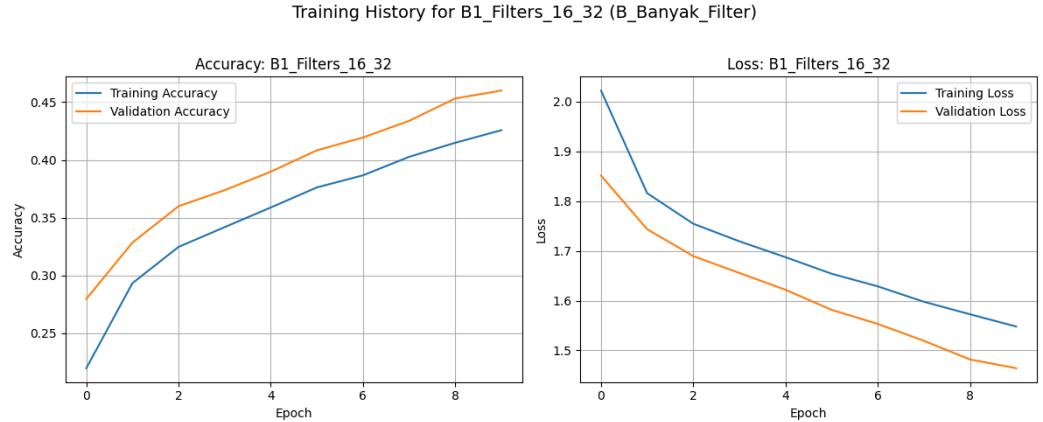
2.2.1.2.1 Hasil Akhir Prediksi

Variasi jumlah konvolusi layer yang digunakan adalah konvolusi 2 layer dengan filter masing masing (16, 32) , (32, 64), dan (64, 128).

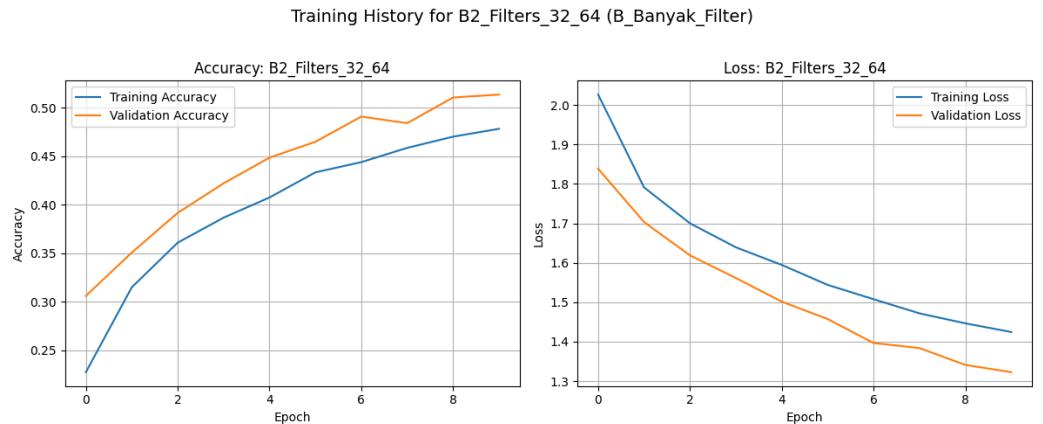
model_name	f1_keras	f1_scratch
B1_Filters_16_32	0.499015	0.499015
B2_Filters_32_64	0.467274	0.467274
B3_Filters_64_128	0.599246	0.599246

2.2.1.2.2 Grafik Training Loss dan Validation Loss

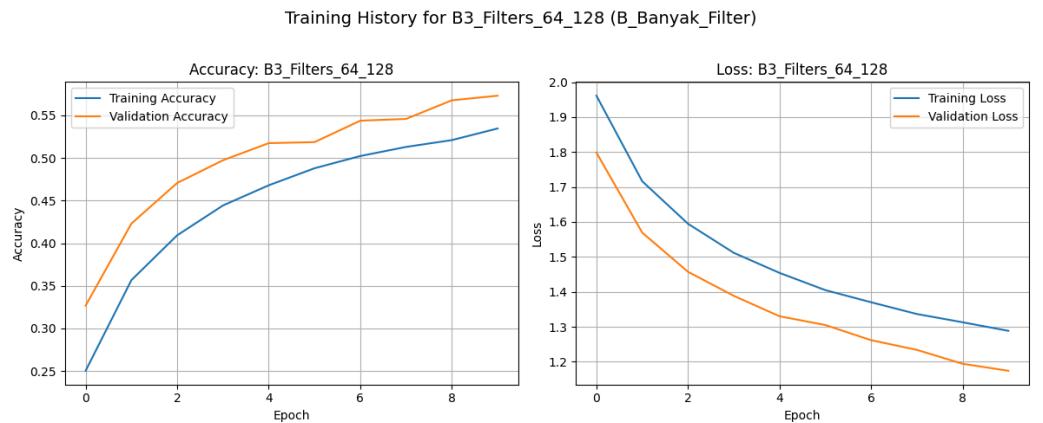
2.2.1.2.2.1 16 filter dan 32 filter



2.2.1.2.2.2 32 filter dan 64 filter



2.2.1.2.2.3 64 filter dan 128 filter



2.2.1.2.3 Analisis Pengaruh Banyak Filter per Layer Konvolusi

Pengaruh banyak filter pada arsitektur CNN menunjukkan bahwa model dengan banyak filter yaitu 64 dan 128 filter memberikan hasil prediksi terbaik dibandingkan model dengan 2 layer maupun 1 layer. Hal ini terlihat dari nilai F1-Macro sebesar 0.5992 dan akurasi 0.53, yang lebih tinggi dari nilai pada konfigurasi layer lainnya.

Namun demikian, berdasarkan tingkat akurasi ke-3 konfigurasi dengan perbedaan jumlah filter yang cukup signifikan dengan kenaikan 2 kali lipat tiap konfigurasi perubahan yang terjadi tidak terlalu berdampak dengan kenaikan akurasi yang hanya sekitar 5-10% saja. Terlebih lagi, mengingat percobaan sebelumnya mengenai konvolusi layer dengan jumlah 2 dengan filter berjumlah (32, 64). Kenaikan jumlah konvolusi memberikan dampak yang lebih besar dibandingkan dengan kenaikan filter yang ada.

2.2.1.3 Pengaruh Ukuran Filter per Layer Konvolusi

2.2.1.3.1 Hasil Akhir Prediksi

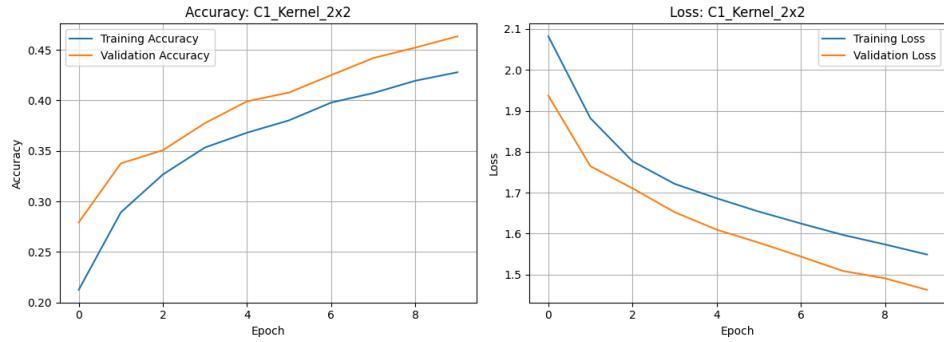
Variasi jumlah konvolusi layer yang digunakan adalah konvolusi 2 layer dengan filter berjumlah 64 dengan kernel masing masing berukuran [(2,2), (2,2)], [(3,3), (3,3)], dan [(5,5), (5,5)].

model_name	f1_keras	f1_scratch
C1_Kernel_2x2	0.441513	0.355307
C2_Kernel_3x3	0.472829	0.472829
C3_Kernel_5x5	0.548385	0.548385

2.2.1.3.2 Grafik Training Loss dan Validation Loss

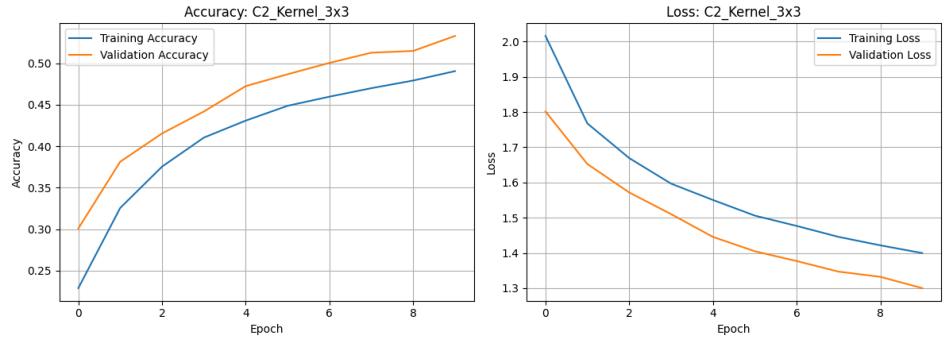
2.2.1.3.2.1 Kernel dengan layer 1 (2,2) dan layer 2 (2,2)

Training History for C1_Kernel_2x2 (C_Ukuran_Filter)



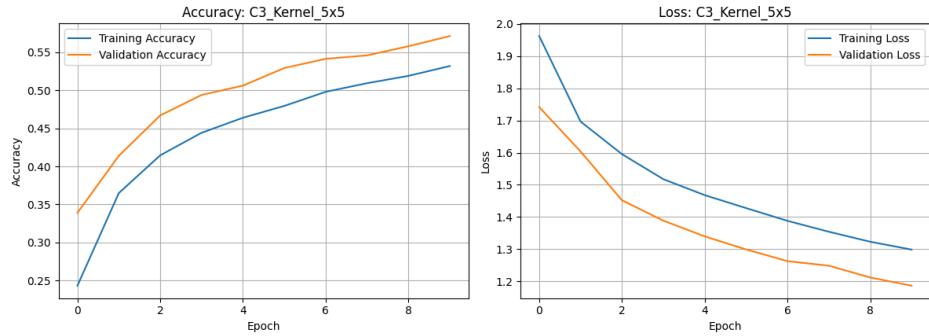
2.2.1.3.2.1 Kernel dengan layer 1 (3,3) dan layer 2 (3,3)

Training History for C2_Kernel_3x3 (C_Ukuran_Filter)



2.2.1.3.2.3 Kernel dengan layer 1 (5,5) dan layer 2 (5,5)

Training History for C3_Kernel_5x5 (C_Ukuran_Filter)



2.2.1.3.3 Analisis Pengaruh Ukuran Filter per Layer Konvolusi

Pengaruh banyak filter pada arsitektur CNN menunjukkan bahwa model dengan jumlah kernel untuk 2 convolution layer yaitu (5,5) dan (5,5) memberikan hasil prediksi terbaik dibandingkan model dengan kernel lainnya. Hal ini terlihat dari nilai F1-Macro sebesar 0.5442 dan akurasi 0.58, yang lebih tinggi dari nilai pada konfigurasi layer lainnya.

Pada percobaan ini, hasil yang diberikan cukup stabil dengan kenaikan yang tidak terlalu besar namun konsisten dengan kenaikan F1 score yaitu sekitar 0.03 tiap penambahan ukuran kernel. Ini menunjukkan penambahan ukuran kernel memberikan hasil pasti untuk kenaikan accuracy yang ada. Namun, kita tetap perlu memperhatikan mengenai adanya kemungkinan overfit karena semakin besar ukuran kernel maka ada kemungkinan besar model gagal mendapatkan sebuah pattern penting didalamnya.

2.2.1.4 Pengaruh Jenis Pooling Layer yang Digunakan

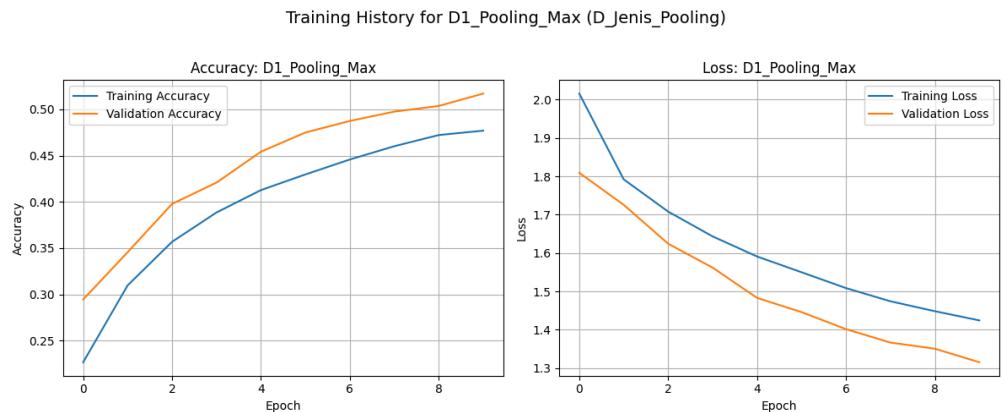
2.2.1.4.1 Hasil Akhir Prediksi

Variasi jenis pooling yang digunakan yaitu MaxPooling dan AveragePooling.

model_name	f1_keras	f1_scratch
D1_Pooling_Max	0.497829	0.497829
D2_Pooling_Avg	0.430238	0.341209

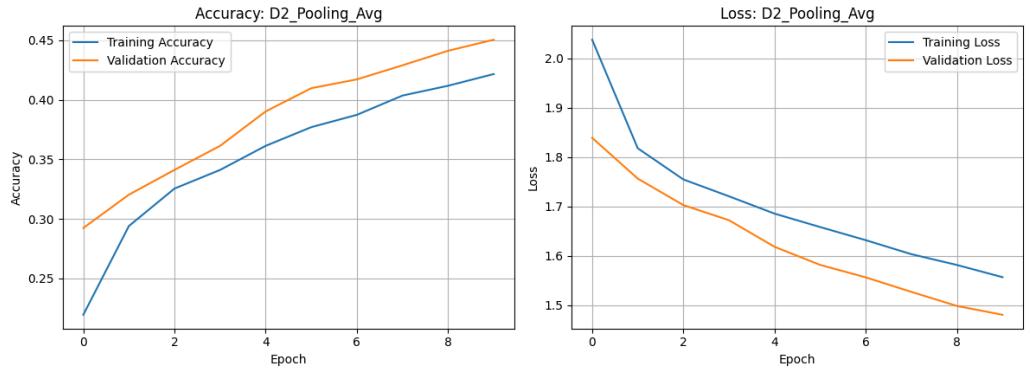
2.2.1.4.2 Grafik Training Loss dan Validation Loss

2.2.1.4.2.1 Max Pooling



2.2.1.4.2.1 Average Pooling

Training History for D2_Pooling_Avg (D_Jenis_Pooling)



2.2.1.4.3 Analisis Pengaruh Jenis Pooling Layer yang Digunakan

Pengaruh banyak jenis pooling pada arsitektur CNN menunjukkan bahwa model yang menggunakan max pooling mendapatkan hasil terbaik pada akhir epoch yaitu 0.497 untuk F1 score nya dengan 0.53 untuk accuracy-nya.

Ini menunjukkan bahwa penggunaan max pooling akan lebih akurat dibandingkan dengan average pooling. Yang berarti nilai maksimal sebuah weight dapat menunjukkan pattern yang lebih baik dibandingkan rata-rata.

2.2.2.4 Perbandingan From Scratch dengan Keras Model

2.2.2.4.1 Hasil Akhir Prediksi

	model_name	f1_keras	f1_scratch
0	A1_1_ConvBlock	0.440476	0.440476
1	A2_2_ConvBlocks	0.476779	0.476779
2	A3_3_ConvBlocks	0.691220	0.691220
3	B1_Filters_16_32	0.499015	0.499015
4	B2_Filters_32_64	0.467274	0.467274
5	B3_Filters_64_128	0.599246	0.599246
6	C1_Kernel_2x2	0.441513	0.355307
7	C2_Kernel_3x3	0.472829	0.472829
8	C3_Kernel_5x5	0.548385	0.548385
9	D1_Pooling_Max	0.497829	0.497829
10	D2_Pooling_Avg	0.430238	0.341209

2.2.2.4.2 Analisis Perbandingan

Berdasarkan hasil percobaan untuk prediksi antara f1_keras dengan f1_scratch untuk arsitektur CNN. kami mendapatkan bahwa hampir tidak ada perbedaan antara hasil prediksi oleh keras dengan algoritma CNN yang kami buat. Hanya terdapat beberapa hal yang menjadi pembeda tepatnya pada Pooling dengan average dan kernel berukuran [(2,2), (2,2)].

Perbedaan ini dapat muncul karena untuk 2x2 kernel ini terjadi karena pada tensorflow “Same” padding dengan kernel berjumlah genap tidak selalu symmetric, sedangkan kami mengimplementasikannya dengan langsung membagi nya menjadi 2 lalu menambahkan offset setelahnya yang mengakibatkan terjadinya perubahan nilai activation function.

2.2.2 RNN

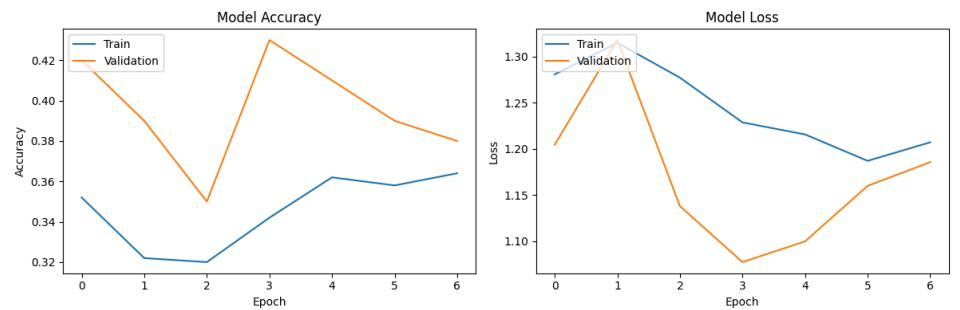
2.2.2.1 Pengaruh Jumlah Layer RNN

2.2.2.1.1 Hasil Akhir Prediksi

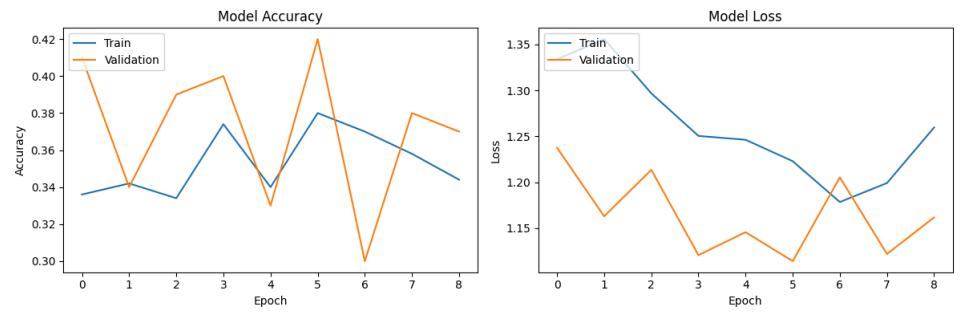
~~~~ Experimental Results ~~~~  
Experiment 1: Impact of Number of SimpleRNN Layers  
25 layers: F1-Macro = 0.3081, Accuracy = 0.4150  
50 layers: F1-Macro = 0.3662, Accuracy = 0.4150  
100 layers: F1-Macro = 0.3464, Accuracy = 0.3875

#### 2.2.2.1.2 Grafik Training Loss dan Validation Loss

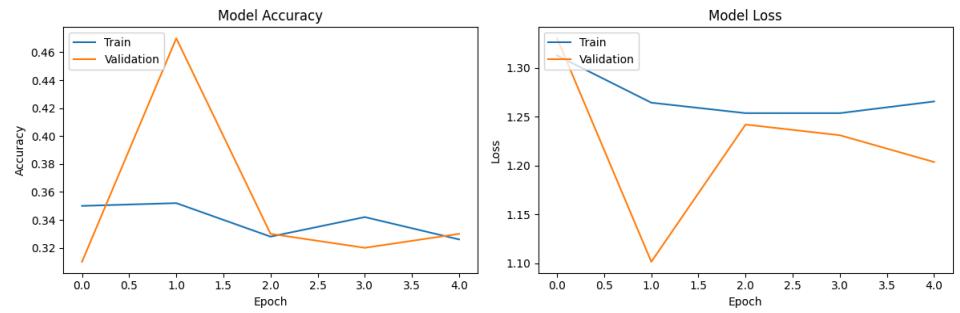
##### 2.2.2.1.2.1 Layers = 25



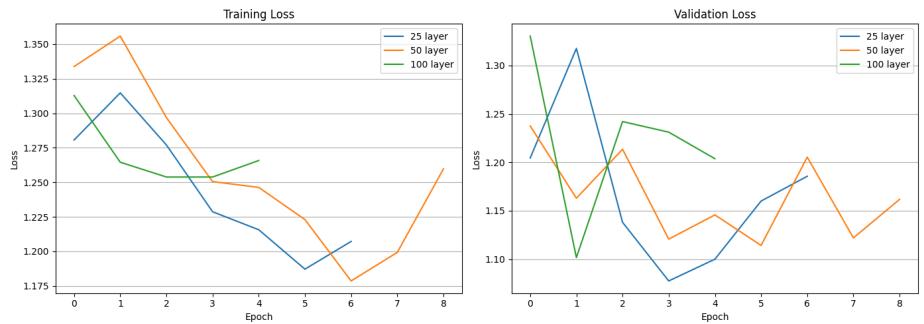
##### 2.2.2.1.2.2 Layers = 50



##### 2.2.2.1.2.3 Layers = 100



#### 2.2.2.1.2.4 Grafik Gabungan



#### 2.2.2.1.3 Analisis Pengaruh Jumlah Layer RNN

Pengaruh jumlah layer pada arsitektur SimpleRNN menunjukkan bahwa model dengan 50 layer memberikan hasil prediksi terbaik dibandingkan model dengan 25 maupun 100 layer. Hal ini terlihat dari nilai F1-Macro sebesar 0.3662 dan akurasi 0.4150, yang lebih tinggi dari nilai pada konfigurasi layer lainnya. Model dengan 25 layer cenderung underfitting, ditunjukkan oleh performa F1-Macro yang lebih rendah (0.3081), kemungkinan karena kapasitas model yang terlalu kecil untuk menangkap kompleksitas pola data. Sementara itu, model dengan 100 layer menunjukkan adanya penurunan performa (F1-Macro 0.3464 dan akurasi 0.3875), yang mengindikasikan potensi overfitting atau kesulitan dalam pelatihan akibat arsitektur yang terlalu dalam.

Dari grafik training loss dan validasi loss, terlihat bahwa model dengan 50 layer memiliki tren penurunan loss yang cukup stabil pada data pelatihan dan relatif konsisten pada validasi. Sebaliknya, model dengan 25 layer menunjukkan perubahan naik turun yang lebih besar pada validation loss. Hal ini menandakan bahwa model belum belajar dengan cukup baik. Model dengan 100 layer memang memiliki penurunan awal yang cepat pada validation loss, namun, model terlalu cepat mencapai kondisi stabil tanpa menunjukkan perkembangan yang signifikan, yang bisa jadi pertanda proses belajar terhenti atau terhambat akibat hilangnya gradien (*vanishing gradient*).

Kesimpulannya, jumlah layer yang terlalu sedikit maupun terlalu banyak dapat berdampak negatif terhadap kinerja model. Jumlah layer 50 memberikan keseimbangan terbaik antara kapasitas pembelajaran dan kestabilan pelatihan, sehingga menghasilkan prediksi yang paling optimal baik dari segi akurasi maupun F1-Macro.

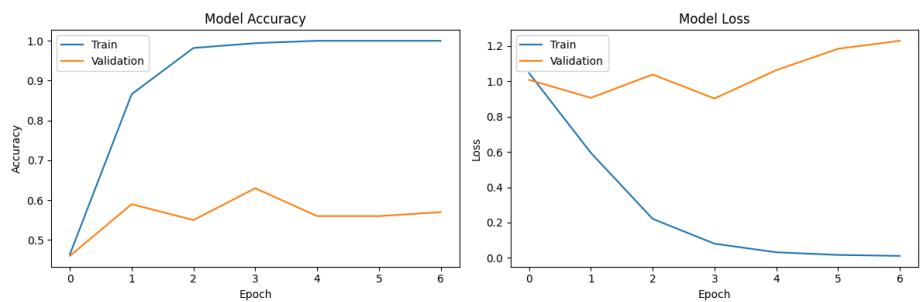
## 2.2.2.2 Pengaruh Banyak Cell RNN per Layer

### 2.2.2.2.1 Hasil Akhir Prediksi

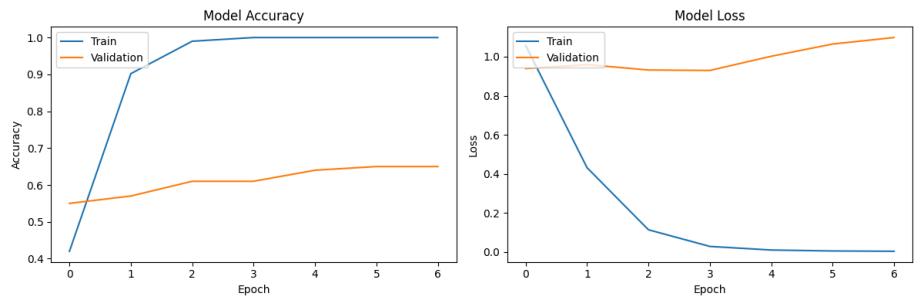
```
Experiment 2: Impact of Number of SimpleRNN Cells per Layer
32 units: F1-Macro = 0.6218, Accuracy = 0.6250
64 units: F1-Macro = 0.6316, Accuracy = 0.6425
128 units: F1-Macro = 0.5832, Accuracy = 0.6000
```

### 2.2.2.2.2 Grafik Training Loss dan Validation Loss

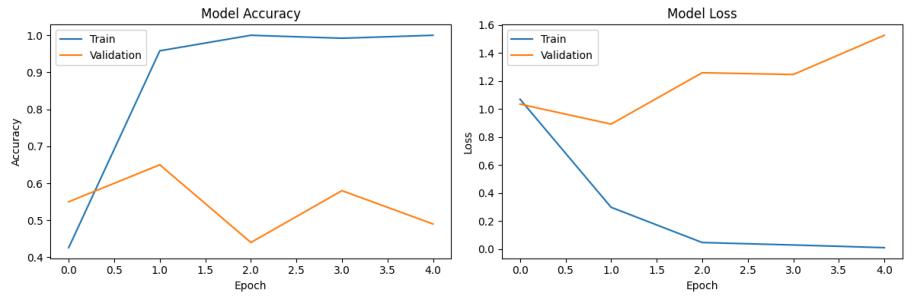
#### 2.2.2.2.2.1 Cells = 32



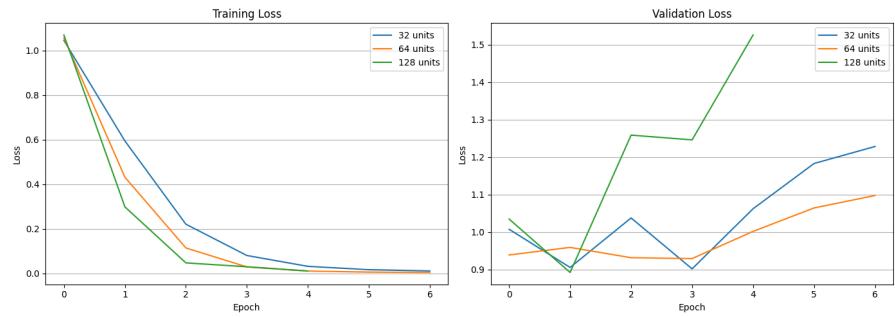
#### 2.2.2.2.2.2 Cells = 64



#### 2.2.2.2.2.3 Cells = 128



#### 2.2.2.2.4 Grafik Gabungan



#### 2.2.2.2.3 Analisis Pengaruh Banyak Cell RNN per Layer

Pengaruh jumlah cell (unit) RNN per layer menunjukkan bahwa penggunaan 64 unit menghasilkan performa terbaik dibandingkan konfigurasi lain. Hal ini terlihat dari hasil prediksi pada eksperimen, di mana model dengan 64 unit mencatat nilai F1-Macro sebesar 0.6316 dan akurasi sebesar 0.6425. Nilai ini lebih tinggi dibandingkan model dengan 32 unit (F1-Macro 0.6218) maupun 128 unit (F1-Macro 0.5832), yang menunjukkan bahwa 64 unit memberikan kapasitas pembelajaran yang paling seimbang dan cukup kompleks untuk mengenali pola dalam data tanpa mengalami overfitting.

Dari sisi grafik training dan validation loss, model dengan 128 unit memang menunjukkan penurunan training loss yang paling cepat dan signifikan, yang berarti model sangat cepat belajar dari data pelatihan. Namun, hal ini tidak dibarengi dengan performa yang baik pada validation loss karena bisa terlihat justru grafik validation loss nya meningkat drastis setelah beberapa epoch, ini menjadi indikasi kuat terjadinya overfitting. Sebaliknya, model dengan 64 unit memperlihatkan penurunan training loss yang stabil serta validation loss yang lebih terkendali dibandingkan 128 unit. Model dengan 32 unit memiliki tren stabil namun tidak sebaik 64 unit dalam performa validasi.

Kesimpulannya, jumlah unit yang terlalu sedikit dapat menyebabkan model kurang mampu menangkap kompleksitas data, sementara jumlah unit yang terlalu besar berisiko menyebabkan overfitting. Dalam eksperimen ini, jumlah unit yang sedang, yaitu 64 unit memberikan performa terbaik karena terhadap data validasi.

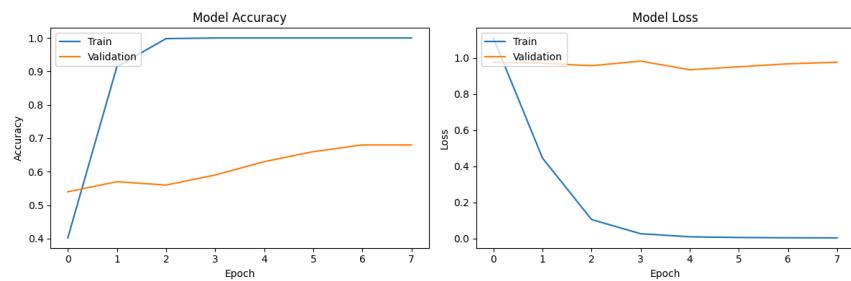
### 2.2.2.3 Pengaruh Jenis Layer RNN berdasarkan Arah

#### 2.2.2.3.1 Hasil Akhir Prediksi

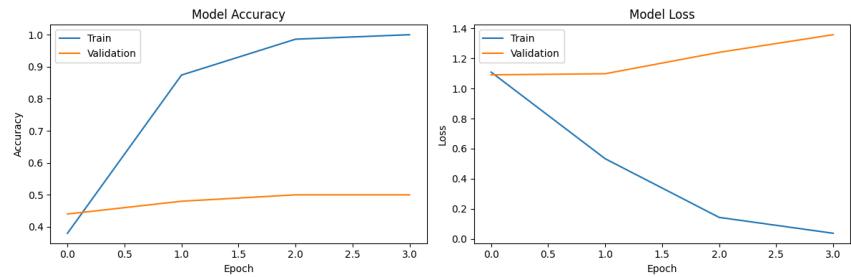
Experiment 3: Impact of SimpleRNN Layer Type by Direction  
bidirectional: F1-Macro = 0.5861, Accuracy = 0.5950  
unidirectional: F1-Macro = 0.4386, Accuracy = 0.5050

#### 2.2.2.3.2 Grafik Training Loss dan Validation Loss

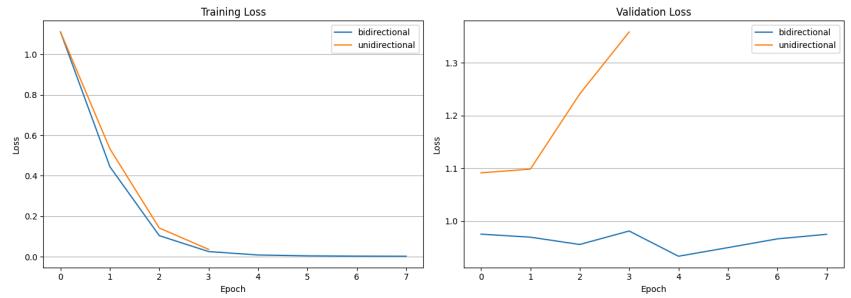
##### 2.2.2.3.2.1 Bidirectional



##### 2.2.2.3.2.2 Unidirectional



##### 2.2.2.3.2.3 Grafik Gabungan



#### 2.2.2.3.3 Analisis Pengaruh Jenis Layer RNN berdasarkan Arah

Pengaruh jenis layer RNN berdasarkan arah menunjukkan bahwa penggunaan layer bidirectional memberikan hasil yang jauh lebih baik dibandingkan dengan layer unidirectional. Berdasarkan hasil prediksi, model dengan bidirectional RNN memperoleh F1-Macro sebesar 0.5861

dan akurasi sebesar 0.5950, jauh melampaui model unidirectional yang hanya mencapai F1-Macro sebesar 0.4386 dan akurasi 0.5050. Hasil ini menunjukkan bahwa kemampuan bidirectional RNN dalam memproses informasi secara maju dan mundur dalam urutan teks memungkinkan model untuk menangkap konteks yang lebih kaya dan kompleks dari data, sehingga meningkatkan kualitas prediksi.

Dari segi grafik, terlihat bahwa meskipun kedua jenis model mengalami penurunan training loss yang signifikan dan serupa, tren validation loss menunjukkan perbedaan mencolok. Validation loss pada model bidirectional tetap relatif stabil dan rendah sepanjang epoch, menunjukkan kemampuan generalisasi yang baik terhadap data baru. Sebaliknya, validation loss pada model unidirectional terus meningkat secara konsisten, ini merupakan indikasi kuat terjadinya overfitting yaitu model belajar terlalu baik pada data pelatihan namun gagal mempertahankan performa pada data validasi.

Kesimpulannya, penggunaan layer bidirectional dalam arsitektur RNN lebih baik dalam menangkap konteks sekuensial. Hal ini tidak hanya menghasilkan metrik prediksi yang lebih tinggi, tetapi juga menunjukkan kestabilan dalam pelatihan yang lebih baik dan kemampuan generalisasi yang lebih kuat dibandingkan dengan layer unidirectional.

## 2.2.2.4 Perbandingan From Scratch dengan Keras Model

### 2.2.2.4.1 Hasil Akhir Prediksi

```
===== Prediction Distribution =====
Class counts in true labels:
  Class 0: 153
  Class 1: 96
  Class 2: 151

Class counts in Keras predictions:
  Class 0: 243
  Class 1: 125
  Class 2: 32

Class counts in Scratch predictions:
  Class 0: 150
  Class 1: 123
  Class 2: 127

Calculating evaluation metrics...

~~~~ Model Comparison ~~~~
Keras Model Macro F1: 0.3364
Scratch Model Macro F1: 0.3288
Keras Model Accuracy: 0.3750
Scratch Model Accuracy: 0.3350
~~~~~


Keras Model Classification Report:
      precision    recall   f1-score  support
negative        0.40     0.63     0.48    153
neutral         0.28     0.36     0.32     96
positive        0.59     0.13     0.21    151

accuracy          -         -     0.38    400
macro avg       0.42     0.37     0.34    400
weighted avg    0.44     0.38     0.34    400


Scratch Model Classification Report:
      precision    recall   f1-score  support
negative        0.38     0.37     0.38    153
neutral         0.24     0.30     0.26     96
positive        0.38     0.32     0.35    151

accuracy          -         -     0.34    400
macro avg       0.33     0.33     0.33    400
weighted avg    0.34     0.34     0.34    400


~~~~ Process Completed! ~~~~
```

### 2.2.2.4.2 Analisis Perbandingan

Perbedaan antara model Keras dan implementasi dari scratch yang terlihat pada hasil evaluasi dapat disebabkan oleh beberapa faktor. Salah satu penyebab utama kemungkinan berasal dari perbedaan dalam implementasi

`forward_pass`, khususnya pada bagian model dari scratch tidak secara eksplisit menangani padding, sedangkan model Keras biasanya sudah memiliki mekanisme masking untuk mengabaikan token `<PAD>` selama pelatihan dan prediksi. Hal ini menyebabkan model dari scratch cenderung menghitung hidden state berdasarkan data yang seharusnya tidak berpengaruh, yang akhirnya bisa mempengaruhi hasil akhir.

Selain itu, perbedaan juga bisa muncul karena model Keras menggunakan backend TensorFlow yang sangat dioptimalkan dan mendukung berbagai fitur stabilisasi numerik seperti clipping, precision tinggi, dan hardware acceleration, sedangkan model dari scratch sepenuhnya menggunakan operasi NumPy tanpa fitur-fitur tersebut. Meskipun pada akhirnya kedua model menghasilkan performa yang relatif mendekati, yaitu dengan skor macro F1 sebesar 0.3364 untuk model Keras dan 0.3288 untuk model dari scratch, serta akurasi masing-masing 37.50% dan 33.050%, perbedaan ini cukup wajar apalagi mengingat detail kompleksitas dalam implementasi dan perbedaan dukungan teknis backend dari masing-masing pendekatan.

## 2.2.3 LSTM

### 2.2.3.1 Pengaruh Jumlah Layer LSTM

#### 2.2.3.1.1 Hasil Akhir Prediksi

##### 2.2.3.1.1.1 Satu Layer Bi-Directional LSTM

```
Mengevaluasi model pada data test...
Test Loss: 6.1082
Test Accuracy: 0.3975
13/13 ━━━━━━ 1s 38ms/step
Test F1 Score: 0.3815

Melakukan prediksi pada beberapa data test...
1/1 ━━━━━━ 0s 34ms/step
Ke-1
Label Asli: positive
Label Prediksi: negative

Ke-2
Label Asli: neutral
Label Prediksi: negative

Ke-3
Label Asli: negative
Label Prediksi: positive

Ke-4
Label Asli: positive
Label Prediksi: negative

Ke-5
Label Asli: neutral
Label Prediksi: neutral
```

##### 2.2.3.1.1.2 Dua Layer Bi-Directional LSTM

```
Mengevaluasi model pada data test...
Test Loss: 7.7029
Test Accuracy: 0.4075
13/13 ━━━━━━ 0s 21ms/step
Test F1 Score: 0.3582

Melakukan prediksi pada beberapa data test...
1/1 ━━━━━━ 0s 28ms/step
Ke-1
Label Asli: positive
Label Prediksi: positive

Ke-2
Label Asli: neutral
Label Prediksi: negative

Ke-3
Label Asli: negative
Label Prediksi: negative

Ke-4
Label Asli: positive
Label Prediksi: positive

Ke-5
Label Asli: neutral
Label Prediksi: negative
```

#### 2.2.3.1.1.3 Tiga Layer Bi-Directional LSTM

```
Mengevaluasi model pada data test...
Test Loss: 7.9650
Test Accuracy: 0.3900
13/13 ━━━━━━━━ 1s 65ms/step
Test F1 Score: 0.3616

Melakukan prediksi pada beberapa data test...
1/1 ━━━━━━ 0s 32ms/step
Ke-1
Label Asli: positive
Label Prediksi: positive

Ke-2
Label Asli: neutral
Label Prediksi: negative

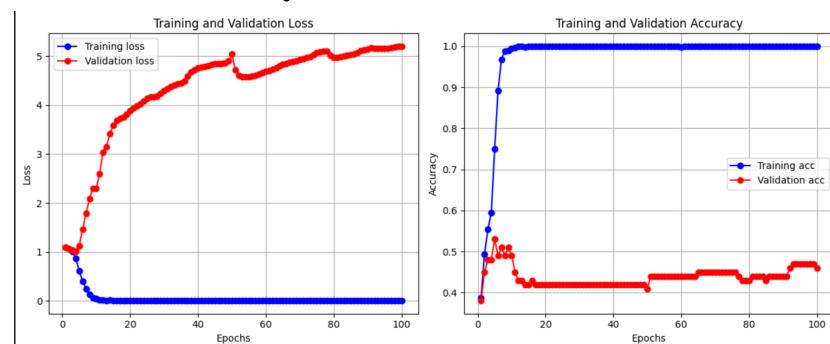
Ke-3
Label Asli: negative
Label Prediksi: positive

Ke-4
Label Asli: positive
Label Prediksi: negative

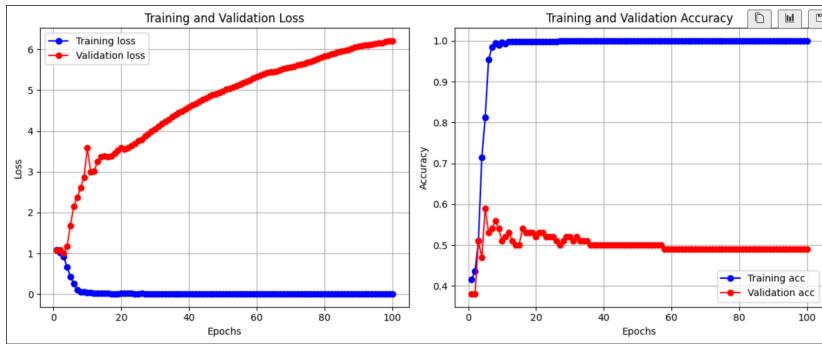
Ke-5
Label Asli: neutral
Label Prediksi: neutral
```

#### 2.2.3.1.2 Grafik Training Loss dan Validation Loss

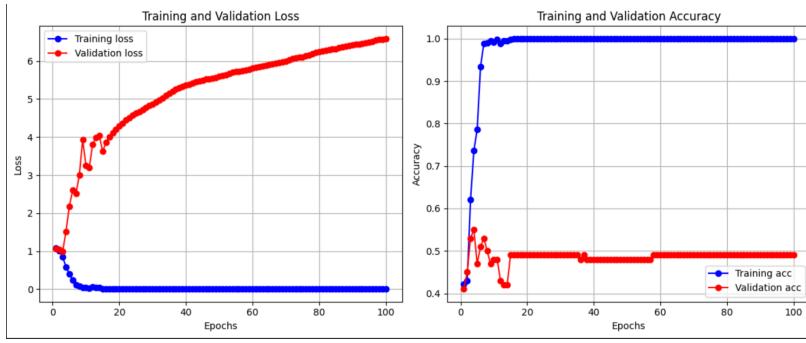
##### 2.2.3.1.2.1 Satu Layer Bi-Directional LSTM



#### 2.2.3.1.2.2 Dua Layer Bi-Directional LSTM



#### 2.2.3.1.2.3 Tiga Layer Bi-Directional LSTM



#### 2.2.3.1.3 Analisis Perbandingan

Penambahan jumlah layer LSTM pada kasus ini tidak meningkatkan kinerja model dan justru memperburuk masalah overfitting.

Model dengan 1 layer LSTM sudah menggunakan dropout, tetapi overfitting tetap parah. Model yang lebih dalam mungkin memerlukan dropout rate yang lebih tinggi atau early stopping yang lebih agresif untuk bisa berkinerja baik. (Early stopping pada kasus ini kami nonaktifkan karena epoch belajar menjadi sangat sedikit.

Data training tidak cukup besar dan kurang beragam untuk mendukung generalisasi. Model yang **lebih sederhana** (1 layer) akan memberikan hasil yang lebih baik pada data *test*.

### 2.2.3.2 Pengaruh Banyak Cell LSTM per Layer

#### 2.2.3.2.1 Hasil Akhir Prediksi

##### 2.2.3.2.1.1 Cells = 32

```
Mengevaluasi model pada data test...
Test Loss: 1.7672
Test Accuracy: 0.5200
13/13 ━━━━━━━━ 0s 19ms/step
Test F1 Score: 0.4209

Melakukan prediksi pada beberapa data test...
1/1 ━━━━━━━━ 0s 20ms/step
Ke-1
Label Asli: positive
Label Prediksi: positive

Ke-2
Label Asli: neutral
Label Prediksi: negative

Ke-3
Label Asli: negative
Label Prediksi: negative

Ke-4
Label Asli: positive
Label Prediksi: positive

Ke-5
Label Asli: neutral
Label Prediksi: negative
```

### 2.2.3.2.1.2 Cells = 64

```
Mengevaluasi model pada data test...
Test Loss: 1.7249
Test Accuracy: 0.4525
13/13 ━━━━━━━━ 0s 22ms/step
Test F1 Score: 0.3414

Melakukan prediksi pada beberapa data test...
1/1 ━━━━━━━━ 0s 37ms/step
Ke-1
Label Asli: positive
Label Prediksi: positive

Ke-2
Label Asli: neutral
Label Prediksi: negative

Ke-3
Label Asli: negative
Label Prediksi: negative

Ke-4
Label Asli: positive
Label Prediksi: positive

Ke-5
Label Asli: neutral
Label Prediksi: negative
```

### 2.2.3.2.1.3 Cells = 128

```
Mengevaluasi model pada data test...
Test Loss: 1.5671
Test Accuracy: 0.4675
13/13 ━━━━━━ 1s 32ms/step
Test F1 Score: 0.3712

Melakukan prediksi pada beberapa data test...
1/1 ━━━━━━ 0s 28ms/step
Ke-1
Label Asli: positive
Label Prediksi: positive

Ke-2
Label Asli: neutral
Label Prediksi: negative

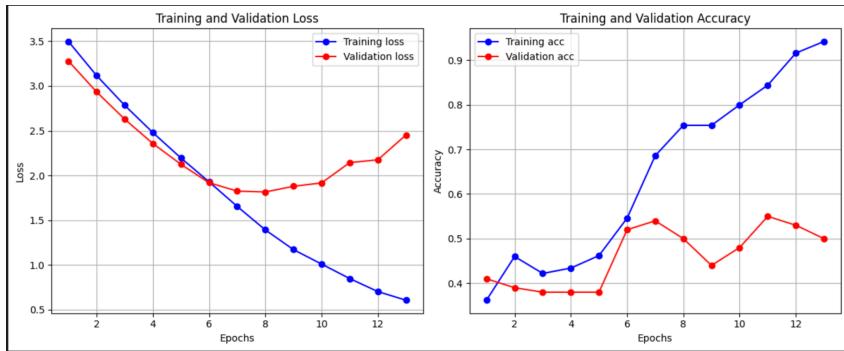
Ke-3
Label Asli: negative
Label Prediksi: positive

Ke-4
Label Asli: positive
Label Prediksi: positive

Ke-5
Label Asli: neutral
Label Prediksi: neutral
```

### 2.2.3.2.2 Grafik Training Loss dan Validation Loss

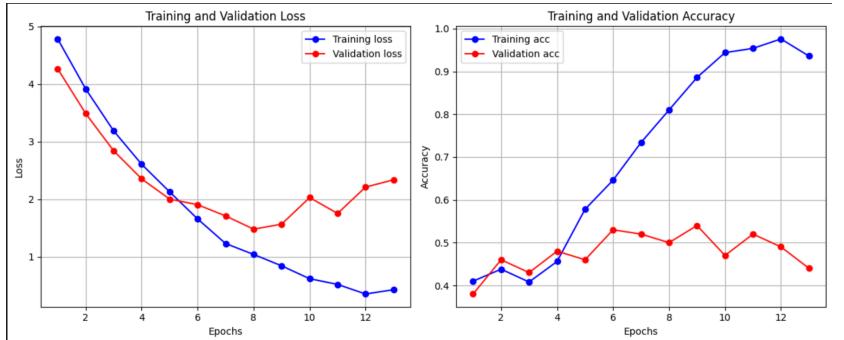
#### 2.2.3.2.2.1 Cells = 32



#### 2.2.3.2.2.1 Cells = 64



#### 2.2.3.2.2.1 Cells = 128



### 2.2.3.2.3 Analisis Pengaruh Banyak Cell LSTM per Layer

Menambah jumlah cell meningkatkan kemampuan model untuk belajar, tetapi pada data yang terbatas atau dengan regularisasi yang kurang, ini akan meningkatkan kecenderungan overfitting. Sebaliknya, mengurangi jumlah cell bisa mengurangi overfitting, tetapi berisiko underfitting (Dalam kasus ini makin sedikit jumlah unit makin baik karena makin rendah *chance* overfitting).

Lebih banyak cell berarti lebih banyak parameter untuk dilatih, sehingga membutuhkan lebih banyak waktu untuk melakukan training.

Menentukan jumlah cell juga harus melalui percobaan dengan memperhatikan struktur model, jumlah data, regularisasi, dan lain-lain.

### 2.2.3.3 Pengaruh Jenis Layer LSTM berdasarkan Arah

#### 2.2.3.3.1 Hasil Akhir Prediksi

##### 2.2.3.3.1.1 Bidirectional

```
Mengevaluasi model pada data test...
Test Loss: 2.0611
Test Accuracy: 0.4025
13/13 ━━━━━━━━ 0s 18ms/step
Test F1 Score: 0.3650

Melakukan prediksi pada beberapa data test...
1/1 ━━━━━━━━ 0s 20ms/step
Ke-1
Label Asli: positive
Label Prediksi: neutral

Ke-2
Label Asli: neutral
Label Prediksi: negative

Ke-3
Label Asli: negative
Label Prediksi: negative

Ke-4
Label Asli: positive
Label Prediksi: positive

Ke-5
Label Asli: neutral
Label Prediksi: positive
```

### 2.2.3.3.1.2 Unidirectional

```
Mengevaluasi model pada data test...
Test Loss: 1.4604
Test Accuracy: 0.5275
13/13 ━━━━━━━━ 0s 13ms/step
Test F1 Score: 0.4004

Melakukan prediksi pada beberapa data test...
1/1 ━━━━━━━━ 0s 18ms/step
Ke-1
Label Asli: positive
Label Prediksi: positive

Ke-2
Label Asli: neutral
Label Prediksi: negative

Ke-3
Label Asli: negative
Label Prediksi: positive

Ke-4
Label Asli: positive
Label Prediksi: positive

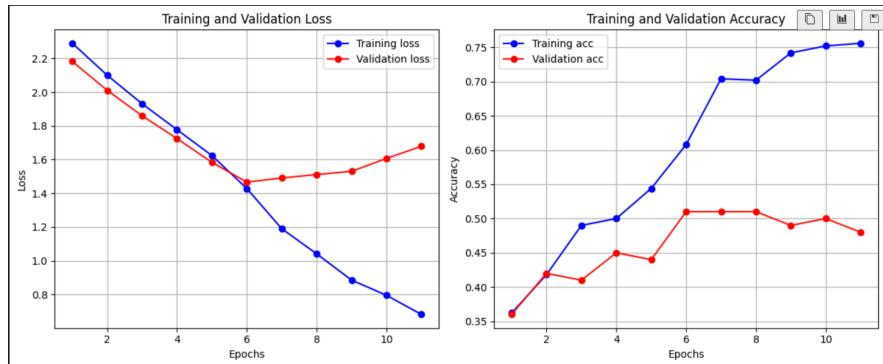
Ke-5
Label Asli: neutral
Label Prediksi: negative
```

### 2.2.3.3.2 Grafik Training Loss dan Validation Loss

#### 2.2.3.3.2.1 Bidirectional



#### 2.2.3.3.2.1 Unidirectional



### 2.2.3.3.3 Analisis Pengaruh Jenis Layer RNN berdasarkan Arah

Bidirectional LSTM secara teoritis lebih unggul karena kemampuannya memahami konteks dari masa lalu dan masa depan secara bersamaan. Ini sering kali menghasilkan akurasi yang lebih tinggi. Namun, dalam kasus data latih yang sedikit, kemungkinan untuk terjadi overfitting menjadi lebih besar dari Unidirectional LSTM karena jumlah parameter Bidirectional LSTM lebih banyak 2 kali dibanding Unidirectional LSTM.

Model Unidirectional LSTM lebih cepat dilatih dan lebih cepat dalam melakukan prediksi karena komputasinya lebih sedikit.

### **2.2.3.4 Perbandingan From Scratch dengan Keras Model**

#### **2.2.3.4.1 Hasil Akhir Prediksi**

```
--- Perbandingan Hasil ---
F1 Score (Macro) Keras : 0.3630
F1 Score (Macro) Scratch: 0.3178
```

#### **2.2.3.4.2 Analisis Perbandingan**

Perbedaan hasil pada model Keras dan model dari scratch dapat terjadi karena berbagai faktor yang saling mempengaruhi. Salah satu faktor utamanya adalah aspek optimisasi. Layer pada Keras sudah dilengkapi dengan mekanisme optimisasi yang sangat baik, termasuk dalam hal inisialisasi bobot, perhitungan gate, serta fungsi aktivasi yang telah terintegrasi dengan berbagai pengecekan dan penyesuaian numerik.

Sebaliknya, implementasi dari scratch cenderung lebih sederhana dan tidak memiliki lapisan optimisasi tambahan maupun validasi terhadap hasil komputasi, sehingga lebih rentan terhadap ketidaktepatan hasil. Selain itu, penggunaan layer Bidirectional pada Keras juga memberikan keunggulan tersendiri karena Keras telah mengatur secara internal bagaimana proses masking dan penggabungan output dari arah maju dan mundur dilakukan secara efisien. Di sisi lain, model dari scratch tidak menyertakan dropout layer yang umum digunakan dalam pelatihan model Keras untuk mengurangi overfitting, sehingga dapat mempengaruhi hasil perhitungan akhir.

Penggunaan embedding layer juga menjadi faktor penting, karena kemungkinan besar implementasi dari scratch belum mampu menangani variasi kasus dengan sekompelks Keras. Akhirnya, perbedaan presisi numerik dalam perhitungan juga dapat menyebabkan selisih hasil, mengingat Keras mengandalkan backend TensorFlow yang sangat dioptimalkan, sementara model dari scratch hanya menggunakan perhitungan dasar berbasis NumPy.

## BAB 3 KESIMPULAN DAN SARAN

### 3.1 Kesimpulan

Secara keseluruhan, hasil analisis terhadap implementasi CNN, RNN, dan LSTM menunjukkan bahwa setiap arsitektur memiliki kekuatan dan kelemahannya masing-masing tergantung pada jenis data dan permasalahan yang dihadapi. Pada arsitektur CNN, model dengan tiga layer konvolusi dan jumlah filter yang besar seperti (64, 128) memberikan performa prediksi terbaik dengan nilai F1-Macro sebesar 0.6912, menunjukkan kemampuan dalam menangkap pola spasial yang kompleks. Sementara itu, pada arsitektur RNN, jumlah layer dan jumlah unit sangat mempengaruhi performa. RNN dengan 50 layer dan 64 unit menunjukkan keseimbangan yang baik antara kapasitas pembelajaran dan kestabilan pelatihan, menghasilkan F1-Macro sebesar 0.6316. Arsitektur bidirectional pada RNN dan LSTM terbukti lebih efektif dibandingkan unidirectional dalam menangkap konteks sekvensial dua arah, meskipun berisiko overfitting jika jumlah data terbatas.

Untuk LSTM, penambahan layer justru cenderung memperparah overfitting jika tidak diimbangi dengan teknik regularisasi atau data yang cukup besar. Model LSTM dengan satu layer bidirectional terbukti cukup efektif pada data terbatas. Perbandingan antara model Keras dan implementasi from scratch menunjukkan bahwa meskipun performanya mendekati, model Keras memiliki akurasi dan F1-score yang umumnya lebih tinggi, seperti terlihat pada kasus RNN dengan F1 Keras 0.3364 dan Scratch 0.3288, serta pada CNN yang hampir tidak menunjukkan perbedaan. Hal ini menegaskan bahwa optimisasi internal, mekanisme masking, dan stabilisasi numerik dalam Keras memberi kontribusi besar terhadap performa akhir model.

### 3.1 Saran

Agar performa implementasi from scratch dapat lebih mendekati model dari framework seperti TensorFlow, beberapa saran berikut dapat dipertimbangkan. Pertama, peningkatan teknik inisialisasi bobot sangat krusial. Penggunaan metode seperti Xavier atau He initialization dapat membantu menjaga kestabilan distribusi nilai dalam jaringan dan mengurangi risiko vanishing atau exploding gradient. Kedua, penerapan algoritma optimasi modern seperti Adam atau RMSprop dapat meningkatkan efisiensi proses pembelajaran dibandingkan metode dasar seperti SGD. Ketiga, implementasi mekanisme automatic differentiation seperti yang digunakan dalam TensorFlow akan meningkatkan akurasi dan efisiensi perhitungan gradien secara signifikan. Selain itu, untuk model RNN dan LSTM, penanganan padding secara eksplisit, penggunaan dropout sebagai bentuk regularisasi, serta penyesuaian jumlah layer dan unit agar sesuai dengan kompleksitas data sangat disarankan. Terakhir, evaluasi lebih lanjut terhadap performa model sebaiknya dilakukan dengan mempertimbangkan ukuran dataset, variasi input, dan jenis tugas agar desain arsitektur dapat disesuaikan secara optimal.

## **Pembagian Tugas**

| <b>Tugas</b> | <b>Anggota</b>                                                                                         |
|--------------|--------------------------------------------------------------------------------------------------------|
| CNN Forward  | 13522157 - Muhammad Davis Adhipramana                                                                  |
| RNN Forward  | 13522164 - Valentino Chryslie Triadi                                                                   |
| LSTM Forward | 13522134 - Shabrina Maharani                                                                           |
| Laporan      | 13522134 - Shabrina Maharani<br>13522157 - M Davis Adhipramana<br>13522164 - Valentino Chryslie Triadi |

## Referensi

Mitchell, T. (1997). *Machine Learning*. McGraw Hill.

PPT Pembelajaran Mata Kuliah Pembelajaran Mesin Tahun 2025

Raschka, S., et al. (2022). *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing Ltd. (Chapter 15).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. (Chapter 10).

Zhu, S., & Chollet, F. (2023). *Working with RNNs | TensorFlow Core*. Retrieved from [https://www.tensorflow.org/guide/keras/working\\_with\\_rnn](https://www.tensorflow.org/guide/keras/working_with_rnn)

TensorFlow. (2023). *Convolutional Neural Network (CNN)*. Retrieved from [https://www.tensorflow.org/tutorials/images/cnn#evaluate\\_the\\_model](https://www.tensorflow.org/tutorials/images/cnn#evaluate_the_model)

TensorFlow. (2023). *tf.keras.layers.LSTM*. Retrieved from [https://www.tensorflow.org/api\\_docs/python/tf/keras/layers/LSTM](https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM)