

Tugas Besar 1
IF3270 PEMBELAJARAN MESIN
CNN, RNN, DAN LSTM



Disusun Oleh:

Wilson Yusda 13522019

Enrique Yanuar 13522077

Mesach Harmasendro 13522117

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

DAFTAR ISI.....	2
BAB 1 DESKRIPSI persoalan.....	5
BAB 2 IMPLEMENTASI.....	7
2.1. CNN.....	7
2.1.1. Deskripsi Kelas dan fungsi.....	7
2.1.1.1. Kelas Scratch Model.....	7
2.1.1.2. Kelas ScratchConv2D.....	9
2.1.1.3. Kelas ScratchDense.....	11
2.1.1.4. Kelas ScratchFlatten.....	12
2.1.1.5. Kelas ScratchMaxPooling2D.....	12
2.1.1.6. Kelas ScratchAveragePooling2D.....	13
2.1.1.7. Kelas ScratchGlobalAveragePooling2D.....	14
2.1.1.8. Kelas ScratchGlobalMaxPooling2D.....	15
2.1.1.9. Algoritma Luar Untuk Pelatihan dan Plot.....	15
2.1.1.10. Konfigurasi fungsi aktivasi.....	17
2.1.2. Penjelasan Forward Propagation.....	17
2.2. RNN.....	21
2.2.1. Deskripsi Kelas.....	21
2.2.1.1. Kelas SimpleRNNLayer.....	21
2.2.1.2. Kelas EmbeddingLayer.....	23
2.2.1.3. Kelas BidirectionalRNNLayer.....	24
2.2.1.4. Kelas SimpleRNNKeras.....	25
2.2.1.5. Kelas SimpleRNNManual.....	31
2.2.1.6. Kelas FFNN.....	38
2.2.2. Penjelasan Forward Propagation.....	38
2.3. LSTM.....	43
2.3.1. Deskripsi Kelas.....	43
2.3.1.1. Kelas ScratchLSTMClassifier.....	43
2.3.1.2. Kelas ScratchLSTM.....	46
2.3.1.3. Kelas ScratchEmbedding.....	50
2.3.1.4. Kelas ScratchDense.....	51
2.3.2. Deskripsi Fungsi.....	52
2.3.2.1. Fungsi build_lstm_model.....	52
2.3.3. Penjelasan Forward Propagation dan Backward Propagation.....	53
BAB 3 PENGUJIAN.....	58
3.1. Pengujian CNN.....	58
3.1.1. Pengaruh jumlah layer konvolusi.....	58
3.1.1.1. Konfigurasi Eksperimen.....	58

3.1.1.2. Hasil Pengujian.....	58
3.1.1.3. Analisis dan Kesimpulan.....	60
3.1.2. Pengaruh banyak filter per layer konvolusi.....	61
3.1.2.1. Konfigurasi Eksperimen.....	61
3.1.2.2. Hasil Pengujian.....	61
3.1.2.3. Analisis dan Kesimpulan.....	63
3.1.3. Pengaruh ukuran filter per layer konvolusi.....	64
3.1.3.1. Konfigurasi Eksperimen.....	64
3.1.3.2. Hasil Pengujian.....	64
3.1.3.3. Analisis dan Kesimpulan.....	66
3.1.4. Pengaruh jenis pooling layer yang digunakan.....	67
3.1.4.1. Konfigurasi Eksperimen.....	67
3.1.4.2. Hasil Pengujian.....	67
3.1.4.3. Analisis dan Kesimpulan.....	69
3.2. Pengujian RNN.....	70
3.2.1. Pengaruh Jumlah Layer RNN.....	70
3.2.1.1. Konfigurasi Eksperimen.....	70
3.2.1.2. Hasil Pengujian.....	71
3.2.1.3. Analisis dan Kesimpulan.....	74
3.2.2. Pengaruh Jumlah Neuron Pada Layer RNN.....	75
3.2.2.1. Konfigurasi Eksperimen.....	75
3.2.2.2. Hasil Pengujian.....	76
3.2.2.3. Analisis dan Kesimpulan.....	79
3.2.3. Pengaruh Bidirectional Layer dan Unidirectional Layer Pada RNN.....	80
3.2.3.1. Konfigurasi Eksperimen.....	80
3.2.3.2. Hasil Pengujian.....	81
3.2.3.3. Analisis dan Kesimpulan.....	84
3.3. Pengujian LSTM.....	85
3.3.1. Pengaruh Jumlah Layer LSTM.....	85
3.3.1.1. Konfigurasi Eksperimen.....	85
3.3.1.2. Hasil Pengujian.....	86
3.3.1.3. Analisis dan Kesimpulan.....	88
3.3.2. Pengaruh Jumlah Neuron Pada Layer LSTM.....	89
3.3.2.1. Konfigurasi Eksperimen.....	89
3.3.2.2. Hasil Pengujian.....	91
3.3.2.3. Analisis dan Kesimpulan.....	92
3.3.3. Pengaruh Bidirectional Layer dan Unidirectional Layer Pada LSTM.....	93
3.3.3.1. Konfigurasi Eksperimen.....	93
3.3.3.2. Hasil Pengujian.....	95
3.3.3.3. Analisis dan Kesimpulan.....	97

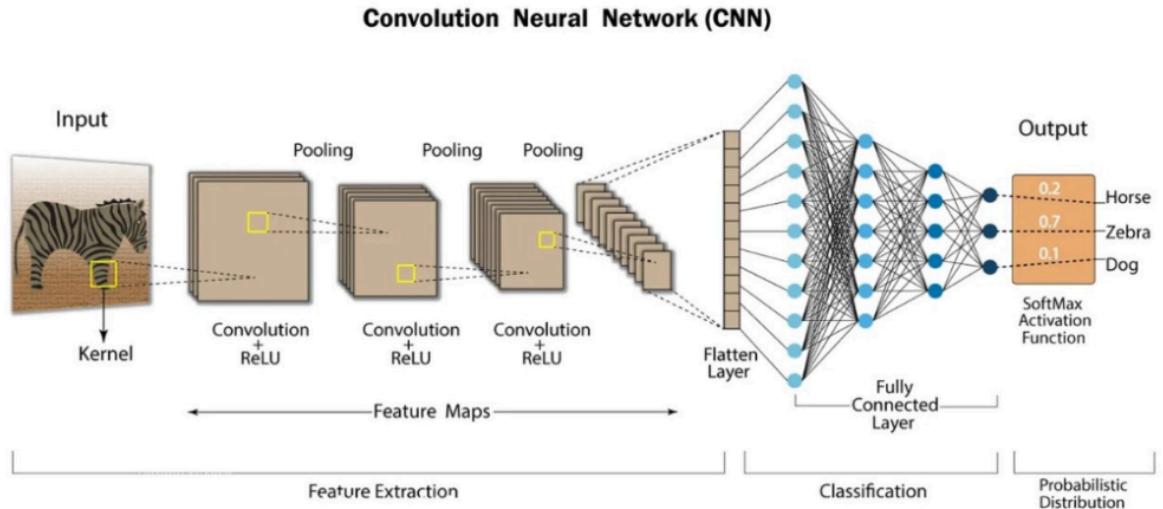
BAB 4 KESIMPULAN DAN SARAN.....	99
4.1. Kesimpulan.....	99
4.2. Saran.....	100
REFERENSI.....	101
LAMPIRAN.....	102

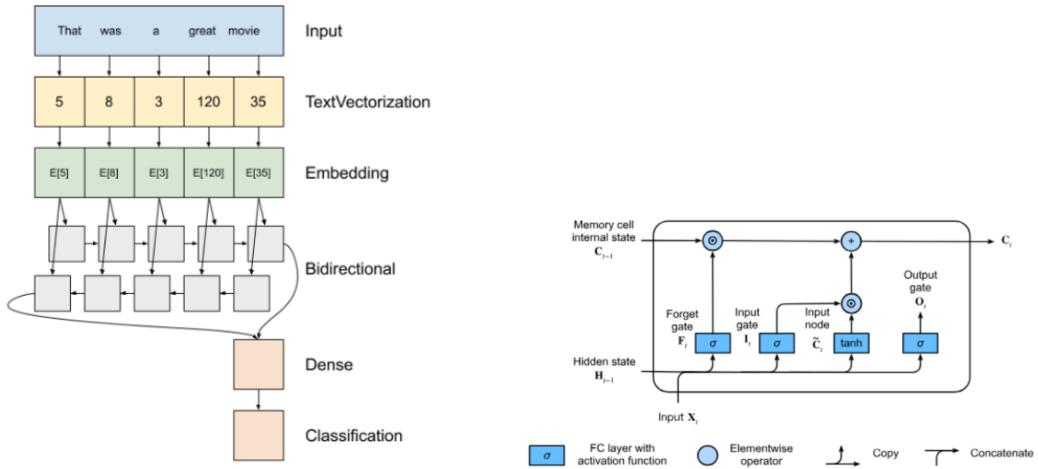
BAB 1

DESKRIPSI PERSOALAN

Dalam era informasi modern, volume data digital yang terus meningkat menuntut metode yang efisien dan akurat dalam proses klasifikasi dan analisis data, baik dalam bentuk gambar maupun teks. Salah satu pendekatan yang terbukti efektif untuk mengatasi tantangan ini adalah melalui pemanfaatan Deep Learning, khususnya arsitektur jaringan syaraf tiruan modern seperti Convolutional Neural Network (CNN) dan Recurrent Neural Network (RNN), termasuk variannya yaitu Long Short-Term Memory (LSTM).

Tugas Besar II pada mata kuliah IF3270 Pembelajaran Mesin dirancang untuk memberikan pemahaman praktis dan mendalam mengenai implementasi CNN dan RNN. Kami ditugaskan untuk tidak hanya menggunakan library high-level seperti Keras dalam pelatihan model, tetapi juga mengembangkan dan menguji modul forward propagation dari nol (from scratch). Hal ini mencakup pemrosesan data, pelatihan model klasifikasi gambar menggunakan CNN dengan dataset CIFAR-10, serta pelatihan model klasifikasi teks berbahasa Indonesia menggunakan RNN dan LSTM dengan dataset NusaX-Sentiment.





Permasalahan dalam tugas ini adalah bagaimana mengimplementasikan modul forward propagation dari CNN, RNN, dan LSTM secara modular dan membandingkan hasilnya dengan Keras menggunakan *macro F1-score*. Selain itu, tugas ini juga menuntut analisis pengaruh variasi *hyperparameter* seperti jumlah layer, banyaknya filter atau unit, arah layer, serta jenis *pooling* terhadap performa model.

Melalui penyelesaian tugas ini, peserta kuliah diharapkan mampu memahami secara konseptual dan praktis cara kerja jaringan syaraf tiruan modern, serta meningkatkan kemampuan analisis terhadap pengaruh struktur arsitektur terhadap akurasi model klasifikasi. Selain itu, mahasiswa juga diharapkan dapat menuliskan dan mendokumentasikan hasil eksperimen secara sistematis dalam bentuk laporan teknis.

BAB 2

IMPLEMENTASI

2.1. CNN

2.1.1. Deskripsi Kelas dan fungsi

2.1.1.1. Kelas *Scratch Model*

```
import numpy as np
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
Flatten, Dense, Dropout, AveragePooling2D,
GlobalAveragePooling2D, GlobalMaxPooling2D

from models.cnn.scratch_convolution import ScratchConv2D
from models.cnn.scratch_dense import ScratchDense
from models.cnn.scratch_flatten import ScratchFlatten
from models.cnn.scratch_pool import ScratchMaxPooling2D,
ScratchAveragePooling2D, ScratchGlobalAveragePooling2D,
ScratchGlobalMaxPooling2D
from models.cnn.scratch_dropout import ScratchDropout
class ScratchModel:
    def __init__(self, keras_model_path):
        keras_model = load_model(keras_model_path)
        self.layers = []
        for layer in keras_model.layers:
            if isinstance(layer, Conv2D):
                w, b = layer.get_weights()
                self.layers.append(ScratchConv2D(w, b,
layer.padding, layer.strides, layer.activation.__name__))
            elif isinstance(layer, MaxPooling2D):
                self.layers.append(ScratchMaxPooling2D(layer.pool_size,
layer.strides))
            elif isinstance(layer, AveragePooling2D):
                self.layers.append(ScratchAveragePooling2D(layer.pool_size,
layer.strides))
            elif isinstance(layer, GlobalAveragePooling2D):
                self.layers.append(ScratchGlobalAveragePooling2D())
            elif isinstance(layer, GlobalMaxPooling2D):
                self.layers.append(ScratchGlobalMaxPooling2D())
```

```

        elif isinstance(layer, Flatten):
            self.layers.append(ScratchFlatten())
        elif isinstance(layer, Dense):
            w, b = layer.get_weights()
            self.layers.append(ScratchDense(w, b,
layer.activation.__name__))
        elif isinstance(layer, Dropout):
            self.layers.append(ScratchDropout())

    def predict(self, x, batch_size=None):
        if batch_size is None:
            return self._predict_batch(x)
        else:
            preds = []
            for start in range(0, len(x), batch_size):
                print(f"Processing batch from index {start}")
                end = start + batch_size
                batch = x[start:end]
                batch_pred = self._predict_batch(batch)
                preds.append(batch_pred)
            return np.concatenate(preds, axis=0)

    def _predict_batch(self, x):
        out = x
        for layer in self.layers:
            out = layer.forward(out)
        return out

```

Kelas ini merepresentasikan sebuah model CNN yang dibangun dari layer-layer hasil konversi model Keras menjadi implementasi manual (scratch). Model ini digunakan untuk melakukan proses inferensi (*forward propagation*) terhadap data input, dan mendukung proses prediksi dalam bentuk batch.

Atribut:

1. Layers

List berisi objek layer hasil konversi dari model Keras. Layer-layer ini merupakan implementasi manual seperti *ScratchConv2D*, *ScratchDense*, *ScratchMaxPooling2D*, dan sebagainya. Setiap layer memiliki method *forward()*.

Metode:

1. *__init__*(self, keras_model_path)

Konstruktor kelas. Memuat model Keras dari file .h5 dan mengonversi setiap layer ke bentuk layer manual (scratch layer). Layer-layer ini disimpan secara berurutan untuk digunakan dalam proses prediksi.

2. predict(self, x, batch_size=None)

Melakukan proses forward propagation terhadap input x. Jika batch_size diberikan, input akan dibagi ke dalam beberapa batch, dan diproses secara bertahap.

3. _predict_batch(self, x)

Fungsi internal untuk melakukan forward propagation pada satu batch penuh tanpa pemrosesan bertahap.

2.1.1.2. Kelas ScratchConv2D

```
import numpy as np
from models.cnn.tensor_activation import
activation_functions_np

class ScratchConv2D:
    def __init__(self, weights, biases, padding='valid',
strides=(1, 1), activation=None):
        self.weights = weights
        self.biases = biases
        self.padding = padding
        self.strides = strides
        self.activation = activation

    def forward(self, x):
        n_samples, height, width, n_channels = x.shape
        filter_height, filter_width, _, n_filters =
self.weights.shape
        if self.padding == 'valid':
            out_height = (height - filter_height) //
self.strides[0] + 1
            out_width = (width - filter_width) //
self.strides[1] + 1
            padded_x = x
        elif self.padding == 'same':
            out_height = int(np.ceil(height / self.strides[0]))
            out_width = int(np.ceil(width / self.strides[1]))
            pad_h = max((out_height - 1) * self.strides[0] +
filter_height - height, 0)
            pad_w = max((out_width - 1) * self.strides[1] +
filter_width - width, 0)
            pad_top, pad_bottom = pad_h // 2, pad_h - (pad_h //
2)
            pad_left, pad_right = pad_w // 2, pad_w - (pad_w //
```

```

2)
        padded_x = np.pad(x, ((0,0), (pad_top, pad_bottom),
(pad_left, pad_right), (0,0)), mode='constant')

        output = np.zeros((n_samples, out_height, out_width,
n_filters))
        for i in range(out_height):
            for j in range(out_width):
                h_start, h_end = i * self.strides[0], i *
self.strides[0] + filter_height
                w_start, w_end = j * self.strides[1], j *
self.strides[1] + filter_width
                patch = padded_x[:, h_start:h_end,
w_start:w_end, :]
                for k in range(n_filters):
                    output[:, i, j, k] = np.sum(
                        patch * self.weights[..., k][None, :, :],
axis=(1, 2, 3)
                    ) + self.biases[k]

        # Generic activation support
        if self.activation in activation_functions_np:
            func = activation_functions_np[self.activation]
            output = func(output)
        return output

```

ScratchConv2D adalah kelas yang merepresentasikan implementasi manual dari layer konvolusi 2D (Conv2D) pada jaringan CNN. Fungsinya adalah untuk melakukan ekstraksi fitur spasial dari input berupa gambar atau feature map, menggunakan filter (kernel) yang digeser di sepanjang dimensi tinggi dan lebar. Layer ini mendukung opsi padding (valid atau same), stride, dan fungsi aktivasi.

Atribut:

1. weights: Tensor bobot konvolusi dengan shape (filter_height, filter_width, input_channels, output_channels) yang berisi kernel filter untuk setiap output channel.
2. biases: Vektor bias dengan shape (output_channels,) yang ditambahkan ke hasil konvolusi masing-masing filter.
3. padding: String yang menentukan jenis padding yang digunakan: 'valid' (tanpa padding) atau 'same' (output size dipertahankan).

4. strides: Tuple (stride_height, stride_width) yang menentukan seberapa jauh filter digeser saat konvolusi.
5. activation: Nama fungsi aktivasi (misalnya 'relu', 'sigmoid') yang diambil dari dictionary activation_functions_np.

Metode:

1. *forward(self, x)* melakukan operasi forward propagation terhadap input x:
Jika padding == 'same', padding nol akan ditambahkan agar ukuran output tetap. Filter digeser ke seluruh area spasial input (mengikuti stride), dan dilakukan operasi konvolusi per patch. Setelah konvolusi, bias ditambahkan dan fungsi aktivasi diterapkan jika ada. Output berupa feature map dengan jumlah channel sesuai jumlah filter (n_filters).

2.1.1.3. Kelas ScratchDense

```
import numpy as np
from models.cnn.tensor_activation import
activation_functions_np

class ScratchDense:
    def __init__(self, weights, biases, activation=None):
        self.weights = weights
        self.biases = biases
        self.activation = activation

    def forward(self, x):
        output = np.dot(x, self.weights) + self.biases
        if self.activation in activation_functions_np:
            output =
activation_functions_np[self.activation](output)
        return output
```

ScratchDense adalah kelas yang merepresentasikan implementasi manual dari layer Dense (fully connected) pada jaringan neural. Fungsinya adalah untuk melakukan transformasi linier terhadap input melalui operasi dot product dengan bobot (weights) dan penambahan bias (biases), lalu menerapkan fungsi aktivasi jika tersedia. Layer ini umum digunakan setelah flatten layer atau pada bagian akhir jaringan untuk menghasilkan prediksi.

Atribut:

1. weights: Matriks bobot dengan shape (input_dim, output_dim) yang digunakan untuk transformasi linier terhadap input.

2. biases: Vektor bias dengan shape (`output_dim`) yang ditambahkan ke hasil dot product.
3. activation: String nama fungsi aktivasi (misal 'relu', 'sigmoid', 'softmax') yang diambil dari dictionary `activation_functions_np`. Jika `None`, tidak ada aktivasi yang diterapkan.

Metode

1. `forward(self, x)` melakukan operasi forward propagation pada input `x`. Caranya adalah dengan menghitung `x @ weights + biases` untuk mendapatkan output linier, kemudian menerapkan fungsi aktivasi jika disediakan. Output berupa tensor dengan dimensi sesuai jumlah neuron pada layer.

2.1.1.4. Kelas ScratchFlatten

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

ScratchFlatten adalah kelas yang merepresentasikan implementasi manual dari layer `Flatten` pada jaringan neural. Fungsinya adalah untuk mengubah tensor multidimensi (biasanya hasil dari layer konvolusi) menjadi vektor satu dimensi per sampel, agar dapat diproses oleh layer fully connected (dense) berikutnya.

Metode:

1. `forward(self, x)` yang melakukan operasi flatten terhadap input `x`. Caranya adalah dengan mengubah setiap sampel dalam batch menjadi vektor satu dimensi menggunakan `reshape`, sambil mempertahankan ukuran batch (`x.shape[0]`) sebagai dimensi pertama.

2.1.1.5. Kelas ScratchMaxPooling2D

```
class ScratchMaxPooling2D:
    def __init__(self, pool_size=(2, 2), strides=None):
        self.pool_size = pool_size
        self.strides = strides if strides else pool_size

    def forward(self, x):
        n, h, w, c = x.shape
        ph, pw = self.pool_size
        sh, sw = self.strides
        out_h = (h - ph) // sh + 1
        out_w = (w - pw) // sw + 1
        out = np.zeros((n, out_h, out_w, c))
```

```

for i in range(out_h):
    for j in range(out_w):
        h_start, h_end = i * sh, i * sh + ph
        w_start, w_end = j * sw, j * sw + pw
        window = x[:, h_start:h_end, w_start:w_end, :]
        out[:, i, j, :] = np.max(window, axis=(1, 2))
return out

```

ScratchMaxPooling2D adalah kelas yang merepresentasikan implementasi manual dari layer *Max Pooling 2D* pada jaringan CNN. Fungsinya adalah untuk mereduksi dimensi spasial (tinggi dan lebar) dari feature map dengan cara membagi area menjadi patch sesuai ukuran pooling, lalu mengambil nilai maksimum dari setiap patch. Ini membantu mempertahankan fitur paling dominan sambil mengurangi dimensi data.

Atribut:

1. *pool_size*: Tuple (*pool_height*, *pool_width*) yang menentukan ukuran jendela pooling. Default adalah (2, 2).
2. *strides*: Tuple (*stride_height*, *stride_width*) yang menentukan pergeseran jendela pooling saat bergerak melintasi input. Jika tidak diberikan, nilainya akan disamakan dengan *pool_size*.

Metode:

1. *forward(self, x)* yang melakukan operasi *max pooling* terhadap input x. Caranya adalah dengan menggeser jendela *pooling* di atas input dan mengambil nilai maksimum dari setiap area patch pada masing-masing channel. Output adalah tensor dengan ukuran spasial yang lebih kecil, namun tetap mempertahankan jumlah channel.

2.1.1.6. Kelas ScratchAveragePooling2D

```

class ScratchAveragePooling2D:
    def __init__(self, pool_size=(2, 2), strides=None):
        self.pool_size = pool_size
        self.strides = strides if strides else pool_size

    def forward(self, x):
        n_samples, height, width, n_channels = x.shape
        pool_height, pool_width = self.pool_size
        stride_height, stride_width = self.strides
        out_height = (height - pool_height) // stride_height +
1
        out_width = (width - pool_width) // stride_width + 1

```

```

        output = np.zeros((n_samples, out_height, out_width,
n_channels))

        for i in range(out_height):
            for j in range(out_width):
                h_start = i * stride_height
                h_end = h_start + pool_height
                w_start = j * stride_width
                w_end = w_start + pool_width
                window = x[:, h_start:h_end, w_start:w_end, :]
                output[:, i, j, :] = np.mean(window, axis=(1,
2))
        return output
    
```

ScratchAveragePooling2D adalah kelas yang merepresentasikan implementasi manual dari layer Average Pooling 2D pada jaringan CNN. Fungsinya adalah untuk mereduksi dimensi spasial (tinggi dan lebar) dari feature map dengan cara membagi area menjadi patch sesuai ukuran pooling, lalu menghitung rata-rata nilai dari tiap patch. Ini membantu dalam mengekstraksi informasi penting sambil mengurangi kompleksitas data.

Atribut:

1. *pool_size*: Tuple (pool_height, pool_width) yang menentukan ukuran jendela pooling. Default adalah (2, 2).
2. *strides*: Tuple (stride_height, stride_width) yang menentukan pergeseran jendela pooling pada tiap langkah. Jika tidak diberikan, akan disamakan dengan *pool_size*.

Metode:

1. *forward(self, x)* yang melakukan operasi average pooling terhadap input x. Caranya adalah dengan menggeser jendela pooling di atas input dan menghitung rata-rata nilai dari setiap area patch pada setiap channel. Output adalah tensor dengan resolusi spasial yang lebih kecil namun tetap mempertahankan jumlah channel.

2.1.1.7. Kelas ScratchGlobalAveragePooling2D

```

class ScratchGlobalAveragePooling2D:
    def forward(self, x):
        return np.mean(x, axis=(1, 2))
    
```

ScratchGlobalAveragePooling2D adalah kelas yang merepresentasikan implementasi manual dari layer Global Average Pooling 2D pada jaringan CNN. Fungsinya adalah untuk mengambil

nilai rata-rata dari seluruh dimensi spasial (tinggi dan lebar) dari setiap feature map (*channel*), sehingga output dari tiap channel hanya berupa satu nilai rata-rata.

Metode:

1. *forward(self, x)* yang melakukan operasi global average pooling terhadap input x. Caranya adalah dengan menghitung nilai rata-rata dari setiap channel pada seluruh area spasial (height dan width).

2.1.1.8. Kelas ScratchGlobalMaxPooling2D

```
class ScratchGlobalMaxPooling2D:  
    def forward(self, x):  
        return np.max(x, axis=(1, 2))
```

ScratchGlobalMaxPooling2D adalah kelas yang merepresentasikan implementasi manual dari layer *Global Max Pooling 2D* pada jaringan CNN. Fungsinya adalah untuk mengambil nilai maksimum dari seluruh dimensi spasial (tinggi dan lebar) dari setiap feature map (*channel*), sehingga output dari tiap channel hanya berupa satu nilai maksimum.

Metode:

1. *forward(self, x)* yang melakukan operasi global max pooling terhadap input x. Caranya adalah mengambil nilai maksimum dari setiap channel pada seluruh area spasial (height dan width).

2.1.1.9. Algoritma Luar Untuk Pelatihan dan Plot

```
def train_model(model, version):  
    os.environ['PYTHONHASHSEED'] = '42'  
    random.seed(42)  
    np.random.seed(42)  
    tf.random.set_seed(42)  
    tf.config.experimental.enable_op_determinism()  
    tf.keras.utils.set_random_seed(42)  
    model.compile(  
        optimizer='adam',  
  
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
        metrics=['accuracy'])  
    history = model.fit(  
        x_train, y_train,  
        epochs=20,
```

```

        validation_data=(x_val, y_val),
        batch_size=64,
        shuffle=False,
        verbose=2
    )
    test_loss, test_acc = model.evaluate(x_test, y_test,
verbose=0)
    print(f"Versi {version} - Akurasi Test: {test_acc:.4f}")
    return history
histories = []
for i, create_fn in enumerate([create_model_v1], start=1):
    os.environ['PYTHONHASHSEED'] = '42'
    random.seed(42)
    np.random.seed(42)
    tf.random.set_seed(42)
    tf.config.experimental.enable_op_determinism()
    tf.keras.utils.set_random_seed(42)
    model = create_fn()
    print(f"\n--- Training Model Versi {i} ---")
    history = train_model(model, version=i)
    histories.append(history)

def plot_histories(histories, train_labels, val_labels,
mode='both'):
    assert len(histories) == len(train_labels) ==
len(val_labels), "Lists must be the same length"
    assert mode in ('both', 'train', 'val'), "mode must be
'both', 'train' or 'val'"
    epochs = range(1, len(histories[0].history['loss']) + 1)
    plt.figure(figsize=(10, 8))
    for h, tl, vl in zip(histories, train_labels, val_labels):
        if mode in ('both', 'train'):
            plt.plot(epochs, h.history['loss'], label=tl)
        if mode in ('both', 'val'):
            plt.plot(epochs, h.history['val_loss'],
linestyle='--', label=vl)
    plt.title('Training vs. Validation Loss' if mode=='both'
else
              'Training Loss' if mode=='train' else
              'Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

```

Metode eksperimen yang digunakan yaitu dengan mendeklarasikan model sekuensial dalam sebuah variabel, contoh *create_model_v1*, yang kemudian dilatih , namun jejak historis , yaitu *loss* disimpan dalam variabel yang berbeda untuk tiap model.

Fungsi plot yang digunakan dapat menerima *array* kumpulan *history* serta *legends* yang beragam sehingga lebih mudah dalam membentuk *stacked line plot*.

2.1.1.10. Konfigurasi fungsi aktivasi

```
import numpy as np

def linear(x): return x
def relu(x): return np.maximum(0, x)
def sigmoid(x): return 1 / (1 + np.exp(-x))
def tanh(x): return np.tanh(x)
def softsign(x): return x / (1 + np.abs(x))
def leaky_relu(x, alpha=0.01): return np.where(x > 0, x, alpha * x)

activation_functions_np = {
    'linear': linear,
    'relu': relu,
    'sigmoid': sigmoid,
    'tanh': tanh,
    'softsign': softsign,
    'leaky_relu': leaky_relu,
}
```

Hanya berupa fungsi tambahan , jika ada kondisi fungsi aktivasi selama konvolusi menggunakan fungsi aktivasi lain selain ReLU.

2.1.2. Penjelasan Forward Propagation

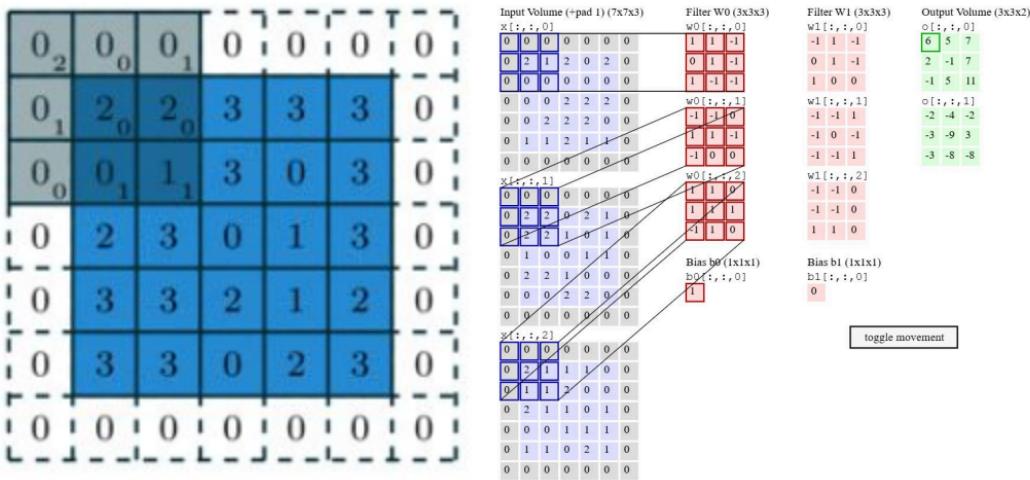
Algoritma forward propagation dalam arsitektur CNN seperti yang diimplementasikan oleh kelas-kelas *ScratchConv2D*, *ScratchMaxPooling2D*, *ScratchAveragePooling2D*, *ScratchGlobalAveragePooling2D*, *ScratchGlobalMaxPooling2D*, *ScratchFlatten*, dan *ScratchDense* bekerja secara berurutan untuk mengekstraksi fitur dari input (biasanya berupa gambar) dan menghasilkan output prediksi.

```

1  class ScratchConv2D:
2      def __init__(self, weights, biases, padding='valid', strides=(1, 1), activation=None):
3          self.weights = weights
4          self.biases = biases
5          self.padding = padding
6          self.strides = strides
7          self.activation = activation
8
9      def forward(self, x):
10         n_samples, height, width, n_channels = x.shape
11         filter_height, filter_width, _, n_filters = self.weights.shape
12         if self.padding == 'valid':
13             out_height = (height - filter_height) // self.strides[0] + 1
14             out_width = (width - filter_width) // self.strides[1] + 1
15             padded_x = x
16         elif self.padding == 'same':
17             out_height = int(np.ceil(height / self.strides[0]))
18             out_width = int(np.ceil(width / self.strides[1]))
19             pad_h = max((out_height - 1) * self.strides[0] + filter_height - height, 0)
20             pad_w = max((out_width - 1) * self.strides[1] + filter_width - width, 0)
21             pad_top, pad_bottom = pad_h // 2, pad_h - (pad_h // 2)
22             pad_left, pad_right = pad_w // 2, pad_w - (pad_w // 2)
23             padded_x = np.pad(x, ((0,0), (pad_top, pad_bottom), (pad_left, pad_right), (0,0)), mode='constant')
24
25         output = np.zeros((n_samples, out_height, out_width, n_filters))
26         for i in range(out_height):
27             for j in range(out_width):
28                 h_start, h_end = i * self.strides[0], i * self.strides[0] + filter_height
29                 w_start, w_end = j * self.strides[1], j * self.strides[1] + filter_width
30                 patch = padded_x[:, h_start:h_end, w_start:w_end, :]
31                 for k in range(n_filters):
32                     output[:, i, j, k] = np.sum(
33                         patch * self.weights[..., k][None, :, :, :], axis=(1, 2, 3)
34                     ) + self.biases[k]
35
36         if self.activation in activation_functions_np:
37             func = activation_functions_np[self.activation]
38             output = func(output)
39
40         return output

```

Pertama-tama, kelas *ScratchConv2D* berfungsi sebagai layer konvolusi yang menerapkan beberapa filter (kernel) pada input 4D berukuran (batch_size, height, width, channels). Filter ini digeser (sliding) di sepanjang dimensi spasial input menggunakan *strides*. Untuk setiap posisi, dilakukan operasi dot product antara input dan bobot filter, kemudian ditambahkan bias dan fungsi aktivasi (umumnya ReLU). Operasi ini menghasilkan *feature map*, yaitu representasi fitur lokal dari input. Jika padding diset ke 'same', input akan dipadatkan terlebih dahulu dengan padding nol agar dimensi output tetap.



Kedua gambar diatas merupakan gambaran jika ada padding dan jika dilihat dari kode, jika *same*, maka akan ditambahkan nilai konstan 0 pada input X.

```

1
2 class ScratchMaxPooling2D:
3     def __init__(self, pool_size=(2, 2), strides=None):
4         self.pool_size = pool_size
5         self.strides = strides if strides else pool_size
6
7     def forward(self, x):
8         n, h, w, c = x.shape
9         ph, pw = self.pool_size
10        sh, sw = self.strides
11        out_h = (h - ph) // sh + 1
12        out_w = (w - pw) // sw + 1
13        out = np.zeros((n, out_h, out_w, c))
14        for i in range(out_h):
15            for j in range(out_w):
16                h_start, h_end = i * sh, i * sh + ph
17                w_start, w_end = j * sw, j * sw + pw
18                window = x[:, h_start:h_end, w_start:w_end, :]
19                out[:, i, j, :] = np.max(window, axis=(1, 2))
20
21    return out

```

Setelah ekstraksi fitur oleh layer konvolusi, biasanya dilakukan reduksi dimensi spasial menggunakan *layer pooling*. Pada *ScratchMaxPooling2D*, input dibagi ke dalam patch-berukuran tetap (*pool_size*) dan mengambil nilai maksimum dari setiap patch, menekankan fitur paling dominan. Sementara *ScratchAveragePooling2D* mengambil rata-rata nilai dalam patch, menghasilkan representasi yang lebih merata dan stabil. Untuk mengurangi seluruh dimensi spasial menjadi satu vektor per channel, digunakan *ScratchGlobalMaxPooling2D* atau *ScratchGlobalAveragePooling2D*, yang mengambil nilai maksimum atau rata-rata dari seluruh area spasial masing-masing channel. Outputnya berupa tensor berdimensi (*batch_size*, *channels*). Untuk menghindari perulangan, hanya akan diperlihatkan algoritma *ScratchMaxPooling2D*.

Saat *forward propagation*, program menghitung dimensi output berdasarkan ukuran input, ukuran *pooling*, dan *stride*. Kemudian, algoritma menggunakan dua loop (i, j) untuk menggeser jendela pooling secara vertikal dan horizontal sesuai dengan *stride*, mengambil area patch dari input, lalu menghitung nilai maksimum pada setiap patch di sepanjang dimensi height dan width (axis 1 dan 2). Perkalian dalam loop umumnya hanya penyesuaian pergerakan *stride*.

Setelah itu, *ScratchFlatten* digunakan untuk mengubah output multidimensi dari layer sebelumnya menjadi vektor 1 dimensi per sampel agar bisa diproses oleh layer *fully connected*.

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

Proses ini penting karena layer dense (*ScratchDense*) hanya menerima input 2D (*batch_size*, *features*). Pada *ScratchDense*, input dikalikan dengan bobot, ditambahkan bias, dan kemudian dilalui fungsi aktivasi jika disediakan. *Dense Layer* umumnya merupakan *FC Layer* sehingga cara kerjanya sama dengan *forward propagation* pada umumnya, yakni aktivasi dari perkalian antar input dengan bobot ditambah bias.

```
● ● ●
```

```
1  class ScratchDense:
2      def __init__(self, weights, biases, activation=None):
3          self.weights = weights
4          self.biases = biases
5          self.activation = activation
6
7      def forward(self, x):
8          output = np.dot(x, self.weights) + self.biases
9          if self.activation in activation_functions_np:
10              output = activation_functions_np[self.activation](output)
11
12      return output
```

Secara keseluruhan, proses forward propagation CNN bekerja secara bertahap dari ekstraksi fitur lokal melalui konvolusi, peringkasan spasial dengan *pooling*, normalisasi dimensi dengan *flattening*, hingga menghasilkan output akhir melalui *dense layer*.

2.2. RNN

2.2.1. Deskripsi Kelas

2.2.1.1. Kelas *SimpleRNNLayer*

```
import torch as tc
import numpy as np
from models.nn.activations import activation_functions

class SimpleRNNLayer:
    def __init__(self, Wx: np.ndarray, Wh: np.ndarray, b: np.ndarray, activation='tanh'):
        self.Wx = tc.tensor(Wx, dtype=tc.float32)
        self.Wh = tc.tensor(Wh, dtype=tc.float32)
        self.b = tc.tensor(b, dtype=tc.float32)
        self.activation, _, _ =
activation_functions[activation]

    def forward(self, x: tc.Tensor, return_sequences=False):
        batch_size, seq_len, _ = x.shape
        h = tc.zeros(batch_size, self.b.shape[0])
        outputs = []
```

```

        for t in range(seq_len):
            h = self.activation(x[:, t, :] @ self.Wx + h @
self.Wh + self.b)
            if return_sequences:
                outputs.append(h.unsqueeze(1))

        if return_sequences:
            return tc.cat(outputs, dim=1)
        else:
            return h
    
```

Kelas ini merepresentasikan sebuah layer Recurrent Neural Network (RNN) sederhana yang digunakan untuk memproses data sekuensial. Layer ini dirancang agar dapat melakukan forward propagation baik dengan mengembalikan seluruh hidden state per timestep maupun hasil akhir saja.

Atribut:

1. Wx

Tensor bobot input-to-hidden dengan bentuk (input_dim, hidden_dim). Digunakan untuk mengalikan input pada setiap timestep.

2. Wh

Tensor bobot hidden-to-hidden dengan bentuk (hidden_dim, hidden_dim). Digunakan untuk mempertahankan informasi dari hidden state sebelumnya.

3. b

Tensor bias dengan bentuk (hidden_dim,). Ditambahkan pada hasil linear sebelum fungsi aktivasi diterapkan.

4. activation

Fungsi aktivasi non-linear (misalnya tanh, relu) yang digunakan untuk menghitung nilai akhir dari hidden state. Fungsi diambil dari dictionary activation_functions.

Method:

1. __init__(self, Wx, Wh, b, activation='tanh')

Konstruktor kelas. Melakukan inisialisasi bobot input, bobot hidden, bias, dan fungsi aktivasi sesuai parameter.

2. forward(self, x, return_sequences=False)

Melakukan proses forward propagation terhadap input x dengan bentuk (batch_size, sequence_length, input_dim). Jika return_sequences=True, akan mengembalikan seluruh urutan hidden state; jika tidak, hanya mengembalikan hidden state pada timestep terakhir.

2.2.1.2. Kelas *EmbeddingLayer*

```
import torch as tc
import numpy as np

class EmbeddingLayer:
    def __init__(self, weights: np.ndarray):
        self.weights = tc.tensor(weights, dtype=tc.float32)

    def forward(self, x: tc.Tensor):
        return self.weights[x]
```

Kelas ini merepresentasikan implementasi sederhana dari embedding layer yang berfungsi untuk memetakan token ID menjadi vektor berdimensi tetap. Layer ini secara langsung menggunakan bobot hasil pelatihan dari model Keras (dalam bentuk array NumPy) dan digunakan sebagai bagian dari proses forward propagation manual.

Atribut:

1. weights

Tensor dua dimensi dengan bentuk (vocab_size, embedding_dim) yang merepresentasikan vektor embedding untuk setiap token dalam kosakata. Bobot ini diinisialisasi dari array hasil pelatihan dan disimpan dalam bentuk tensor PyTorch.

Method:

1. __init__(weights)

Konstruktor yang menerima bobot embedding hasil pelatihan dan mengkonversinya menjadi tensor PyTorch.

2. forward(x)

Menerima input tensor x berisi token ID (dengan bentuk (batch_size, sequence_length)) dan mengembalikan tensor hasil embedding dengan bentuk (batch_size, sequence_length, embedding_dim). Operasi ini dilakukan dengan cara mengindeks langsung bobot embedding menggunakan nilai ID dalam x.

2.2.1.3. Kelas *BidirectionalRNNLayer*

```
import torch as tc
from .simple_rnn_layer import SimpleRNNLayer

class BidirectionalRNNLayer:
    def __init__(self, forward_weights, backward_weights,
                 activation='tanh'):
        self.rnn_f = SimpleRNNLayer(*forward_weights,
                                    activation=activation)
        self.rnn_b = SimpleRNNLayer(*backward_weights,
                                    activation=activation)

    def forward(self, x: tc.Tensor, return_sequences=False):
        h_f_out = self.rnn_f.forward(x,
                                     return_sequences=return_sequences)

        h_b_out_for_flipped_input =
        self.rnn_b.forward(tc.flip(x, dims=[1]),
                           return_sequences=return_sequences)

        if return_sequences:
            h_b_aligned = tc.flip(h_b_out_for_flipped_input,
                                  dims=[1])
            h_f_final = h_f_out
            h_b_final = h_b_aligned
        else:
            h_f_final = h_f_out.unsqueeze(1)
            h_b_final = h_b_out_for_flipped_input.unsqueeze(1)

        h = tc.cat([h_f_final, h_b_final], dim=-1)

        if not return_sequences:
            h = h.squeeze(1)

    return h
```

Kelas ini merepresentasikan sebuah layer Recurrent Neural Network (RNN) dua arah (bidirectional) yang memproses input sekuensial dari dua arah: maju (forward) dan mundur (backward). Tujuan penggunaan arsitektur bidirectional adalah untuk memanfaatkan konteks dari masa lalu dan masa depan secara simultan, sehingga model memiliki pemahaman yang lebih lengkap terhadap keseluruhan sekuens input.

Atribut:

1. rnn_f

Objek SimpleRNNLayer yang memproses input dari arah maju (kiri ke kanan). Menggunakan bobot forward (forward_weights) untuk perhitungan hidden state.

2. rnn_b

Objek SimpleRNNLayer yang memproses input dari arah mundur (kanan ke kiri). Menggunakan bobot backward (backward_weights) dan menerima input sekuens yang telah dibalik.

Method:

1. __init__(self, forward_weights, backward_weights, activation='tanh')

Konstruktor kelas. Menginisialisasi dua buah SimpleRNNLayer, masing-masing untuk arah maju dan mundur, dengan bobot serta fungsi aktivasi yang sama.

2. forward(self, x, return_sequences=False)

Melakukan forward propagation secara bidirectional terhadap input tensor x yang berdimensi (batch_size, sequence_length, input_dim). Layer rnn_f memproses input secara normal, sedangkan rnn_b memproses input yang dibalik. Hasil dari kedua arah kemudian dikombinasikan melalui konkatenasi pada dimensi fitur. Jika return_sequences=True, hasil konkatenasi dari semua timestep dikembalikan; jika False, hanya hasil akhir yang dikembalikan.

2.2.1.4. Kelas *SimpleRNNKeras*

```
import numpy as np
from sklearn.metrics import f1_score
from tensorflow.keras.layers import Embedding, Dropout, Dense,
SimpleRNN, Bidirectional
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt

class SimpleRNNKeras:
    def __init__(
        self,
        max_vocab=10000,
        max_len=100,
        embedding_dim=128,
        rnn_units=[64],
        rnn_activations=None,
        dense_units=[3],
```

```

        dense_activations=['softmax'],
        bidirectional=True,
        dropout=0.5,
        learning_rate=1e-3
    ):

        self.max_vocab = max_vocab
        self.max_len = max_len
        self.embedding_dim = embedding_dim
        self.rnn_units = rnn_units

        if rnn_activations is None:
            self.rnn_activations = ['tanh'] * len(rnn_units)
        elif isinstance(rnn_activations, str):
            self.rnn_activations = [rnn_activations] *
len(rnn_units)
        elif isinstance(rnn_activations, list):
            if len(rnn_activations) != len(rnn_units):
                raise ValueError("Length of rnn_activations
must match length of rnn_units.")
            self.rnn_activations = rnn_activations
        else:
            raise ValueError("rnn_activations must be a string,
a list of strings, or None.")

        self.dense_units = dense_units
        self.dense_activations = dense_activations
        self.bidirectional = bidirectional
        self.dropout = dropout
        self.learning_rate = learning_rate
        self.num_classes = self.dense_units[-1] if
self.dense_units else None

        self.model = None

    def set_vectorized_data(self, X_train, y_train, X_valid,
y_valid, X_test, y_test):
        self.X_train = X_train
        self.y_train = y_train
        self.X_valid = X_valid
        self.y_valid = y_valid
        self.X_test = X_test
        self.y_test = y_test

    def build_model(self):
        layers = []

```

```

        layers.append(Embedding(input_dim=self.max_vocab,
output_dim=self.embedding_dim))

        for i, units in enumerate(self.rnn_units):
            current_rnn_activation = self.rnn_activations[i]
            print(f"Adding RNN layer {i+1} with {units} units
and activation {current_rnn_activation}")
            rnn_layer_instance = SimpleRNN(units,
activation=current_rnn_activation, return_sequences=(i <
len(self.rnn_units) - 1))
            if self.bidirectional:
                rnn_layer_instance =
Bidirectional(rnn_layer_instance)
            layers.append(rnn_layer_instance)

        layers.append(Dropout(self.dropout))

        for units, activation in zip(self.dense_units,
self.dense_activations):
            layers.append(Dense(units, activation=activation))

    self.model = Sequential(layers)
    metrics_to_use = ['accuracy']
    self.model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=Adam(learning_rate=self.learning_rate),
        metrics=metrics_to_use
    )

    def train(self, epochs=10, batch_size=64, shuffle=True,
verbose = 0):
        self.history = self.model.fit(
            self.X_train,
            self.y_train,
            validation_data=(self.X_valid, self.y_valid),
            epochs=epochs,
            batch_size=batch_size,
            shuffle=shuffle,
            verbose=verbose
        )

        return self.history

    def evaluate(self):
        y_pred = np.argmax(self.model.predict(self.X_test),
axis=1)

```

```

        return (y_pred, f1_score(self.y_test, y_pred,
average='macro'))

def plot_loss(self):
    if not hasattr(self, 'history'):
        print("Model belum dilatih.")
        return

    train_loss = self.history.history['loss']
    val_loss = self.history.history['val_loss']

    plt.figure(figsize=(6, 4))
    plt.plot(train_loss, label='Train Loss', color='blue')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Train Loss per Epoch')
    plt.grid(True)
    plt.show()

    plt.figure(figsize=(6, 4))
    plt.plot(val_loss, label='Validation Loss',
color='orange')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Validation Loss per Epoch')
    plt.grid(True)
    plt.show()

    plt.figure(figsize=(6, 4))
    plt.plot(train_loss, label='Train Loss', color='blue')
    plt.plot(val_loss, label='Validation Loss',
color='orange')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Train vs Validation Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

def save(self, path="model_simple_rnn.h5"):
    self.model.save(path)

def save_full_npy(self, path="model_simple_rnn.npy"):
    data = {
        "weights": self.model.get_weights(),
        "config": {

```

```

        "rnn_units": self.rnn_units,
        "rnn_activations": self.rnn_activations,
        "dense_units": self.dense_units,
        "dense_activations": self.dense_activations,
        "embedding_dim": self.embedding_dim,
        "max_vocab": self.max_vocab,
        "max_len": self.max_len,
        "bidirectional": self.bidirectional,
        "dropout": self.dropout,
        "learning_rate": self.learning_rate
    }
}
np.save(path, data, allow_pickle=True)
print(f"Saved full model to {path}")

```

Kelas ini merepresentasikan implementasi model Simple Recurrent Neural Network (RNN) berbasis Keras untuk tugas klasifikasi teks. Model ini dibangun secara modular dengan dukungan layer embedding, beberapa layer RNN (baik unidirectional maupun bidirectional), dropout, serta dense layer untuk klasifikasi. Selain membangun dan melatih model, kelas ini juga mendukung evaluasi, visualisasi loss, dan penyimpanan bobot model.

Atribut:

1. max_vocab
Ukuran maksimum kosakata yang digunakan dalam embedding.
2. max_len
Panjang maksimum input sekuen yang diterima model.
3. embedding_dim
Dimensi vektor hasil embedding untuk setiap token.
4. rnn_units
Daftar jumlah unit pada setiap layer RNN. Misalnya [64, 64] berarti dua layer RNN dengan masing-masing 64 unit neuron di dalamnya.
5. rnn_activations
Daftar fungsi aktivasi untuk masing-masing layer RNN. Jika tidak diberikan, nilai default adalah tanh.
6. dense_units
Daftar jumlah neuron pada dense layer. Umumnya berakhir dengan jumlah kelas output.

7. `dense_activations`

Daftar fungsi aktivasi untuk masing-masing dense layer. Misalnya ['relu', 'softmax'].

8. `bidirectional`

Boolean yang menentukan apakah layer RNN menggunakan Bidirectional atau tidak.

9. `dropout`

Rasio dropout yang digunakan untuk regularisasi setelah layer RNN.

10. `learning_rate`

Laju pembelajaran untuk optimizer Adam.

11. `num_classes`

Jumlah kelas target (diambil dari elemen terakhir `dense_units`).

12. `model`

Objek Keras Sequential yang berisi arsitektur lengkap model.

13. `History`

Objek history hasil training model, berisi metrik dan loss per epoch.

Method:

1. `__init__(...)`

Konstruktor untuk inisialisasi parameter model seperti embedding, arsitektur RNN/Dense, dropout, dan lainnya.

2. `set_vectorized_data(X_train, y_train, X_valid, y_valid, X_test, y_test)`

Menyimpan data latih, validasi, dan uji yang sudah dalam bentuk vektorisasi (numerik) ke dalam atribut kelas.

3. `build_model()`

Menyusun arsitektur Keras Sequential berdasarkan konfigurasi RNN dan Dense. Kompilasi model dilakukan dengan `sparse_categorical_crossentropy` dan optimizer Adam.

4. `train(epochs=10, batch_size=64, shuffle=True, verbose=0)`

Melatih model menggunakan data yang sudah disiapkan. Mengembalikan objek history.

5. `evaluate()`

Menghitung prediksi pada data uji dan mengembalikan tuple berupa y_pred serta nilai macro F1-score.

6. plot_loss()

Menampilkan grafik loss untuk data latih dan validasi per epoch. Terdiri dari 3 grafik: train, valid, dan keduanya digabung.

7. save(path="model_simple_rnn.h5")

Menyimpan model Keras dalam format .h5.

8. save_full_npy(path="model_simple_rnn.npy")

Menyimpan bobot model dan konfigurasi model dalam satu file NumPy (.npy) agar bisa digunakan kembali untuk implementasi dari nol (from scratch).

2.2.1.5. Kelas *SimpleRNNManual*

```
from .simple_rnn_layer import SimpleRNNLayer
from .bidirectional_rnn_layer import BidirectionalRNNLayer
from .embedding_layer import EmbeddingLayer
import tensorflow as tf
from models.nn.ffnn import FFNN
import torch as tc
import numpy as np
import matplotlib.pyplot as plt

class SimpleRNNManual:
    def __init__(self):
        self.embedding = None
        self.rnn_layers = []
        self.dense = None

    def forward(self, x_token_ids: tc.Tensor):
        x = self.embedding.forward(x_token_ids)

        for i, rnn_layer_instance in enumerate(self.rnn_layers):
            return_sequences = (i < len(self.rnn_layers) - 1)
            x = rnn_layer_instance.forward(x,
                                           return_sequences=return_sequences)

        if self.dense is not None:
            return self.dense.forward(x)
        return x
```

```

def predict(self, x_token_ids: tc.Tensor):
    logits = self.forward(x_token_ids)
    return tc.argmax(logits, dim=1)

def plot_loss(self, history):
    if not history:
        print("History kosong.")
        return

    train_loss = history['train_loss']
    val_loss = history.get('val_loss', [])

    plt.figure(figsize=(6, 4))
    plt.plot(train_loss, label='Train Loss', color='blue')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Train Loss per Epoch')
    plt.grid(True)
    plt.show()

    if val_loss:
        plt.figure(figsize=(6, 4))
        plt.plot(val_loss, label='Validation Loss',
color='orange')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title('Validation Loss per Epoch')
        plt.grid(True)
        plt.show()

    plt.figure(figsize=(6, 4))
    plt.plot(train_loss, label='Train Loss', color='blue')
    if val_loss:
        plt.plot(val_loss, label='Validation Loss',
color='orange')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title('Train vs Validation Loss')
        plt.legend()
        plt.grid(True)
        plt.show()

def evaluate(self, x_token_ids: tc.Tensor, y_true:
tc.Tensor):

```

```

from sklearn.metrics import f1_score
y_pred = self.predict(x_token_ids).cpu().numpy()
y_true = y_true.cpu().numpy()
return f1_score(y_true, y_pred, average='macro')

def load_full_npy(self, path: str):
    data = np.load(path, allow_pickle=True).item()
    config = data['config']
    weights = data['weights']
    self.load_from_config_and_weights(config, weights)

    def load_from_config_and_weights(self, config: dict,
weights: list):
        idx = 0
        self.embedding = EmbeddingLayer(weights[idx]); idx += 1

        self.rnn_layers = []

        rnn_activations_from_config =
config.get("rnn_activations")

        if rnn_activations_from_config is None:
            global_rnn_activation =
config.get("rnn_activation", "tanh")
            rnn_activations_from_config =
[global_rnn_activation] * len(config["rnn_units"])
            elif len(rnn_activations_from_config) !=
len(config["rnn_units"]):
                print(f"Warning: Length mismatch between
rnn_activations ({len(rnn_activations_from_config)}) and
rnn_units ({len(config['rnn_units'])}). Using first activation
or tanh.")
                if rnn_activations_from_config:
                    rnn_activations_from_config =
[rnn_activations_from_config[0]] * len(config["rnn_units"])
                else:
                    rnn_activations_from_config = ['tanh'] *
len(config["rnn_units"])

        for i in range(len(config["rnn_units"])):
            current_rnn_activation =
rnn_activations_from_config[i]
            if config["bidirectional"]:
                Wx_f, Wh_f, b_f = weights[idx:idx+3]; idx += 3
                Wx_b, Wh_b, b_b = weights[idx:idx+3]; idx += 3

```

```

        self.rnn_layers.append(
            BidirectionalRNNLayer((Wx_f, Wh_f, b_f),
(Wx_b, Wh_b, b_b), activation=current_rnn_activation)
        )
    else:
        Wx, Wh, b = weights[idx:idx+3]; idx += 3
        self.rnn_layers.append(SimpleRNNLayer(Wx, Wh,
b, activation=current_rnn_activation))

    dense_weights, dense_biases = [], []
    while idx < len(weights):
        if idx + 1 < len(weights):
            W = weights[idx]; idx += 1
            b = weights[idx]; idx += 1
            dense_weights.append(W)
            dense_biases.append(b.reshape(1, -1))
        else:
            break

    if not dense_weights:
        self.dense = None
        return

    layer_sizes = [dense_weights[0].shape[0]] + [w.shape[1]]
for w in dense_weights:
    dense_activations = config.get("dense_activations")

    if dense_activations is None:
        if len(layer_sizes) - 1 == 1:
            dense_activations = ['softmax']
        elif len(layer_sizes) - 1 > 1:
            dense_activations = ['relu'] *
(len(layer_sizes) - 2) + ['softmax']
        else:
            dense_activations = []

    self.dense = FFNN(
        layer_sizes,
        activations_list=dense_activations,
        loss_function='cce',
        weight_inits=['manual'] * (len(layer_sizes) - 1) if
layer_sizes and len(layer_sizes) > 1 else []
    )
    for i, layer in enumerate(self.dense.layers):
        layer.weights = tc.tensor(dense_weights[i],
dtype=tc.float32)

```

```

        layer.biases = tc.tensor(dense_biases[i],
dtype=tc.float32)

    def load_from_keras_h5_auto_config(self, keras_h5_path:
str):
        model = tf.keras.models.load_model(keras_h5_path)
        weights = model.get_weights()
        keras_layer_configs = model.get_config()['layers']

        idx = 0
        self.embedding = EmbeddingLayer(weights[idx]); idx += 1

        self.rnn_layers = []

        keras_rnn_layer_idx_in_weights = 0

        for layer_conf in keras_layer_configs:
            if layer_conf['class_name'] == 'SimpleRNN':
                rnn_activation =
layer_conf['config']['activation']
                Wx, Wh, b = weights[idx:idx+3]; idx += 3
                self.rnn_layers.append(SimpleRNNLayer(Wx, Wh,
b, activation=rnn_activation))
                keras_rnn_layer_idx_in_weights +=1

            elif layer_conf['class_name'] == 'Bidirectional':
                wrapped_rnn_conf =
layer_conf['config']['layer']['config']
                rnn_activation = wrapped_rnn_conf['activation']

                Wx_f, Wh_f, b_f = weights[idx:idx+3]; idx += 3
                Wx_b, Wh_b, b_b = weights[idx:idx+3]; idx += 3
                self.rnn_layers.append(
                    BidirectionalRNNLayer((Wx_f, Wh_f, b_f),
(Wx_b, Wh_b, b_b), activation=rnn_activation)
                )
                keras_rnn_layer_idx_in_weights +=1

        dense_weights, dense_biases = [], []
        dense_activations = []

        for layer_conf in keras_layer_configs:
            if layer_conf['class_name'] == 'Dense':
                if idx + 1 < len(weights):
                    W = weights[idx]; idx += 1
                    b = weights[idx]; idx += 1

```

```

        dense_weights.append(W)
        dense_biases.append(b.reshape(1, -1))

dense_activations.append(layer_conf['config']['activation'])
else:
    print(f"Warning: Not enough weights for all Dense layers in H5 config. Processed {len(dense_weights)} Dense layers.")
    break

if not dense_weights:
    self.dense = None
    return

layer_sizes = [dense_weights[0].shape[0]] + [w.shape[1] for w in dense_weights]
self.dense = FFNN(layer_sizes,
activations_list=dense_activations, loss_function='cce',
weight_inits=['manual'] * len(dense_weights) if dense_weights else [])
for i, layer_obj in enumerate(self.dense.layers):
    layer_obj.weights = tc.tensor(dense_weights[i],
dtype=tc.float32)
    layer_obj.biases = tc.tensor(dense_biases[i],
dtype=tc.float32)

```

Kelas ini merepresentasikan implementasi manual dari arsitektur Simple RNN (baik unidirectional maupun bidirectional) yang dapat memuat bobot dari model Keras dan digunakan untuk evaluasi forward propagation tanpa menggunakan library deep learning high-level seperti TensorFlow atau Keras. Kelas ini dirancang untuk menguji kesesuaian antara hasil model dari Keras dengan implementasi dari nol menggunakan PyTorch.

Atribut:

1. embedding

Objek dari kelas EmbeddingLayer yang digunakan untuk mengubah token ID menjadi representasi vektor berdimensi tetap.

2. rnn_layers

List dari layer RNN yang digunakan dalam model, bisa terdiri dari SimpleRNNLayer atau BidirectionalRNNLayer.

3. dense

Objek dari kelas FFNN (feedforward neural network) yang digunakan sebagai classifier setelah layer RNN. Class FFNN disini adalah class FFNN yang sudah dibuat di tubes 1 sebelumnya.

Method:

1. `__init__()`

Konstruktor yang menginisialisasi atribut utama model: embedding, rnn_layers, dan dense.

2. `forward(x_token_ids)`

Melakukan forward propagation dari input token ID. Data akan melalui embedding, berlapis-lapis RNN, lalu masuk ke dense layer.

3. `predict(x_token_ids)`

Mengembalikan prediksi kelas akhir dari hasil forward propagation dengan mengambil argmax dari logits output.

4. `plot_loss(history)`

Menampilkan grafik training loss dan optional validation loss berdasarkan dictionary history yang berisi daftar loss per epoch.

5. `evaluate(x_token_ids, y_true)`

Menghitung prediksi dan mengembalikan macro F1-score berdasarkan label ground truth `y_true`.

6. `load_full_npy(path)`

Memuat konfigurasi model dan bobot dari file .npy hasil penyimpanan Keras yang telah dikonversi sebelumnya.

7. `load_from_config_and_weights(config, weights)`

Menginisialisasi model secara manual berdasarkan konfigurasi dan bobot. Termasuk setup embedding, RNN, dan dense layer. Method ini akan digunakan di fungsi `load_full_npy`.

8. `load_from_keras_h5_auto_config(keras_h5_path)`

Mengambil arsitektur dan bobot dari file model .h5 Keras dan mengkonversinya menjadi bentuk manual. Digunakan untuk validasi kesesuaian forward propagation antara Keras dan implementasi manual.

2.2.1.6. Kelas *FFNN*

Kelas ini menggunakan implementasi dari tubes 1 sebelumnya jadi tidak akan dijelaskan pada laporan ini. Kelas-kelas lain yang berhubungan dengan kelas ini seperti kelas layer, kelas activation, kelas loss_function dan kelas initializer juga digunakan kembali pada tubes ini namun penjelasannya tidak akan dijelaskan disini karena penjelasannya sudah ada di tubes sebelumnya.

2.2.2. Penjelasan Forward Propagation

Pada tugas ini, kami mengimplementasikan proses forward propagation secara manual menggunakan PyTorch untuk membangun model klasifikasi teks berbasis Recurrent Neural Network (RNN). Proses forward propagation dilakukan melalui beberapa tahapan utama, mulai dari vectorization, embedding token, pemrosesan sekuens dengan RNN (baik unidirectional maupun bidirectional), hingga klasifikasi melalui fully connected layer.

A. Proses Awal: Vectorization

Sebelum teks dapat diproses oleh jaringan, perlu dilakukan vectorization terlebih dahulu. Proses ini mengubah teks mentah menjadi deretan angka melalui tokenisasi dan pemetaan indeks dari kosakata yang telah dipelajari. Dalam implementasi kami, TextVectorization dari Keras digunakan dan dibungkus ke dalam kelas TextPreprocessor. Output dari vectorization ini adalah tensor integer berdimensi tetap [batch_size, max_len], yang kemudian digunakan sebagai input ke tahap berikutnya yaitu embedding.

B. Embedding Layer

Langkah pertama dalam forward propagation adalah mentransformasikan token ID hasil vectorization menjadi vektor berdimensi tetap melalui EmbeddingLayer. Layer ini bekerja dengan mengambil representasi vektor dari matriks embedding:



Vektor embedding ini kemudian menjadi representasi angka dari kata-kata dalam kalimat, dan digunakan sebagai input awal yang akan diproses secara berurutan oleh jaringan RNN.

C. Simple RNN Layer

Vektor hasil embedding kemudian diteruskan ke SimpleRNNLayer, yang merupakan komponen utama dalam arsitektur RNN manual ini. Layer ini bekerja dengan memproses input satu per satu untuk setiap waktu (timestep). Pada setiap waktu t , layer menghitung hidden state saat ini (h_t) berdasarkan input di waktu itu (x_t) dan hidden state sebelumnya (h_{t-1}). Rumus yang digunakan adalah:

$$h_t = \text{activation}(x_t \times W_x + h_{t-1} \times W_h + b)$$

Rumus tersebut diterapkan secara langsung menggunakan perulangan for untuk setiap langkah waktu dalam input. Dengan cara ini, kita bisa melihat bagaimana informasi mengalir dari awal hingga akhir kalimat secara bertahap. Proses ini menyerupai cara manusia membaca kalimat, yaitu satu kata demi satu kata dari kiri ke kanan. Berikut potongan kode dari method forward() di kelas SimpleRNNLayer:

```
1 def forward(self, x: tc.Tensor, return_sequences=False):
2     batch_size, seq_len, _ = x.shape
3     h = tc.zeros(batch_size, self.b.shape[0])
4     outputs = []
5
6     for t in range(seq_len):
7         h = self.activation(x[:, t, :] @ self.Wx + h @ self.Wh + self.b)
8         if return_sequences:
9             outputs.append(h.unsqueeze(1))
10
11    if return_sequences:
12        return tc.cat(outputs, dim=1)
13    else:
14        return h
```

Jika `return_sequences=True`, maka layer akan mengembalikan seluruh urutan hidden state dalam bentuk tensor berdimensi (`batch_size, seq_len, hidden_dim`). Jika `False`, hanya hidden state terakhir yang dikembalikan, yang biasanya digunakan untuk klasifikasi sekuen secara keseluruhan.

D. Bidirectional RNN Layer

Untuk membantu model memahami konteks kata secara lebih menyeluruh, kami menggunakan BidirectionalRNNLayer. Layer ini menggabungkan dua buah SimpleRNNLayer: satu membaca urutan kata dari awal ke akhir (maju/forward), dan satu lagi membaca dari akhir ke awal (mundur/backward). Dengan cara ini, model bisa menangkap informasi penting yang muncul baik sebelum maupun sesudah suatu kata dalam kalimat.

Input untuk arah mundur dibalik urutannya menggunakan `torch.flip`, lalu kedua hasil hidden state dari arah maju dan mundur digabungkan (concatenate) menggunakan `torch.cat`. Gabungan ini menjadi output akhir yang lebih informatif. Berikut adalah potongan kode utamanya:

```
1 def forward(self, x: tc.Tensor, return_sequences=False):
2     h_f_out = self.rnn_f.forward(x, return_sequences=return_sequences)
3
4     h_b_out_for_flipped_input = self.rnn_b.forward(tc.flip(x, dims=[1]), return_sequences=return_sequences)
5
6     if return_sequences:
7         h_b_aligned = tc.flip(h_b_out_for_flipped_input, dims=[1])
8         h_f_final = h_f_out
9         h_b_final = h_b_aligned
10    else:
11        h_f_final = h_f_out.unsqueeze(1)
12        h_b_final = h_b_out_for_flipped_input.unsqueeze(1)
13
14    h = tc.cat([h_f_final, h_b_final], dim=-1)
15
16    if not return_sequences:
17        h = h.squeeze(1)
18
19    return h
```

Dengan cara ini, model bisa memahami arti kata berdasarkan konteks sebelum dan sesudahnya secara bersamaan, yang sangat berguna untuk tugas seperti klasifikasi sentimen.

E. Dense Layer

Output dari layer RNN, baik yang unidirectional maupun bidirectional, diteruskan ke fully-connected layer untuk melakukan klasifikasi. Layer ini bertugas mengubah representasi akhir dari sekueens menjadi prediksi kelas.

Implementasi FFNN yang digunakan di sini merupakan hasil modifikasi dari Tugas Besar 1, di mana model terdiri dari beberapa layer linear berurutan, masing-masing disertai dengan fungsi aktivasi seperti ReLU atau Softmax. Proses forward dilakukan secara berantai dimana output dari satu layer menjadi input bagi layer berikutnya.

Setiap layer dalam FFNN memiliki metode forward() yang melakukan transformasi input menjadi output aktivasi menggunakan rumus dasar:

$$out = activation(X \times W + b)$$

dan diterapkan dalam kode sebagai berikut:

```
1 def forward(self, X):
2     self.input = X
3     self.net = X @ self.weights + self.biases
4     self.out = self.activation(self.net)
5     return self.out
```

Kumpulan layer ini dikemas dalam kelas FFNN, yang juga memiliki metode forward() untuk menjalankan semua layer secara berurutan:



```
1 def forward(self, X):
2     output = X
3     for layer in self.layers:
4         output = layer.forward(output)
5     return output
```

Dengan struktur ini, hasil keluaran dari RNN (misalnya berdimensi [batch_size, hidden_dim]) bisa langsung dikirim ke FFNN. Setiap layer dense akan mengubah bentuk data secara bertahap hingga menjadi output akhir dengan jumlah kelas yang diinginkan (misalnya 3 kelas untuk sentimen). Di akhir, digunakan fungsi aktivasi softmax untuk menghasilkan nilai probabilitas dari setiap kelas.

Seluruh komponen di atas diintegrasikan dalam kelas SimpleRNNManual, yang menjadi kerangka utama dari model kami. Proses forward dalam kelas ini dimulai dari input berupa token ID hasil vektorisasi teks. Token ini diteruskan ke EmbeddingLayer untuk mendapatkan representasi vektor tiap kata. Kemudian, data vektor tersebut diproses oleh setiap layer RNN secara berurutan. Jika terdapat lebih dari satu layer RNN, maka hanya layer terakhir yang tidak mengembalikan urutan (return_sequences=False), sementara layer sebelumnya akan mengembalikan seluruh hidden state (return_sequences=True). Jika model juga memiliki layer dense (untuk klasifikasi), maka hasil dari RNN diteruskan ke FFNN untuk menghasilkan prediksi akhir. Berikut potongan kode forward() yang digunakan:



```
1  def forward(self, x_token_ids: tc.Tensor):
2      x = self.embedding.forward(x_token_ids)
3
4      for i, rnn_layer_instance in enumerate(self.rnn_layers):
5          return_sequences = (i < len(self.rnn_layers) - 1)
6          x = rnn_layer_instance.forward(x, return_sequences=return_sequences)
7
8      if self.dense is not None:
9          return self.dense.forward(x)
10     return x
```

Alur ini mencerminkan transformasi data dari token ke embedding, lalu diproses oleh RNN, diteruskan ke dense layer, hingga menghasilkan output akhir, semuanya dikendalikan secara manual tanpa menggunakan abstraksi dari library tingkat tinggi.

2.3. LSTM

2.3.1. Deskripsi Kelas

2.3.1.1. Kelas *ScratchLSTMClassifier*

```
import numpy as np
from models.lstm.embedding_layer import ScratchEmbedding
from models.lstm.lstm_layer import ScratchLSTM
from models.lstm.dense_layer import ScratchDense

class ScratchLSTMClassifier:
    def __init__(self, emb_w=None, lstm_specs=None, d_w=None,
                 d_b=None):
        if emb_w is not None and lstm_specs is not None and d_w is not None:
            self.embedding = ScratchEmbedding(emb_w)
            self.dense = ScratchDense(d_w, d_b)
            self.lstm_specs = lstm_specs
            self.lstm_layers = []

    def forward(self, x):
        H = self.embedding.forward(x)
```

```

self.lstm_layers = []
for spec in self.lstm_specs:
    typ, return_seq, *weights = spec
    if typ == 'unidir':
        lstm = ScratchLSTM(*weights[0])
        H = lstm.forward(H,
return_sequences=return_seq)
        self.lstm_layers.append((typ, lstm))
    else:
        f_lstm = ScratchLSTM(*weights[0])
        b_lstm = ScratchLSTM(*weights[1])
        out_f = f_lstm.forward(H,
return_sequences=return_seq)
        out_b = b_lstm.forward(H[:, :, :-1, :],
return_sequences=return_seq)
        if return_seq:
            out_b = out_b[:, :, :-1, :]
            H = np.concatenate([out_f, out_b], axis=2)
        else:
            H = np.concatenate([out_f, out_b], axis=1)
        self.lstm_layers.append((typ, f_lstm, b_lstm))
if H.ndim == 3:
    H = H[:, -1, :]
self.last_input = H
return self.dense.forward(H)

def backward(self, dL_dout):
    dH = self.dense.backward(dL_dout)

    for layer in reversed(self.lstm_layers):
        if layer[0] == 'bidir':
            f_lstm, b_lstm = layer[1], layer[2]
            if dH.ndim == 3:
                D = dH.shape[2] // 2
                dH_f = dH[:, :, :D]
                dH_b = dH[:, :, D:]
                dX_f = f_lstm.backward(dH_f)
                dX_b = b_lstm.backward(dH_b[:, :, :-1, :])
                dH = dX_f + dX_b[:, :, :-1, :]
            else:
                D = dH.shape[1] // 2
                dH_f = dH[:, :D]
                dH_b = dH[:, D:]
                dX_f = f_lstm.backward(dH_f)
                dX_b = b_lstm.backward(dH_b)
                dH = dX_f + dX_b

```

```

        self.embedding.backward(dH)

    def save_npy(self, path):
        data = {
            "embedding": self.embedding.W,
            "dense_W": self.dense.W,
            "dense_b": self.dense.b,
            "lstm_specs": self.lstm_specs
        }
        np.save(path, data)

    def load_npy(self, path):
        data = np.load(path, allow_pickle=True).item()
        self.embedding = ScratchEmbedding(data["embedding"])
        self.dense = ScratchDense(data["dense_W"],
        data["dense_b"])
        self.lstm_specs = data["lstm_specs"]

```

Kelas ini merepresentasikan sebuah model klasifikasi berbasis LSTM dari awal (scratch). Model ini terdiri dari tiga komponen utama. Embedding layer untuk mengubah input integer (token) menjadi vektor. LSTM layer yang bisa terdiri dari satu atau beberapa layer, dengan opsi unidirectional atau bidirectional. Dense layer untuk klasifikasi akhir.

Atribut:

1. Embedding
Objek dari ScratchEmbedding, memetakan token ke vektor embedding.
2. lstm_layers
List yang menyimpan layer-layer LSTM, baik unidirectional maupun bidirectional.
3. lstm_specs
Konfigurasi setiap LSTM layer berupa tipe (unidir / bidir), apakah mengembalikan seluruh urutan, dan bobot-bobot masing-masing layer.
4. dense
Objek dari ScratchDense, digunakan untuk menghasilkan output klasifikasi dari hidden state terakhir.
5. last_input
Input terakhir yang diberikan ke layer dense (untuk keperluan backpropagation).

Method:

1. __init__(self, emb_w=None, lstm_specs=None, d_w=None, d_b=None)
Konstruktor untuk inisialisasi model. Jika semua parameter tersedia, maka layer-layer (embedding, dense, dan lstm_specs) akan diinisialisasi

- emb_w merupakan bobot awal untuk embedding (biasanya matrix vocab × dimensi)
 - lstm_specs merupakan list konfigurasi untuk setiap LSTM layer
 - d_w, d_b merupakan bobot dan bias untuk dense layer
2. forward(self, x)
- Melakukan forward propagation dari input x
- Input x (dalam bentuk token ID) masuk ke layer embedding hasil: (batch_size, seq_len, emb_dim).
 - Hasil embedding masuk ke tiap LSTM layer sesuai lstm_specs
 - Untuk LSTM bidirectional
- Hasil forward dan backward digabung (concat) secara tepat. Jika return_seq = False, hanya hidden terakhir yang digunakan.
- Jika hasil akhir dari LSTM berbentuk sekuens (ndim == 3), hanya timestep terakhir yang diambil
 - Output akhir masuk ke layer dense menjadi hasil akhir klasifikasi
3. backward(self, dL_dout)
- Melakukan backpropagation dari loss hingga ke input
- Gradien dari loss (dL_dout) disalurkan ke dense.backward()
 - Gradien dari dense dilanjutkan ke layer-layer LSTM secara terbalik (reversed). Untuk bidirectional LSTM, gradien dibagi dua: forward dan backward, dan di-backprop masing-masing lalu digabung kembali
 - Setelah LSTM, gradien terakhir dikembalikan ke embedding.backward()
4. save_npy(self, path)
- Menyimpan parameter model ke file .npy. Yang disimpan
- Bobot embedding
 - Bobot & bias dense layer
 - Konfigurasi lstm_specs (termasuk semua bobot LSTM)
5. load_npy(self, path)
- Memuat kembali model dari file .npy
- Membuat ulang layer embedding dan dense berdasarkan bobot yang tersimpan.
 - Menyimpan konfigurasi lstm_specs, tetapi belum membuat ulang LSTM karena membutuhkan forward() dulu untuk tahu bentuk input dan menginisialisasi ulang.

2.3.1.2. Kelas *ScratchLSTM*

```
import numpy as np

class ScratchLSTM:
    def __init__(self, W, U, b):
        self.W, self.U, self.b = W, U, b
        self.units = U.shape[0]
```

```

def forward(self, x, return_sequences=False):
    self.x = x
    B, T, _ = x.shape
    H = np.zeros((B, self.units), np.float32)
    C = np.zeros((B, self.units), np.float32)
    self.cache = []

    outputs = []
    for t in range(T):
        xt = x[:, t, :]
        z = xt @ self.W + H @ self.U + self.b
        i, f, g, o = np.split(z, 4, axis=1)
        i = 1 / (1 + np.exp(-i))
        f = 1 / (1 + np.exp(-f))
        g = np.tanh(g)
        o = 1 / (1 + np.exp(-o))
        C = f * C + i * g
        H = o * np.tanh(C)
        self.cache.append((xt, H.copy(), C.copy(), i, f, g,
                           o))
    if return_sequences:
        outputs.append(H[:, None, :])
    self.return_sequences = return_sequences
    return np.concatenate(outputs, axis=1) if return_sequences else H

def backward(self, dL_dH):
    B, T, D = self.x.shape
    dW = np.zeros_like(self.W)
    dU = np.zeros_like(self.U)
    db = np.zeros_like(self.b)
    dX_all = np.zeros_like(self.x)
    dH_next = np.zeros((B, self.units))
    dC_next = np.zeros((B, self.units))

    range_T = range(T - 1, -1, -1) if self.return_sequences
    else [T - 1]

    for t in range_T:
        xt, H, C, i, f, g, o = self.cache[t]
        dH = dL_dH[:, t, :] if self.return_sequences else
dL_dH
        dH += dH_next

        tanhC = np.tanh(C)
        dO = dH * tanhC * o * (1 - o)

```

```

dC = dH * o * (1 - tanhC ** 2) + dC_next

dF = dC * self.cache[t - 1][2] * f * (1 - f) if t >
0 else dC * 0
dI = dC * g * i * (1 - i)
dG = dC * i * (1 - g ** 2)

dz = np.concatenate([dI, dF, dG, dO], axis=1)

dW += xt.T @ dz
dU += self.cache[t - 1][1].T @ dz if t > 0 else 0
db += dz.sum(axis=0)

dX_step = dz @ self.W.T
dH_next = dz @ self.U.T
dC_next = dC * f

dX_all[:, t, :] = dX_step

self.dW = dW
self.dU = dU
self.db = db
return dX_all

```

Kelas ini merepresentasikan implementasi manual (from scratch) dari layer LSTM (Long Short-Term Memory) yang digunakan untuk memproses data sekuensial dalam model RNN.

Atribut:

1. W
Matriks bobot input (dari x ke gate), bentuk (input_dim, 4 * units)
2. U
Matriks bobot recurrent (dari h_prev ke gate), bentuk (units, 4 * units)
3. b
Bias untuk gate, bentuk (4 * units,)
4. units
Jumlah unit LSTM (hidden size), diturunkan dari U.shape[0]
5. cache
Menyimpan semua nilai intermediate per timestep untuk keperluan backpropagation

Method:

1. __init__(self, W, U, b)
Konstruktor untuk inisialisasi parameter bobot input-to-hidden (W), hidden-to-hidden (U), dan bias (b)
2. backward(self, dL_dH)

Melakukan backward propagation dari loss terhadap output H, baik full sequence (B, T, units) atau timestep terakhir (B, units)

1. Inisialisasi gradien:
 - dW, dU, db merupakan gradien dari parameter
 - dX_all merupakan gradien dari input x, akan diisi per timestep
 - dH_next, dC_next merupakan hidden dan cell state gradien dari timestep berikutnya (di-set awal ke nol)
 2. Loop mundur (backpropagation through time):
 - Ambil cache per timestep: xt, H, C, i, f, g, o
 - Tambahkan gradien timestep saat ini ke dH
 - Hitung gradien untuk output gate o, cell state C, input gate i, forget gate f, dan candidate g
 - Gabungkan semua gradien gate jadi dz
 - Hitung dan akumulasikan gradien ke dW, dU, dan db
 - Hitung dX_step (gradien ke input timestep xt)
 - Update dH_next dan dC_next untuk timestep sebelumnya
 - Simpan dX_step ke dX_all
 3. Output:
 - Return dX_all, yaitu gradien terhadap input x dengan shape (B, T, D)
3. forward(self, x, return_sequences=False)
- Melakukan **forward propagation** pada input x dengan shape (batch_size, time_steps, input_dim).
1. Inisialisasi
 - H merupakan hidden state awal, di-set ke nol (B, units)
 - C merupakan cell state awal, di-set ke nol (B, units)
 - Outputs berupa list untuk menyimpan output tiap timestep jika return_sequences=True
 2. Loop melalui setiap timestep t
 - xt: Input pada timestep t
 - z: Linear combination dari xt dan H untuk semua gate
 - Pisah z menjadi i, f, g, dan o (input, forget, cell, output gate)
 - Hitung aktivasi gate
 - i. i, f, o: sigmoid
 - ii. g: tanh
 - Update cell state C dan hidden state H
 - Simpan ke cache untuk digunakan pada backward
 - Simpan H jika return_sequences=True
 3. Output:
 - Jika return_sequences=True: gabungkan semua output menjadi (B, T, units)

- Jika False: hanya output timestep terakhir (B, units)

2.3.1.3. Kelas *ScratchEmbedding*

```
import numpy as np

class ScratchEmbedding:
    def __init__(self, W: np.ndarray):
        self.W = W # shape (vocab, dim)

    def forward(self, x_int):
        self.x_int = x_int
        return self.W[x_int]

    def backward(self, dL_dout):
        self.dW = np.zeros_like(self.W)
        np.add.at(self.dW, self.x_int, dL_dout)
        return None # No gradient w.r.t. input ids
```

Kelas ini merepresentasikan layer embedding buatan sendiri, seperti tf.keras.layers.Embedding. Layer ini mengubah token ID menjadi vektor representasi (embedding) berdasarkan matriks bobot.

Atribut:

1. W
Matriks bobot embedding dengan bentuk (vocab_size, embed_dim), di mana setiap baris merepresentasikan vektor dari satu token unik.
2. x_int
Input integer (token ID), disimpan untuk keperluan backward.
3. dW
Gradien dari loss terhadap matriks embedding W, dihitung selama backward.

Method:

1. __init__(self, W)
Konstruktor untuk menginisialisasi bobot embedding.
 - W: Matriks awal dengan bentuk (vocab_size, embed_dim)
2. forward(self, x, return_sequences=False)
Melakukan lookup embedding dari token ID ke vektor embedding.
 - Input: x_int dengan shape (batch_size, sequence_length)
 - Output: vektor embedding (batch_size, sequence_length, embed_dim) (mengakses baris W[x_int])
3. backward(self, dL_dout)
Menghitung gradien terhadap bobot embedding berdasarkan output layer sebelumnya.
 - Input

dL_dout adalah gradien dari loss terhadap output embedding, bentuk sama dengan hasil forward, yaitu (`batch_size`, `sequence_length`, `embed_dim`)

- Proses
 - `np.add.at(self.dW, self.x_int, dL_dout)` akan menjumlahkan gradien ke baris yang sesuai dengan ID token.
- Output: None

2.3.1.4. Kelas *ScratchDense*

```
import numpy as np

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

    def forward(self, x):
        self.last_input = x
        z = x @ self.W + self.b
        e = np.exp(z - z.max(axis=1, keepdims=True))
        self.out = e / e.sum(axis=1, keepdims=True)
        return self.out

    def backward(self, dL_dout):
        x = self.last_input
        dL_dz = self.out * (dL_dout - (dL_dout *
        self.out).sum(axis=1, keepdims=True))
        self.dW = x.T @ dL_dz
        self.db = dL_dz.sum(axis=0)
        dL_dx = dL_dz @ self.W.T
        return dL_dx
```

Kelas ini merepresentasikan layer dense (fully connected) terakhir dalam jaringan neural untuk klasifikasi, dengan aktivasi softmax. Layer ini biasanya digunakan sebagai output layer dalam model klasifikasi multi-kelas.

Atribut:

1. `W`
Bobot dense layer, shape (`input_dim`, `output_dim`)
2. `b`
Bias dense layer, shape (`output_dim`,
3. `last_input`
Input yang disimpan selama forward pass, digunakan untuk backward
4. `out`
Output softmax dari forward pass, juga disimpan untuk backward

5. dW, db

Gradien terhadap W dan b, dihitung saat backward

Method:

1. __init__(self, W, b)

Konstruktor untuk inisialisasi bobot dan bias. W merupakan Matriks bobot. b merupakan Vektor bias

2. forward(self, x)

Melakukan forward propagation dari input x

1. Simpan input x untuk backward.

2. Hitung output linear: $z = x @ W + b$

3. Terapkan softmax untuk normalisasi ke distribusi probabilitas

- $e = np.exp(z - z.max(axis=1, keepdims=True))$ digunakan untuk numerik stabil

- $out = e / e.sum(axis=1, keepdims=True)$ menghasilkan probabilitas untuk setiap kelas

3. backward(self, dL_dout)

Melakukan backward propagation terhadap loss.

1. dL_dout: gradien dari loss terhadap output softmax, bentuk (batch_size, num_classes)

2. dL_dz: gradien loss terhadap input linear z,

3. Hitung gradien terhadap bobot dan bias:

- $dW = x.T @ dL_dz$

- $db = dL_dz.sum(axis=0)$

4. Hitung gradien terhadap input layer sebelumnya:

- $dL_dx = dL_dz @ W.T$

2.3.2. Deskripsi Fungsi

2.3.2.1. Fungsi build_lstm_model

```
from tensorflow.keras import Model, Input
from tensorflow.keras.layers import Embedding, LSTM,
    Bidirectional, Dropout, Dense

def build_lstm_model(n_layers, units, bidirectional, max_len,
    max_tokens, embed_dim, num_classes):
    inp = Input(shape=(max_len,), dtype='int32')
    x = Embedding(input_dim=max_tokens,
        output_dim=embed_dim)(inp)
    for i in range(n_layers):
        lstm_cls = LSTM if not bidirectional else lambda
        *a, **kw: Bidirectional(LSTM(*a, **kw))
        return_seq = i < n_layers - 1
        x = lstm_cls(units[i], return_sequences=return_seq)(x)
```

```
x = Dropout(0.5)(x)
out = Dense(num_classes, activation='softmax')(x)
model = Model(inp, out)
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam', metrics[])
return model
```

Fungsi ini membuat dan mengembalikan model LSTM berbasis Keras (`tf.keras.Model`) untuk klasifikasi sekuensial. Model dapat berupa unidirectional maupun bidirectional LSTM, tergantung parameter.

Parameter:

- `n_layers` merupakan Jumlah layer LSTM
- `units` merupakan list jumlah unit (neuron) pada tiap LSTM layer, panjang list harus = `n_layers`
- `bidirectional` merupakan boolean, apakah semua layer LSTM menggunakan Bidirectional
- `max_len` merupakan panjang maksimum sekuens input (panjang padding/truncation)
- `max_tokens` merupakan ukuran vocab (jumlah token unik)
- `embed_dim` merupakan dimensi vektor embedding
- `num_classes` merupakan Jumlah kelas target untuk klasifikasi

2.3.3. Penjelasan Forward Propagation dan Backward Propagation

Tahap pertama dimulai saat input berupa angka-angka masuk ke layer *ScratchEmbedding*. Setiap angka akan dicocokkan dengan baris pada matriks embedding untuk mengubahnya menjadi vektor berdimensi tetap. Hasil dari tahap ini adalah tensor 3 dimensi berisi representasi kata dalam bentuk angka, satu vektor untuk setiap kata dalam setiap kalimat.

```

1  class ScratchLSTMClassifier:
2      def __init__(self, emb_w=None, lstm_specs=None, d_w=None, d_b=None):
3          if emb_w is not None and lstm_specs is not None and d_w is not None:
4              self.embedding = ScratchEmbedding(emb_w)
5              self.dense = ScratchDense(d_w, d_b)
6              self.lstm_specs = lstm_specs
7              self.lstm_layers = []
8
9      def forward(self, x):
10         H = self.embedding.forward(x)
11         self.lstm_layers = []
12         for spec in self.lstm_specs:
13             typ, return_seq, *weights = spec
14             if typ == 'unidir':
15                 lstm = ScratchLSTM(*weights[0])
16                 H = lstm.forward(H, return_sequences=return_seq)
17                 self.lstm_layers.append((typ, lstm))
18             else:
19                 f_lstm = ScratchLSTM(*weights[0])
20                 b_lstm = ScratchLSTM(*weights[1])
21                 out_f = f_lstm.forward(H, return_sequences=return_seq)
22                 out_b = b_lstm.forward(H[:, ::-1, :], return_sequences=return_seq)
23                 if return_seq:
24                     out_b = out_b[:, ::-1, :]
25                     H = np.concatenate([out_f, out_b], axis=2)
26                 else:
27                     H = np.concatenate([out_f, out_b], axis=1)
28                     self.lstm_layers.append((typ, f_lstm, b_lstm))
29         if H.ndim == 3:
30             H = H[:, -1, :]
31         self.last_input = H
32         return self.dense.forward(H)

```

Selanjutnya, hasil embedding masuk ke satu atau lebih *layer* LSTM sesuai dengan spesifikasi (*lstm_specs*). Jika *layer* bertipe *unidirectional*, maka input diproses dari awal hingga akhir kalimat (kiri ke kanan). Setiap langkah waktu menghitung *hidden state* dan *cell state* berdasarkan input saat itu dan hasil dari langkah sebelumnya. LSTM menggunakan empat komponen utama (gate): *forget gate*, *input gate*, *candidate value*, dan *output gate* untuk mengatur informasi apa yang harus disimpan, ditambahkan, atau dilupakan. Jika *return_sequences=True*, maka hidden state dari setiap langkah waktu disimpan, jika tidak, hanya hidden state terakhir yang digunakan.

Jika layer bertipe *bidirectional*, maka input akan diproses dua kali: satu dari kiri ke kanan (forward) dan satu lagi dari kanan ke kiri (backward). Dua buah objek LSTM dibuat dan masing-masing menghitung hidden state berdasarkan arah prosesnya. Setelah itu, hasil dari dua

arah ini digabung, baik per langkah waktu maupun hanya pada hasil akhir. Dengan cara ini, model bisa memahami konteks sebelum dan sesudah setiap kata, yang sangat berguna dalam memahami makna kalimat secara utuh. Untuk implementasi tiap tiap gate, dapat dilihat pada gambar dibawah:

```
● ● ●

1  class ScratchLSTM:
2      def __init__(self, W, U, b):
3          self.W, self.U, self.b = W, U, b
4          self.units = U.shape[0]
5
6      def forward(self, x, return_sequences=False):
7          self.x = x
8          B, T, _ = x.shape
9          H = np.zeros((B, self.units), np.float32)
10         C = np.zeros((B, self.units), np.float32)
11         self.cache = []
12
13         outputs = []
14         for t in range(T):
15             xt = x[:, t, :]
16             z = xt @ self.W + H @ self.U + self.b
17             i, f, g, o = np.split(z, 4, axis=1)
18             i = 1 / (1 + np.exp(-i))
19             f = 1 / (1 + np.exp(-f))
20             g = np.tanh(g)
21             o = 1 / (1 + np.exp(-o))
22             C = f * C + i * g
23             H = o * np.tanh(C)
24             self.cache.append((xt, H.copy(), C.copy(), i, f, g, o))
25             if return_sequences:
26                 outputs.append(H[:, None, :])
27         self.return_sequences = return_sequences
28         return np.concatenate(outputs, axis=1) if return_sequences else H
```

Dari kode diatas, terlihat untuk setiap *time step*, dicari nilai z untuk tiap nilai *gate*. Tiap tiap *gate* kemudian dijalankan dengan fungsi aktivasi yang seharusnya , baik *tanh* maupun *sigmoid*. Kemudian dilakukan pembaruan *hidden state* sebagai *output* dari *time step* ini.

$$\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
\tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t &= o_t * \tanh(C_t)
\end{aligned}$$

```

● ● ●

1 import numpy as np
2
3 class ScratchDense:
4     def __init__(self, W, b):
5         self.W = W
6         self.b = b
7
8     def forward(self, x):
9         self.last_input = x
10    z = x @ self.W + self.b
11    e = np.exp(z - z.max(axis=1, keepdims=True))
12    self.out = e / e.sum(axis=1, keepdims=True)
13    return self.out
14
15    def backward(self, dL_dout):
16        x = self.last_input
17        dL_dz = self.out * (dL_dout - (dL_dout * self.out).sum(axis=1, keepdims=True))
18        self.dW = x.T @ dL_dz
19        self.db = dL_dz.sum(axis=0)
20        dL_dx = dL_dz @ self.W.T
21        return dL_dx

```

Setelah semua tahapan diatas selesai, barulah dilanjutkan ke *dense layer*. Tidak seperti FFNN pada umumnya, pada FC *Layer* ini sudah diset untuk menggunakan *sigmoid* sebagai fungsi aktivasinya. Terkait dari cara kerjanya, sama dengan FFNN atau FC yang sudah dijelaskan pada bagian-bagian sebelumnya.

Pada proses backward dari *ScratchLSTM*, model menghitung turunan (gradien) terhadap parameter dan input secara berurutan dari waktu terakhir ke waktu pertama. Pada setiap langkah waktu t , gradien dari loss terhadap hidden state h ditambahkan ke gradien dari waktu selanjutnya (dH_{next}). Model kemudian menghitung gradien terhadap gate output, cell state, forget gate, input gate, dan kandidat nilai menggunakan turunan fungsi aktivasi (sigmoid dan tanh). Gradien

tersebut digabung menjadi satu (dz) dan digunakan untuk menghitung turunan terhadap bobot input W , bobot hidden state U , serta bias b .

Selanjutnya, gradien input terhadap langkah waktu saat itu (dX_{step}) dihitung dan disimpan ke tensor dX_{all} . Model juga menghitung gradien yang akan diteruskan ke waktu sebelumnya (dH_{next} dan dC_{next}). Jika `return_sequences=True`, maka gradien diberikan untuk semua langkah waktu; jika `False`, hanya gradien dari langkah terakhir yang dihitung. Semua gradien disimpan di atribut dW , dU , dan db untuk digunakan dalam update parameter saat training.

BAB 3

PENGUJIAN

3.1. Pengujian CNN

Pengujian ini bertujuan untuk mengevaluasi performa model *Convolutional Neural Network* (CNN) dalam melakukan klasifikasi gambar pada dataset CIFAR-10. Pengujian dilakukan dengan berbagai variasi konfigurasi model untuk mengamati pengaruhnya terhadap hasil klasifikasi. Selain itu, pengujian juga dilakukan untuk memastikan bahwa implementasi *forward propagation* secara manual telah berjalan dengan benar, dengan cara membandingkan hasilnya dengan model bawaan dari Keras.

3.1.1. Pengaruh jumlah layer konvolusi

3.1.1.1. Konfigurasi Eksperimen

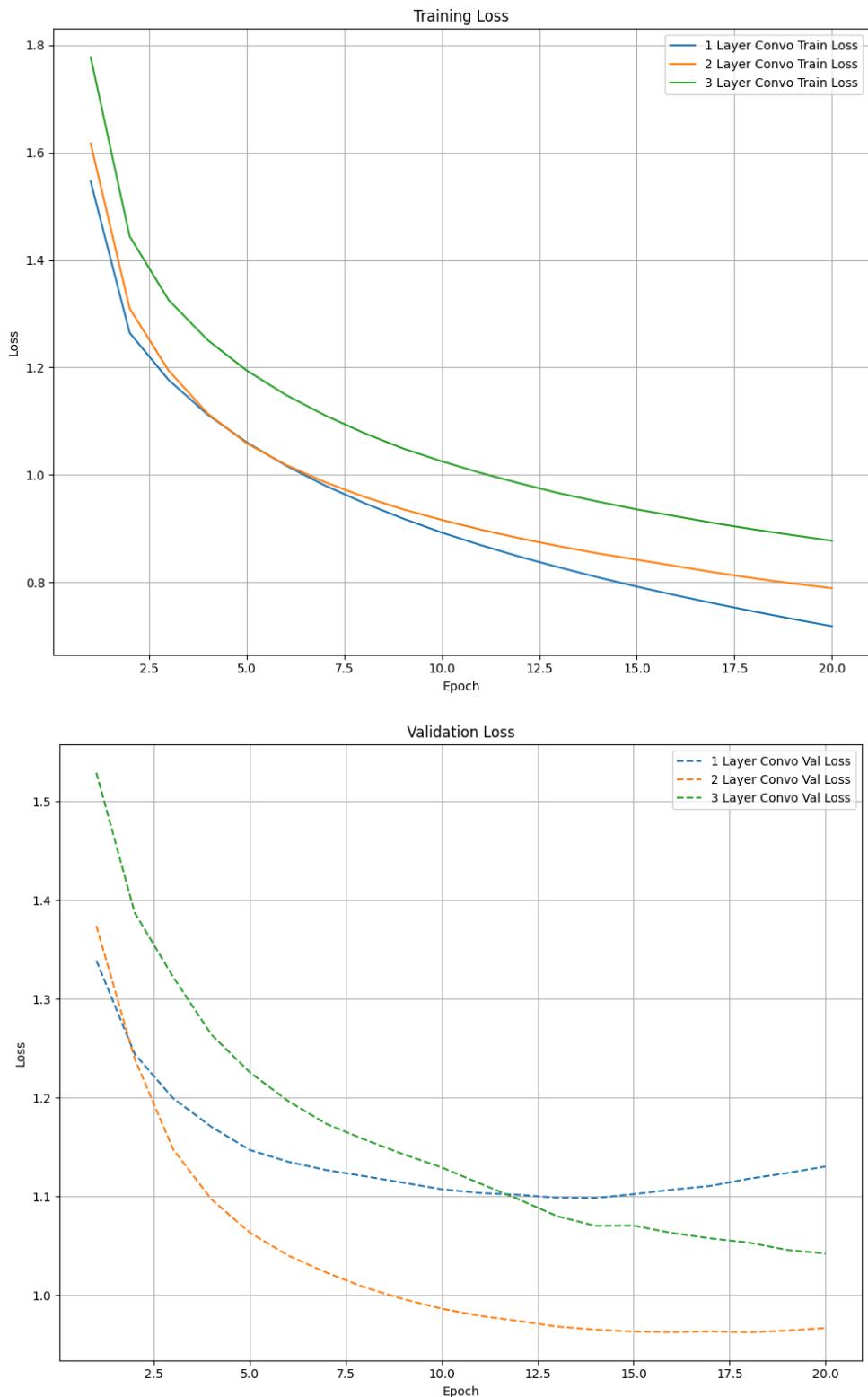
Pada pengujian ini, pelatihan dilakukan selama 20 epoch dengan satu lapisan konvolusi yang terdiri dari 32 filter berukuran 3×3 . Selanjutnya, pada bagian ini akan diterapkan berbagai variasi jumlah layer konvolusi untuk mengevaluasi bagaimana perubahan kedalaman jaringan mempengaruhi kinerja model. Selain itu, pada eksperimen ini ditambahkan 1 layer *Max Pooling 2D*, 1 *Layer Flatten* dan *Dense Layer* dengan *Output* 10.

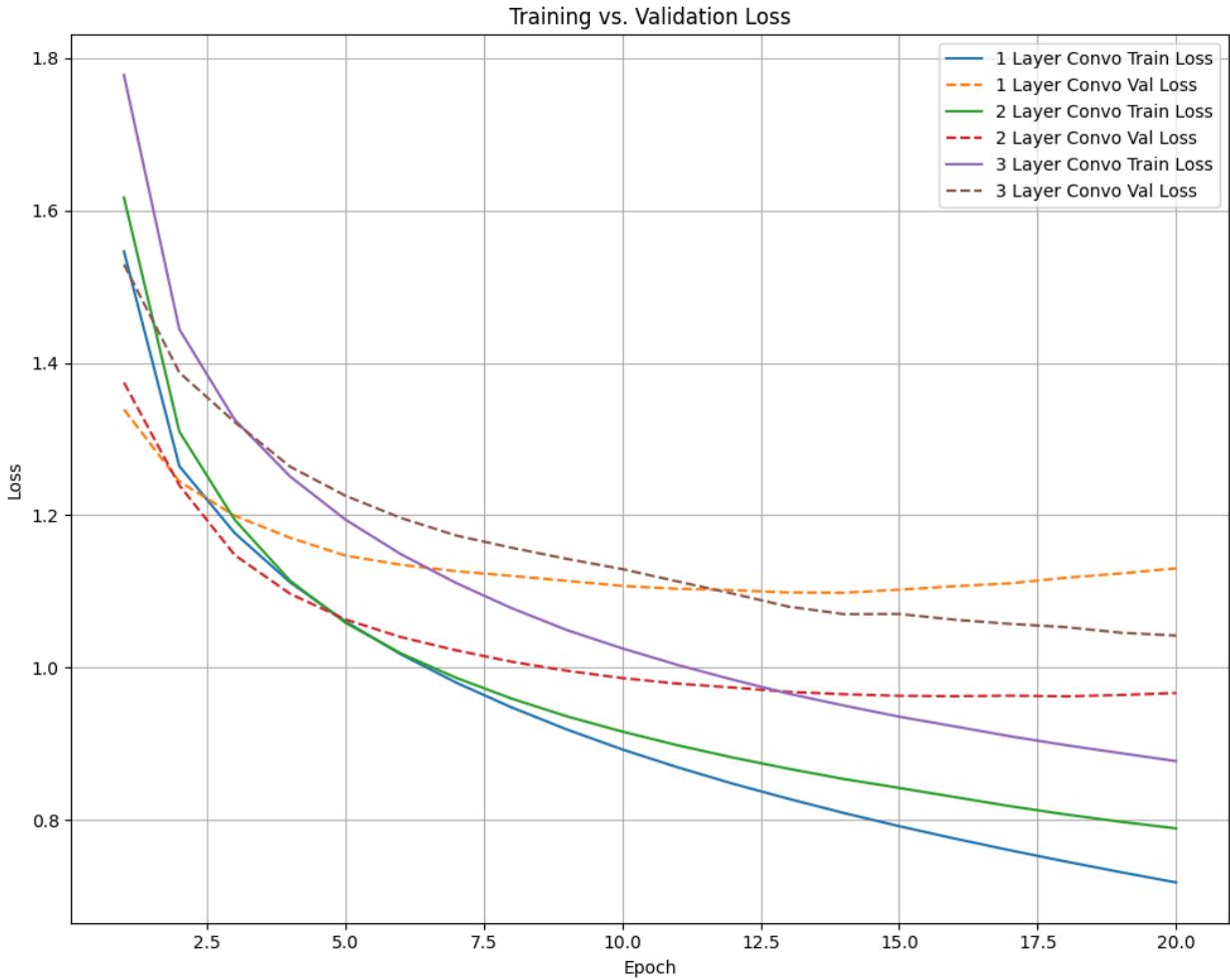
3.1.1.2. Hasil Pengujian

Berikut merupakan hasil evaluasi model dengan berbagai variasi jumlah layer konvolusi yang telah dilakukan. Evaluasi dilakukan pada data uji dan menggunakan metrik macro F1-score.

Jumlah Layer Konvolusi	Konfigurasi	Waktu Latih (detik)	F1-Score (Keras)	F1-Score (Manual)
1 layer	[32]	57.3	0.61663	0.61663
2 layer	[32, 32]	84.2	0.6668	0.6668
3 layer	[32, 32, 32]	97.1	0.6345	0.6345

Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini





3.1.1.3. Analisis dan Kesimpulan

Dari hasil konfigurasi diatas, terbukti bahwa 2 *layer* konvolusi memberikan hasil yang lebih optimal. Hal ini sebenarnya lebih lazim , karena sebenarnya terlihat bahwa jika jumlah konvolusi terlalu kecil, model lebih tidak mengandalkan pola pada data latih. Terlihat pada konvolusi 3 lapis, karakteristik pada data latih dan validasi sama, mereka mulai dengan *loss* yang tinggi, namun masih bisa mengejar hasil optimal.

Sebaliknya, konvolusi 1 *layer* terlihat tidak memiliki performa yang sama dengan data validasi, dimana model sulit mengalami penurunan nilai *loss* , berbeda dengan pada data latih. Layer 2 menyeimbangkan antara keduanya, sehingga tidak begitu bergantung pada pola yang ada.

Sebenarnya kembali lagi , tidak ada penentu pasti mengenai peningkatan akurasi prediksi. Hanya saja , semakin dalam konvolusi , bisa saja terjadi *overfitting*, namun menurut penulis , pada kasus ini model bersifat *overfit* pada data latih , karena memiliki penurunan yang kurang

tajam dibandingkan pada data latih. Sedangkan untuk 2 *layer*, pada pelatihan memiliki performa yang mirip dengan 1 *layer*, namun, lebih *fit* dari 1 *layer* yang menurut penulis kurang *general* akibat kurangnya konvolusi sehingga tidak cukup mampu dalam melakukan prediksi, *layer* 2 mempertahankan konsistensi untuk data latih dan validasi sehingga memberikan hasil lebih baik.

3.1.2. Pengaruh banyak filter per layer konvolusi

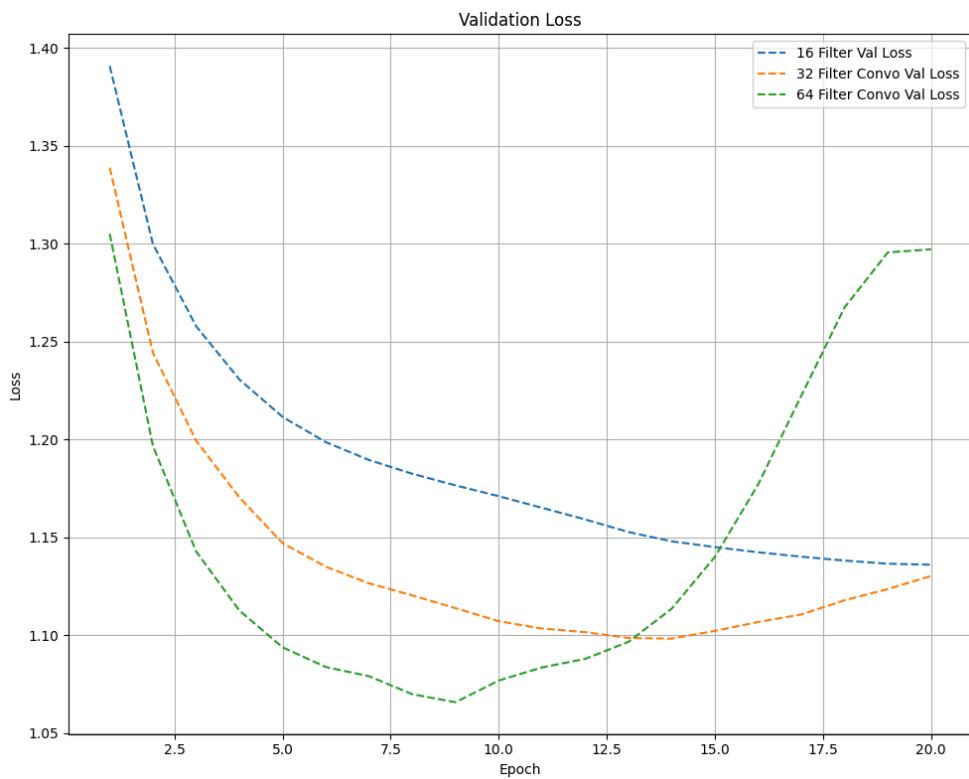
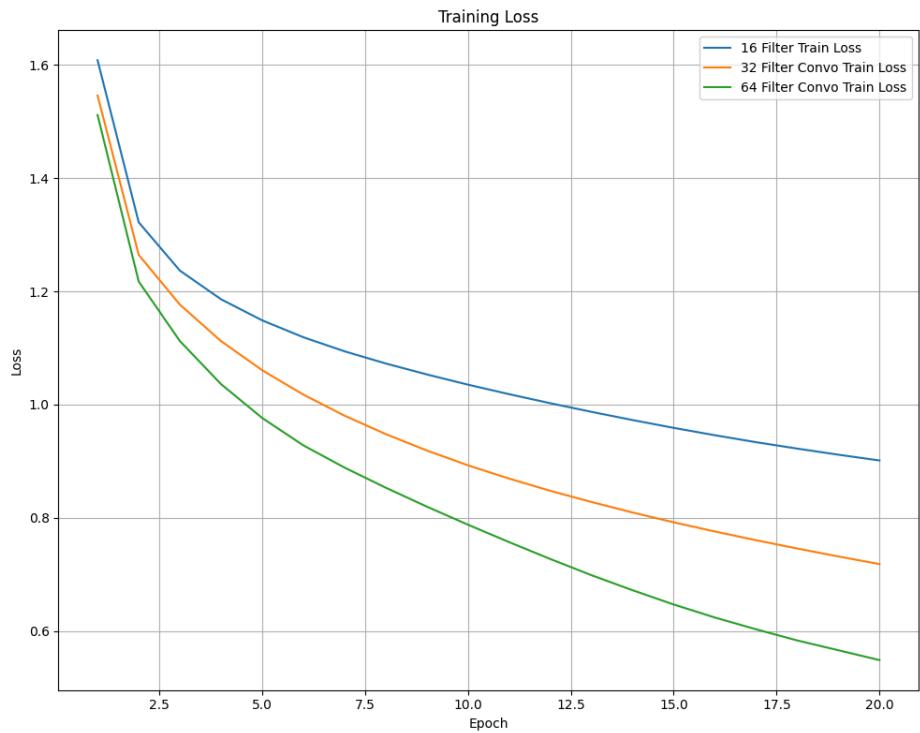
3.1.2.1. Konfigurasi Eksperimen

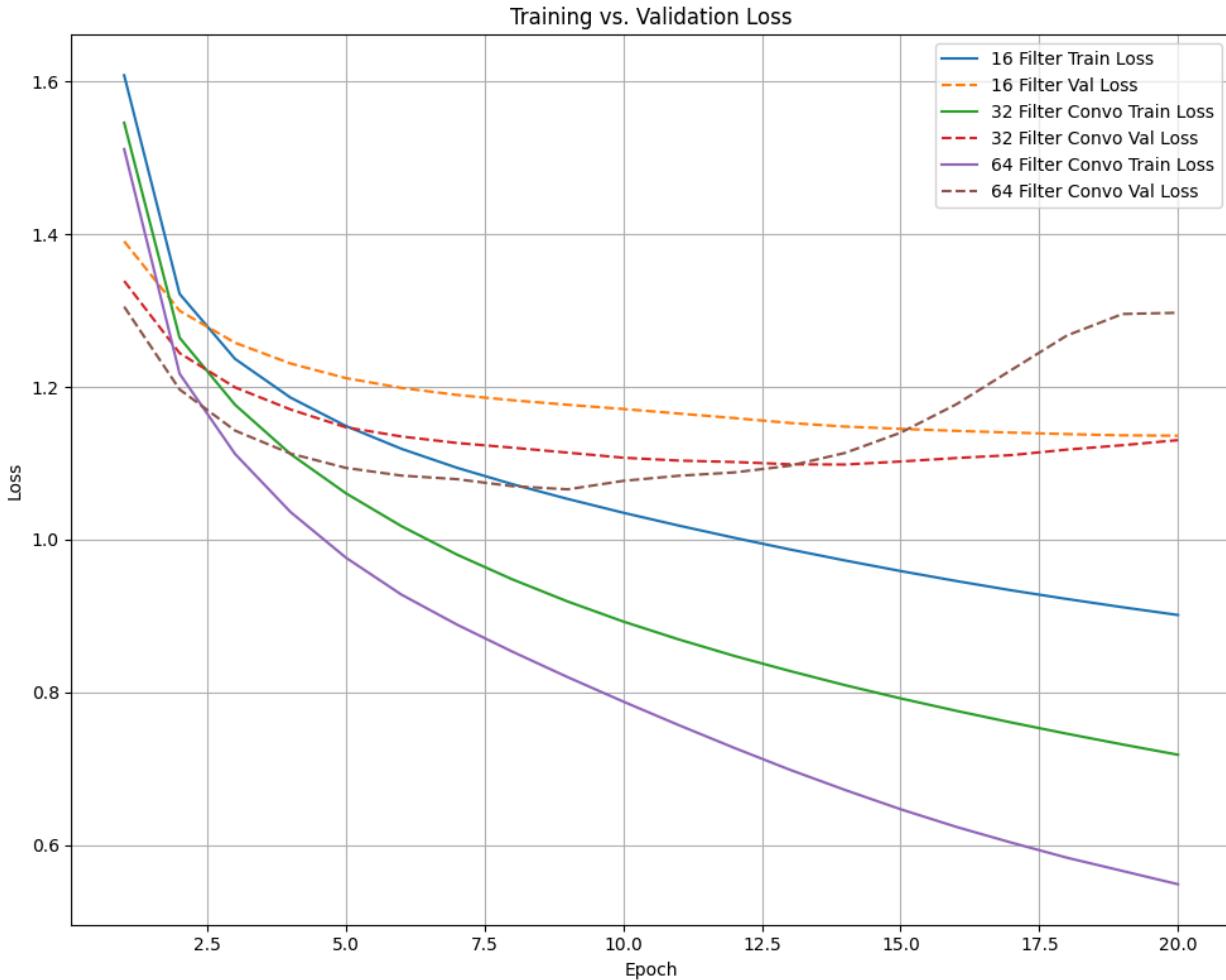
Pengujian kali ini dilakukan selama 20 epoch dengan konfigurasi satu layer konvolusi yang menggunakan filter berukuran 3×3 . Pada bagian ini, akan dilakukan berbagai variasi jumlah filter konvolusi untuk melihat pengaruhnya terhadap performa model. Selain itu, pada eksperimen ini ditambahkan 1 layer *Max Pooling 2D*, 1 *Layer Flatten* dan *Dense Layer* dengan *Output* 10.

3.1.2.2. Hasil Pengujian

Jumlah Filter Konvolusi	Konfigurasi	Waktu Latih (detik)	F1-Score (Keras)	F1-Score (Manual)
16 filter	[16]	48.3	0.6101	0.6101
32 filter	[32]	54.5	0.6166	0.6166
64 filter	[64]	86.3	0.5991	0.5991

Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini





3.1.2.3. Analisis dan Kesimpulan

Penulis merasa bahwa dari grafik sudah cukup terus terang mengenai pengaruh jumlah filter dibandingkan dengan performa model. Dengan lebih banyak filter, performa model dapat lebih banyak mempelajari pola dalam model. Terlihat pada grafik *train loss*, semakin banyak filter memiliki penurunan yang lebih baik pula.

Namun, seperti yang sudah dijelaskan sebelumnya, semakin banyak menghafal pola, semakin banyak resiko *overfitting*. Disini bisa dilihat bahwa pada titik *epoch* tertentu , model yang memiliki jumlah filter yang banyak mulai mengalami lonjakan pada nilai *loss*. Jika kita hentikan sampai 5 iterasi mungkin tidak akan terlihat , namun karena dilanjutkan sampai 20 iterasi, model yang lebih banyak filter mulai *overfit*.

Fenomena *overfitting* dimulai oleh model dengan 64 filter , yang kemudian diikuti dengan model yang memiliki 32 filter. Namun, karena dibatasi sampai 20 iterasi, pada kasus ini , model dengan filter sebanyak 32 buah masih memimpin skor. Namun skor tersebut hanya berbeda sedikit

dengan 16 filter , yang menandakan adanya kemungkinan bahwa model bisa saja lebih overfit jika iterasi diperbanyak.

Kesimpulannya yaitu untuk mengecek *loss* terhadap iterasi untuk memastikan limitasi dari *overfitting* itu sendiri agar memastikan keseimbangan antara iterasi dengan konvolusi.

3.1.3. Pengaruh ukuran filter per layer konvolusi

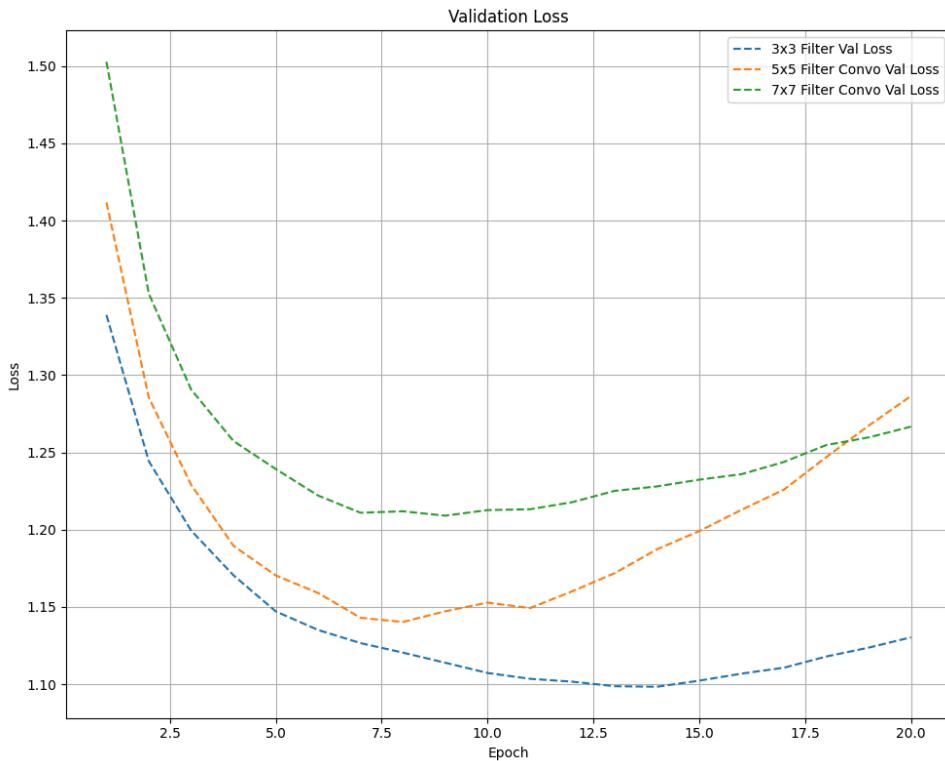
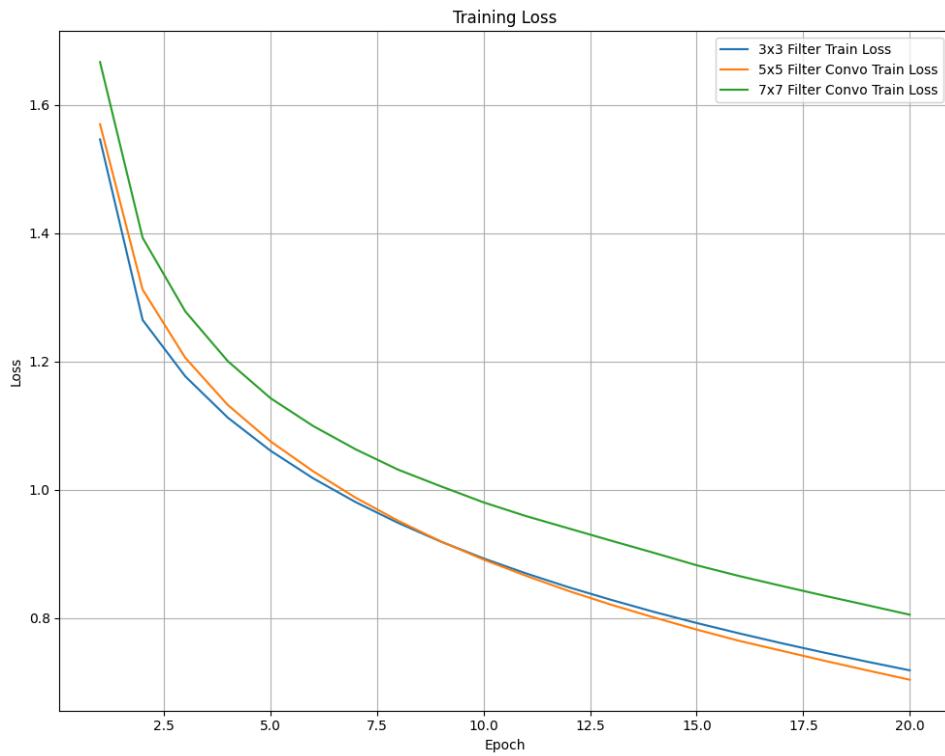
3.1.3.1. Konfigurasi Eksperimen

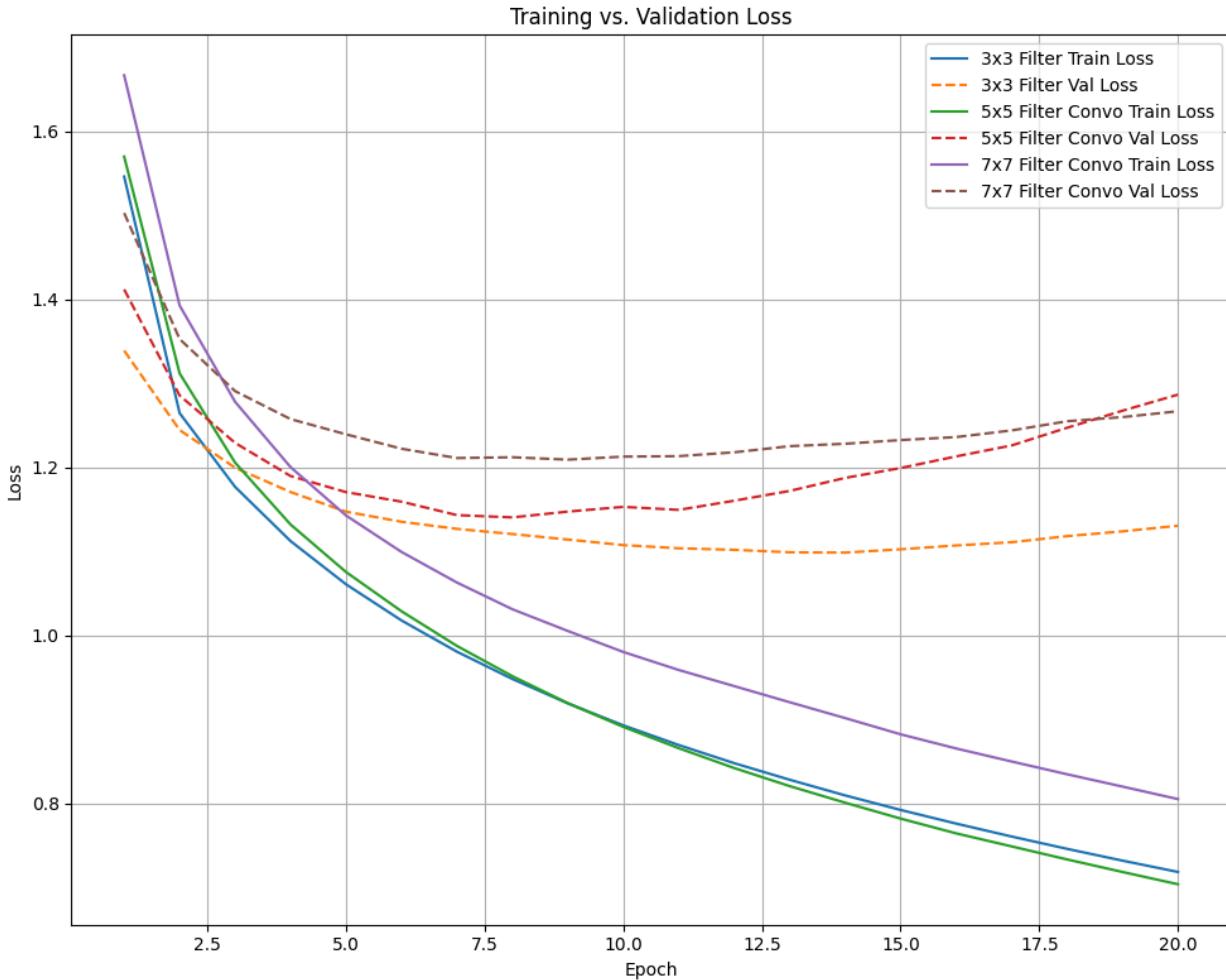
Pengujian kali ini dilakukan selama 20 epoch dengan konfigurasi satu layer konvolusi yang memiliki 32 filter. Pada bagian ini, akan dilakukan berbagai variasi ukuran filter konvolusi untuk menganalisis pengaruh ukuran filter terhadap kinerja model dalam proses pelatihan. Selain itu, pada eksperimen ini ditambahkan 1 layer *Max Pooling 2D*, 1 *Layer Flatten* dan *Dense Layer* dengan *Output* 10.

3.1.3.2. Hasil Pengujian

<i>Ukuran Filter Konvolusi</i>	<i>Konfigurasi</i>	<i>Waktu Latih (detik)</i>	<i>F1-Score (Keras)</i>	<i>F1-Score (Manual)</i>
3	[3x3]	60	0.61663	0.61663
5	[5x5]	67.8	0.59988	0.59988
7	[7x7]	78.2	0.5939	0.5939

Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini





3.1.3.3. Analisis dan Kesimpulan

Performa paling bagus ada pada ukuran filter 3x3. Sebenarnya konfigurasi ini lebih mengarah pada data yang dimiliki. Ukuran kecil menandakan detil yang lebih baik serta menangkap fitur kecil yang mungkin bersifat penting , sedangkan ukuran merupakan *trade off* detil dengan konteks gambar.

CIFAR sendiri sebenarnya merupakan dataset yang cukup sederhana. Bahkan ukurannya hanya 32x32 yang setelah dicek secara langsung memiliki berbagai elemen pada foto yang cukup kecil. Ukuran 3x3 hanya kira kira 1/10 dari 32x32, yang mungkin masuk akal untuk konteks kecil seperti telinga hewan , roda , dan lain sebagainya. 5x5 juga mungkin masih bisa dengan sedikit *trade off* tertentu.

Namun, menurut penulis, 7x7 memang sedikit lebih luas. 7x7 berarti hampir 1/5 dari keseluruhan foto. Hanya saja , dataset CIFAR lebih mengandalkan fitur fitur kecil yang menjadi pendukung dalam memprediksi secara keseluruhan.

Dari grafik sebenarnya sudah bisa dilihat dengan mudah bahwa ukuran 7×7 memang lebih tidak optimal, mengingat bukan hanya *loss start* yang tinggi , namun juga kesulitan dalam menurunkan nilai *loss* tersebut. Model dengan filter 5×5 memiliki *trade off* nya , dimana terlihat semakin banyak *epoch*, mulai mengalami *loss* yang tinggi, asumsinya adalah karena kurangnya merekam fitur fitur yang menjadi detail penting.

Kesimpulannya, penentuan ukuran filter harus disesuaikan dengan dataset, apakah harus menangkap detail kecil, atau perlu untuk mengambil konteks dari gambar. Pada percobaan ini , 3×3 terbukti lebih baik dalam menarik pola dari gambar.

3.1.4. Pengaruh jenis pooling layer yang digunakan

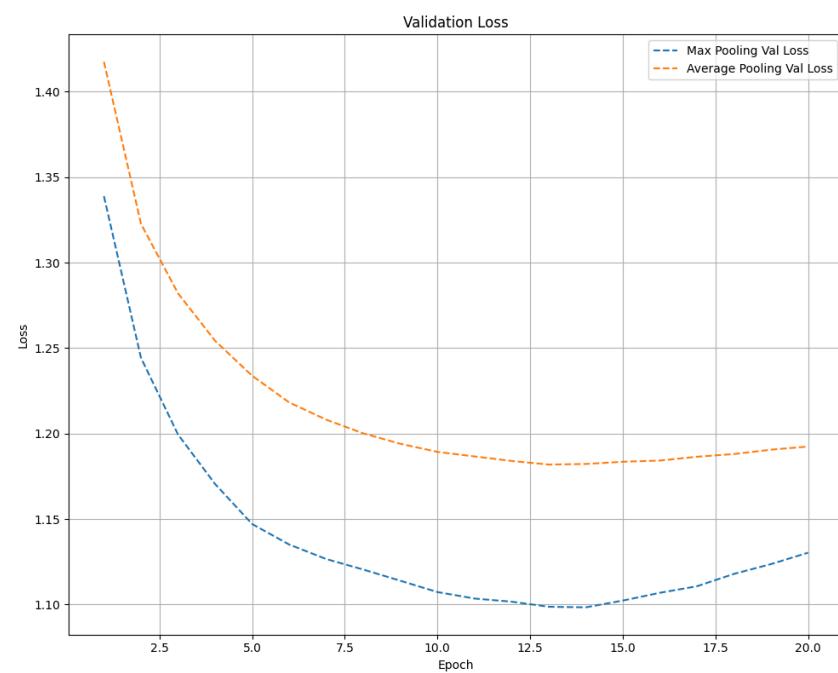
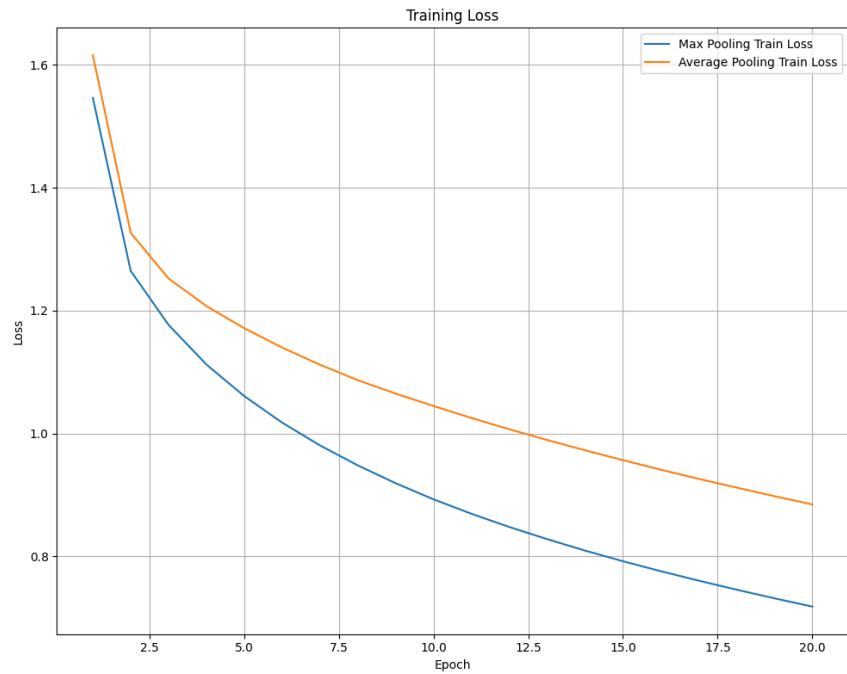
3.1.4.1. Konfigurasi Eksperimen

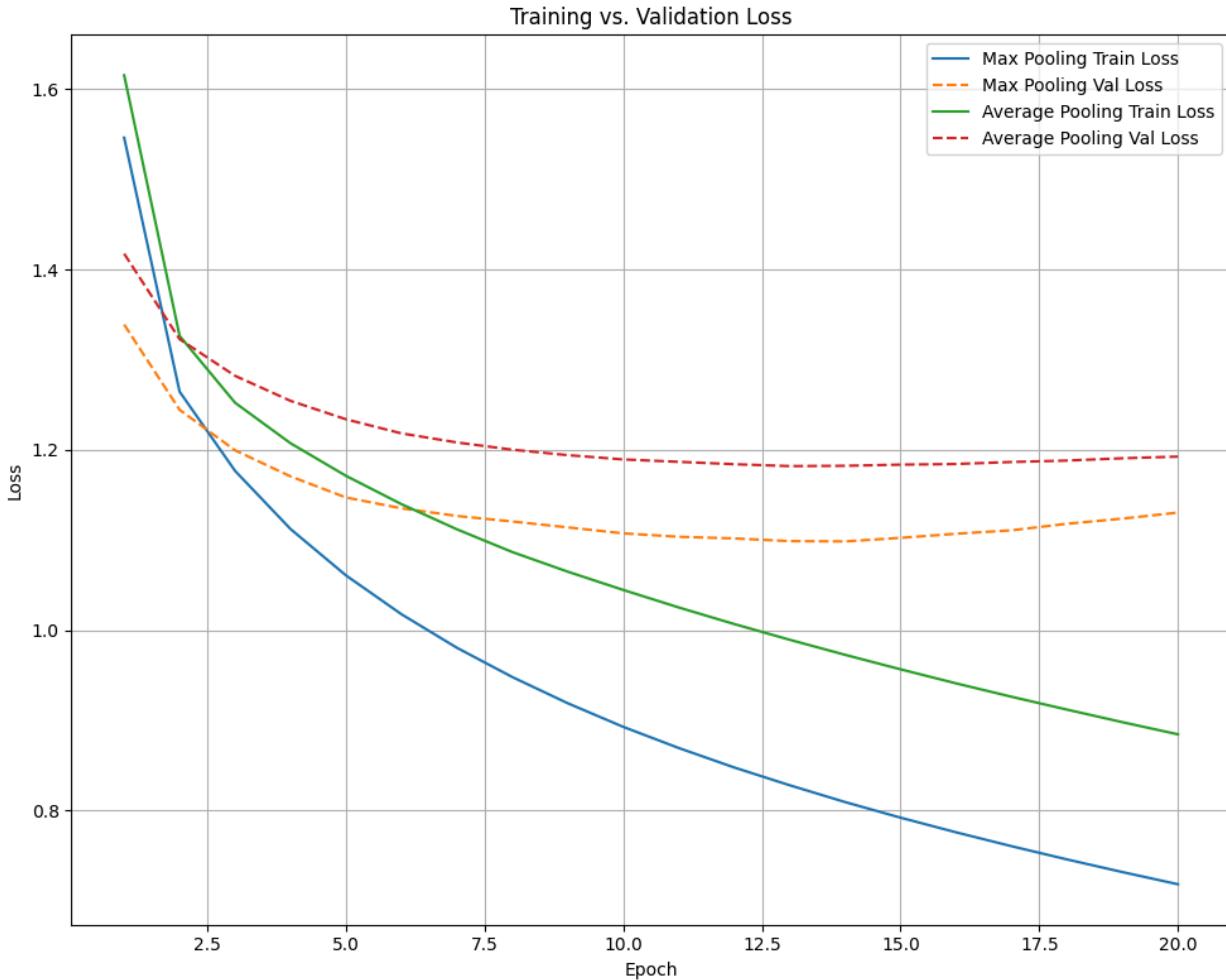
Pengujian kali ini dilakukan selama 20 epoch dengan konfigurasi satu layer konvolusi yang menggunakan 32 filter dan ukuran kernel 3×3 . Pada bagian ini, akan dilakukan berbagai variasi metode pooling untuk mengevaluasi pengaruh jenis pooling yang digunakan terhadap kinerja model, khususnya dalam hal ekstraksi fitur dan generalisasi. Selain itu, pada eksperimen ini ditambahkan 1 *Layer Flatten* dan *Dense Layer* dengan *Output* 10.

3.1.4.2. Hasil Pengujian

Jenis Pooling	Waktu Latih (detik)	F1-Score (Keras)	F1-Score (Manual)
Max Pooling	60	0.61663	0.61663
Average Pooling	60	0.5891	0.5891

Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini





3.1.4.3. Analisis dan Kesimpulan

Max Pooling digunakan untuk memberikan penekanan pada fitur yang penting, sedangkan *average pooling* berfokus pada representasi. Mungkin hal ini hanya berupa asumsi penulis, namun penulis merasa penggunaan *max pooling* cocok untuk CIFAR karena data yang terlalu beragam.

Ragamnya data membuat bahwa pasti akan ada 1 fitur yang menjadi pembeda antara label yang ada. Hal hal seperti telinga, roda, sayap pesawat, dan lain sebagainya tidak hadir pada gambar kelas lainnya.

Tidak ada analisis spesial, bahkan dilihat dari grafik *loss* sekalipun karena memang hal ini tidak berhubungan dengan *overfitting* ataupun permasalahan umum lainnya , melainkan hanya pengguna tipe *pooling* pada dataset yang salah.

Kesimpulannya adalah, *max pooling* menekankan pada satu fitur. Jika dirasa bahwa akan ada suatu fitur yang dapat menjadi kunci perbedaan antar berbagai kelas, maka *max pooling* seharusnya digunakan. Selain kasus tersebut, jika dataset terlalu general, dan pengujian ingin istilahnya “main aman”, dapat menggunakan *average pooling* agar model tidak terpaku pada satu fitur yang bisa saja merupakan mis-interpretasi dari konteks gambar tersebut.

3.2. Pengujian RNN

Pengujian ini bertujuan mengevaluasi performa model Recurrent Neural Network (RNN) dalam klasifikasi sentimen pada teks Bahasa Indonesia menggunakan dataset NusaX-Sentiment. Pengujian dilakukan dengan berbagai variasi konfigurasi model untuk melihat pengaruhnya terhadap hasil klasifikasi. Selain itu, pengujian juga dilakukan untuk memastikan bahwa implementasi forward propagation manual sudah berjalan dengan benar, dengan cara membandingkan hasilnya dengan model dari Keras. Seluruh kode pengujian dapat dilihat di file src/simple_rnn.ipynb.

3.2.1. Pengaruh Jumlah Layer RNN

3.2.1.1. Konfigurasi Eksperimen

Eksperimen ini bertujuan untuk menganalisis pengaruh jumlah layer RNN terhadap performa model dalam melakukan klasifikasi sentimen pada dataset NusaX-Sentiment berbahasa Indonesia. Dataset ini diunduh dari tautan resmi pada spesifikasi tugas dan telah disediakan dalam tiga bagian terpisah: train, validation, dan test, sehingga tidak diperlukan proses pembagian data secara manual.

Seluruh model menggunakan konfigurasi dasar yang sama, yaitu:

- Maksimum kosakata (max_vocab): 10000 token
- Panjang sekuens input (max_len): 100 token
- Embedding: 128 dimensi
- Jumlah unit per layer RNN: 64
- Fungsi aktivasi pada setiap layer RNN: tanh
- Jenis RNN: Bidirectional SimpleRNN
- Dropout: 0.3
- Optimizer: Adam (lr=1e-3)
- Loss function: sparse_categorical_crossentropy
- Dense layer: [32, 3] dengan aktivasi ['relu', 'softmax']
- Epoch: 20
- Batch size: 64



```
1 model = SimpleRNNKeras(  
2     max_vocab=10000,  
3     max_len=100,  
4     rnn_units=config,  
5     embedding_dim=128,  
6     rnn_activations=rnn_activations,  
7     dense_units=[32, 3],  
8     dense_activations=['relu', 'softmax'],  
9     bidirectional=True,  
10    dropout=0.3,  
11    learning_rate=1e-3  
12 )
```

Tiga variasi jumlah layer RNN yang akan diuji adalah:

- 1 Layer
- 2 Layer
- 3 Layer

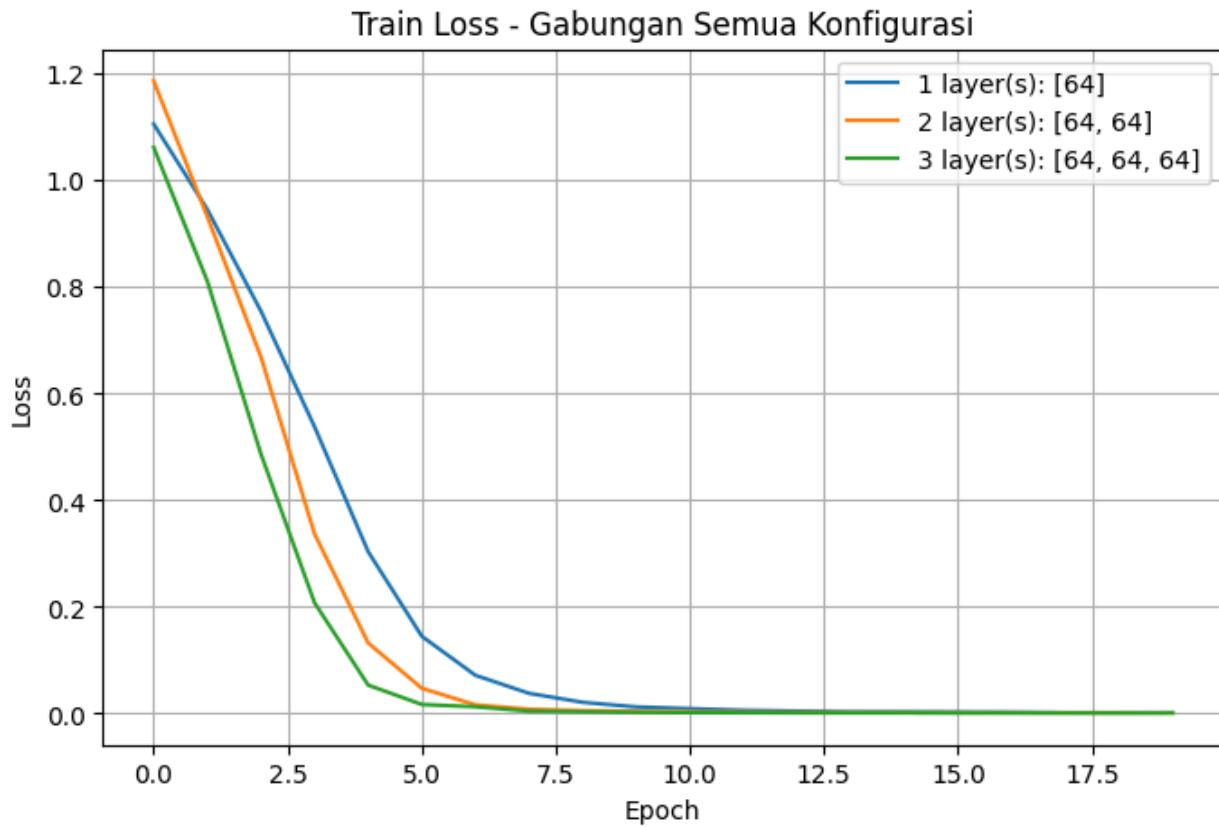
3.2.1.2. Hasil Pengujian

Berikut merupakan hasil evaluasi model dengan berbagai variasi jumlah layer RNN yang telah dilakukan. Evaluasi dilakukan pada data uji dan menggunakan metrik macro F1-score.

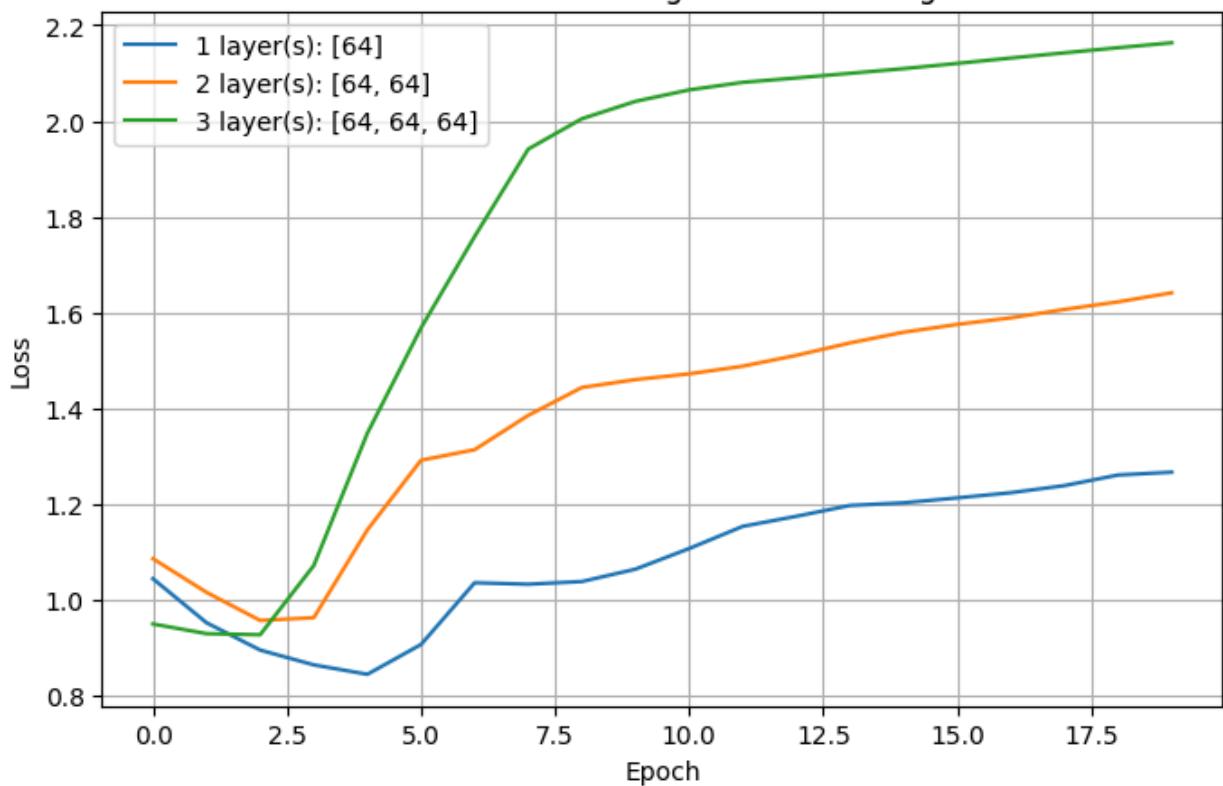
Jumlah Layer	Konfigurasi	Waktu Latih	F1-Score	F1-Score
--------------	-------------	-------------	----------	----------

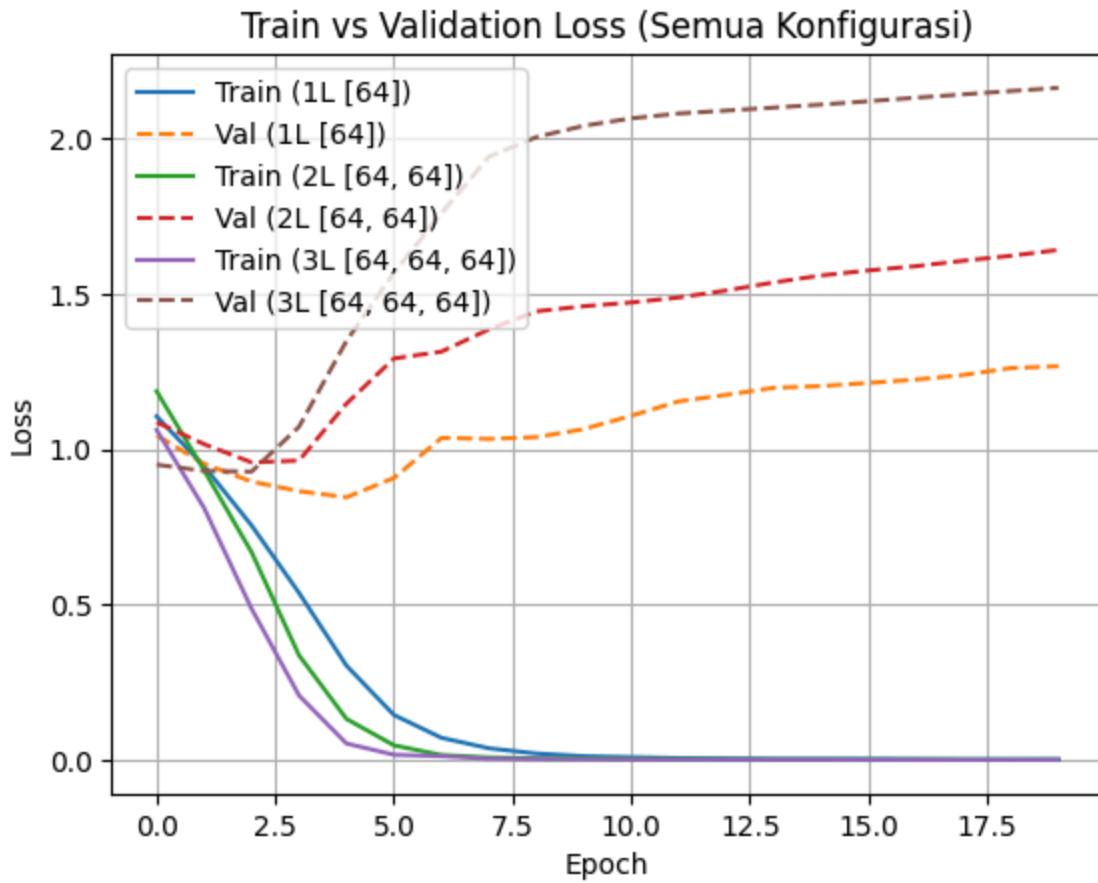
RNN		(detik)	(Keras)	(Manual)
1 layer	[64]	5.83	0.6011	0.6011
2 layer	[64, 64]	10.23	0.4694	0.4694
3 layer	[64, 64, 64]	16.52	0.4928	0.4928

Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini



Validation Loss - Gabungan Semua Konfigurasi





3.2.1.3. Analisis dan Kesimpulan

Dari hasil pengujian, model dengan 1 layer RNN memberikan performa paling baik dengan nilai macro F1-score sebesar 0.6011. Sementara itu, model dengan 2 layer justru mengalami penurunan performa menjadi 0.4694, dan model 3 layer hanya sedikit lebih baik di angka 0.4928. Menariknya, ketika model diuji kembali menggunakan implementasi forward propagation manual, hasil F1-score-nya sama persis. Ini menunjukkan bahwa kode manual yang dibuat sudah berjalan dengan benar dan sesuai dengan model aslinya dari Keras.

Selain dari sisi akurasi, waktu pelatihan juga jadi faktor penting. Semakin banyak layer, waktu yang dibutuhkan juga makin lama sekitar 5.8 detik untuk 1 layer, 10.2 detik untuk 2 layer, dan 16.5 detik untuk 3 layer. Hal ini wajar karena jumlah parameter dan komputasi yang dilakukan juga lebih besar.

Kalau dilihat dari grafik train loss, model dengan 2 dan 3 layer memang bisa menurunkan loss lebih cepat di awal dibandingkan model 1 layer. Hal ini karena arsitektur yang lebih dalam normalnya punya kemampuan lebih besar untuk “menyesuaikan” diri ke data latih. Tapi di sisi lain, model yang terlalu dalam juga lebih rentan mengalami overfitting, seperti yang terlihat di

grafik validation loss, dimana loss validasinya justru meningkat terus setelah beberapa epoch. Jadi, walaupun loss latih turun, performa model di data baru malah memburuk.

Perlu dicatat juga bahwa penurunan performa karena menambah layer itu bukan hal yang pasti. Mungkin saja di kasus lain, jumlah layer yang lebih banyak justru membantu meningkatkan akurasi, apalagi kalau disertai tuning hyperparameter atau regularisasi yang lebih optimal. Tapi di eksperimen ini, hasilnya justru menurun karena model mungkin terlalu kompleks untuk datanya.

Kesimpulannya, model dengan 1 layer RNN jadi pilihan paling seimbang dalam hal akurasi, kecepatan pelatihan, dan kestabilan performa. Tambahan layer justru menambah beban tanpa meningkatkan hasil, dan validasi dari hasil manual menunjukkan bahwa sistem yang dibangun sudah benar.

3.2.2. Pengaruh Jumlah Neuron Pada Layer RNN

3.2.2.1. Konfigurasi Eksperimen

Pada eksperimen ini, dilakukan pengujian terhadap pengaruh jumlah unit (neuron) di dalam layer RNN terhadap performa model klasifikasi sentimen Bahasa Indonesia menggunakan dataset NusaX-Sentiment. Sama seperti eksperimen sebelumnya, dataset dari awalnya sudah disediakan dalam tiga bagian terpisah: train, validation, dan test, sehingga tidak dilakukan pembagian data secara manual.

Model yang digunakan tetap menggunakan arsitektur dasar yang sama seperti sebelumnya, namun dengan satu layer RNN dan jumlah unit yang divariasikan. Detail konfigurasinya adalah sebagai berikut:

- Maksimum kosakata (max_vocab): 10000 token
- Panjang sekuens input (max_len): 100 token
- Embedding: 128 dimensi
- Jumlah layer RNN: 1
- Fungsi aktivasi pada setiap layer RNN: tanh
- Jenis RNN: Bidirectional SimpleRNN
- Dropout: 0.3
- Optimizer: Adam (lr=1e-3)
- Loss function: sparse_categorical_crossentropy
- Dense layer: [32, 3] dengan aktivasi ['relu', 'softmax']
- Epoch: 20
- Batch size: 64



```
1 model = SimpleRNNKeras(  
2     max_vocab=10000,  
3     max_len=100,  
4     rnn_units=config,  
5     embedding_dim=128,  
6     rnn_activations=['tanh'],  
7     dense_units=[32, 3],  
8     dense_activations=['relu', 'softmax'],  
9     bidirectional=True,  
10    dropout=0.3,  
11    learning_rate=1e-3  
12 )
```

Model diuji pada konfigurasi dengan 1 layer RNN dan variasi unit sebesar 64, 128, dan 256. Tujuan dari pengujian ini adalah untuk melihat apakah peningkatan kapasitas neuron dalam satu layer RNN dapat memperbaiki performa model dalam melakukan klasifikasi teks.

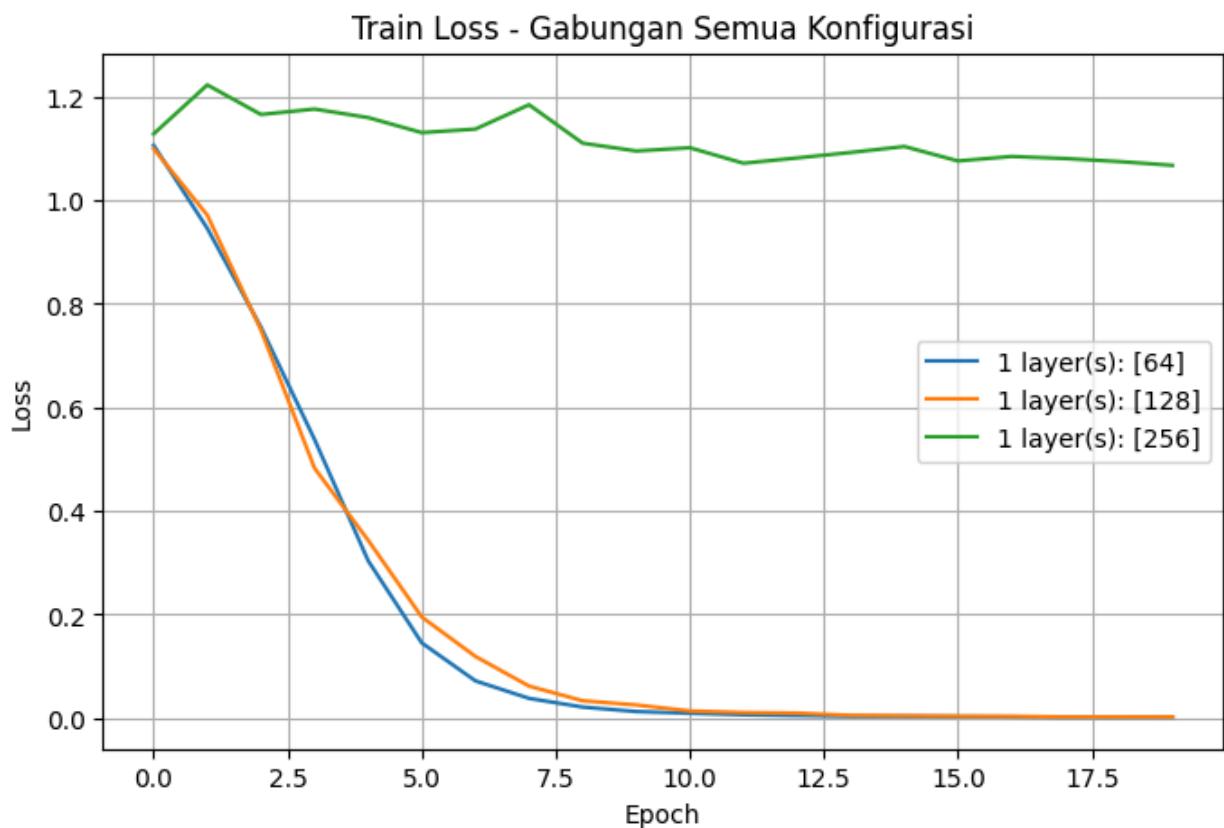
3.2.2.2. Hasil Pengujian

Berikut merupakan hasil evaluasi model dengan berbagai variasi jumlah unit dalam satu layer RNN yang telah dilakukan. Evaluasi dilakukan pada data uji dan menggunakan metrik macro F1-score.

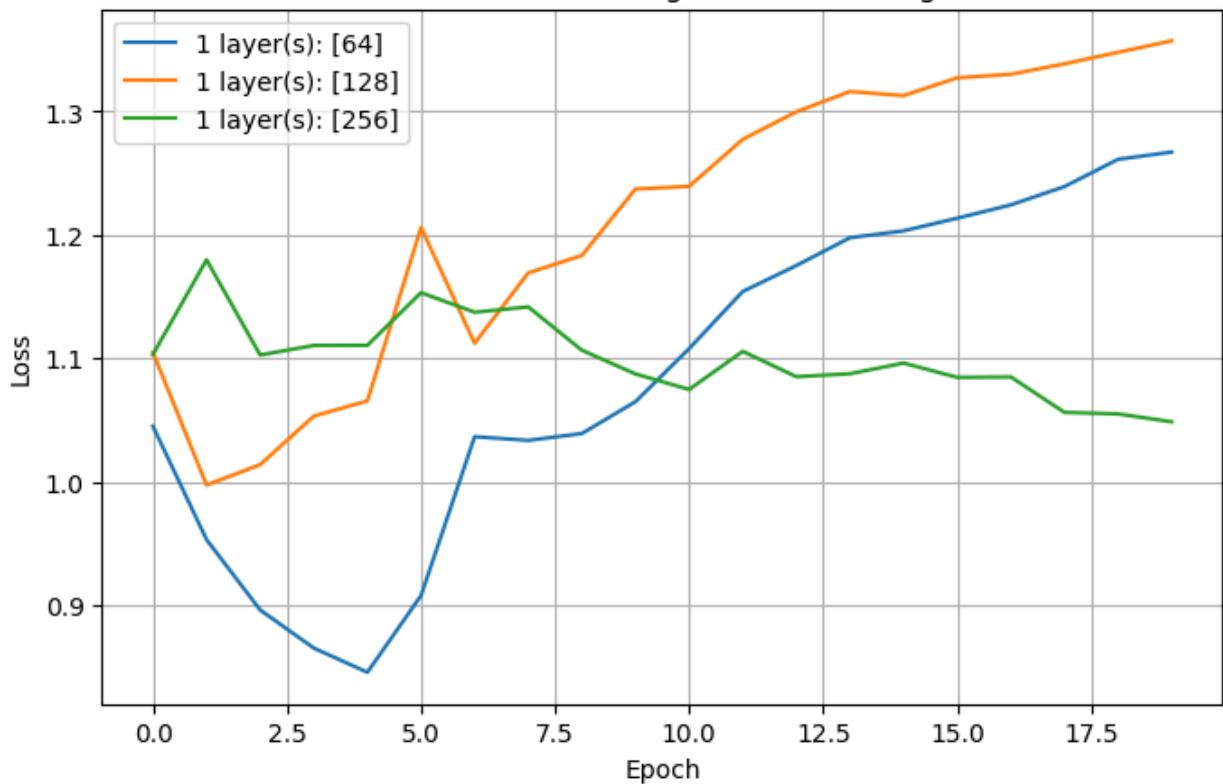
Jumlah Unit	Konfigurasi	Waktu Latih (detik)	F1-Score (Keras)	F1-Score (Manual)
-------------	-------------	---------------------	------------------	-------------------

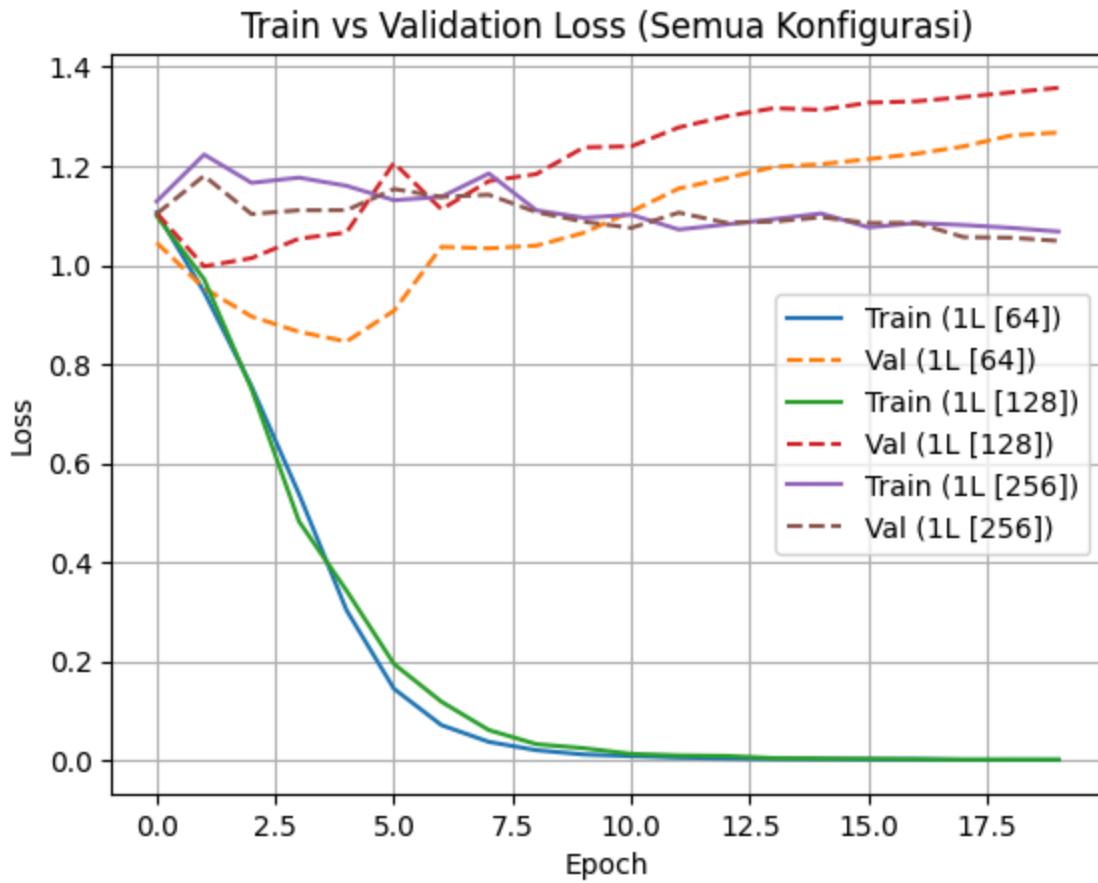
64 unit	[64]	6.44	0.6011	0.6011
128 unit	[128]	7.83	0.4727	0.4727
256 unit	[256]	9.06	0.3208	0.3208

Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini



Validation Loss - Gabungan Semua Konfigurasi





3.2.2.3. Analisis dan Kesimpulan

Pengujian ini dilakukan untuk melihat pengaruh jumlah unit (cell) dalam satu layer RNN terhadap performa model. Hasilnya, model dengan 64 unit memberikan nilai macro F1-score terbaik yaitu 0.6011. Saat jumlah unit ditambah jadi 128, performanya justru menurun menjadi 0.4727, dan ketika dinaikkan lagi jadi 256, nilainya makin turun ke 0.3208. Ini menunjukkan bahwa menambah unit tidak selalu bikin model lebih bagus.

Dari sisi waktu pelatihan, makin besar jumlah unit, makin lama juga proses training-nya. Model 64 unit selesai dalam 6.4 detik, 128 unit butuh 7.8 detik, dan 256 unit butuh 9.0 detik. Waktu ini wajar meningkat karena jumlah parameter yang dihitung juga lebih banyak.

Kalau kita lihat dari grafik train loss, model 64 dan 128 unit sama-sama berhasil menurunkan loss dengan baik selama training. Tapi model 256 unit justru train loss-nya cenderung datar, nggak ada penurunan yang jelas. Ini tanda bahwa model underfitting alias gagal belajar dari data latih, kemungkinan karena terlalu besar dan susah dioptimasi.

Dari grafik validation loss, kelihatan juga kalau model 128 unit lebih overfit dibandingkan 64 unit. Meskipun loss latihnya turun, val loss-nya justru naik terus setelah beberapa epoch.

Sebaliknya, model 256 unit punya val loss yang naik sedikit di awal, lalu perlahan turun. Tapi karena train loss-nya nggak turun, performanya tetap jelek. Jadi bisa dibilang, 256 unit nggak overfit, tapi juga nggak belajar apa-apa.

Selain itu, semua model juga diuji pakai implementasi manual dari forward propagation. Hasil F1-score-nya sama persis dengan model dari Keras, artinya kode manual yang dibuat sudah berjalan dengan benar.

Kesimpulannya, menambah jumlah unit atau cell pada layer RNN memang mempengaruhi kinerja model, tetapi tidak selalu membuat performa model menjadi lebih baik. Dalam pengujian ini, model dengan 64 unit memberikan hasil terbaik karena memiliki keseimbangan antara kemampuan belajar, waktu pelatihan yang efisien, dan kestabilan saat validasi. Saat jumlah unit dinaikkan menjadi 128, model mulai menunjukkan gejala overfitting. Sementara itu, saat jumlah unit dinaikkan lagi menjadi 256, model justru mengalami underfitting karena tidak berhasil menurunkan loss dengan baik. Ini menunjukkan bahwa semakin banyak unit memang menambah kapasitas model, tetapi jika tidak diimbangi dengan pengaturan pelatihan dan regularisasi yang tepat, performanya bisa menurun.

3.2.3. Pengaruh Bidirectional Layer dan Unidirectional Layer Pada RNN

3.2.3.1. Konfigurasi Eksperimen

Pengujian ini bertujuan untuk mengetahui apakah arah propagasi dalam layer RNN, yaitu bidirectional atau unidirectional, berpengaruh terhadap performa model klasifikasi sentimen. Dataset yang digunakan tetap sama, yaitu NusaX-Sentiment berbahasa Indonesia, yang telah dibagi ke dalam tiga bagian: train, validation, dan test, sehingga tidak dilakukan pembagian ulang.

Arsitektur model yang digunakan tetap sederhana dan konsisten, hanya arah layer RNN yang divariasikan. Detail konfigurasi model adalah sebagai berikut:

- Maksimum kosakata (max_vocab): 10000 token
- Panjang sekuens input (max_len): 100 token
- Embedding: 128 dimensi
- Jumlah layer RNN: 1
- Jumlah unit pada layer RNN: 64
- Fungsi aktivasi pada setiap layer RNN: tanh
- Dropout: 0.3
- Optimizer: Adam (lr=1e-3)
- Loss function: sparse_categorical_crossentropy
- Dense layer: [32, 3] dengan aktivasi ['relu', 'softmax']
- Epoch: 20
- Batch size: 64



```
1 model = SimpleRNNKeras(  
2     max_vocab=10000,  
3     max_len=100,  
4     rnn_units=[64],  
5     embedding_dim=128,  
6     rnn_activations=['tanh'],  
7     dense_units=[32, 3],  
8     dense_activations=['relu', 'softmax'],  
9     bidirectional=config,  
10    dropout=0.3,  
11    learning_rate=1e-3  
12 )
```

Pengujian dilakukan dengan hanya mengubah arah layer RNN, sementara semua parameter lainnya dijaga tetap sama. Tujuannya adalah untuk mengevaluasi sejauh mana bidirectional RNN dapat memberikan keuntungan dibandingkan dengan unidirectional dalam konteks pemrosesan teks Bahasa Indonesia.

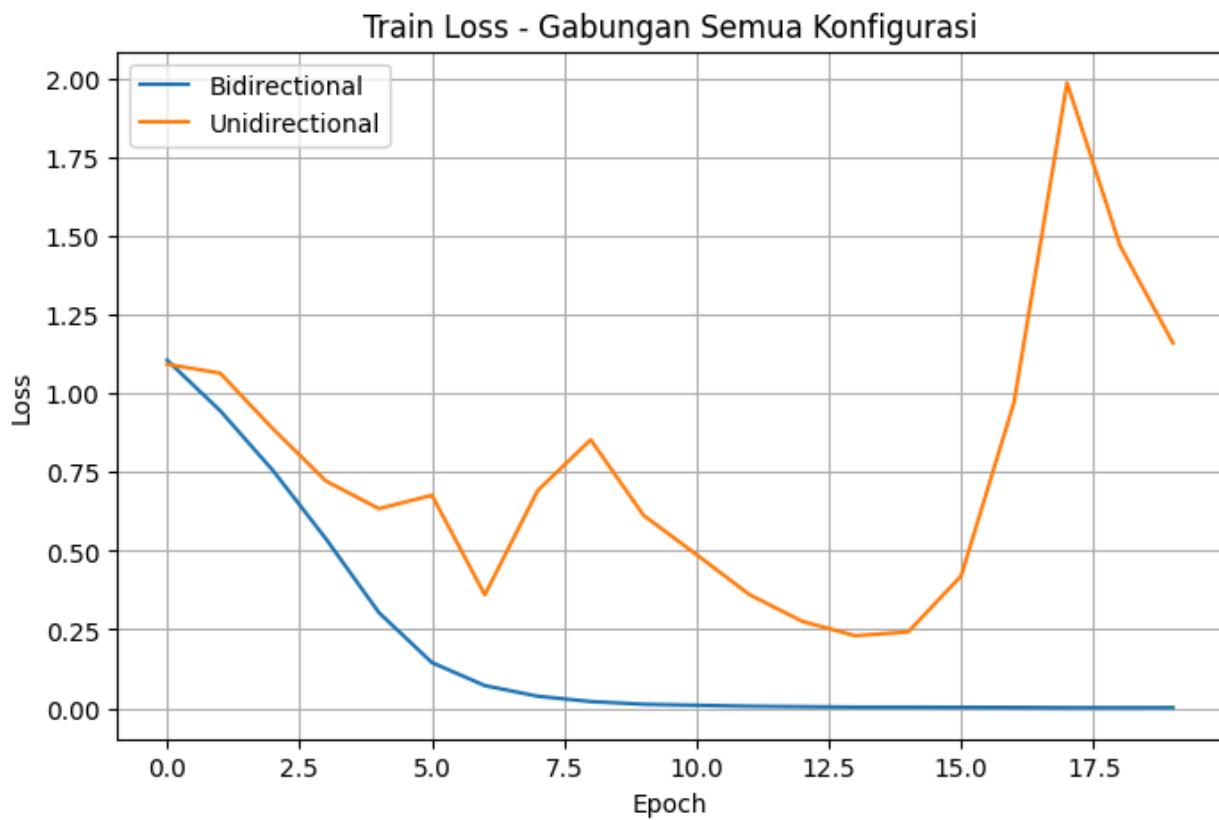
3.2.3.2. Hasil Pengujian

Pengujian dilakukan pada dua konfigurasi arah layer RNN, yaitu bidirectional dan unidirectional, dengan parameter lain dijaga tetap sama. Hasil evaluasi ditampilkan dalam tabel berikut:

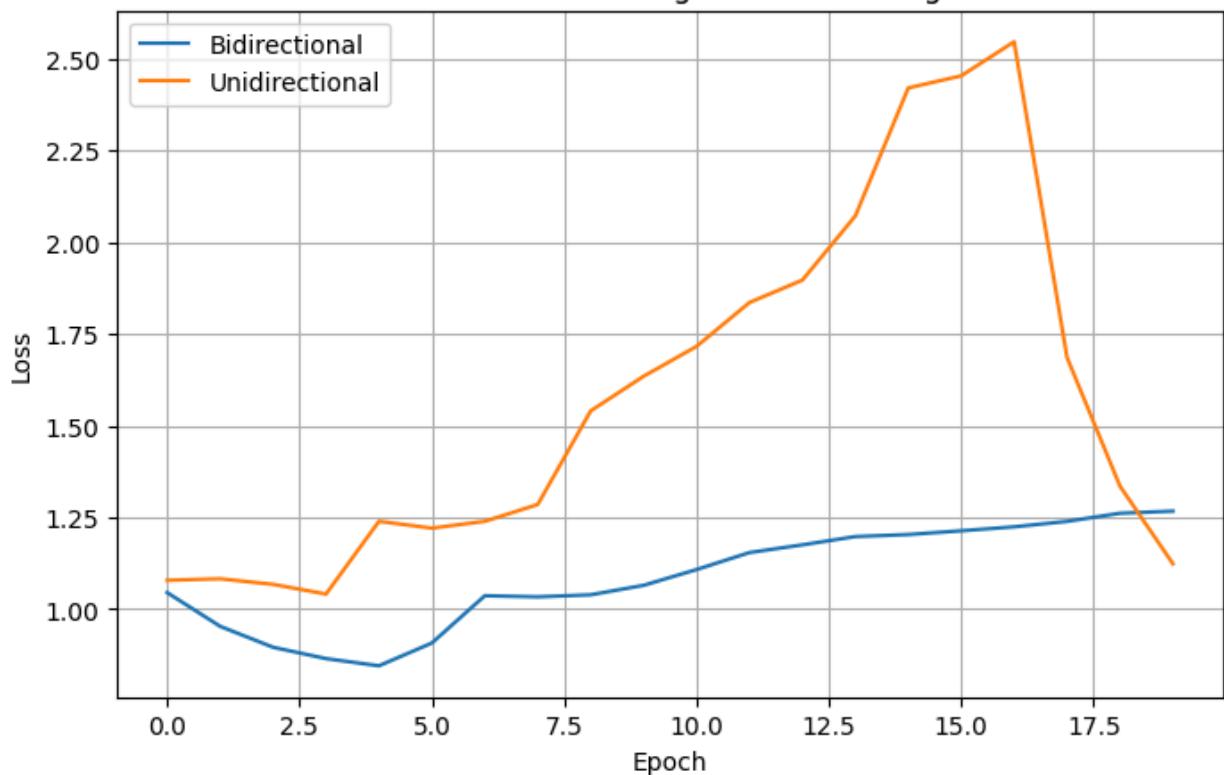
Arah RNN	Waktu Latih (detik)	F1-Score (Keras)	F1-Score (Manual)
Bidirectional	7.40	0.6011	0.6011

Unidirectional	4.76	0.2917	0.2917
----------------	------	--------	--------

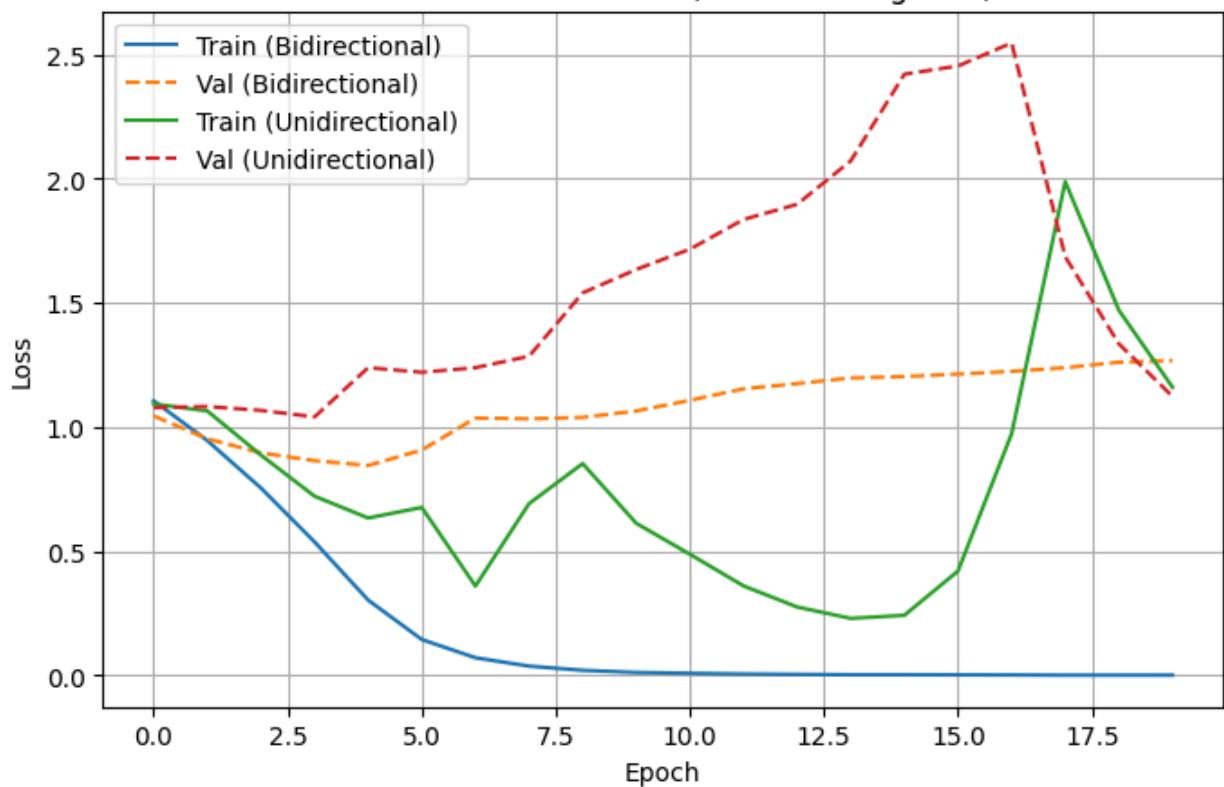
Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini



Validation Loss - Gabungan Semua Konfigurasi



Train vs Validation Loss (Semua Konfigurasi)



3.2.3.3. Analisis dan Kesimpulan

Pengujian ini menunjukkan bahwa arah propagasi pada layer RNN memiliki pengaruh besar terhadap performa model. Ketika menggunakan arah bidirectional, model menghasilkan macro F1-score sebesar 0.6011, sedangkan pada arah unidirectional, performanya jauh menurun menjadi 0.2917. Seluruh parameter lainnya seperti jumlah unit, fungsi aktivasi, dan konfigurasi pelatihan dijaga tetap sama agar hasilnya bisa dibandingkan secara adil.

Dari segi waktu pelatihan, model bidirectional memerlukan waktu sekitar 7.4 detik, lebih lama dibandingkan model unidirectional yang hanya memerlukan 4.7 detik. Perbedaan ini terjadi karena bidirectional RNN memproses input dari dua arah, sehingga jumlah bobot yang harus dilatih menjadi dua kali lebih banyak dibandingkan unidirectional.

Grafik *train loss* menunjukkan bahwa model bidirectional belajar dengan stabil. Loss-nya turun secara konsisten dari awal hingga akhir pelatihan. Sebaliknya, model unidirectional mengalami penurunan yang tidak stabil. Setelah sempat turun, train loss-nya malah naik drastis menjelang akhir epoch, yang menunjukkan bahwa proses belajarnya tidak berjalan lancar.

Kalau dilihat dari validation loss, model bidirectional juga lebih stabil. Memang ada sedikit kenaikan, tapi pelan dan tidak ekstrem. Sebaliknya, model unidirectional mengalami val loss yang terus naik selama pelatihan, lalu tiba-tiba turun di akhir. Penurunan ini kelihatan bagus, tapi terlalu terlambat dan tidak cukup untuk menyaingi performa model bidirectional secara keseluruhan.

Bidirectional bisa memberikan hasil yang lebih baik karena dalam Bahasa Indonesia, informasi penting dalam kalimat bisa muncul di berbagai posisi, baik di awal, tengah, maupun akhir kalimat. Model unidirectional hanya membaca dari kiri ke kanan sehingga ada kemungkinan melewatkannya konteks yang muncul setelahnya. Sebaliknya, model bidirectional membaca dari dua arah, sehingga dapat memahami hubungan antar kata secara lebih menyeluruh dan menangkap arti kalimat dengan lebih baik.

Selain itu, hasil evaluasi dari implementasi manual forward propagation juga menunjukkan F1-score yang sama persis dengan hasil model Keras. Ini membuktikan bahwa kode manual yang dibuat sudah sesuai dan bekerja dengan baik.

Kesimpulannya, arah propagasi dalam RNN sangat memengaruhi hasil pelatihan dan akurasi model. Bidirectional RNN memberikan hasil yang lebih stabil dan akurat karena mampu menangkap konteks dari dua arah. Walaupun waktu pelatihannya sedikit lebih lama karena jumlah bobot yang lebih banyak, hasil akhirnya jauh lebih baik. Untuk kasus klasifikasi sentimen dalam Bahasa Indonesia, bidirectional RNN terbukti jauh lebih efektif dibandingkan unidirectional.

3.3. Pengujian LSTM

3.3.1. Pengaruh Jumlah Layer LSTM

3.3.1.1. Konfigurasi Eksperimen

Eksperimen ini bertujuan untuk menganalisis pengaruh jumlah layer LSTM terhadap performa model dalam melakukan klasifikasi sentimen pada dataset NusaX-Sentiment berbahasa Indonesia. Dataset ini telah disediakan dalam tiga bagian terpisah: train, validation, dan test, sehingga tidak diperlukan proses pembagian data secara manual.

Seluruh model menggunakan konfigurasi dasar yang sama:

- Maksimum kosakata (max_vocab): 10000 token
- Panjang sekuens input (max_len): 100 token
- Embedding: 128 dimensi
- Jumlah unit per layer LSTM: 64
- Fungsi aktivasi pada setiap layer LSTM: tanh
- Jenis LSTM: Bidirectional LSTM
- Dropout: 0.5
- Optimizer: Adam (lr=1e-3)
- Loss function: sparse_categorical_crossentropy
- Dense layer: dengan aktivasi 'softmax'
- Epoch: 10
- Batch size: 32

```

1 max_tokens = 10000
2 max_len    = 100
3 embed_dim  = 128
4
5 vectorizer = TextVectorization(max_tokens=max_tokens, output_sequence_length=max_len)
6 vectorizer.adapt(texts_train)
7
8 def make_dataset(texts, labels, batch=32, shuffle=True):
9     x = vectorizer(tf.constant(texts))
10    ds = tf.data.Dataset.from_tensor_slices((x, labels))
11    if shuffle: ds = ds.shuffle(1024)
12    return ds.batch(batch).prefetch(1)
13
14 print("Sample vocab:", vectorizer.get_vocabulary()[:10])
15 print("Max index:", len(vectorizer.get_vocabulary()))
16
17 ds_train = make_dataset(texts_train, y_train)
18 ds_val   = make_dataset(texts_val,   y_val,   shuffle=False)
19 ds_test  = make_dataset(texts_test,  y_test,  shuffle=False)
20
21 def train_and_eval(params, name):
22     print(f"\n==== Running {name}: {params}")
23     model = build_lstm_model(**params, max_len=max_len, max_tokens=max_tokens, embed_dim=embed_dim, num_classes=num_classes)
24     hist = model.fit(ds_train, validation_data=ds_val, epochs=10,
25                       shuffle=False)
26     y_pred = np.argmax(model.predict(ds_test), axis=-1)
27     f1 = f1_score(y_test, y_pred, average='macro')
28     print(f"\n{name} macro-F1: {f1:.4f}")
29     return model, hist, f1
30
31 for n in [1, 2, 3]:
32     print(f"\n===== [LSTM] Jumlah Layer = {n} =====")
33     config = {
34         'n_layers': n,
35         'units': [64] * n,
36         'bidirectional': True
37     }
38     start = time.time()
39     model, hist, f1 = train_and_eval(config, f'layers={n}')
40     end = time.time()
41
42     label = f'{n} Layer'
43     trainloss_dict[label] = hist.history['loss']
44     valloss_dict[label] = hist.history['val_loss']
45     durations.append((label, round(end - start, 2)))
46     exp1.append((label, hist))

```

Tiga variasi jumlah layer LSTM yang diuji:

1. 1 Layer
2. 2 Layer
3. 3 Layer

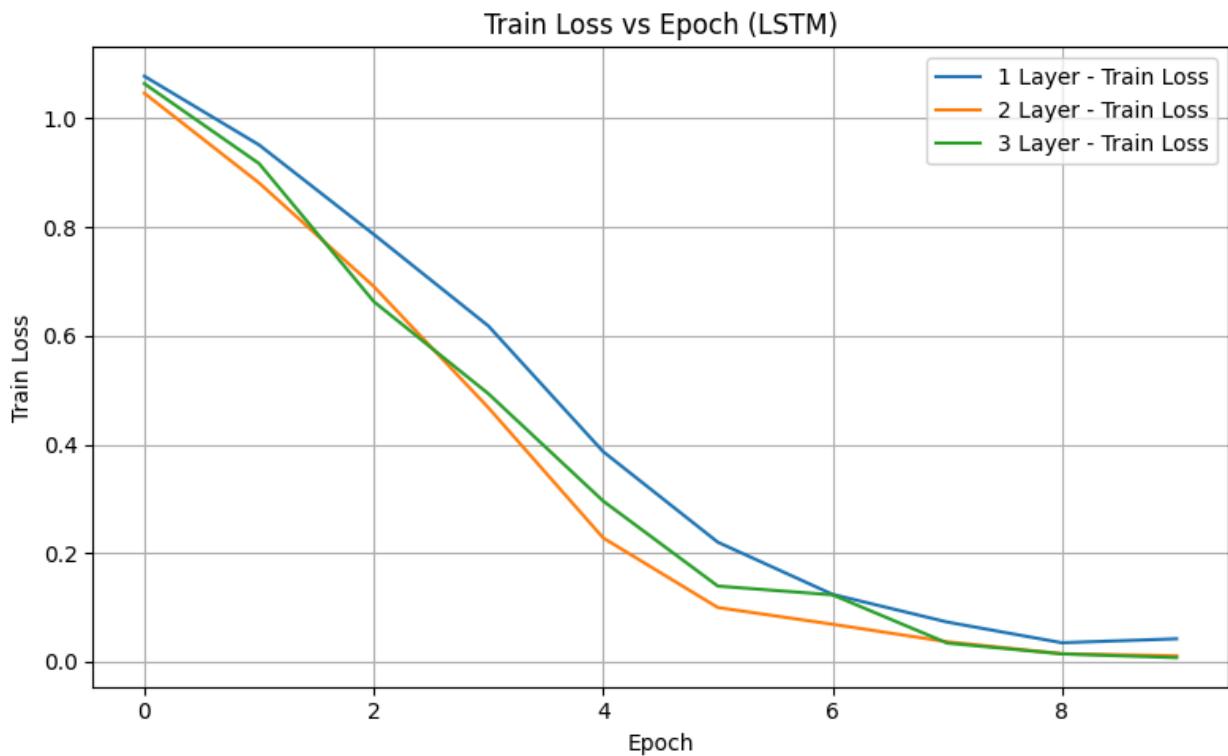
3.3.1.2. Hasil Pengujian

Berikut merupakan hasil evaluasi model dengan berbagai variasi jumlah layer LSTM yang telah dilakukan. Evaluasi dilakukan pada data uji dan menggunakan metrik macro F1-score.

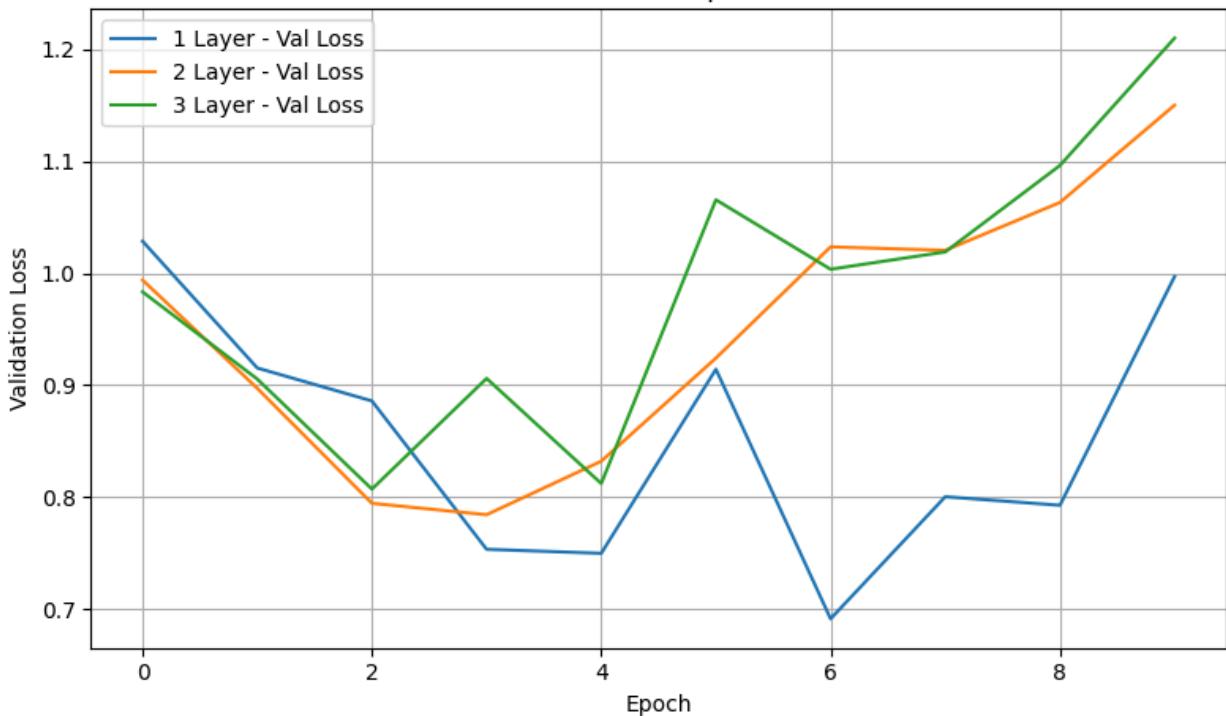
<i>Jumlah Layer RNN</i>	<i>Konfigurasi</i>	<i>Waktu Latih (detik)</i>	<i>F1-Score (Keras)</i>	<i>F1-Score (Manual)</i>
1 layer	[64]	3.90	0.6376	0.6376

2 layer	[64, 64]	7.17	0.7560	0.7560
3 layer	[64, 64, 64]	10.93	0.7317	0.7317

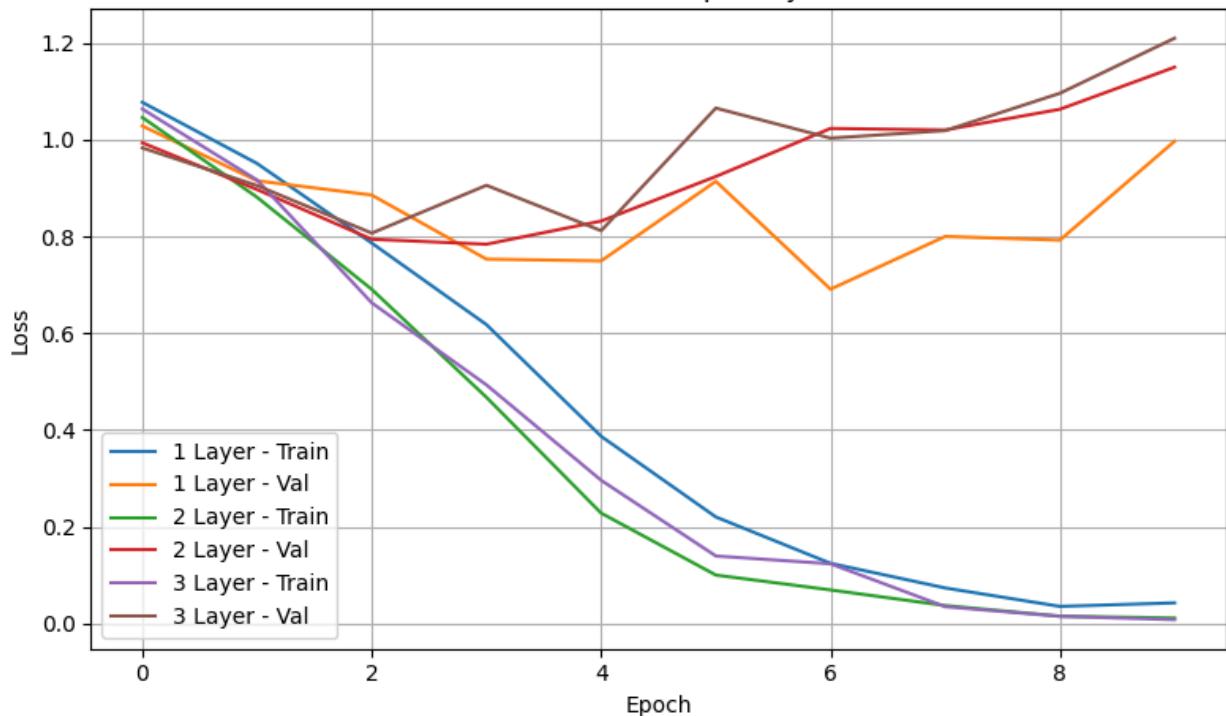
Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini



Validation Loss vs Epoch (LSTM)



Train vs Validation Loss per Layer (LSTM)



3.3.1.3. Analisis dan Kesimpulan

Berdasarkan grafik *train vs validation loss per layer* (LSTM), terlihat perbedaan yang cukup jelas antara performa model LSTM dengan 1, 2, dan 3 layer. Model dengan 1 layer menunjukkan

tren penurunan training loss yang konsisten dan stabil, sementara *validation loss*-nya juga relatif rendah dibandingkan dengan model lain hingga akhir *epoch*, walaupun tetap sedikit fluktuatif. Ini menandakan bahwa model dengan 1 layer memiliki kemampuan generalisasi yang cukup baik terhadap data validasi dan tidak menunjukkan gejala overfitting yang berat.

Pada model 2 layer, performa training loss bahkan lebih cepat turun dibandingkan model 1 layer, yang menandakan kapasitas model meningkat dan lebih mampu menangkap pola dalam data training. Validation loss juga menunjukkan performa yang lebih stabil pada awalnya, namun mulai sedikit fluktuatif setelah epoch ke-5. Meskipun begitu, fluktuasinya masih wajar dan model ini masih dapat dianggap sebagai kompromi yang baik antara kapasitas dan generalisasi.

Sementara itu, model dengan 3 layer menunjukkan performa yang berbeda. Meskipun training loss turun dengan cepat dan mencapai nilai sangat rendah (mendekati nol), *validation loss* justru naik secara bertahap setelah epoch ke-2 hingga akhir. Ini adalah indikasi kuat terjadinya *overfitting*, di mana model terlalu menyesuaikan diri pada data training tetapi gagal mengenali pola general pada data validasi.

Secara keseluruhan, model 2 layer memberikan hasil paling seimbang antara akurasi pelatihan dan kemampuan generalisasi. Model 1 layer cukup stabil dan aman dari overfitting, cocok untuk tugas sederhana, namun secara keseluruhan penulis merasa performanya masih terlalu tanggung. Sementara model 3 layer perlu diwaspadai, karena meskipun sangat akurat di training set, ia gagal mempertahankan performa di validation set, menandakan bahwa model terlalu kompleks atau menandakan indikasi *overfitting*.

3.3.2. Pengaruh Jumlah Neuron Pada Layer LSTM

3.3.2.1. Konfigurasi Eksperimen

Pada eksperimen ini, dilakukan pengujian terhadap pengaruh jumlah unit (neuron) di dalam layer LSTM terhadap performa model klasifikasi sentimen Bahasa Indonesia menggunakan dataset NusaX-Sentiment. Sama seperti eksperimen sebelumnya, dataset dari awalnya sudah disediakan dalam tiga bagian terpisah: train, validation, dan test, sehingga tidak dilakukan pembagian data secara manual.

Model yang digunakan tetap menggunakan arsitektur dasar yang sama seperti sebelumnya, namun dengan satu layer LSTM dan jumlah unit yang divariasikan. Detail konfigurasinya adalah sebagai berikut:

- Maksimum kosakata (max_vocab): 10000 token
- Panjang sekuens input (max_len): 100 token
- Embedding: 128 dimensi
- Jumlah layer LSTM: 1
- Fungsi aktivasi pada setiap layer LSTM: tanh
- Jenis LSTM: Bidirectional LSTM

- Dropout: 0.5
- Optimizer: Adam (lr=1e-3)
- Loss function: sparse_categorical_crossentropy
- Dense layer: dengan aktivasi 'softmax'
- Epoch: 10
- Batch size: 32

```

● ● ●
1 max_tokens = 10000
2 max_len    = 100
3 embed_dim  = 128
4
5 vectorizer = TextVectorization(max_tokens=max_tokens, output_sequence_length=max_len)
6 vectorizer.adapt(texts_train)
7 def make_dataset(texts, labels, batch=32, shuffle=True):
8     x = vectorizer(tf.constant(texts))
9     ds = tf.data.Dataset.from_tensor_slices((x, labels))
10    if shuffle: ds = ds.shuffle(1024)
11    return ds.batch(batch).prefetch(1)
12
13 print("Sample vocab:", vectorizer.get_vocabulary()[:10])
14 print("Max index:", len(vectorizer.get_vocabulary()))
15
16 ds_train = make_dataset(texts_train, y_train)
17 ds_val   = make_dataset(texts_val,   y_val,   shuffle=False)
18 ds_test  = make_dataset(texts_test,  y_test,  shuffle=False)
19
20 def train_and_eval(params, name):
21     print(f"\n==== Running {name}: {params}")
22     model = build_lstm_model(*params, max_len=max_len, max_tokens=max_tokens, embed_dim=embed_dim, num_classes=num_classes)
23     hist = model.fit(ds_train, validation_data=ds_val, epochs=10,
24                       shuffle=False)
25     y_pred = np.argmax(model.predict(ds_test), axis=-1)
26     f1 = f1_score(y_test, y_pred, average='macro')
27     print(f'{name} macro-F1: {f1:.4f}')
28     return model, hist, f1
29
30 for u in [64, 128, 256]:
31     print(f"\n==== [LSTM] Jumlah Unit = {u} ====")
32     config = {
33         'n_layers': 1,
34         'units': [u],
35         'bidirectional': True
36     }
37     start = time.time()
38     model, hist, f1 = train_and_eval(config, f'units={u}')
39     end = time.time()
40
41     label = f'{u} Units'
42     trainloss_dict[label] = hist.history['loss']
43     valloss_dict[label] = hist.history['val_loss']
44     durations.append((label, round(end - start, 2)))
45     exp2.append((label, hist))

```

Tiga variasi jumlah unit LSTM yang diuji:

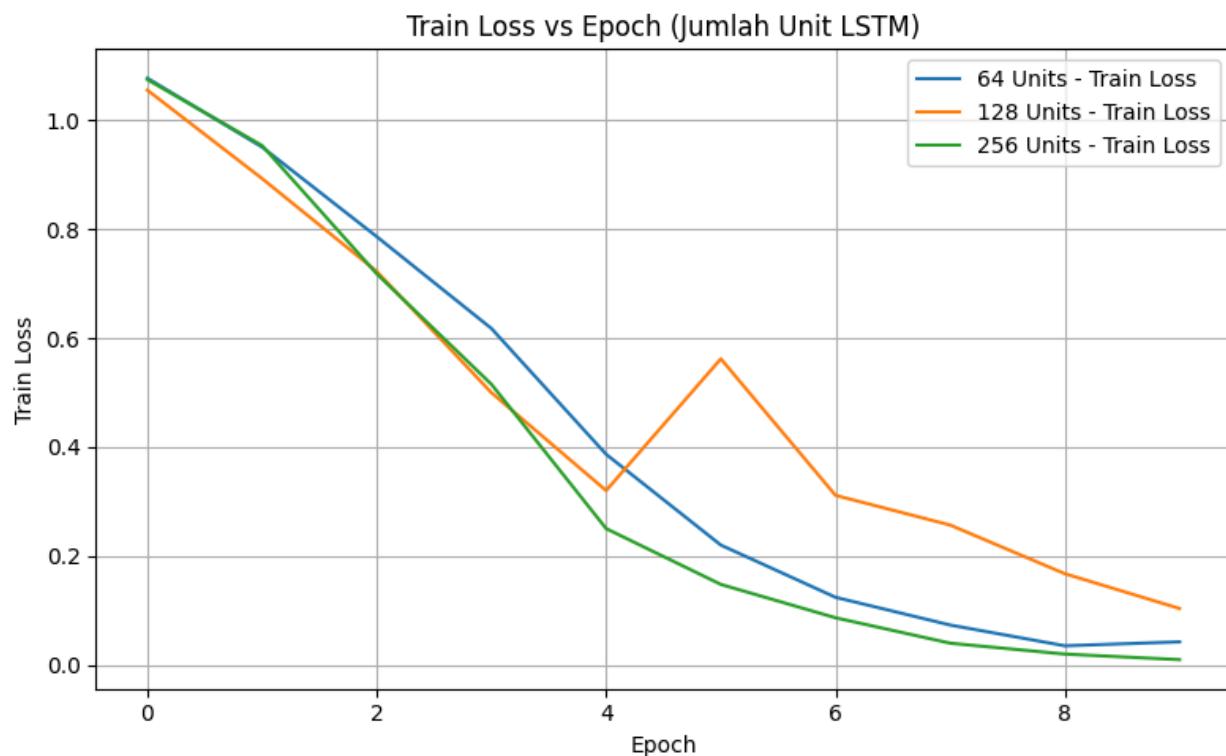
1. 64 unit
2. 128 unit
3. 256 unit

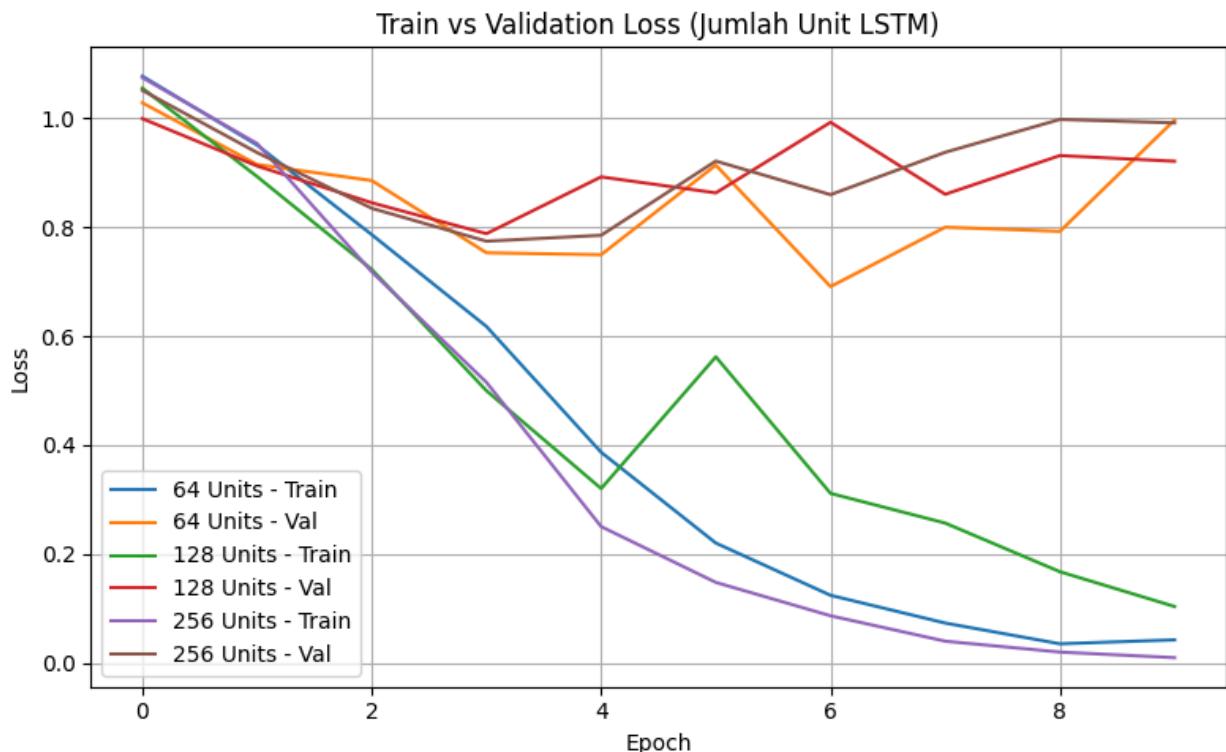
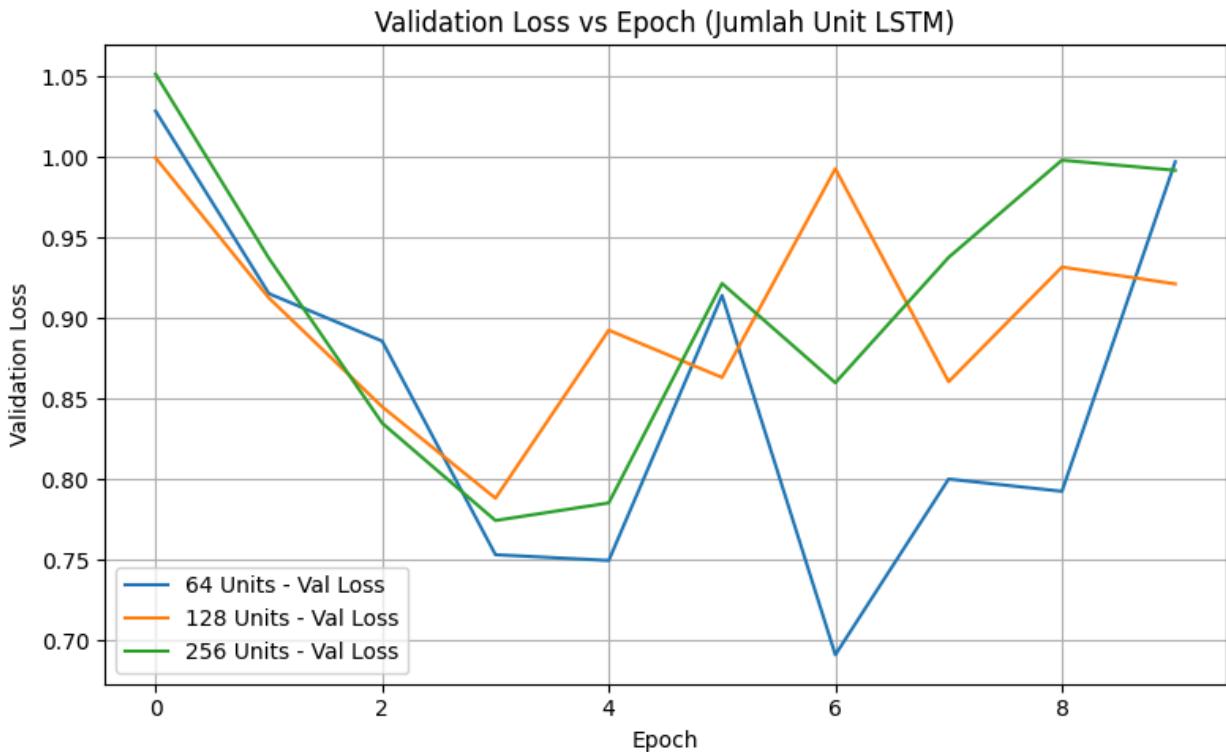
3.3.2.2. Hasil Pengujian

Berikut merupakan hasil evaluasi model dengan berbagai variasi jumlah unit dalam satu layer LSTM yang telah dilakukan. Evaluasi dilakukan pada data uji dan menggunakan metrik macro F1-score.

Jumlah Unit	Konfigurasi	Waktu Latih (detik)	F1-Score (Keras)	F1-Score (Manual)
64 unit	[64]	4.29	0.6376	0.6376
128 unit	[128]	6.60	0.6856	0.6856
256 unit	[256]	12.75	0.7419	0.7419

Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini





3.3.2.3. Analisis dan Kesimpulan

Penulis mengambil pandangan yang berbeda dengan implementasi pada CNN. Penulis yakin , bahwa perbedaan konfigurasi merupakan sebuah *trade off* antara hasil prediksi dengan

konsistensi (*overfitting*). Dari hasil prediksi, semakin banyak unit memberikan hasil yang lebih baik. Namun apakah selalu seperti itu? Atau apakah hal ini terjadi akibat penggunaan iterasi yang dapat dibilang relatif lebih rendah?

Penulis merasa bahwa dari analisis grafik diatas, semakin membuktikan asumsi bahwa harus ada keseimbangan antara jumlah iterasi dengan kompleksitas model. Dari *plot train loss*, performnya terbilang cukup baik dengan adanya fluktuasi pada iterasi kelima yang bisa dibilang berupa *miss* pada saat iterasi, dan penulis yakin dapat terjadi pada model apapun.

Dataset yang digunakan , menurut penulis, termasuk dalam jenjang menengah, dimana beberapa bersifat sangat panjang dan beberapa bersifat pendek. Hal ini mungkin memberikan ambiguitas untuk model 128 units yang dalam hal ini kita sebut “menengah”.

Jika kita melihat kuantitas, data validasi sangat sedikit , sehingga hal ini menjadi sebuah pengaruh dalam menganalisa kompleksitas. Namun, meskipun begitu, terdapat lumayan banyak *long text*, yang mungkin cocok untuk model dengan kompleksitas tinggi seperti model dengan 256 unit.

Meskipun begitu, penulis yakin bahwa dalam eksperimen kali ini, memang lebih baik digunakan kompleksitas yang lebih tinggi dari sisi unit, mengingat analisis terhadap dataset tersebut lebih mempertimbangkan konteks pernyataan.

Kesimpulannya adalah , perlu adanya penyesuaian antara penggunaan unit dengan kebutuhan analisis. Selain itu perlu diingat mengenai resiko *overfitting* dimana terlihat bahwa pada kasus ini , jumlah layer hanya 1 dan iterasi lebih rendah sehingga mungkin saja untuk percobaan kali ini gejala *overfitting* belum muncul.

3.3.3. Pengaruh Bidirectional Layer dan Unidirectional Layer Pada LSTM

3.3.3.1. Konfigurasi Eksperimen

Pengujian ini bertujuan untuk mengetahui apakah arah propagasi dalam layer LSTM, yaitu bidirectional atau unidirectional, berpengaruh terhadap performa model klasifikasi sentimen. Dataset yang digunakan tetap sama, yaitu NusaX-Sentiment berbahasa Indonesia, yang telah dibagi ke dalam tiga bagian: train, validation, dan test, sehingga tidak dilakukan pembagian ulang.

Arsitektur model yang digunakan tetap sederhana dan konsisten, hanya arah layer LSTM yang divariasikan. Detail konfigurasi model adalah sebagai berikut:

- Maksimum kosakata (max_vocab): 10000 token
- Panjang sekuen input (max_len): 100 token

- Embedding: 128 dimensi
- Jumlah layer LSTM: 1
- Jumlah unit pada layer LSTM: 64
- Fungsi aktivasi pada setiap layer LSTM: tanh
- Dropout: 0.5
- Optimizer: Adam (lr=1e-3)
- Loss function: sparse_categorical_crossentropy
- Dense layer: dengan aktivasi 'softmax'
- Epoch: 10
- Batch size: 32

```

● ● ●

1 max_tokens = 10000
2 max_len    = 100
3 embed_dim  = 128
4
5 vectorizer = TextVectorization(max_tokens=max_tokens, output_sequence_length=max_len)
6 vectorizer.adapt(texts_train)
7
8 def make_dataset(texts, labels, batch=32, shuffle=True):
9     x = vectorizer(tf.constant(texts))
10    ds = tf.data.Dataset.from_tensor_slices((x, labels))
11    if shuffle: ds = ds.shuffle(1024)
12    return ds.batch(batch).prefetch(1)
13
14 print("Sample vocab:", vectorizer.get_vocabulary()[:10])
15 print("Max index:", len(vectorizer.get_vocabulary()))
16
17 ds_train = make_dataset(texts_train, y_train)
18 ds_val   = make_dataset(texts_val, y_val,   shuffle=False)
19 ds_test  = make_dataset(texts_test, y_test,  shuffle=False)
20 def train_and_eval(params, name):
21     print("\n==== Running {name}: {params}")
22     model = build_lstm_model(**params, max_len=max_len, max_tokens=max_tokens, embed_dim=embed_dim, num_classes=num_classes)
23     hist = model.fit(ds_train, validation_data=ds_val, epochs=10,
24                       shuffle=False)
25     y_pred = np.argmax(model.predict(ds_test), axis=-1)
26     f1 = f1_score(y_test, y_pred, average='macro')
27     print(f"\n{name} macro-F1: {f1:.4f}")
28     return model, hist, f1
29
30 for b in [False, True]:
31     arah = "Bidirectional" if b else "Unidirectional"
32     print(f"\n===== [LSTM] Arah = {arah} =====")
33     config = {
34         'n_layers': 1,
35         'units': [64],
36         'bidirectional': b
37     }
38     start = time.time()
39     model, hist, f1_keras = train_and_eval(config, arah)
40     end = time.time()
41
42     label = f"\n{arah}"
43     trainloss_dict[label] = hist.history['loss']
44     valloss_dict[label] = hist.history['val_loss']
45     durations.append((label, round(end - start, 2)))
46     exp3.append((label, hist))

```

Dua variasi arah propagasi LSTM yang diuji:

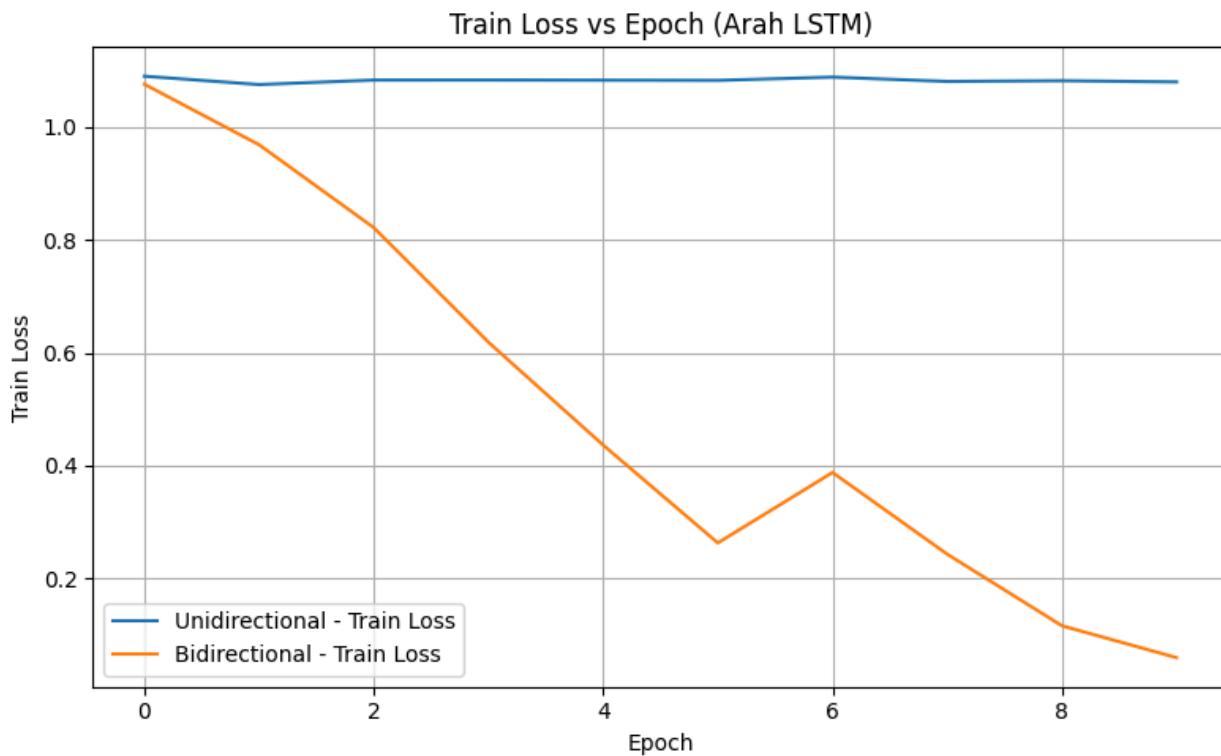
1. Unidirectional LSTM
2. Bidirectional LSTM

3.3.3.2. Hasil Pengujian

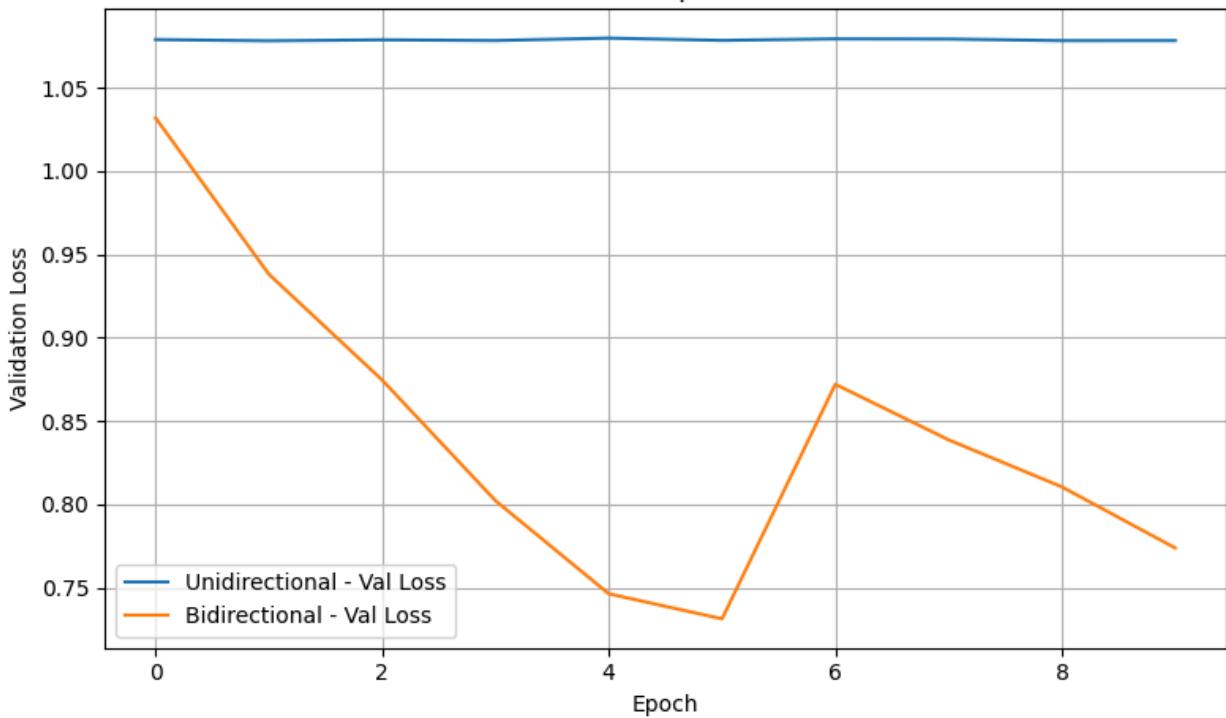
Pengujian dilakukan pada dua konfigurasi arah layer LSTM, yaitu bidirectional dan unidirectional, dengan parameter lain dijaga tetap sama. Hasil evaluasi ditampilkan dalam tabel berikut:

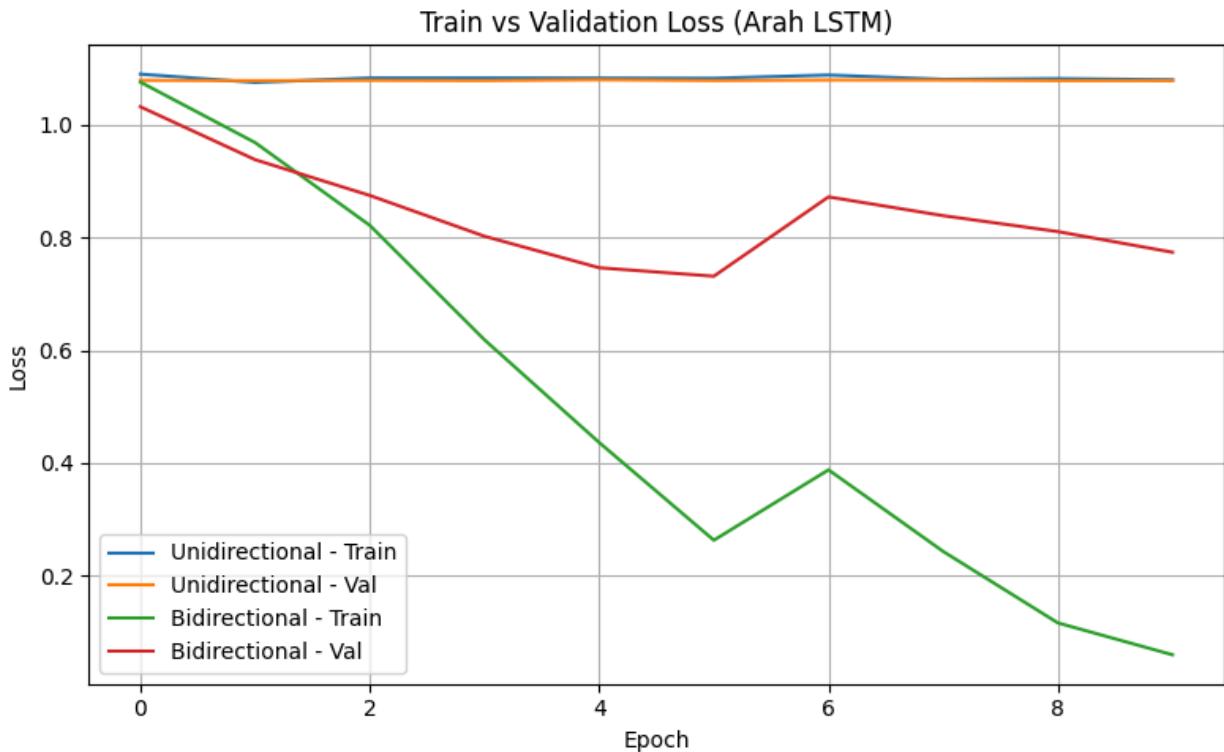
Arah RNN	Waktu Latih (detik)	F1-Score (Keras)	F1-Score (Manual)
Bidirectional	3.93	0.7217	0.7217
Unidirectional	3.75	0.1844	0.1844

Grafik visualisasi loss, baik validation loss maupun train loss, akan ditampilkan di bawah ini



Validation Loss vs Epoch (Arah LSTM)





3.3.3.3. Analisis dan Kesimpulan

Penulis merasa tidak perlu lagi menyampaikan asumsi pribadi, karena hasil visualisasi plot training loss dan validation loss telah menunjukkan perbedaan performa yang sangat mencolok antara model unidirectional dan bidirectional. Secara umum, tugas yang dikerjakan dalam eksperimen ini dapat dikategorikan sebagai upaya untuk mendeteksi konteks dalam teks, di mana makna suatu pernyataan tidak hanya ditentukan oleh kata-kata yang muncul sebelumnya, tetapi juga oleh kata-kata yang muncul setelahnya.

Kombinasi susunan kata dalam kalimat mencerminkan suatu nilai atau makna tertentu, baik itu bernuansa positif, negatif, maupun netral. Oleh karena itu, kemampuan model untuk memahami konteks secara menyeluruh sangatlah penting. Semakin baik pemahaman terhadap konteks, semakin akurat pula model dalam menentukan maksud sebenarnya dari suatu pernyataan. Hasil pengujian menunjukkan bahwa model Bidirectional LSTM menghasilkan akurasi sebesar 0.7217, jauh lebih tinggi dibandingkan Unidirectional LSTM yang hanya mencapai 0.1844. Hal ini menjadi indikator yang jelas bahwa kemampuan memahami konteks dua arah berkontribusi signifikan terhadap peningkatan akurasi klasifikasi.

Perbandingan ini sebenarnya agak timpang, karena dalam analisis teks, terutama yang bergantung pada nuansa dan makna kalimat, model yang mampu membaca informasi baik dari arah depan maupun belakang tentu memiliki keunggulan yang jelas. Bidirectional LSTM memiliki akses penuh terhadap keseluruhan struktur kalimat, sehingga mampu membentuk

representasi yang lebih kaya dan kontekstual, sementara unidirectional LSTM hanya mengandalkan informasi masa lalu, dan cenderung kehilangan makna yang lebih utuh.

Sebagai kesimpulan, dalam eksperimen klasifikasi teks yang dilakukan, model bidirectional terbukti memberikan hasil yang jauh lebih baik. Hal ini dikarenakan kemampuannya dalam menangkap konteks secara menyeluruh dari sebuah pernyataan, sehingga lebih efektif dalam memahami dan mengklasifikasikan makna dari data teks yang dianalisis.

BAB 4

KESIMPULAN DAN SARAN

4.1. Kesimpulan

Dalam eksperimen klasifikasi gambar menggunakan CNN pada dataset CIFAR-10, model dengan 2 layer konvolusi dan filter berukuran 3×3 memberikan hasil paling optimal. Hal ini dapat dijelaskan karena CIFAR-10 merupakan dataset gambar berukuran kecil (32×32) yang menekankan pada fitur lokal seperti tepi, bentuk, dan detail objek kecil (asumsi yang ada seperti telinga kucing). Ukuran filter kecil seperti 3×3 cocok untuk menangkap pola mikro tersebut tanpa kehilangan resolusi spasial. Sementara itu, penambahan layer atau filter yang terlalu banyak justru memperbesar risiko overfitting, seperti terlihat pada konfigurasi 3-layer dan 64 filter, di mana training loss turun tapi validation loss mulai naik. Hal ini menunjukkan pentingnya menjaga trade-off antara kapasitas model dan kompleksitas data. Selain itu, juga harus menyesuaikan dengan kebutuhan analisis, terutama dalam menentukan metode *pooling*.

Berbeda dengan gambar, tugas klasifikasi teks seperti sentimen analysis memerlukan pemahaman terhadap urutan dan konteks kalimat, yang dieksplorasi melalui RNN dan LSTM. Model RNN dengan 1 layer dan 64 unit menunjukkan performa paling stabil dan general di dataset NusaX-Sentiment. Penambahan layer atau neuron tidak otomatis meningkatkan akurasi; justru model menjadi lebih lambat dan rawan overfitting atau bahkan underfitting bila arsitektur terlalu besar tanpa regularisasi yang memadai. Dalam hal ini, kompleksitas berlebih pada data yang tidak seimbang atau pendek justru kontraproduktif.

LSTM hadir sebagai solusi atas kelemahan RNN dalam menangani *long-term dependencies*. Hasil eksperimen menunjukkan bahwa LSTM 2 layer atau dengan unit besar (256) memberikan hasil terbaik dalam klasifikasi sentimen, terutama karena kemampuannya menangkap informasi jangka panjang. Namun, hal ini tetap memerlukan kontrol terhadap overfitting, karena model dengan 3 layer atau tanpa cukup validasi bisa menurunkan performanya. Penggunaan Bidirectional LSTM juga terbukti unggul secara signifikan, dengan makro *F1-score* jauh di atas unidirectional, karena dapat menggunakan informasi sebelum dan sesudah token untuk memahami konteks secara utuh, hal yang krusial dalam analisis teks.

Secara keseluruhan, pendekatan terbaik sangat bergantung pada sifat data dan tujuan analisis. CNN cocok untuk pola spasial lokal dalam gambar, RNN cukup untuk urutan sederhana, dan LSTM (khususnya bidirectional) unggul untuk pemrosesan teks kontekstual. Namun, semua model memerlukan pemilihan arsitektur yang seimbang, jumlah layer, unit, dan arah agar dapat menghindari overfitting serta mempertahankan performa yang kuat di data validasi.

4.2. Saran

Penulis merasa bahwa saran yang paling baik adalah untuk terus menambah eksplorasi. Seperti yang disebutkan sebelumnya , berbagai pergantian konfigurasi tidak memiliki ungkapan penentu, lebih ke sebuah pertukaran antar berbagai penilaian. Penulis menyarankan untuk mencoba memperluas analisis , baik dengan menambah iterasi, memperbanyak dataset , ataupun menggunakan kombinasi konfigurasi yang lebih unik pula.

REFERENSI

- [1] Institut Teknologi Bandung. (2025). IF3270 - RNN Part 2. [online] Available at: https://edunexcontentprodhot.blob.core.windows.net/edunex/nullfile/1747619176390_IF3270-Mgg11-RNN-part2 [Accessed 15 May 2025].
- [2] Institut Teknologi Bandung. (2025). IF3270 - CNN. [online] Available at: https://edunexcontentprodhot.blob.core.windows.net/edunex/nullfile/1743979407057_IF3270-Mgg0506-CNN [Accessed 17 May 2025].
- [3] Institut Teknologi Bandung. (2025). IF3270 - RNN Part 1. [online] Available at: https://edunexcontentprodhot.blob.core.windows.net/edunex/nullfile/1745821108485_IF3270-Mgg09-RNN-part1 [Accessed 20 May 2025].

LAMPIRAN

Link Github:

<https://github.com/mybajwk/ML-2-8>

Pembagian Tugas

Nama (NIM)	Pembagian Tugas
Wilson Yusda (13522019)	CNN
Enrique Yanuar (13522077)	LSTM
Mesach Harmasendro (13522117)	RNN