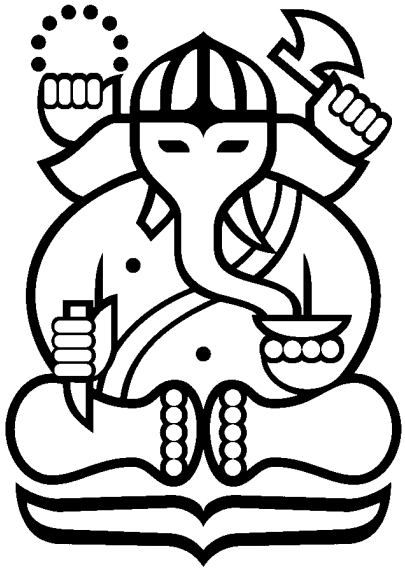
LAPORAN TUGAS BESAR 2

IF3270 PEMBELAJARAN MESIN

Convolutional Neural Network & Recurrent Neural Network



Disusun Oleh:

13522102 Hayya Zuhailii Kinasih

13522104 Diana Tri Handayani

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

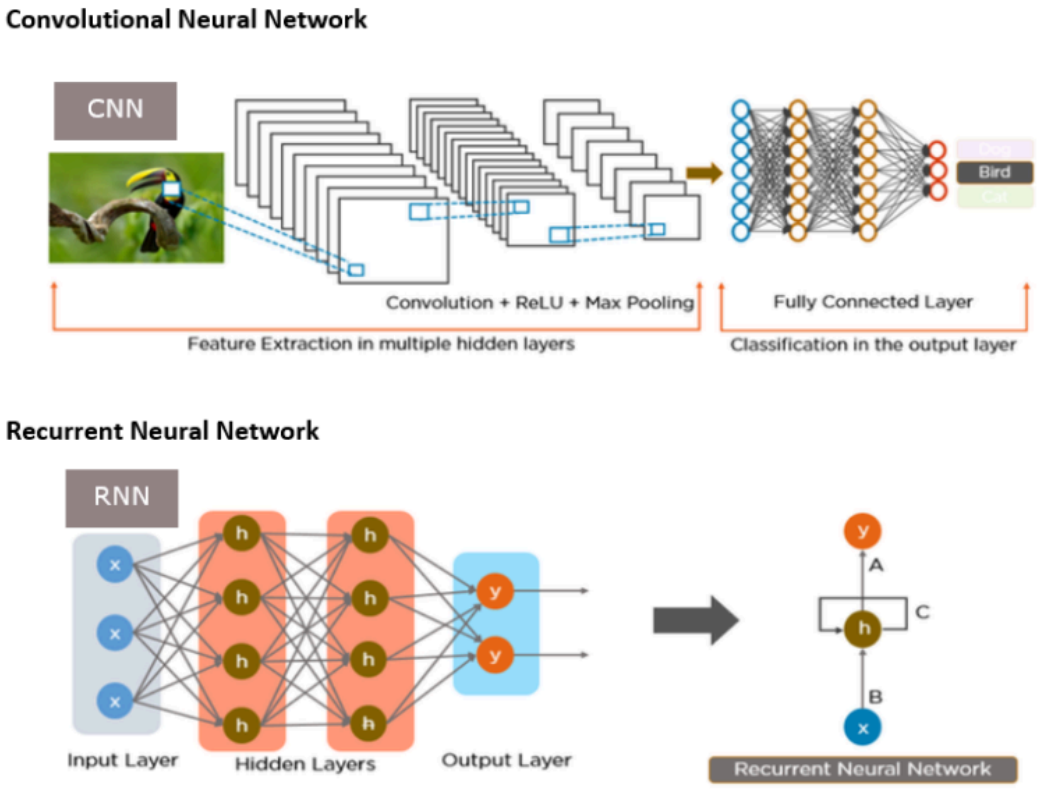
2025

DAFTAR ISI

DAFTAR GAMBAR

BAB I
DESKRIPSI MASALAH

1.1. Deskripsi Persoalan



Gambar 1.1.1 Ilustrasi Arsitektur CNN dan RNN

(Sumber: Spesifikasi Tugas Besar 2 Pembelajaran Mesin)

Convolutional Neural Network (CNN) dan Recurrent Neural Network (RNN) adalah dua buah implementasi Artificial Neural Network (ANN). Sesuai namanya, algoritma tersebut terinspirasi dari cara kerja jaringan neuron pada otak manusia. ANN terdiri dari tiga komponen utama, yaitu *input layer*, *hidden layer(s)*, dan *output layer*. Setiap jenis *layer* tersebut terdiri atas *neuron* yang jumlahnya sesuai dengan kebutuhan persoalan.

CNN adalah penerapan ANN menggunakan *layer* konvolusi dan melakukan operasi dengan topologi *grid*, sehingga CNN paling cocok digunakan dengan *input* berupa gambar, karena dapat meng *capture* fitur spasial dari data. *Layer* CNN juga menggunakan *sparse connectivity* dan *parameter sharing* yang membuat perhitungan lebih akurat dan efisien.

RNN adalah penerapan ANN yang mempunyai *forward* dan *backward link*, sehingga cocok untuk *input* dimana urutan data penting. Satu unit RNN memiliki *loop* yang mengirimkan informasi antar *neuron*, sehingga dapat memahami konteks keseluruhan *input*. LSTM adalah salah satu modifikasi atas RNN yang bertujuan untuk menjaga dari masalah *vanishing gradient* yang rentan terjadi pada RNN.

1.2. Spesifikasi Tugas

Tugas ini adalah untuk mengimplementasikan fungsi *forward propagation* untuk beberapa arsitektur berikut:

1. CNN
2. Simple RNN
3. LSTM

Selain implementasi, terdapat ketentuan untuk melakukan pelatihan:

1. Suatu model CNN untuk *image classification* dengan *library* Keras dan dengan dataset CIFAR-10 serta pengujian dengan variasi jumlah *layer* konvolusi, banyak filter per *layer* konvolusi, ukuran *filter* per *layer* konvolusi, dan jenis *pooling layer*.
2. Suatu model RNN untuk *text classification* dengan dataset NusaX-Sentiment (Bahasa Indonesia) dan dengan menggunakan Keras serta pengujian dengan variasi jumlah *layer* RNN, jumlah *cell* setiap *layer* RNN, dan tipe *layer* RNN.
3. Suatu model LSTM untuk *text classification* dengan dataset NusaX-Sentiment (Bahasa Indonesia) dan dengan menggunakan Keras serta pengujian dengan variasi jumlah *layer* LSTM, banyak *cell* LSTM per *layer,* dan jenis *layer* LSTM berdasarkan arah.

Seluruh implementasi menggunakan bahasa *python*.

BAB II
PEMBAHASAN

1. Penjelasan Implementasi

1.1. Deskripsi Kelas dan Fungsi

1.1.1. CNN

Nama Kelas dan Atribut	Method	Deskripsi
Conv2D *Atribut:* W (weight) b (bias) stride padding	forward(self, x) x berupa array 4D: batch, height, width, channels.	Melakukan tahap *convolution* pada *forward propagation* CNN
Activation	linear(x) relu(x) softmax(x) sigmoid(x) tanh(x) x berupa array angka	Melakukan operasi *activation function* baik untuk *detector stage* maupun *fully connected* pada CNN
MaxPooling2D *Atribut:* pool_size stride	forward(self, x) x berupa array 4D: batch, height, width, channels.	Melakukan tahap *pooling* dengan jenis *max pooling* pada *forward propagation* CNN
AveragePooling2D *Atribut:* pool_size stride	forward(self, x) x berupa array 4D: batch, height, width, channels.	Melakukan tahap *pooling* dengan jenis *average pooling* pada *forward propagation* CNN
Dense *Atribut:* W (weight) b (bias)	forward(self, x) x berupa array 2D: batch, input_dimension	Melakukan tahap *dense layer* atau *fully connected* pada arsitektur CNN
Flatten	forward(self, x) x berupa array 4D: batch, height, width, channels.	Melakukan pengubahan *input* multidimensi menjadi vektor 1 dimensi per sampel *(batch_size, features*), sehingga bisa dihubungkan ke *layer dense* (*fully connected*).

1.1.2. Simple RNN

Nama Fungsi	Deskripsi
rnn_unidirectional(x, W_x, W_h, b, return_sequences=False)	Melakukan *forward propagation* pada *layer* RNN pada satu arah.
rnn_bidirectional(x, W_x_f, W_h_f, b_f, W_x_b, W_h_b, b_b, return_sequences=False)	Melakukan *forward propagation* pada *layer* RNN pada dua arah.

1.1.3. LSTM

Nama Fungsi	Deskripsi
lstm_cell(x_t, h_prev, c_prev, W_x, W_h, b)	Menghitung *output* dari *gate-gate* pada *cell* LSTM.
lstm_unidirectional(x, W_x, W_h, b, return_sequences=False)	Melakukan *forward propagation* pada *layer* LSTM pada satu arah.
lstm_bidirectional(x, W_x_f, W_h_f, b_f, W_x_b, W_h_b, b_b, return_sequences=False)	Melakukan *forward propagation* pada *layer* LSTM pada dua arah.

1.1.4. Utils

Nama Fungsi	Deskripsi
embedding(x, embedding_matrix)	Melakukan *forward propagation* pada *layer* Embedding.
dense(x, W, b)	Melakukan *forward propagation* pada *layer* Dense dengan fungsi aktivasi *linear*.
softmax(x)	Melakukan perhitungan softmax.
sigmoid(x)	Melakukan perhitungan sigmoid.

1.2. *Forward Propagation*

1.2.1. CNN

Implementasi untuk *convolution stage* pada CNN sebagai berikut:

```
import numpy as np

class Conv2D:
    def __init__(self, W, b, stride=1, padding='same'):
        self.W = W  # shape: (kh, kw, in_channels, out_channels)
        self.b = b
        self.stride = stride
        self.padding = padding

    def forward(self, x):
        batch_size, in_h, in_w, in_c = x.shape
        kh, kw, _, out_c = self.W.shape

        if self.padding == 'same':
            pad_h = (kh - 1) // 2
            pad_w = (kw - 1) // 2
        else:
            pad_h = pad_w = 0

        out_h = (in_h + 2 * pad_h - kh) // self.stride + 1
        out_w = (in_w + 2 * pad_w - kw) // self.stride + 1

        x_padded = np.pad(x, ((0, 0), (pad_h, pad_h), (pad_w, pad_w), (0, 0)),
mode='constant')

        output = np.zeros((batch_size, out_h, out_w, out_c))

        for b in range(batch_size):
            for oc in range(out_c):
                for i in range(0, out_h):
                    for j in range(0, out_w):
                        for ic in range(in_c):
                            region = x_padded[b,
                                              i * self.stride:i * self.stride
+ kh,
                                              j * self.stride:j * self.stride
+ kw,
```

```
                                                    ic]
                            output[b, i, j, oc] += np.sum(region * self.W[:,
:, ic, oc])
                output[b, :, :, oc] += self.b[oc]

        return output
```

Kelas ini menerima bobot W (berukuran (kernel_height, kernel_width, in_channels, out_channels)) dan bias b, serta parameter opsional seperti stride dan padding. Pada metode forward, input x akan diproses dengan cara menggeser kernel konvolusi di atas setiap channel input (dengan padding jika diperlukan), lalu menghitung hasil perkalian elemen-sejajar antara patch input dan kernel, yang kemudian dijumlahkan dan ditambahkan bias untuk menghasilkan output. Proses ini dilakukan untuk setiap sampel dalam batch dan untuk setiap output channel. Padding 'same' akan mempertahankan dimensi output sama dengan input (jika stride=1), sedangkan padding 'valid' tidak menambahkan padding.

Implementasi untuk *activation function* pada CNN sebagai berikut:

```
import numpy as np

class Activation:
    @staticmethod
    def linear(x):
        return x

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

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

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

    @staticmethod
    def tanh(x):
```

```
        return np.tanh(x)
```

Setiap fungsi aktivasi diimplementasikan berdasarkan definisi masing-masing fungsi secara matematis.

Implementasi untuk *pooling stage* pada CNN sebagai berikut:

```
import numpy as np

class MaxPooling2D:
    def __init__(self, pool_size=2, stride=2):
        self.pool_size = pool_size
        self.stride = stride

    def forward(self, x):
        batch_size, h, w, c = x.shape
        out_h = (h - self.pool_size) // self.stride + 1
        out_w = (w - self.pool_size) // self.stride + 1
        output = np.zeros((batch_size, out_h, out_w, c))

        for i in range(out_h):
            for j in range(out_w):
                x_slice = x[:,
                            i*self.stride:i*self.stride+self.pool_size,
                            j*self.stride:j*self.stride+self.pool_size,
                            :]
                output[:, i, j, :] = np.max(x_slice, axis=(1, 2))
        return output

class AveragePooling2D:
    def __init__(self, pool_size=2, stride=2):
        self.pool_size = pool_size
        self.stride = stride

    def forward(self, x):
        batch_size, h, w, c = x.shape
        out_h = (h - self.pool_size) // self.stride + 1
        out_w = (w - self.pool_size) // self.stride + 1
        output = np.zeros((batch_size, out_h, out_w, c))

        for i in range(out_h):
            for j in range(out_w):
```

```
                x_slice = x[:,
                            i*self.stride:i*self.stride+self.pool_size,
                            j*self.stride:j*self.stride+self.pool_size,
                            :]
                output[:, i, j, :] = np.mean(x_slice, axis=(1, 2))
        return output
```

Pada setiap *patch*, MaxPooling2D mengambil nilai maksimum, sedangkan AveragePooling2D mengambil nilai rata-rata. Operasi ini dilakukan secara terpisah untuk setiap channel dan untuk semua batch secara bersamaan.

Implementasi untuk *flatten* dan *dense layer* pada CNN sebagai berikut:

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

class Dense:
    def __init__(self, W, b):
        self.W = W
        self.b = b

    def forward(self, x):
        return x @ self.W + self.b
```

Kelas Flatten mengubah input multidimensi menjadi vektor 1 dimensi. Sementara itu, kelas Dense merepresentasikan *layer fully connected* yang melakukan transformasi linier terhadap input menggunakan bobot (W) dan bias (b) yang diberikan, dengan operasi x *dot product* W + b.

Implementasi untuk membandingkan F1-*score* hasil implementasi dengan Keras dan implementasi *from scratch*:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from sklearn.metrics import f1_score
from conv_layer import Conv2D
from pooling import MaxPooling2D, AveragePooling2D
from activation import Activation
from dense_layer import Dense, Flatten

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype("float32") / 255.0
```

```
x_test = x_test.astype("float32") / 255.0
y_train = y_train.flatten()
y_test = y_test.flatten()

# testing subset data (100 data uji)
x_test, y_test = x_test[:100], y_test[:100]

# Model Keras
keras_model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(8, 3, activation='relu', padding='same',
input_shape=(32, 32, 3)),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(16, 3, activation='relu', padding='same'),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

keras_model.compile(optimizer='adam',
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])

keras_model.fit(x_train, y_train, epochs=3, batch_size=32, verbose=1)

keras_model.save("cnn_cifar10_model.h5")

loaded_model = tf.keras.models.load_model("cnn_cifar10_model.h5")

keras_preds = loaded_model.predict(x_test).argmax(axis=1)
f1_keras = f1_score(y_test, keras_preds, average='macro')
print("Macro F1-score (Keras loaded):", f1_keras)

w1, b1 = loaded_model.layers[0].get_weights()
w2, b2 = loaded_model.layers[2].get_weights()
w3, b3 = loaded_model.layers[5].get_weights()
w4, b4 = loaded_model.layers[6].get_weights()

# Model from Scratch
x = x_test
x = Conv2D(w1, b1).forward(x)
```

```
x = Activation.relu(x)
x = MaxPooling2D().forward(x)
x = Conv2D(w2, b2).forward(x)
x = Activation.relu(x)
x = MaxPooling2D().forward(x)
x = Flatten().forward(x)
x = Dense(w3, b3).forward(x)
x = Activation.relu(x)
x = Dense(w4, b4).forward(x)
predictions = Activation.softmax(x)

predicted_labels = predictions.argmax(axis=1)
f1_manual = f1_score(y_test, predicted_labels, average='macro')
print("Macro F1-score (Manual forward):", f1_manual)
print(f"Selisih    : {abs(f1_keras-f1_manual)}")
```

Implementasi dilakukan dengan membuat suatu model dengan Keras terlebih dahulu, lalu menyimpan bobot yang terbentuk dalam file h5. Bobot yang tersimpan lalu dimuat untuk perhitungan pada *forward propagation from scratch*. Hasil dari implementasi di atas sebagai berikut:

```
Macro F1-score (Keras loaded): 0.6376809124501822
Macro F1-score (Manual forward): 0.6376809124501822
Selisih    : 0.0
```

Implementasi untuk membandingkan dapat divariasikan dengan cara mengubah kode pada bagian model keras dan juga model *from scratch* yang sesuai.

1.2.2. Simple RNN

Implementasi *forward propagation* untuk Simple RNN satu arah adalah sebagai berikut:

```
function rnn_unidirectional(X, W_x, W_h, b, return_sequences=false) returns
output of neural network
    batch_size, time_steps, _ ← X.shape
    h ← np.zeros((batch_size, W_h.shape[0]))

    outputs ← []
    for each timestep do
        h ← np.tanh(np.dot(x[:, t, :], W_x) + np.dot(h, W_h) + b)
        outputs ← outputs.append(h)
```

```
    if return_sequences then
        → np.stack(outputs, axis=1)
    else
        → h
```

Untuk setiap *time step*, dilakukan perhitungan dengan rumus

$$h_t = tanh(W\_x \cdot x_t + (W\_h \cdot h_{t-1} + b))$$

Sedangkan untuk Simple RNN dua arah, dilakukan dengan memanfaatkan fungsi di atas dalam implementasi berikut:

```
function rnn_bidirectional(X, W_x_f, W_h_f, b_f, W_x_b, W_h_b, b_b,
return_sequences=false) returns output of neural network
    h_f ← rnn_unidirectional(x, W_x_f, W_h_f, b_f, return_sequences)
    h_b ← rnn_unidirectional(x[:, ::-1, :], W_x_b, W_h_b, b_b,
return_sequences)

    → np.concatenate([h_f, h_b], axis=-1)
```

Pengujian perbandingan implementasi *forward propagation from scratch* dengan implementasi Keras dilakukan dengan arsitektur model seperti berikut:

```
<Embedding name=embedding_8, built=True>,
<SimpleRNN name=simple_rnn_11, built=True>,
<Bidirectional name=bidirectional_10, built=True>,
<Dropout name=dropout_8, built=True>,
<Dense name=dense_8, built=True>
```

Hasilnya adalah sebagai berikut:

```
Keras Macro F1: 0.5017517713182456
Manual Macro F1: 0.5017517713182456
```

Implementasi manual dan implementasi Keras menghasilkan nilai F1 Score yang sama.

1.2.3. LSTM

Implementasi *forward propagation* untuk LSTM satu arah adalah sebagai berikut:

```
function lstm_unidirectional(X, W_x, W_h, b, return_sequences=false) returns
output of neural network
    batch_size, time_steps, _ ← X.shape
    h ← np.zeros((batch_size, W_h.shape[0]))
    c ← np.zeros((batch_size, W_h.shape[0]))

    outputs ← []
    for each timestep do
        h, c ← lstm_cell(x[:, t, :], h, c, W_x, W_h, b)
        outputs ← outputs.append(h)
    if return_sequences then
        → np.stack(outputs, axis=1)
    else
        → h
```

Untuk setiap *time step*, dilakukan perhitungan untuk *cell* dalam LSTM dengan implementasi berikut:

```
function lstm_cell(x_t, h_prev, c_prev, W_x, W_h, b) returns output of neural
network and cell state
    hidden_size ← h_prev.shape[1]
    z ← np.dot(x_t, W_x) + np.dot(h_prev, W_h) + b
    i ← sigmoid(z[:, :hidden_size])
    f ← sigmoid(z[:, hidden_size:2*hidden_size])
    c_hat ← np.tanh(z[:, 2*hidden_size:3*hidden_size])
    o ← sigmoid(z[:, 3*hidden_size:])

    c_t ← f * c_prev + i * c_hat
    h_t ← o * np.tanh(c_t)

    → h_t, c_t
```

Perhitungan yang dilakukan pada fungsi tersebut adalah sebagai berikut:

$$i = \sigma(W\_xi \cdot x_t + (W\_hi \cdot h_{t-1} + b))$$

$$f = \sigma(W\_xf \cdot x_t + (W\_hf \cdot h_{t-1} + b))$$

$$c_{hat} = tanh(W\_xc \cdot x_t + (W\_hc \cdot h_{t-1} + b))$$

$$o = \sigma(W\_xo \cdot x_t + (W\_ho \cdot h_{t-1} + b))$$

$$c_t = f \odot c_{t-1} + i \odot c_{hat}$$

$$h_t = tanh(c_t) \odot o$$

Sedangkan untuk Simple LSTM dua arah, dilakukan dengan memanfaatkan fungsi di atas dalam implementasi berikut:

```
function lstm_bidirectional(X, W_x_f, W_h_f, b_f, W_x_b, W_h_b, b_b,
return_sequences=false) returns output of neural network
    h_f ← lstm_unidirectional(x, W_x_f, W_h_f, b_f, return_sequences)
    h_b ← lstm_unidirectional(x[:, ::-1, :], W_x_b, W_h_b, b_b,
return_sequences)

    → np.concatenate([h_f, h_b], axis=-1)
```

Pengujian perbandingan implementasi *forward propagation from scratch* dengan implementasi Keras dilakukan dengan arsitektur model seperti berikut:

```
<Embedding name=embedding_11, built=True>,
<LSTM name=lstm_17, built=True>,
<Bidirectional name=bidirectional_13, built=True>,
<Dropout name=dropout_11, built=True>,
<Dense name=dense_11, built=True>
```

Hasilnya adalah sebagai berikut:

```
Keras Macro F1: 0.6746899545089828
Manual Macro F1: 0.6746899545089828
```

Implementasi manual dan implementasi Keras menghasilkan nilai F1 Score yang sama.

2. Hasil Pengujian

2.1. CNN

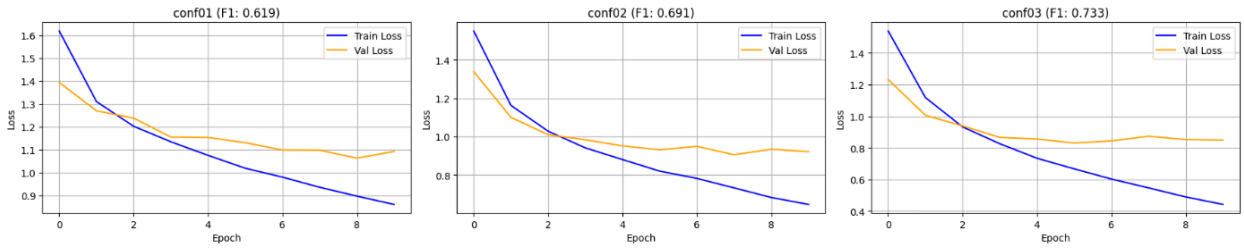
Berikut adalah parameter yang digunakan dalam pengujian:

Jumlah *Layer* Konvolusi	Jumlah Filter	Ukuran Kernel	*Pooling*
Pengaruh Jumlah *Layer* Konvolusi			

1	[32]	[3]	*max*
2	[32, 64]	[3, 3]	*max*
3	[32, 64, 128]	[3, 3, 3]	*max*
Pengaruh Banyak *Filter* per *Layer* Konvolusi			
2	[16, 32]	[3, 3]	*max*
2	[32, 64]	[3, 3]	*max*
2	[64, 128]	[3, 3]	*max*
Pengaruh Ukuran *Filter* per *Layer* Konvolusi			
2	[32, 64]	[3, 3]	*max*
2	[32, 64]	[5, 5]	*max*
2	[32, 64]	[7, 7]	*max*
Pengaruh Jenis *Pooling Layer*			
2	[32, 64]	[3, 3]	*max*
2	[32, 64]	[3, 3]	*average*

Dengan parameter *batch size* sebesar 64 dan *epochs* sebanyak 10.

2.1.1. Pengaruh jumlah *layer* konvolusi



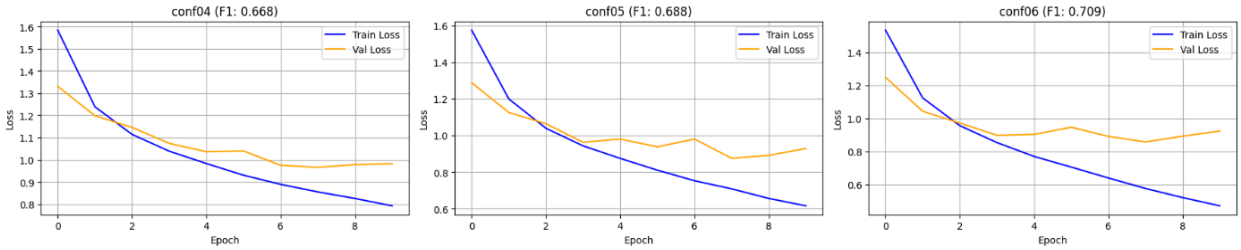
Gambar 2.1.1.1 Grafik Loss pada CNN dengan jumlah *layer* 1, 2, dan 3 (kiri ke kanan)

Jumlah *Layer*	**Macro F1 *Score***
1	0.619
2	0.691

3	0.733

Berdasarkan hasil percobaan yang menghasilkan grafik *loss* dan nilai *macro* F1-*score* di atas, dapat disimpulkan bahwa semakin banyak jumlah *layer* pada suatu model, maka performa prediksinya semakin baik. Pada eksperimen ini, model dengan 3 *layer* konvolusi menunjukkan nilai F1-*score* lebih tinggi dibanding 1 dan 2 *layer*. Model dengan lebih banyak *layer* konvolusi cenderung belajar fitur yang lebih kompleks dan abstrak sehingga memberikan performa prediksi yang lebih baik. Namun, jumlah *layer* yang terlalu banyak bisa menyebabkan *overfitting* jika data pelatihan tidak cukup besar atau regularisasi kurang optimal (ditandai dengan kenaikan val_loss setelah epoch tertentu), meskipun pada eksperimen kali ini tidak terjadi.

2.1.2. Pengaruh banyak *filter* per *layer* konvolusi



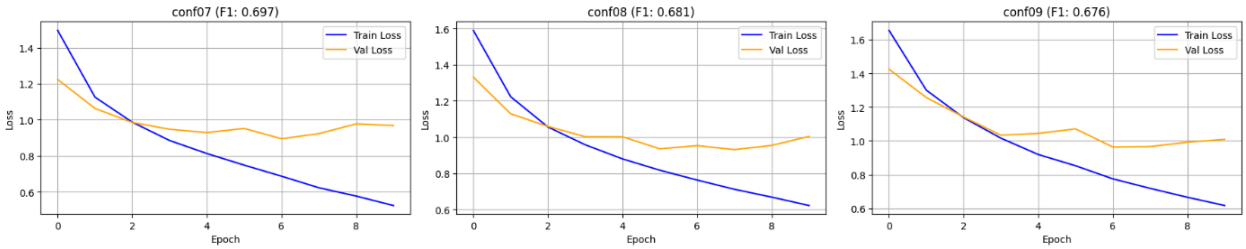
Gambar 2.1.2.1 Grafik Loss pada CNN dengan banyak *filter* [16, 32], [32, 64], dan [64, 128] (kiri ke kanan)

Banyak *Filter*	**Macro F1 *Score***
[16, 32]	0.668
[32, 64]	0.688
[64, 128]	0.709

Berdasarkan hasil percobaan yang menghasilkan grafik *loss* dan nilai *macro* F1-*score* di atas, dapat disimpulkan bahwa semakin banyak *filter* per *layer* pada suatu model, maka performa model cenderung menghasilkan F1-*score* yang lebih baik dan *loss* yang lebih

rendah. Hal tersebut karena *filter* lebih banyak memungkinkan model menangkap lebih banyak fitur penting dari gambar. Namun, jumlah *filter* yang terlalu besar meningkatkan kompleksitas dan waktu pelatihan, dan juga berisiko *overfitting*.

2.1.3. Pengaruh ukuran *filter* per *layer* konvolusi

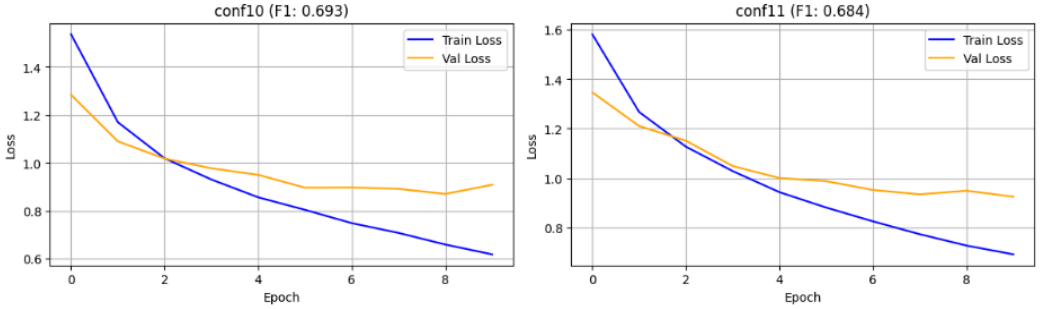


Gambar 2.1.3.1 Grafik Loss pada CNN dengan ukuran filter [3, 3], [5, 5], dan [7, 7] (kiri ke kanan)

Ukuran *Filter*	**Macro F1 *Score***
[3, 3]	0.697
[5, 5]	0.681
[7, 7]	0.676

Berdasarkan hasil percobaan yang menghasilkan grafik *loss* dan nilai *macro* F1-*score* di atas, dapat disimpulkan bahwa semakin kecil ukuran *filter* per *layer* pada suatu model, maka performa prediksinya semakin baik. Hal tersebut karena ukuran kernel *filter* mempengaruhi jangkauan fitur lokal yang dipelajari. *Filter* yang terlalu besar dapat menangkap konteks yang lebih luas, tapi risiko kehilangan detail kecil. Sedangkan *filter* yang lebih kecil memungkinkan model membangun representasi fitur secara bertingkat dan lebih detail. Sehingga dari percobaan, model dengan kernel 3x3 pada semua *layer* memberikan hasil terbaik dengan *loss* lebih kecil dan F1-score lebih tinggi dibanding kombinasi yang menggunakan *filter* 5x5 dan 7x7.

2.1.4. Pengaruh jenis *pooling layer*



Gambar 2.1.4.1 Grafik Loss pada CNN dengan jenis pooling *layer max* (kiri) dan *average* (kanan).

Jenis *Pooling*	**Macro F1 *Score***
max	0.693
average	0.684

Berdasarkan hasil percobaan yang menghasilkan grafik *loss* dan nilai *macro* F1-*score* di atas, dapat disimpulkan bahwa *max pooling* menghasilkan F1-*score* lebih tinggi dan *loss* yang lebih rendah dibanding *average pooling*. Hal tersebut karena *max pooling* cenderung mempertahankan fitur paling dominan, sedangkan *average pooling* mengambil rata-rata fitur dalam area *pooling*. Sehingga *average* pooling cenderung menghasilkan *loss* yang lebih tinggi dan F1-*score* sedikit lebih rendah karena fitur penting kurang terjaga dengan baik.

2.2. Simple RNN

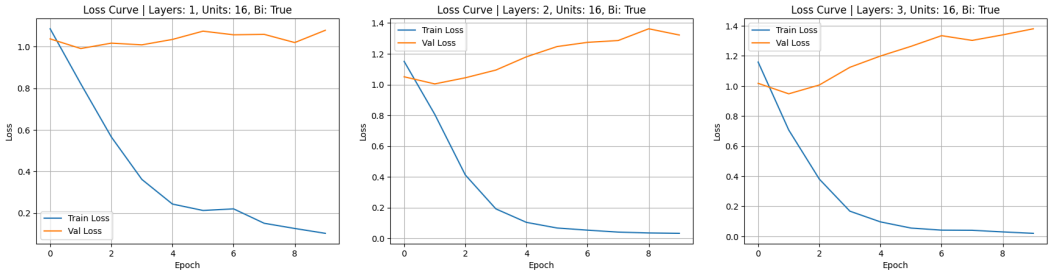
Berikut adalah parameter yang digunakan dalam pengujian:

Jumlah *Layer*	Banyak *Cell* per *Layer*	Jenis *Layer*
Pengaruh Jumlah *Layer* RNN		
1	16	Bidirectional
2	16	Bidirectional
3	16	Bidirectional

Pengaruh Banyak *Cell* RNN per *Layer*		
1	8	Bidirectional
1	16	Bidirectional
1	32	Bidirectional
Pengaruh Jenis *Layer* RNN Berdasarkan Arah		
1	16	Bidirectional
1	16	Unidirectional

Dengan parameter *batch size* sebesar 32 dan *epochs* sebanyak 10.

2.2.1. Pengaruh jumlah *layer* RNN



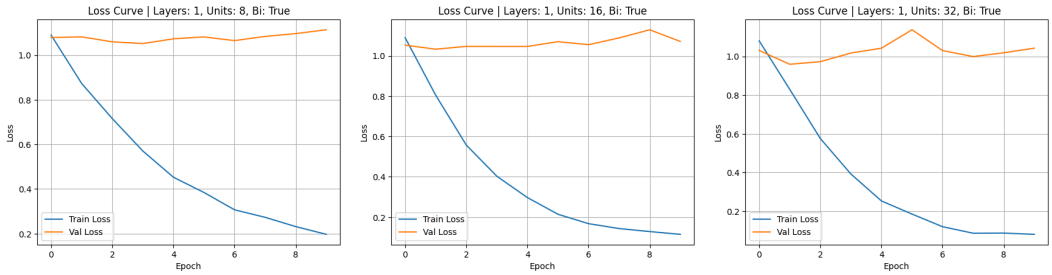
Gambar 2.2.1.1 Grafik Loss pada RNN dengan jumlah *layer* 1, 2, dan 3 (kiri ke kanan)

Jumlah *Layer*	**Macro F1 *Score***
1	0.5021
2	0.4664
3	0.5056

Hasil percobaan tidak menunjukkan suatu tren. Secara teori, penambahan jumlah *layer* pastinya membuat model lebih terlatih. Namun, konsekuensinya adalah pelatihan akan membutuhkan waktu yang lebih lama dan memunculkan kemungkinan *overfitting*. Jika dilihat dari grafik, RNN dengan jumlah *layer* dua dan tiga berhasil mencapai *training loss* mendekati

nol. Namun, pada setiap variasi, *validation loss* semakin meningkat seiring bertambahnya *epoch*. Ini berarti terdapat kemungkinan bahwa sudah terjadi *overfitting* sejak layer pertama.

2.2.2. Pengaruh banyak *cell* RNN per *layer*

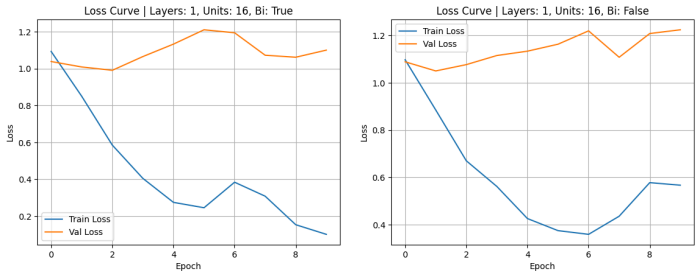


Gambar 2.2.2.1 Grafik Loss pada RNN dengan jumlah cell per *layer* 8, 16, dan 32 (kiri ke kanan)

Jumlah Cell per *Layer*	**Macro F1 *Score***
8	0.4463
16	0.5513
32	0.5199

Berdasarkan hasil eksperimen variasi jumlah *cell* pada setiap *layer* RNN, jumlah *cell* 16 memberikan hasil paling bagus. Secara teori, penambahan jumlah *cell* per *layer* membuat model lebih terlatih. Namun, sama seperti penambahan jumlah *layer*, waktu pelatihan akan lebih lama dan terdapat kemungkinan *overfitting*. Seperti sebelumnya juga, *training loss* turun secara drastis, namun *validation loss* tidak mengalami penurunan. Penurunan performa antara jumlah *cell* 16 dan jumlah *cell* 32 menandakan bahwa jumlah *cell* 32 sudah terlalu besar sehingga terjadi *overfitting*.

2.2.3. Pengaruh jenis *layer* RNN berdasarkan arah



Gambar 2.2.3.1 Grafik Loss pada RNN dengan satu arah dan dua arah (kiri ke kanan)

Tipe *Layer*	**Macro F1 *Score***
Bidirectional	0.5519
Unidirectional	0.3648

Berdasarkan hasil eksperimen variasi tipe *layer* RNN, *layer* Bidirectional memberikan hasil yang jauh lebih baik daripada *layer* Unidirectional. Secara teori, *layer* Bidirectional memang cocok untuk klasifikasi sentimen karena membantu memahami konteks suatu kalimat, dibandingkan dengan *layer* Unidirectional yang tidak mempertimbangkan konteks dari akhir kalimat.

2.3. LSTM

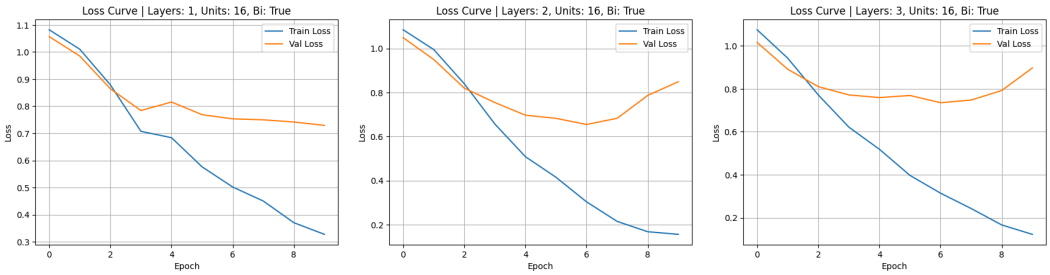
Berikut adalah parameter yang digunakan dalam pengujian:

Jumlah *Layer*	Banyak *Cell* per *Layer*	Jenis *Layer*
Pengaruh Jumlah *Layer* LSTM		
1	16	Bidirectional
2	16	Bidirectional
3	16	Bidirectional
Pengaruh Banyak *Cell* LSTM per *Layer*		
1	8	Bidirectional

1	16	Bidirectional
1	32	Bidirectional
Pengaruh Jenis *Layer* LSTM Berdasarkan Arah		
1	16	Bidirectional
1	16	Unidirectional

Dengan parameter *batch size* sebesar 32 dan *epochs* sebanyak 10.

2.3.1. Pengaruh jumlah *layer* LSTM



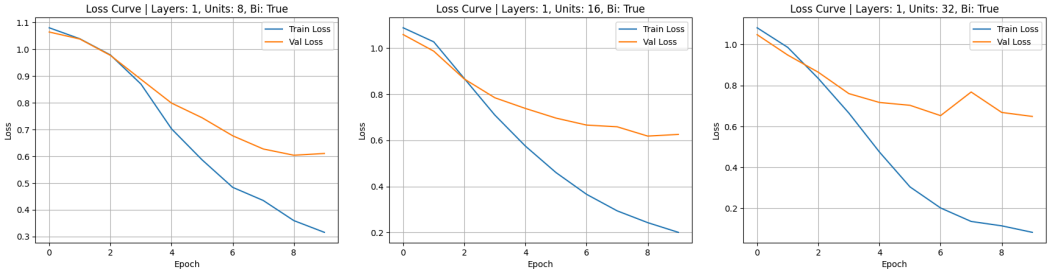
Gambar 2.3.1.1 Grafik Loss pada LSTM dengan jumlah *layer* 1, 2, dan 3 (kiri ke kanan)

Jumlah *Layer*	**Macro F1 *Score***
1	0.7067
2	0.6844
3	0.6877

Hasil percobaan menunjukkan performa yang menjadi lebih buruk dengan penambahan *layer*. Secara teori, penambahan jumlah *layer* pastinya membuat model lebih terlatih. Namun, konsekuensinya adalah pelatihan akan membutuhkan waktu yang lebih lama dan memunculkan kemungkinan *overfitting*. Jika dilihat dari grafik, LSTM dengan jumlah *layer* satu berhasil menurunkan *validation loss*. Namun, pada jumlah *layer* yang semakin banyak, terdapat titik dimana *validation loss* meningkat. Ini berarti terdapat kemungkinan bahwa terjadi *overfitting* pada *layer-layer* tersebut.

LSTM juga memberikan hasil yang jauh lebih baik dibandingkan dengan RNN karena menghindari masalah *vanishing gradient* yang terjadi pada RNN.

2.3.2. Pengaruh banyak cell LSTM per *layer*

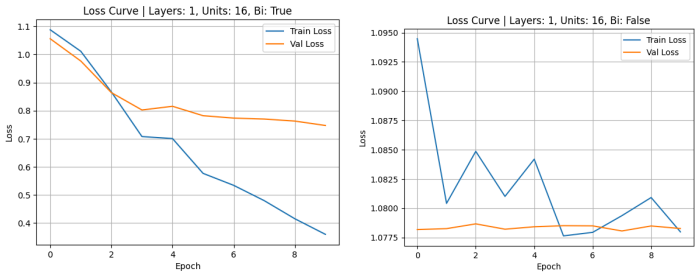


Gambar 2.3.2.1 Grafik Loss pada LSTM dengan jumlah cell per *layer* 8, 16, dan 32 (kiri ke kanan)

Jumlah Cell per *Layer*	**Macro F1 *Score***
8	0.7738
16	0.7944
32	0.7946

Berdasarkan hasil eksperimen variasi jumlah *cell* pada setiap *layer* LSTM, jumlah *cell* 32 memberikan hasil paling bagus, walaupun hanya berbeda tipis dengan jumlah *cell* 16. Secara teori, penambahan jumlah *cell* per *layer* membuat model lebih terlatih. Namun, sama seperti penambahan jumlah *layer*, waktu pelatihan akan lebih lama dan terdapat kemungkinan *overfitting*. Pada semua contoh, terjadi penurunan *validation loss*, walaupun terdapat sedikit lonjakan pada jumlah *cell* 32.

2.3.3. Pengaruh jenis *layer* LSTM berdasarkan arah



Gambar 2.3.3.1 Grafik Loss pada LSTM dengan satu arah dan dua arah (kiri ke kanan)

Tipe *Layer*	**Macro F1 *Score***
Bidirectional	0.6306
Unidirectional	0.1836

Berdasarkan hasil eksperimen variasi tipe *layer* LSTM, *layer* Bidirectional memberikan hasil yang jauh lebih baik daripada *layer* Unidirectional. Secara teori, *layer* Bidirectional memang cocok untuk klasifikasi sentimen karena membantu memahami konteks suatu kalimat, dibandingkan dengan *layer* Unidirectional yang tidak mempertimbangkan konteks dari akhir kalimat.

BAB III
PENUTUP

3.1. Kesimpulan

CNN adalah implementasi *neural network* yang cocok untuk *input* data dengan aspek spasial, sedangkan RNN adalah implementasi *neural network* yang cocok untuk *input* data dengan aspek sekuensial. LSTM adalah variasi dari RNN yang menggunakan *gates* untuk menghindari masalah *vanishing gradient*.

Pada CNN, parameter jumlah *filter* per *layer* konvolusi dan ukuran *filter* pada *layer* konvolusi yang semakin besar akan membuat model semakin terlatih terhadap training data, sehingga membuat hasil prediksi lebih bagus. Sedangkan untuk ukuran filter, semakin kecil akan menghasilkan prediksi yang lebih bagus karena dapat menangkap detail kecil. Pada RNN dan LSTM, jumlah *layer* dan jumlah *cell* per *layer* yang semakin besar akan membuat model semakin terlatih terhadap training data. Namun, seluruh parameter yang terlalu besar (ataupun kecil untuk kasus ukuran filter) dapat membuat waktu pelatihan semakin lama dan memunculkan kemungkinan *overfitting*. *Overfitting* ditunjukkan ketika *training loss* turun tetapi *validation loss* stagnan atau naik.

Untuk parameter jenis *pooling layer* pada CNN, *max pooling* memberikan hasil yang terbaik karena menyimpan fitur paling dominan dari *layer* konvolusi. Untuk parameter jenis *layer* RNN, *bidirectional* memberikan hasil terbaik, terutama untuk tugas sentimen kalimat, karena mempertimbangkan konteks dari keseluruhan kalimat.

Model inferensi CNN, RNN, dan LSTM yang telah dibuat sudah memberikan hasil yang sama dengan inferensi CNN, RNN, dan LSTM dari *library* tensorflow.keras ditunjukkan dengan F1-*score* yang dihasilkan sama.

3.2. Saran

Saran untuk pelatihan CNN & RNN, serta implementasi *forward propagation*-nya *from scratch* ini adalah sebagai berikut:

1. Eksplorasi lebih lanjut atas parameter-parameter yang terlibat dalam pelatihan CNN dan efeknya.
2. Eksplorasi lebih lanjut atas parameter-parameter yang terlibat dalam pelatihan RNN dan efeknya.

PEMBAGIAN TUGAS

NIM	Nama	Tugas
13522102	Hayya Zuhailii Kinasih	○ RNN & LSTM ○ Laporan
13522104	Diana Tri Handayani	○ CNN ○ Laporan

DAFTAR PUSTAKA

[1] Tim Pengajar IF3270. *Materi CNN*. Accessed May 30, 2025.

[2] Tim Pengajar IF3270. *Materi RNN Bagian 1*. Accessed May 30, 2025.

[3] Tim Pengajar IF3270. *Materi RNN Bagian 2*. Accessed May 30, 2025.