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

Disusun oleh:

Kelompok 46

Ariel Herfrison	13522002
Zachary Samuel Tobing	13522016
Imam Hanif Mulyarahman	13522030

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

2025/2026

BAB 1
DESKRIPSI PERSOALAN

Dalam tugas besar ini, terdapat beberapa permasalahan utama yang perlu diselesaikan:

1. Implementasi dan Analisis CNN untuk Image Classification

Permasalahan pertama adalah mengimplementasikan dan menganalisis model CNN menggunakan dataset CIFAR-10 yang terdiri dari 10 kelas objek. Tantangan yang dihadapi meliputi:

- **Desain Arsitektur CNN**: Menentukan kombinasi yang optimal dari Conv2D *layers*, *pooling layers*, dan *dense layers* untuk mencapai performa terbaik pada dataset CIFAR-10.
- **Analisis Hyperparameter**: Memahami bagaimana berbagai hyperparameter CNN mempengaruhi performa model, termasuk:
    - Jumlah layer konvolusi dan dampaknya terhadap kemampuan *feature extraction*
    - Banyaknya filter per layer dan pengaruhnya terhadap representasi fitur
    - Ukuran filter dan efeknya terhadap detail fitur yang dapat ditangkap
    - Jenis *pooling layer* (*max pooling* vs *average pooling*) dan pengaruhnya terhadap *spatial information retention* (mempertahankan informasi spasial)
- **Implementasi Forward Propagation from Scratch**: Membangun implementasi forward propagation CNN tanpa menggunakan *high-level framework*, hanya dengan *library* matematika dasar seperti NumPy.

2. Implementasi dan Analisis RNN untuk Text Classification

Permasalahan kedua adalah mengimplementasikan model Simple RNN untuk klasifikasi sentimen teks menggunakan dataset NusaX-Sentiment (Bahasa Indonesia). Tantangan yang dihadapi meliputi:

- **Text Preprocessing**: Mengonversi data teks mentah menjadi representasi numerik yang dapat diproses oleh neural network melalui tokenization dan embedding.
- **Analisis Hyperparameter RNN**: Memahami bagaimana berbagai aspek RNN mempengaruhi performa model:
    - Jumlah layer RNN dan pengaruhnya terhadap kemampuan model menangkap pola sekuensial yang kompleks
    - Banyaknya RNN *cells per layer* dan dampaknya terhadap kapasitas representasi
    - Arah processing (*bidirectional* vs *unidirectional*) dan pengaruhnya terhadap *context understanding*

- **Implementasi Forward Propagation RNN from Scratch**: Membangun implementasi forward propagation RNN yang dapat menangani sequential data processing.

3. Implementasi dan Analisis LSTM untuk Text Classification

Permasalahan ketiga adalah mengimplementasikan LSTM, yang merupakan varian *advanced* dari RNN, untuk mengatasi masalah *vanishing gradient* dan *long-term dependency*. Tantangan yang dihadapi serupa dengan RNN namun dengan kompleksitas tambahan:

- **LSTM Architecture Complexity**: Memahami dan mengimplementasikan mekanisme gate (*forget gate*, *input gate*, *output gate*) yang membuat LSTM lebih *powerful* dibandingkan Simple RNN.
- **Analisis Hyperparameter LSTM**: Evaluasi pengaruh berbagai parameter LSTM terhadap performa model *text classification*.
- **Implementasi Forward Propagation LSTM from Scratch**: Membangun implementasi yang akurat untuk forward propagation LSTM dengan semua komponen *gate*-nya.

4. Validasi dan Perbandingan Implementasi

Tantangan utama lainnya adalah memastikan bahwa implementasi *from scratch* menghasilkan output yang identik atau sangat mendekati dengan implementasi menggunakan Keras/TensorFlow, yang menunjukkan kebenaran implementasi.

BAB 2
PEMBAHASAN

2.1. Penjelasan Implementasi

2.1.1. Deskripsi Kelas

2.1.1.1. CNN

a. Model

Model CNN dibuat sebagai model *sequential* dari berbagai *layer*, sehingga tidak dibuat dalam sebuah kelas, tetapi model *sequential* langsung digunakan dengan mengandung semua *layer* pada CNN. *Layer* yang digunakan pada model CNN yaitu:

Tabel 2.1.1.1.1 Layer CNN

Layer	Output Shape
conv2d (Conv2D)	(30, 30, 32)
max_pooling2d (MaxPooling2D)	(15, 15, 32)
conv2d_1 (Conv2D)	(13, 13, 64)
max_pooling2d_1 (MaxPooling2D)	(6, 6, 64)
flatten (Flatten)	(2304)
dense (Dense)	(64)
dense_1 (Dense)	(10)

b. Helper Functions

Selain model yang dibuat, terdapat berbagai fungsi pembantu untuk menolong dalam *forward propagation* model CNN tersebut.

Tabel 2.1.1.1.2 Helper Function CNN

Fungsi	Deskripsi
def relu(x)	Mengubah nilai negatif menjadi nol dan nilai positif tetap.
def softmax(x)	Mengubah vektor skor menjadi distribusi probabilitas, di mana jumlah total elemen bernilai 1.
def conv2d_forward(x, w, b)	Melakukan operasi konvolusi 2D antara input x dan *filter* w, ditambah dengan bias

	b.
def maxpool2d_forward(x, size=2, stride=2)	Mengurangi dimensi spasial dari data dengan memilih nilai maksimum dari tiap *patch* (biasanya 2x2) dan melompat sejauh *stride*.
def flatten_forward(x)	Mengubah input multidimensi (misalnya hasil *pooling* 3D) menjadi vektor 1D yang dibutuhkan oleh *dense layer*.
def dense_forward(x, w, b, activation=None)	Melakukan perhitungan *linear* antara input x dan bobot w, ditambah bias b, lalu menerapkan fungsi aktivasi jika ditentukan.
def forward_pass_scratch(x, weights)	Merupakan fungsi utama yang menyusun seluruh proses *forward pass* CNN.

2.1.1.2. Simple RNN

a. SimpleRNNScratch

Kelas SimpleRNNScratch sebagai satuan *layer* dari Simple RNN yang memiliki banyak neuron.

Tabel 2.1.1.2.1 Atribut Kelas SimpleRNNScratch

Atribut	Deskripsi
input_dim	Dimensi input dari *layer*
n_cell	Jumlah sel
return_sequences	Atribut *Boolean* apakah *layer* ingin mengembalikan *sequences*
Wx	Bobot dari input
Wh	Bobot dari *hidden state*
b	Bobot bias

Tabel 2.1.1.2.2 Method Kelas SimpleRNNScratch

Method	Deskripsi
def __init__(self, input_dim, n_cell, return_sequences=False)	Inisialisasi dari kelas ini

def load_weights(self, layer)	Memuat bobot pada *layer*
def tanh(self, x)	Mengembalikan nilai tanh dari x
def forward(self, input)	Melakukan *forward propagation* pada *layer* saat ini

b. ModelRNNScratch
Kelas ModelRNNScratch yang menyimpan setiap *layer* dari SimpleRNNScratch dan melakukan prediksi.

Tabel 2.1.1.2.3 Atribut Kelas ModelRNNScratch

Atribut	Deskripsi
layers	Array berisi kumpulan *layer* SimpleRNN

Tabel 2.1.1.2.4 Method Kelas ModelRNNScratch

Method	Deskripsi
def __init__(self)	Inisialisasi dari model
def load_model(self, model, input_dim):	Memuat bobot pada setiap *layer* SimpleRNN
def forward(self, x_seq):	Melakukan forward propagation pada setiap layer
def predict(self, X_batch)	Melakukan prediksi nilai pada model

c. Dense
Kelas Dense berperan sebagai layer dense suatu model

Tabel 2.1.1.2.5 Atribut Kelas Dense

Atribut	Deskripsi
weights	Bobot dari *dense layer*
biases	Bias dari *dense layer*

Tabel 2.1.1.2.6 Method Kelas Dense

Method	Deskripsi
def __init__(self, weights, bias)	Inisialisasi dari *layer dense*

def softmax(self, x)	Mengembalikan nilai dari fungsi softmax
def forward(self, x_seq):	Melakukan *forward propagation* pada *dense layer*

d. Embedder
Kelas Embedder berperan sebagai kelas *embedder* suatu model

Tabel 2.1.1.2.7 Atribut Kelas Embedder

Atribut	Deskripsi
weights	Bobot dari *embedder*

Tabel 2.1.1.2.8 Method Kelas Embedder

Method	Deskripsi
def __init__(self, weights, bias)	Inisialisasi dari *embedder*
def forward(self, x_seq):	Melakukan *forward propagation* pada *embedder*

2.1.1.3. LSTM

Model LSTM from scratch terdiri dari beberapa kelas, yaitu LSTM, Model, Dense, dan Embedder.

1. Class LSTM

Kelas LSTM berperan sebagai satuan layer LSTM yang terdiri dari banyak neuron, berbeda dengan kelas Model yang berperan sebagai gabungan banyak layer LSTM.

Tabel 2.1.1.3.1.1 Atribut Kelas LSTM

Atribut	Deskripsi
input_dim	Dimensi dari input yang akan diterima oleh *layer*
n_cell	Jumlah *cell*/neuron dalam *layer*
U	*Weight*/bobot dari input ke *layer* LSTM.
W	*Weight*/bobot antara *layer* LSTM dari satu *timestamp* ke *timestamp* berikutnya
b	*Weight*/bobot dari bias

return_sequences	Atribut *boolean* yang menentukan apakah layer akan mengembalikan *sequences*

Tabel 2.1.1.3.1.2 Method Kelas LSTM

Method	Deskripsi
def __init__(self, input_dim, n_cell, return_sequences = False)	Menginisialisasi atribut dari kelas/*layer*
def load_weights(self, layer)	Memuat bobot dari satu *layer* LSTM dalam model Keras
def sigmoid(self, x)	Fungsi bantuan untuk mengkalkulasi sigmoid dari nilai x
def tanh(self, x)	Fungsi bantuan untuk mengkalkulasi tanh dari nilai x
def forward(self, input)	Fungsi untuk menjalankan *forward propagation through time* dari *layer* dan menghasilkan output dari *hidden layer*.

2. Class Model

Kelas Model berperan menyatukan semua *layer* LSTM, berbeda dengan kelas LSTM yang berperan sebagai satuan *layer* LSTM. Kelas Model tidak mencakup layer Embedder dan Dense, melainkan hanya mencakup semua *layer* LSTM.

Tabel 2.1.1.3.2.1. Atribut Kelas Model

Atribut	Deskripsi
input_dim	Atribut bantuan yang menyimpan dimensi yang akan diterima oleh suatu *layer*
layers	Menyimpan *layer*-*layer* LSTM yang terdapat dalam model

Tabel 2.1.1.3.2.2. Method Kelas Model

Method	Deskripsi
def __init__(self)	Menginisialisasi atribut dari kelas/*layer* dengan nilai kosong
def load_model(self, model, input_dim)	Memuat bobot dari semua *layer* LSTM dalam model Keras

def forward(self, x_seq)	Fungsi untuk menjalankan *forward propagation* dari model secara keseluruhan. Fungsi ini akan memanggil fungsi forward dari tiap *layer* LSTM untuk melakukan *forward propagation through time*. Fungsi ini menerima satu *sequence* sebagai input.
def predict(self, X_batch)	Menjalankan *forward propagation* terhadap satu *batch*. Fungsi ini akan memanggil fungsi forward dari kelas Model untuk memproses tiap *sequence* dalam *batch*.

3. Class Dense

Kelas Dense berperan sebagai *layer* Dense dalam suatu model.

Tabel 2.1.1.3.3.1. Atribut Kelas Dense

Atribut	Deskripsi
weights	Menyimpan bobot dari tiap neuron dalam *layer* Dense
biases	Menyimpan bobot bias untuk tiap neuron dalam *layer* Dense

Tabel 2.1.1.3.3.2. Atribut Kelas Dense

Method	Deskripsi
def __init__(self, weights, bias)	Menginisialisasi atribut dari kelas/*layer*
def softmax(self, x)	Fungsi bantuan untuk mengkalkulasi softmax dari nilai x
def forward(self, x)	Fungsi untuk menjalankan *forward propagation* dari layer Dense.

4. Class Embedder

Kelas Embedder berperan sebagai layer Embedder dalam suatu model.

Tabel 2.1.1.3.4.1. Atribut Kelas Embedder

Atribut	Deskripsi

weights	Menyimpan pemetaan token ke dalam bentuk vektor (*weights* dari *layer* Embedder dalam model Keras)

Tabel 2.1.1.3.4.2. Method Kelas Embedder

Method	Deskripsi
def __init__(self, weights)	Menginisialisasi atribut dari kelas/*layer*
def forward(self, X_vectorized)	Memetakan input berupa token ke dalam bentuk vektor

2.1.2. Penjelasan Forward Propagation

2.1.2.1. CNN

Implementasi *forward propagation* dari model CNN di atas dilakukan sepenuhnya dari awal (*from scratch*) menggunakan NumPy tanpa bantuan pustaka *deep learning* seperti TensorFlow atau PyTorch. Proses dimulai dengan mentransposisikan input gambar dari format (tinggi, lebar, channel) menjadi (*channel*, tinggi, lebar) agar sesuai dengan kebutuhan fungsi konvolusi. Selanjutnya, input diproses melalui lapisan konvolusi pertama (conv2d_forward) dengan bobot dan bias tertentu, kemudian hasilnya dilewatkan ke fungsi aktivasi ReLU untuk mengenalkan non-linearitas. Output dari konvolusi ini selanjutnya diperkecil dimensi spasialnya melalui *max pooling*, yang mengambil nilai maksimum pada patch 2x2. Proses ini diulang untuk lapisan konvolusi kedua yang memiliki *filter* berbeda, juga diikuti oleh aktivasi ReLU dan pooling.

Setelah melalui dua blok konvolusi dan *pooling*, data yang semula berupa *tensor* tiga dimensi diubah menjadi vektor satu dimensi melalui proses *flatten*, sehingga dapat diteruskan ke lapisan *fully connected*. Vektor ini kemudian masuk ke *dense layer* pertama dengan aktivasi ReLU, diikuti oleh *dense layer* kedua yang menghasilkan skor akhir dan mengubahnya menjadi probabilitas menggunakan fungsi aktivasi softmax. Hasil akhir dari *forward propagation* ini adalah vektor probabilitas yang merepresentasikan prediksi model terhadap kelas-kelas output. Seluruh proses ini meniru arsitektur CNN konvensional dan ditulis secara eksplisit untuk memperlihatkan mekanisme kerja internal dari setiap tahapan pemrosesan data dalam jaringan saraf konvolusional.

2.1.2.2. Simple RNN

Pertama-tama, input berupa hasil Tokenization dari TextVectorization layer dari keras dimasukkan ke dalam layer **Embedder**, yang telah dimuat dengan bobot dari Keras, untuk menghasilkan input dalam bentuk vektor. Kemudian, input tersebut dimasukkan ke dalam kelas **Model**, yang telah memuat bobot dari model Keras. Forward propagation dalam kelas Model dilakukan melalui fungsi **predict**. Fungsi tersebut akan mengiterasi input secara satu per satu dan memanggil fungsi **forward** dari kelas Model untuk melakukan forward propagation terhadap input tersebut. Fungsi **forward** dari kelas Model akan mengiterasi setiap layer **Simple RNN**

secara satu per satu dan memanggil fungsi **forward** dari kelas/layer Simple RNN untuk melakukan forward propagation.

Fungsi **forward** akan melakukan *dot product* dari nilai input dengan bobot input ditambah nilai *hidden layer* dengan bobotnya. Hal ini membuat nilai dari *hidden layer* terus diperbaharui dan digunakan untuk menghitung nilai data berikutnya. Hasil dari fungsi **forward** dari kelas Model adalah output dari *layer* Simple RNN terakhir. Hasil dari fungsi **predict** adalah output dari semua *layer* Simple RNN untuk semua input. Hasil ini kemudian akan diteruskan menuju *layer* Dense yang terdiri dari 3 neuron untuk menghasilkan prediksi akhir.

2.1.2.3. LSTM

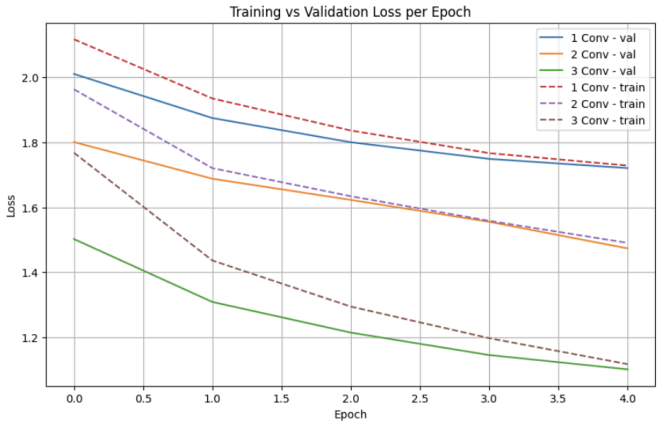
Pertama-tama, input berupa hasil *tokenization* dari TextVectorization *layer* dari keras dimasukkan ke dalam layer **Embedder**, yang telah dimuat dengan bobot dari Keras, untuk menghasilkan input dalam bentuk vektor. Kemudian, input tersebut dimasukkan ke dalam kelas **Model**, yang telah memuat bobot dari model Keras. *Forward propagation* dalam kelas Model dilakukan melalui fungsi **predict**. Fungsi tersebut akan mengiterasi input secara satu per satu dan memanggil fungsi **forward** dari kelas Model untuk melakukan *forward propagation* terhadap input tersebut. Fungsi **forward** dari kelas Model akan mengiterasi setiap *layer* **LSTM** secara satu per satu dan memanggil fungsi **forward** dari kelas/*layer* LSTM untuk melakukan *forward propagation through time*. Hasil dari fungsi **forward** dari kelas Model adalah output dari *layer* LSTM terakhir. Hasil dari fungsi **predict** adalah output dari semua *layer* LSTM untuk semua input. Hasil ini kemudian akan diteruskan menuju *layer* Dense yang terdiri dari 3 neuron untuk menghasilkan prediksi akhir.

2.2. Hasil Pengujian

2.2.1. CNN

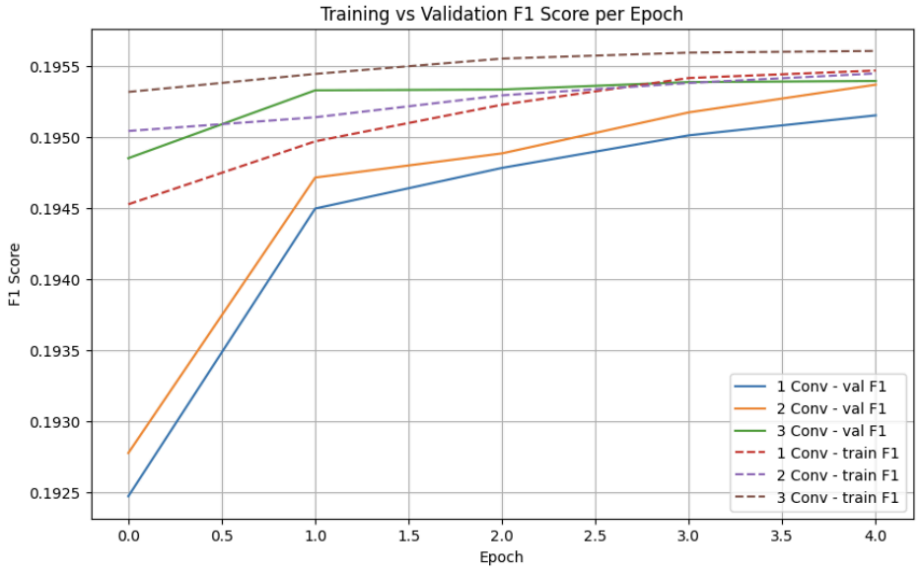
2.2.1.1. Pengaruh jumlah layer konvolusi

Penambahan *layer* konvolusi disertai dengan *layer max pooling* tersendiri. Jumlah *layer* yang digunakan yaitu satu, dua, dan tiga *layer* konvolusi. Banyaknya *epoch* yang digunakan adalah lima.



Gambar 2.2.1.1.1. Grafik Loss Pengaruh Jumlah Layer Konvolusi

Berdasarkan nilai *loss*, dapat dilihat bahwa banyaknya *layer* konvolusi berpengaruh besar pada nilai *loss* awal. Semakin banyak *layer* konvolusi yang digunakan, semakin kecil nilai *loss* awal, baik pada *training* maupun *validation*. Tren ini terus berlanjut sepanjang *epoch* hingga nilai *loss* pada *epoch* terakhir. Perubahan nilai *loss* seiring *epoch* juga cenderung lebih besar pada jumlah *layer* konvolusi yang lebih besar. Nilai *loss* sepanjang *epoch* lebih rendah pada *validation* daripada *training*. Hal ini menunjukkan bahwa model tidak melakukan *overfitting* pada data *training*.



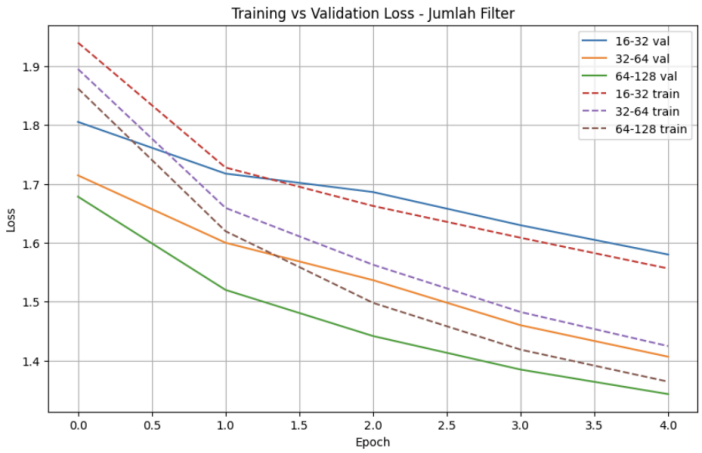
Gambar 2.2.1.1.2. Grafik F1 Score Pengaruh Jumlah Layer Konvolusi

Berdasarkan F1 Score, dapat dilihat bahwa banyaknya *layer* konvolusi memiliki pengaruh yang sama pada F1 Score. Semakin banyak *layer* konvolusi yang digunakan, semakin baik F1 Score yang dihasilkan. Tren ini terus dipertahankan sepanjang *epoch* hingga *epoch* terakhir, baik pada data *training* maupun data *validation*.

Akan tetapi, berbeda dengan nilai *loss* yang memiliki nilai serupa pada data *training* dan data *validation* pada jumlah *layer* konvolusi, dapat dilihat bahwa perbedaan F1 Score awal data *training* dan data *validation* sangat besar untuk satu dan dua *layer*. Perbedaan ini dengan signifikan diperbaiki pada seluruh *epoch* selanjutnya hingga memiliki nilai yang serupa pada *epoch* terakhir.

2.2.1.2. Pengaruh banyak filter per layer konvolusi

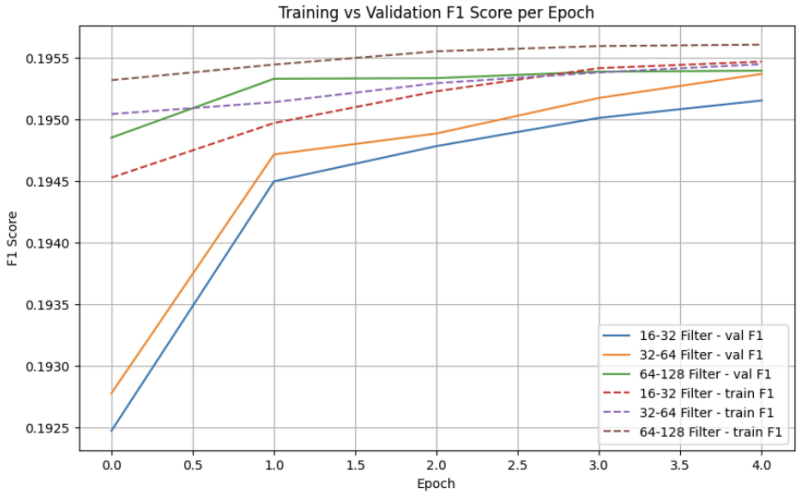
Pada pengujian ini digunakan dua *layer* konvolusi dan *max pooling* dengan setiap *layer* dan setiap pengujian memiliki nilai *filter* yang berbeda-beda. Pengujian dilakukan pada pasangan jumlah *filter* 16 & 32, 32 & 64, dan 64 & 128. Jumlah *epoch* yang digunakan adalah lima.



Gambar 2.2.1.2.1. Grafik Loss Pengaruh Banyak Filter

Berdasarkan nilai *loss*, dapat dilihat bahwa jumlah *filter* yang lebih besar memiliki nilai *loss* yang lebih rendah, baik pada data *training* maupun data *validation*. Tren ini dipertahankan hingga *epoch* terakhir. Penurunan nilai *loss* juga lebih cepat seiring *epoch* pada jumlah *filter* yang lebih besar, dapat diamati pada data *training* maupun *validation*.

Selain itu, dapat dilihat juga bahwa nilai *loss* awal *training* jauh lebih besar dibandingkan nilai *loss* awal *validation* untuk ketiga pasangan jumlah *filter*. Akan tetapi, perbaikan nilai *loss* dengan cepat dilakukan sehingga F1 Score akhir serupa dengan data *validation*.



Gambar 2.2.1.2.2. Grafik F1 Score Pengaruh Banyak Filter

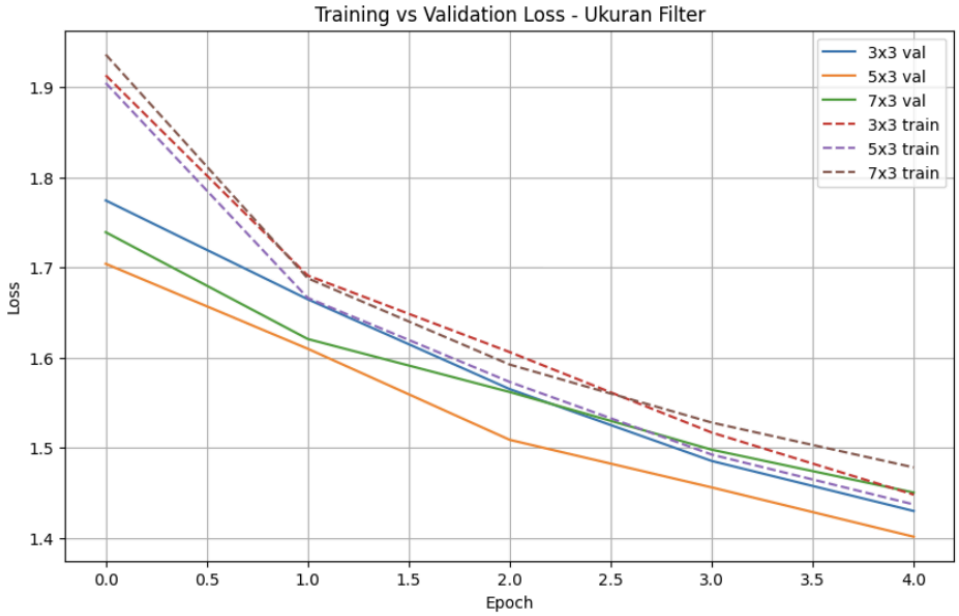
Berdasarkan F1 Score, dapat dilihat pada *epoch* pertama bahwa jumlah *filter* yang lebih kecil memiliki F1 Score yang lebih rendah. Tren tersebut dipertahankan hingga *epoch* terakhir, baik pada data *training* maupun data *validation*. Akan tetapi, terdapat suatu anomali, yaitu F1 Score yang sangat besar pada data *validation* dengan jumlah *filter* terbesar.

Sementara itu, dapat dilihat bahwa F1 Score pada *epoch* pertama untuk data *training* jauh lebih besar dibandingkan data *validation*. Hal ini mungkin disebabkan oleh *overfitting* yang terjadi pada data *training* sehingga bobot yang diperoleh tidak sesuai dengan data *validation*,

menyebabkan F1 Score yang jauh lebih rendah pada *epoch* pertama. Akan tetapi, hal ini juga diperbaiki dengan cepat sehingga pada akhirnya F1 Score pada setiap jenis data dan setiap jumlah *filter* memiliki nilai yang serupa satu dengan yang lainnya.

2.2.1.3. Pengaruh ukuran filter per layer konvolusi

Ukuran filter yang digunakan pada setiap *layer* dibedakan pada salah satu dimensi *filter* saja agar tidak terlalu banyak mengubah parameter antar pengujian. Pengujian dilakukan pada ukuran *filter* 3x3, 5x3, dan 7x3. Jumlah *epoch* yang digunakan adalah lima.

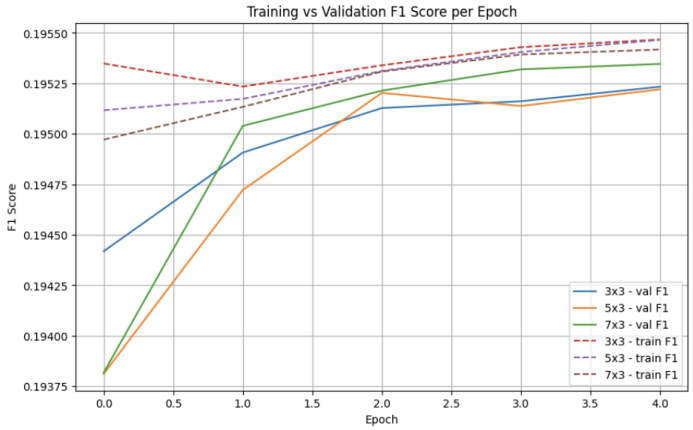


Gambar 2.2.1.3.1. Grafik Loss Pengaruh Ukuran Filter

Berdasarkan nilai *loss*, dapat dilihat suatu tren yang unik. Nilai *loss epoch* pertama untuk data *training* menunjukkan ukuran *filter* terbesar (7x3) sebagai yang terburuk, diikuti oleh yang terkecil (3x3), dan terakhir ukuran 5x3. Akan tetapi, pada *epoch* terakhir dapat dilihat bahwa nilai *loss* terbaik dimiliki oleh ukuran 5x3, diikuti oleh 3x3, dan terakhir 7x3, memutarbalikkan urutan pada *epoch* pertama.

Sementara itu, pada data *validation* dapat dilihat tren yang berbeda lagi. Pada *epoch* pertama, nilai *loss* terendah dimiliki oleh ukuran *filter* 5x3, diikuti oleh ukuran 7x3, dan terakhir ukuran 3x3. Akan tetapi, pada *epoch* terakhir dapat dilihat bahwa tren tersebut dipertahankan hingga akhirnya ukuran 7x3 and 3x3 terbalik pada *epoch* terakhir.

Selain itu, dapat dilihat juga bahwa nilai *loss* pada data *validation* jauh lebih rendah dibandingkan data *training* pada *epoch* awal, menunjukkan bahwa bobot yang diperoleh sesuai dengan data *validation* dan tidak terjadi *overfitting*. Pembelajaran pada data *training* dilakukan dengan lebih cepat sehingga semua nilai *loss* pada semua ukuran dan tipe data sangat serupa satu dengan yang lainnya.



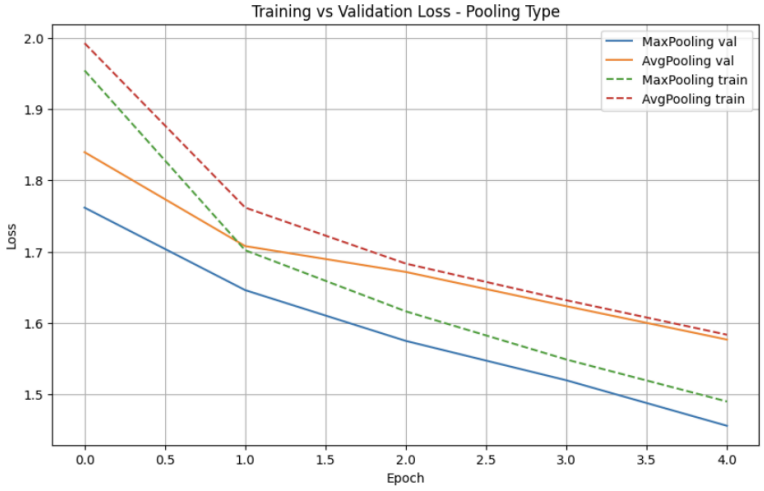
Gambar 2.2.1.3.2. Grafik F1 Score Pengaruh Ukuran Filter

Berdasarkan F1 Score, dapat dilihat suatu tren yang tidak biasa. Pada *epoch* pertama, ukuran kernel terkecil memiliki F1 Score yang paling besar, dengan *margin* yang cukup besar bahkan. Akan tetapi, seiring bertambahnya *epoch*, dapat dilihat bahwa ukuran kernel terbesar (7x3) segera menyusul dan menempati posisi paling tinggi pada *epoch* terakhir, sementara ukuran terkecil (3x3) menempati posisi kedua, dan ukuran 5x3 menjadi yang terendah.

Selain itu, dapat dilihat bahwa F1 Score awal pada data *validation* jauh lebih rendah dibandingkan data *training*. Hal ini memunculkan kemungkinan bahwa dapat terjadi *overfitting* pada data *training* sehingga bobot yang didapatkan ketika *training* sangat tidak sesuai dengan data *validation*, menyebabkan F1 Score awal yang jauh lebih rendah pada data *validation*. Akan tetapi, perbaikan dilakukan dengan cepat sehingga dapat memiliki F1 Score yang serupa dengan data *training* pada *epoch* terakhir.

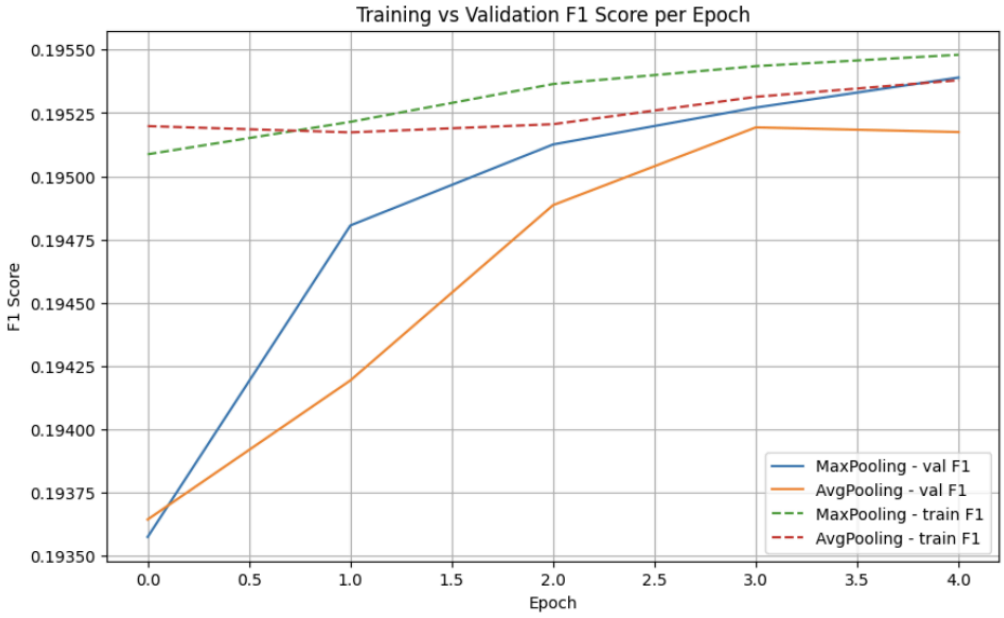
2.2.1.4. Pengaruh jenis pooling layer

Jenis *pooling layer* yang digunakan terdiri dari dua jenis, yaitu *max pooling* dan *average pooling*. Jumlah *epoch* yang digunakan adalah lima.



Gambar 2.2.1.4.1. Grafik Loss Pengaruh Jenis Pooling Layer

Berdasarkan nilai *loss*, dapat dilihat bahwa *max pooling* memiliki nilai *loss* yang lebih kecil dibandingkan *average pooling*, mulai dari *epoch* pertama hingga terakhir. Nilai *loss* pada data *training* lebih tinggi dibandingkan data *validation*, menunjukkan bahwa tidak terjadi *overfitting* pada data *training*. Nilai *loss* awal yang lebih tinggi pada data *training* diperbaiki dengan lebih cepat dibandingkan data *validation* sehingga pada *epoch* terakhir keduanya memiliki nilai yang serupa, baik pada *max pooling* maupun *average pooling*.



Gambar 2.2.1.4.2. Grafik F1 Score Pengaruh Jenis Pooling Layer

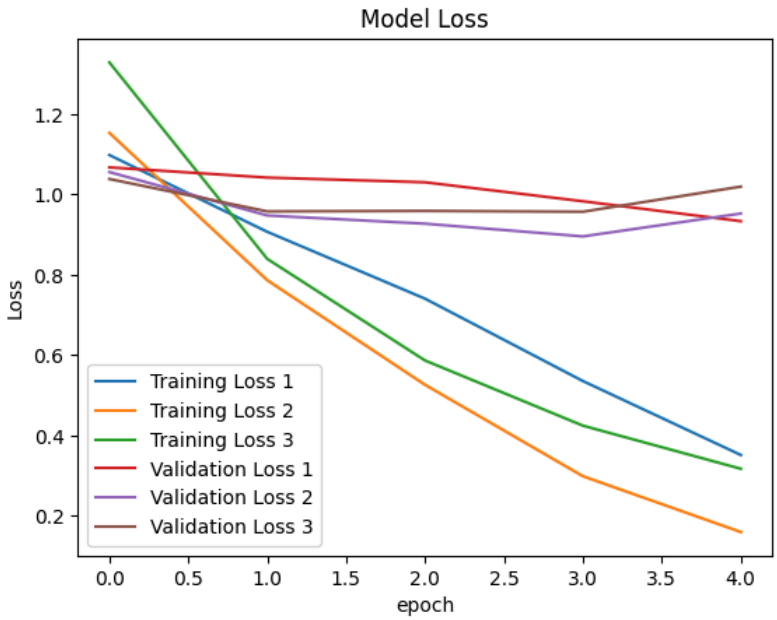
Pola yang sama terlihat juga pada F1 Score dengan sedikit perbedaan. Metode *max pooling* memiliki F1 Score yang lebih rendah dibandingkan *average pooling* pada data *training* maupun data *validation* pada *epoch* pertama. Pada *epoch* kedua hingga *epoch* terakhir, dapat dilihat tren yang sama, yaitu F1 Score yang dimiliki oleh *max pooling* lebih baik dibandingkan *average pooling*.

Selain itu, dapat dilihat bahwa F1 Score awal pada data *validation* jauh lebih rendah dibandingkan data *training*. Hal ini memunculkan kemungkinan bahwa dapat terjadi *overfitting* pada data *training* sehingga bobot yang didapatkan ketika *training* sangat tidak sesuai dengan data *validation*, menyebabkan F1 Score awal yang jauh lebih rendah pada data *validation*. Akan tetapi, perbaikan dilakukan dengan cepat sehingga dapat memiliki F1 Score yang serupa dengan data *training* pada *epoch* terakhir.

2.2.2. Simple RNN

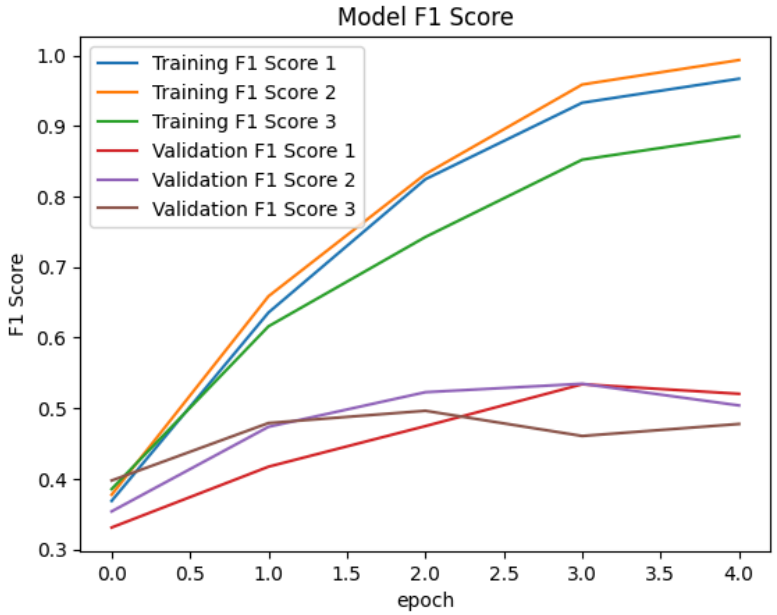
2.2.2.1. Pengaruh jumlah layer RNN

Pada pengujian ini digunakan 3 model *simple* RNN dengan jumlah *layer* yang berbeda-beda. Model 1 memiliki jumlah 1 *layer*, model 2 memiliki jumlah 3 *layer*, dan model 3 memiliki jumlah 7 *layer*. Setiap model menggunakan 16 neuron dengan jumlah *epoch* 5.



Gambar 2.2.2.1.1. Grafik Loss Pengaruh Jumlah Layer RNN

Berdasarkan nilai *loss* nya, dapat dilihat bahwa jumlah *layer* berpengaruh terhadap nilai *loss* model. Model 1 menunjukkan tren yang cukup stagnan yang menunjukkan bahwa model *underfitting*. Model 2 menunjukkan penurunan nilai *loss* yang cukup signifikan. Sedangkan model 3 menunjukkan penurunan nilai *loss* yang baik tetapi pada *epoch* akhir nilai *loss*nya mulai naik lagi. Hal ini menunjukkan model 3 *overfitting* dalam jangka panjang.



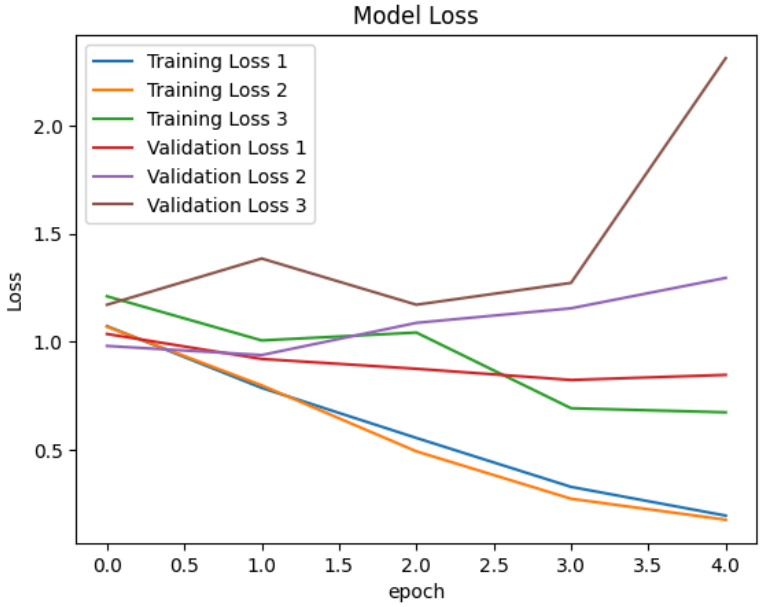
Gambar 2.2.2.1.2. Grafik F1 Score Pengaruh Jumlah Layer RNN

Berdasarkan F1 Scorenya, dapat dilihat bahwa jumlah *layer* berpengaruh terhadap nilai F1 Score model. Model 1 menunjukkan tren yang meningkat cukup baik, tetapi belum sebaik Model 2. Sedangkan model 3 menunjukkan F1 Score yang kurang konsisten pada *validation* model dan secara keseluruhan tidak sebaik model 1 dan model 2. Hal ini mungkin disebabkan oleh *overfitting*.

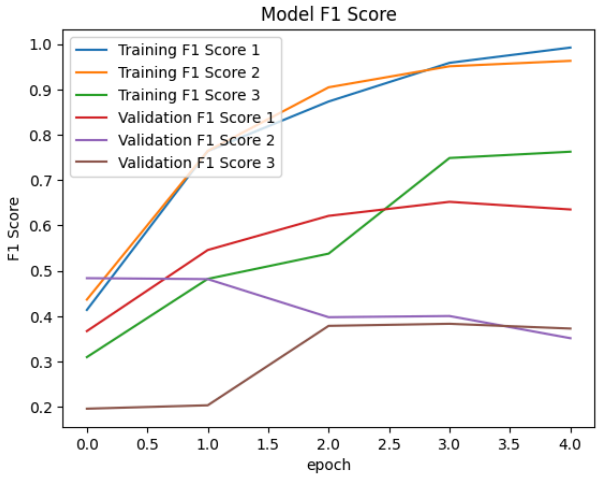
Dari kedua grafik tersebut dapat disimpulkan bahwa peningkatan jumlah *layer* sampai tahap tertentu dapat memperbagus kinerja model. Namun, jika jumlah *layer* terlalu banyak, performa model justru dapat turun disebabkan oleh *overfitting*.

2.2.2.2. Pengaruh banyak cell RNN per layer

Pada pengujian ini digunakan 3 model *simple* RNN dengan jumlah *cell* yang berbeda-beda. Model 1 memiliki jumlah 32 *cell*, model 2 memiliki jumlah 128 *cell*, dan model 3 memiliki jumlah 512 *cell*. Setiap model menggunakan 1 *layer* dengan jumlah *epoch* 5.



Gambar 2.2.2.2.1. Grafik Loss Pengaruh Banyak Cell RNN



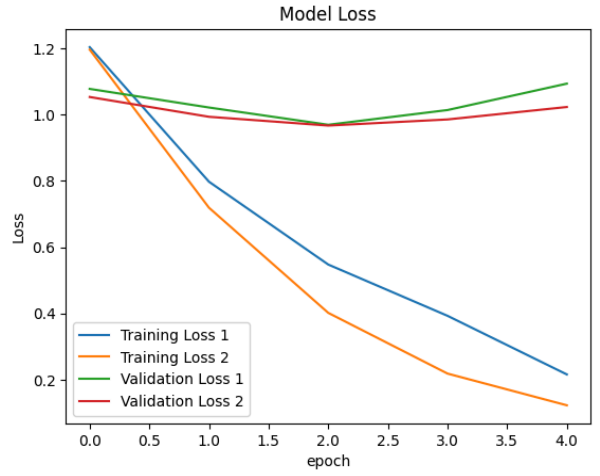
Gambar 2.2.2.2.2. Grafik F1 Score Pengaruh Banyak Cell RNN

Dapat dilihat dari kedua grafik di atas bahwa jumlah *cell* dalam *layer* mempengaruhi performa model. Model 1 dan 2 memiliki performa yang lumayan baik dan tidak terlalu berbeda saat *training*. Namun, pada tahap validasi dapat dilihat bahwa model 1 memiliki performa yang cenderung lebih baik dibandingkan dengan model 2. Selain itu, dapat dilihat bahwa model 3 sangat *overfitting* karena memiliki terlalu banyak *cell*. Hal ini menyebabkan nilai *loss* yang meningkat jauh pada *epoch* akhir dan nilai F1 Score yang rendah.

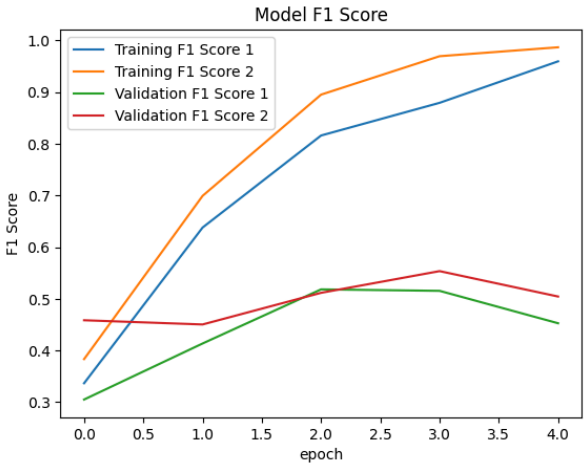
Dari kedua grafik tersebut dapat disimpulkan bahwa peningkatan jumlah *cell* pada *layer* sampai tahap tertentu dapat memperbagus kinerja model. Namun, jika jumlah *cell* pada *layer* terlalu banyak, performa model justru dapat turun disebabkan oleh *overfitting*.

2.2.2.3. Pengaruh jenis layer RNN berdasarkan arah

Pada pengujian ini digunakan 2 model simple RNN dengan arah yang berbeda. Model 1 menggunakan *unidirectional simple* RNN dan model 2 menggunakan *bidirectional simple* RNN. Setiap model menggunakan 3 *layer* dengan jumlah *epoch* 5.



Gambar 2.2.2.3.1. Grafik Loss Pengaruh Arah RNN



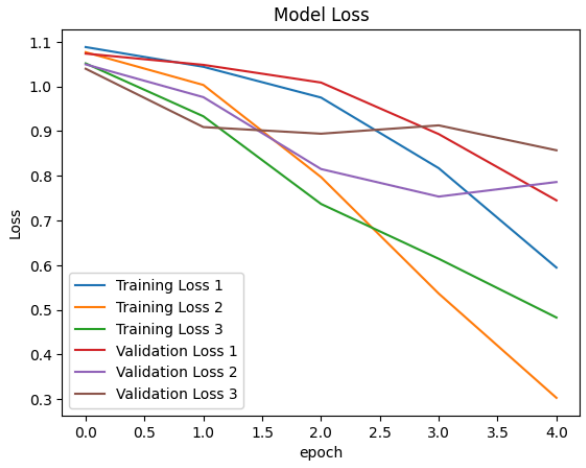
Gambar 2.2.2.3.2. Grafik F1 Score Pengaruh Arah RNN

Dari kedua grafik diatas dapat disimpulkan bahwa secara umum Bidirectional RNN (model 2) memiliki performa lebih baik dibanding Unidirectional RNN (model 1) walau tidak signifikan. Hal ini disebabkan model 2 mampu belajar representasi data lebih kaya dan efektif (*loss* turun lebih banyak dan F1 *training* lebih tinggi) karena akses konteks dua arah, tetapi ada indikasi risiko *overfitting* pada data validasi (validasi F1 lebih rendah secara umum).

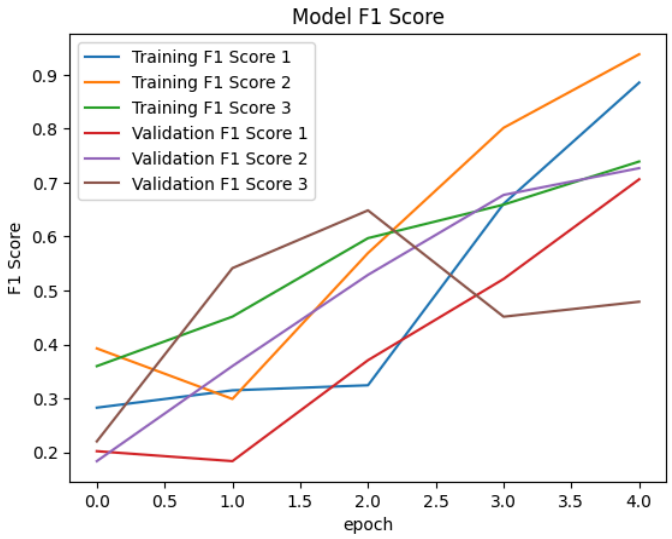
2.2.3. LSTM

2.2.3.1. Pengaruh jumlah layer LSTM

Pada pengujian ini, digunakan 3 model dengan jumlah layer LSTM yang berbeda-beda. Model pertama memiliki 1 *layer* LSTM yang terdiri dari 16 neuron. Model kedua memiliki 3 *layer* LSTM, masing-masing terdiri dari 16 neuron. Model ketiga memiliki 7 *layer* LSTM, masing-masing terdiri dari 16 neuron. Berikut adalah perbandingan hasil training dan validasi dari ketiga model:



Gambar 2.2.3.1.1. Grafik Loss Pengaruh Jumlah Layer LSTM



Gambar 2.2.3.1.2. Grafik F1 Score Pengaruh Jumlah Layer LSTM

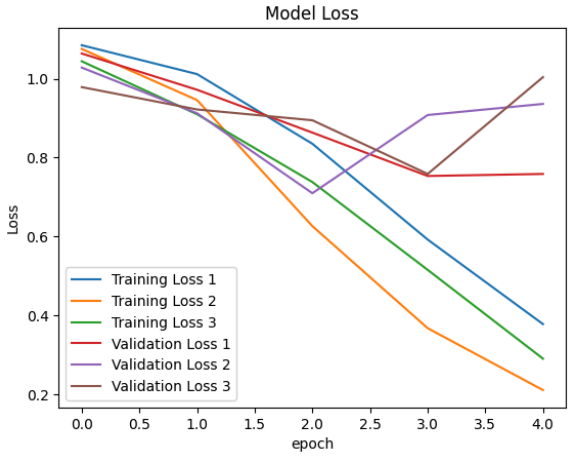
```
Test accuracy 1: 0.6437
Test accuracy 2: 0.6526
Test accuracy 3: 0.5486
```

Gambar 2.2.3.1.3. Perbandingan Akurasi Pengaruh Jumlah Layer LSTM

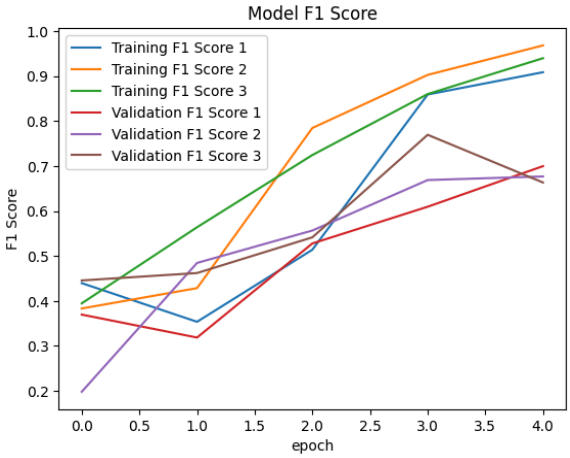
Dapat dilihat bahwa baik dalam tahap *training*, model 3 selalu memiliki *loss* paling rendah di antara ketiga model. Namun, pada tahap validation, meskipun lebih baik pada *epoch-epoch* awal, performa model 3 cenderung lebih buruk pada *epoch-epoch* akhir. Sebaliknya, model 2 cenderung selalu lebih baik daripada model 1. Hasil dari akurasi terhadap *test set* juga menunjukkan bahwa model 3 memiliki hasil terburuk, sedangkan model 2 memiliki hasil terbaik antara ketiga model. Dari eksperimen ini, dapat disimpulkan bahwa peningkatan jumlah *layer* dapat meningkatkan performa model. Namun, penggunaan *layer* yang berlebihan dapat menimbulkan *overfitting* sehingga menimbulkan performa buruk dalam tahap validasi dan testing.

2.2.3.2. Pengaruh banyak cell LSTM per layer

Pada pengujian ini, digunakan 3 model dengan jumlah *cell/neuron* dalam *layer* LSTM yang berbeda-beda. Model pertama memiliki 1 *layer* LSTM yang terdiri dari 32 *neuron*. Model kedua memiliki 1 *layer* LSTM yang terdiri dari 128 *neuron*. Model ketiga memiliki 1 *layer* LSTM yang terdiri dari 512 *neuron*. Berikut adalah perbandingan hasil training dan validasi dari ketiga model:



Gambar 2.2.3.2.1. Grafik Loss Pengaruh Jumlah Cell LSTM



Gambar 2.2.3.2.2. Grafik F1 Score Pengaruh Jumlah Cell LSTM

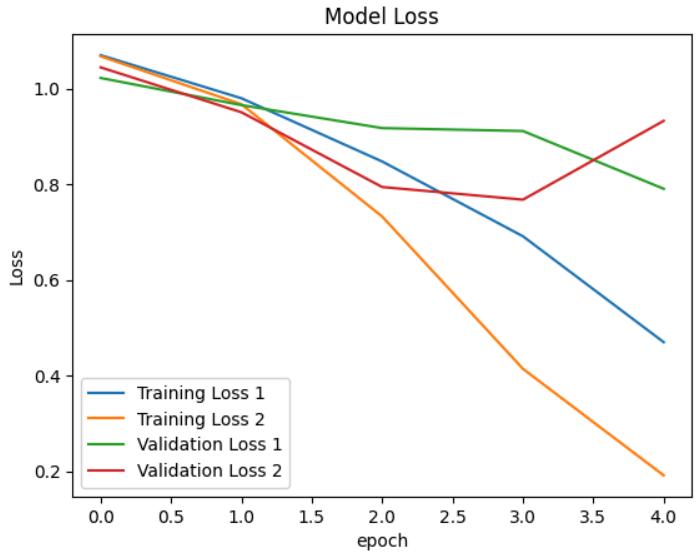
```
Test accuracy 1: 0.6671
Test accuracy 2: 0.6861
Test accuracy 3: 0.6217
```

Gambar 2.2.3.2.3. Perbandingan Akurasi Pengaruh Jumlah Cell LSTM

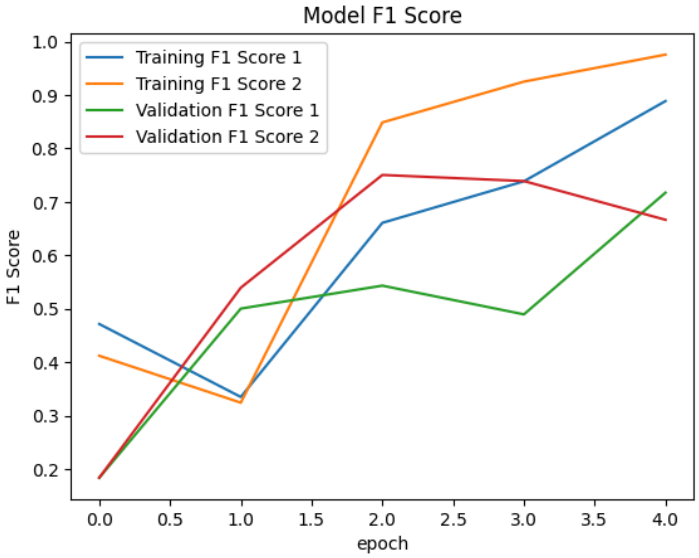
Dapat dilihat bahwa model 3, dengan jumlah *cell* paling banyak, cenderung memiliki performa yang lebih baik pada *epoch-epoch* awal, tetapi memiliki performa yang lebih buruk daripada model 2 pada *epoch-epoch* akhir. Namun, model 2 cenderung selalu lebih baik daripada model 1. Hasil dari akurasi terhadap *test set* juga menunjukkan bahwa model 3 memiliki hasil terburuk, sedangkan model 2 memiliki hasil terbaik antara ketiga model. Dari eksperimen ini, dapat disimpulkan bahwa peningkatan jumlah cell/neuron dapat meningkatkan performa model, tetapi peningkatan yang berlebihan dapat menimbulkan *overfitting* seperti halnya dengan penambahan jumlah *layer*.

2.2.3.3. Pengaruh jenis layer LSTM berdasarkan arah

Pada pengujian ini, digunakan 2 model dengan arah *layer* LSTM yang berbeda. Model pertama memiliki 3 *layer Unidirectional* LSTM yang secara berurut terdiri dari 64, 32, dan 16 *neuron*. Model kedua memiliki 3 *layer Bidirectional* LSTM yang secara berurut terdiri dari 64, 32, dan 16 *neuron*. Berikut adalah perbandingan hasil training dan validasi dari kedua model tersebut:



Gambar 2.2.3.3.1. Grafik Loss Pengaruh Arah LSTM



Gambar 2.2.3.3.2. Grafik F1 Score Pengaruh Arah LSTM

```
Test accuracy 1: 0.6965
Test accuracy 2: 0.6629
```

Gambar 2.2.3.3.3. Perbandingan Akurasi Pengaruh Arah LSTM

Dapat dilihat bahwa tidak terdapat perbedaan yang jelas antara *Unidirectional layer* dan *Bidirectional layer*. Meskipun *Bidirectional layer* memiliki performa yang lebih baik pada *epoch* awal, *Unidirectional layer* memiliki performa yang lebih baik pada *epoch* akhir. Pada hasil *testing* pun, *Unidirectional layer* memiliki hasil yang lebih baik daripada *Bidirectional layer*. Tergantung dengan 'seed' yang diperoleh, *Bidirectional layer* mampu memberikan hasil yang lebih baik daripada *Unidirectional layer*, demikian juga sebaliknya. Namun, *Bidirectional layer* cenderung lebih banyak memberi hasil yang lebih baik. Hal ini mungkin dikarenakan persoalan pemrosesan kalimat, seperti penentuan sentimen, cenderung membutuhkan konteks dari kedua sisi kalimat, tetapi tidak secara signifikan.

BAB 3
KESIMPULAN DAN SARAN

3.1. Kesimpulan

Laporan ini memberikan penjelasan mengenai implementasi dan pengujian berbagai arsitektur model CNN, Simple RNN, dan LSTM. Dari hasil eksperimen dapat disimpulkan hal berikut:

1. Arsitektur LSTM yang lebih dalam (*layer* lebih banyak) dapat meningkatkan performa model, tetapi peningkatan yang berlebihan akan menurunkan performa model, yang ditunjukkan dengan nilai akurasi yang lebih rendah dan nilai *loss* yang lebih tinggi pada *epoch* lebih tinggi. Hal ini terjadi karena terjadinya *overfitting* ketika *layer* model bertambah.
2. Arsitektur LSTM yang lebih lebar (jumlah neuron lebih banyak) dapat meningkatkan performa model, tetapi peningkatan yang berlebihan akan menurunkan performa model, yang ditunjukkan dengan nilai akurasi yang lebih rendah dan nilai *loss* yang lebih tinggi pada *epoch* lebih tinggi. Hal ini terjadi karena terjadinya *overfitting* ketika jumlah neuron model bertambah.
3. Pada arsitektur Simple RNN, jumlah layer, cell dan arah dari model dapat mempengaruhi kinerja model. Jumlah layer dan cell yang lebih banyak (sampai tahap tertentu) dapat memperbagus kinerja model. Tetapi, jumlah cell dan layer terlalu berlebihan dapat menyebabkan overfitting yang justru menurunkan kinerja model secara keseluruhan. Secara umum, kinerja dari model dengan bidirectional layer lebih baik dari model dengan unidirectional layer karena memiliki akses konteks dua arah.
4. Arsitektur CNN menunjukkan bahwa arsitektur yang lebih dalam (*convolution layer* yang bertambah) dapat meningkatkan performa pembelajaran model CNN, ditunjukkan oleh nilai *loss* dan F1 Score yang mempertahankan trennya hingga *epoch* terakhir. Sementara itu, arsitektur yang lebih lebar (jumlah *filter* yang lebih banyak) meningkatkan performa pembelajaran model CNN, dilihat juga dari tren nilai *loss* dan F1 Score yang sama.
5. Arsitektur CNN memiliki perilaku yang beragam ketika menggunakan ukuran *kernel* yang berbeda-beda. Tren yang ditunjukkan berbeda-beda, baik nilai awal maupun akhir, baik nilai *loss* maupun F1 Score, sehingga tidak dapat disimpulkan pengaruhnya pada model CNN.

3.2. Saran

1. Hasil prediksi model dan label sebenarnya dapat di-*plot* menjadi sebuah *heatmap* untuk menyoroti *error* pada kinerja model.

LAMPIRAN

PEMBAGIAN TUGAS

Nim	Nama	Tugas
13522002	**Ariel Herfrison**	**Melakukan implementasi & pengujian LSTM**
13522016	**Zachary Samuel Tobing**	**Melakukan implementasi & pengujian CNN**
13522030	**Imam Hanif Mulyarahman**	**Melakukan implementasi & pengujian Simple RNN**

REFERENSI

Mitchell, T. M. (1997). Machine learning (Vol. 1). McGraw-hill New York.

Repository Github:

https://github.com/Ariel-HS/Tubes2_ML