

LAPORAN TUGAS BESAR 2
IF3270 PEMBELAJARAN MESIN

Convolutional Neural Network dan Recurrent Neural Network



Disusun Oleh:

13522060 Andhita Naura Hariyanto

13522062 Salsabiila

13522082 Keanu Amadius Gonza Wrahatno

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

2025

DAFTAR ISI

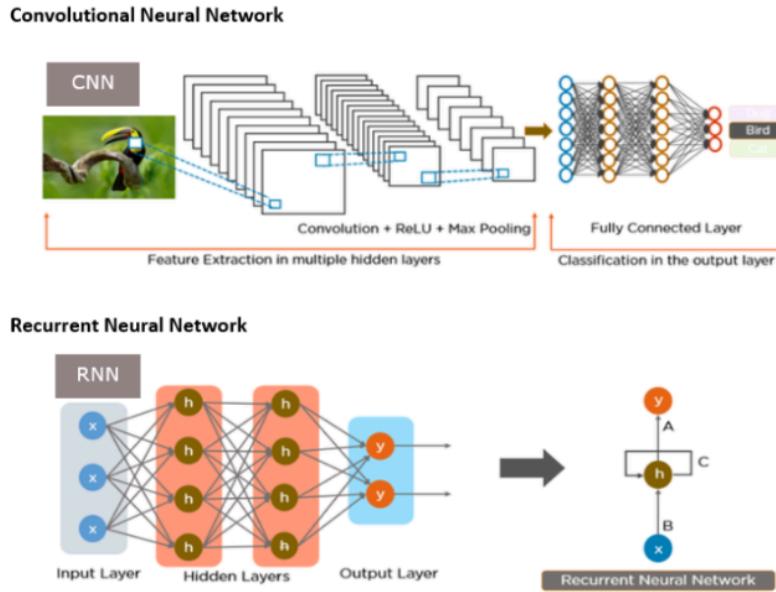
DAFTAR ISI.....	2
DAFTAR GAMBAR.....	3
BAB I	
Deskripsi Persoalan.....	4
BAB II	
Pembahasan.....	7
2.1. Penjelasan Implementasi.....	7
2.1.1. CNN.....	7
2.1.2. RNN.....	13
2.1.3. LSTM.....	20
2.2. Hasil Pengujian.....	29
2.2.1. Hasil Pengujian CNN.....	29
2.2.1.1. Compare model keras vs manual.....	29
2.2.1.3. Pengaruh jumlah filter.....	30
2.2.1.4. Pengaruh ukuran filter.....	31
2.2.1.5. Pengaruh jenis pooling.....	32
2.2.2. Hasil Pengujian RNN.....	33
2.2.2.2. Pengaruh jumlah layer RNN.....	34
2.2.2.3. Pengaruh banyak cell RNN per layer.....	36
2.2.2.4. Pengaruh jenis layer RNN berdasarkan arah.....	38
2.2.3. Hasil Pengujian LSTM.....	40
2.2.3.1. Compare antara model keras dan manual.....	40
2.2.3.2. Pengaruh jumlah layer LSTM.....	40
2.2.3.3. Pengaruh banyak cell LSTM per layer.....	42
2.2.3.4. Pengaruh jenis layer LSTM berdasarkan arah.....	44
BAB III	
Kesimpulan dan Saran.....	47
3.1. Kesimpulan.....	47
3.2. Saran.....	48
LAMPIRAN.....	49
REFERENSI.....	50

DAFTAR GAMBAR

Gambar 1.1 Visualisasi CNN dan RNN	4
Gambar 1.2 Visualisasi CNN	4
Gambar 1.3 Visualisasi RNN	5
Gambar 1.4 Visualisasi LSTM	6
Gambar 2.2.1.1.1 Grafik Training Loss dan Validation Loss Eksperimen Jumlah Layer	32
Gambar 2.2.1.2.1 Grafik Training Loss dan Validation Loss Eksperimen Jumlah Filter	33
Gambar 2.2.1.3.1 Grafik Training Loss dan Validation Loss Eksperimen Ukuran Filter	34
Gambar 2.2.1.4.1 Grafik Training Loss dan Validation Loss Eksperimen Jenis Pooling	35
Gambar 2.2.2.1.1 Grafik Training Loss dan Validation Loss Eksperimen layer	37
Gambar 2.2.2.1.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen layer	37
Gambar 2.2.2.2.1 Grafik Training Loss dan Validation Loss Eksperimen Arah	39
Gambar 2.2.2.2.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Arah	39
Gambar 2.2.2.3.1 Grafik Training Loss dan Validation Loss Eksperimen Arah	40
Gambar 2.2.2.3.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Arah	41
Gambar 2.2.3.1.1 Grafik Training Accuracy dan Validation Accuracy Eksperimen Layer	43
Gambar 2.2.3.1.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Layer	43
Gambar 2.2.3.2.1 Grafik Training Accuracy dan Validation Accuracy Eksperimen Cell	45
Gambar 2.2.3.2.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Cell	45
Gambar 2.2.3.3.1 Grafik Training Accuracy dan Validation Accuracy Eksperimen Arah	47
Gambar 2.2.3.3.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Arah	47

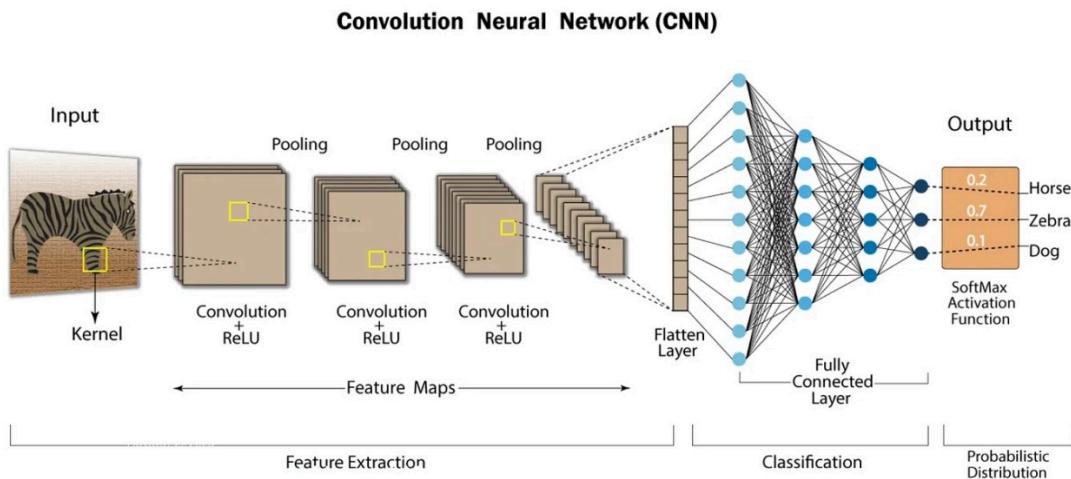
BAB I

Deskripsi Persoalan



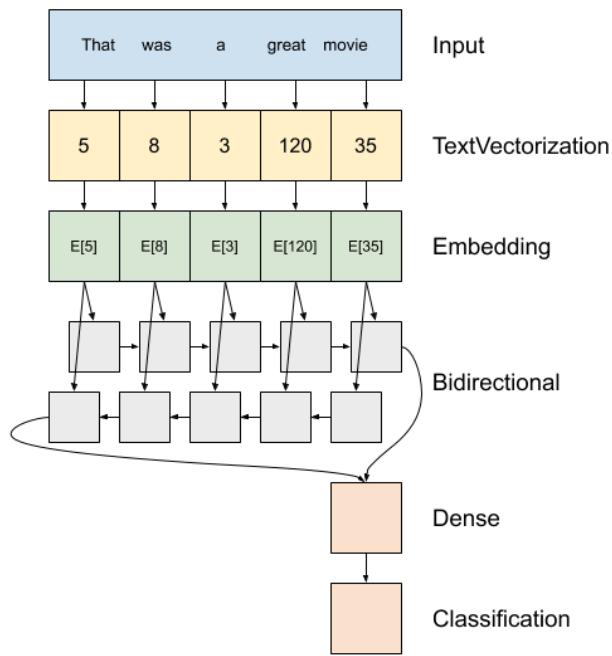
Gambar 1.1 Visualisasi CNN dan RNN

Pada tugas besar ini, akan diimplementasikan sebuah modul Convolutional Neural Network (CNN) serta dua jenis Recurrent Neural Network (RNN), yaitu RNN dan Long Short-Term Memory (LSTM). Implementasi dilakukan from scratch, dan dibandingkan performanya dengan hasil dari framework Keras.



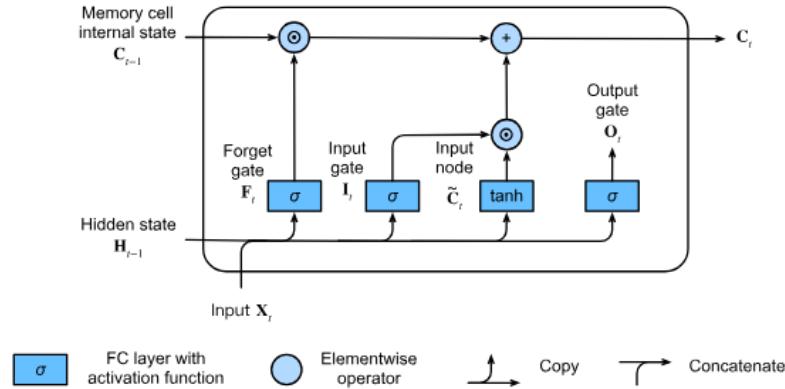
Gambar 1.2 Visualisasi CNN

Pada pengimplementasian CNN, model harus dilatih menggunakan dataset CIFAR-10 dengan Keras. Model harus mencakup layer-layer seperti Conv2D, Pooling, Flatten/Global Pooling, dan Dense. Fungsi loss yang digunakan adalah Sparse Categorical Cross-entropy dan optimizer-nya adalah Adam. Model yang dilatih harus diuji dalam berbagai variasi hyperparameter, yaitu jumlah layer konvolusi, jumlah filter per layer, ukuran filter, dan jenis pooling layer (max/average). Evaluasi performa dilakukan dengan macro F1-score serta analisis grafik training dan validation loss. Hasil pelatihan disimpan untuk digunakan kembali.



Gambar 1.3 Visualisasi RNN

Pada pengimplementasian RNN, model harus dilatih menggunakan dataset NusaX-Sentiment. Preprocessing teks dilakukan melalui tokenization dan embedding, menggunakan TextVectorization dan Embedding dari Keras. Model RNN minimal harus memiliki Embedding Layer, RNN Layer (bidirectional atau unidirectional), Dropout Layer, Dense Layer. Eksperimen dilakukan untuk menganalisis pengaruh jumlah layer RNN, jumlah cell per layer, dan arah (bidirectional vs unidirectional). Hasil pelatihan disimpan dan modul forward propagation RNN diimplementasikan from scratch secara modular. Hasilnya dibandingkan dengan hasil Keras menggunakan test set dan dievaluasi menggunakan macro F1-score.



Gambar 1.4 Visualisasi LSTM

Pada pengimplementasian LSTM, model harus dilatih menggunakan dataset NusaX-Sentiment. Tahapan preprocessing dan arsitektur dasar mirip dengan Simple RNN, namun menggunakan layer LSTM. Eksperimen dilakukan untuk mengevaluasi jumlah layer LSTM, jumlah cell per layer, dan arah layer LSTM (bidirectional vs unidirectional). Hasil pelatihan model disimpan, lalu forward propagation LSTM diimplementasikan secara modular dan diuji untuk mencocokkan output-nya dengan model Keras berdasarkan macro F1-score

BAB II

Pembahasan

2.1. Penjelasan Implementasi

2.1.1. CNN

1. Implementasi CNN Mandiri

a. Kelas CNN

Kelas ini merupakan kelas utama untuk model CNN implementasi mandiri yang memiliki beberapa fungsi seperti `add_layer`, `compile_model`, `_prepare_input` dan `_unprepare_output` untuk menyesuaikan dimensi input dan output, serta forward dan backward yang memanggil fungsi forward ataupun backward pada setiap layer. Fungsi `fit` berfungsi untuk training model dengan menerima parameter `X_train`, `y_train`, `epochs`, `batch_size`, dan `validation` data opsional, kemudian melakukan proses training per epoch dengan mengacak data, membagi menjadi mini-batch, forward-backward pass, dan mengintegrasikan Keras optimizer untuk optimasi parameter. Fungsi ini juga men-track metrics seperti `accuracy` dan `loss` yang disimpan dalam `history` serta melakukan evaluasi `validation` jika tersedia. Untuk prediksi terdapat fungsi `predict_proba` yang melakukan forward propagation dalam mode inference dan mengembalikan probabilitas dengan softmax manual jika diperlukan, serta fungsi `predict` yang menggunakan `np.argmax()` pada hasil `predict_proba()` untuk mendapatkan kelas prediksi. Terdapat juga fitur `load_weights_from_keras` untuk memuat weights dari model Keras.

b. Kelas SparseCategoricalCrossentropy

Kelas `SparseCategoricalCrossentropy` memiliki parameter `from_logits` yang menentukan apakah input berupa raw logits atau sudah dalam bentuk probabilitas. Pada fungsi forward, loss dihitung dengan mengubah logits ke probabilitas menggunakan softmax jika `from_logits=True`, kemudian menerapkan clipping untuk stabilitas numerik dan menghitung negative log-likelihood dari probabilitas kelas yang benar menggunakan indexing `probabilities[np.arange(batch_size), y_true_sparse]`, lalu mengembalikan rata-rata loss dari seluruh batch. Untuk fungsi backward, gradien dihitung dengan membuat copy dari probabilitas, mengurangi 1 pada indeks kelas yang benar untuk setiap sample dalam batch, dan membagi dengan `batch_size` untuk normalisasi gradien.

```

import numpy as np

class SparseCategoricalCrossentropy:
    def __init__(self, from_logits=True):
        self.from_logits = from_logits
        self.probabilities = None

    def forward(self, y_pred_logits_or_probs,
y_true_sparse):
        batch_size = y_pred_logits_or_probs.shape[0]

        if y_true_sparse.ndim > 1 and
y_true_sparse.shape[1] == 1:
            y_true_sparse = y_true_sparse.flatten()

        if self.from_logits:
            exp_logits = np.exp(y_pred_logits_or_probs -
np.max(y_pred_logits_or_probs, axis=1, keepdims=True))
            self.probabilities = exp_logits /
np.sum(exp_logits, axis=1, keepdims=True)
        else:
            self.probabilities = y_pred_logits_or_probs

        epsilon = 1e-12
        self.probabilities = np.clip(self.probabilities,
epsilon, 1. - epsilon)

        correct_log_probs =
-np.log(self.probabilities[np.arange(batch_size),
y_true_sparse])

        loss = np.mean(correct_log_probs)
        return loss

    def backward(self, y_pred_logits_or_probs,
y_true_sparse):
        batch_size = y_pred_logits_or_probs.shape[0]

        if y_true_sparse.ndim > 1 and
y_true_sparse.shape[1] == 1:
            y_true_sparse = y_true_sparse.flatten()

        if self.probabilities is None or
self.probabilities.shape != y_pred_logits_or_probs.shape:
            if self.from_logits:
                exp_logits = np.exp(y_pred_logits_or_probs -
np.max(y_pred_logits_or_probs, axis=1, keepdims=True))
                self.probabilities = exp_logits /
np.sum(exp_logits, axis=1, keepdims=True)
            else:
                self.probabilities =
y_pred_logits_or_probs

        gradient = np.copy(self.probabilities)

```

```

    gradient[np.arange(batch_size), y_true_sparse] == 1
    gradient = gradient / batch_size
    return gradient

```

c. Kelas Layer

Kelas ini merupakan base class untuk setiap layer pada CNN

```

class Layer:
    def __init__(self):
        self.input_tensor = None
        self.output = None

        self.weights = None
        self.biases = None

        self.d_weights = None
        self.d_biases = None

        self.training_mode = True

    def forward(self, input_data):
        raise NotImplementedError

    def backward(self, output_gradient):
        raise NotImplementedError

    def initialize_parameters(self, *args, **kwargs):
        pass

    def set_training_mode(self, training: bool):
        self.training_mode = training

```

d. Kelas Conv2DLayer

Kelas Conv2DLayer merupakan implementasi layer konvolusi 2D yang memiliki parameter num_filters, filter_size, stride, padding, serta mode inisialisasi weight dan bias. Konstruktor melakukan validasi parameter dan fungsi initialize_parameters menginisialisasi weights dengan mode 'he', 'xavier', atau 'zeros' serta biases dengan mode 'zeros' secara default. Fungsi forward melakukan konvolusi dengan menentukan padding berdasarkan mode ('valid' tanpa padding, 'same' dengan padding yang dihitung), kemudian melakukan operasi konvolusi menggunakan nested loop untuk menghitung dot product antara receptive field dengan filter weights ditambah bias. Fungsi backward menghitung gradien untuk backpropagation dimana gradien weights dihitung sebagai akumulasi receptive field dikali output gradient, gradien bias sebagai akumulasi output gradient, dan gradien input sebagai akumulasi weights dikali

output gradient, dengan penanganan khusus untuk mengembalikan dimensi asli input jika menggunakan padding 'same'.

```

def forward(self, input_tensor):
    self.input_tensor_original_shape =
input_tensor.shape
    batch_size, i_h, i_w, i_c = input_tensor.shape

    if self.weights is None:
        self.initialize_parameters(i_c)

    if self._input_channels != i_c:
        raise ValueError(f"Conv2DLayer input channel mismatch. Expected {self._input_channels}, got {i_c}.")"

    if self.padding_mode == 'valid':
        self.pad_dims = (0,0,0,0)
        o_h = (i_h - self.f_h) // self.stride_h + 1
        o_w = (i_w - self.f_w) // self.stride_w + 1
        input_padded = input_tensor
    elif self.padding_mode == 'same':
        o_h = int(np.ceil(float(i_h) /
float(self.stride_h)))
        o_w = int(np.ceil(float(i_w) /
float(self.stride_w)))

        pad_h_total = max((o_h - 1) * self.stride_h +
self.f_h - i_h, 0)
        pad_w_total = max((o_w - 1) * self.stride_w +
self.f_w - i_w, 0)

        pad_top = pad_h_total // 2
        pad_bottom = pad_h_total - pad_top
        pad_left = pad_w_total // 2
        pad_right = pad_w_total - pad_left
        self.pad_dims = (pad_top, pad_bottom,
pad_left, pad_right)

        input_padded = np.pad(input_tensor,
((0,0), (pad_top,
pad_bottom), (pad_left, pad_right), (0,0)),
mode='constant',
constant_values=0)
    else:
        raise ValueError(f"Unsupported padding mode:
{self.padding_mode}. Choose 'valid' or 'same'.")"

    self.input_padded_for_backward = input_padded

    output_tensor = np.zeros((batch_size, o_h, o_w,
self.num_filters))

    for b in range(batch_size):
        for h_out in range(o_h):
            for w_out in range(o_w):
                h_start = h_out * self.stride_h

```

```

        h_end = h_start + self.f_h
        w_start = w_out * self.stride_w
        w_end = w_start + self.f_w

        receptive_field = input_padded[b,
h_start:h_end, w_start:w_end, :]

        for f_idx in range(self.num_filters):
            conv_val = np.sum(receptive_field
* self.weights[:, :, :, f_idx]) + self.biases[0, 0, 0,
f_idx]
            output_tensor[b, h_out, w_out,
f_idx] = conv_val

        self.output = output_tensor
    return self.output

```

e. Kelas DenseLayer

Kelas DenseLayer merupakan implementasi fully connected layer dengan parameter output_dim dan mode inisialisasi weight serta bias. Fungsi initialize_parameters menginisialisasi weights dengan mode 'he', 'xavier', atau 'zeros' berdasarkan input dimension, dan biases dengan mode 'zeros' secara default. Fungsi forward melakukan transformasi linear dengan operasi matrix multiplication np.dot(input_tensor, self.weights) + self.biases setelah memvalidasi dimensi input. Fungsi backward menghitung gradien dimana d_weights dihitung sebagai input_tensor.T @ output_gradient, d_biases sebagai sum dari output_gradient pada axis batch, dan gradien input sebagai output_gradient @ weights.T untuk propagasi ke layer sebelumnya.

```

def forward(self, input_tensor):
    self.input_tensor = input_tensor

    if self.weights is None:
        self.initialize_parameters(input_tensor.shape[1])
        elif self.input_dim != input_tensor.shape[1]:
            raise ValueError(f"DenseLayer input dimension mismatch. Expected {self.input_dim}, got {input_tensor.shape[1]}. Re-initialization might be needed.")

        # Linear transformation: Output = Input @ Weights
        + Biases
        self.output = np.dot(input_tensor, self.weights) +
        self.biases
    return self.output

```

f. Kelas FlattenLayer

Kelas flatten layer berfungsi untuk mengubah output dari layer sebelum dense layer menjadi berdimensi satu.

```
def forward(self, input_tensor):
    self.input_tensor = input_tensor
    self.original_shape = input_tensor.shape

    batch_size = input_tensor.shape[0]
    self.output = input_tensor.reshape(batch_size, -1)
    return self.output
```

g. Kelas PoolingLayer

Kelas ini merupakan implementasi layer pooling yang memiliki parameter pool_size, stride, dan mode pooling ('max' atau 'average'). Konstruktor akan melakukan validasi pool_size sebagai tuple dan menentukan stride yang defaultnya sama dengan pool_size jika tidak ditentukan. Pada fungsi forward, operasi pooling dilakukan dengan nested loop untuk batch, channel, dan output dimensions dimana untuk setiap receptive field akan menghitung nilai maksimum atau rata-rata sesuai mode yang dipilih, dengan menyimpan posisi nilai maksimum ke dalam cache untuk max pooling guna keperluan backward pass. Untuk fungsi backward, gradien dihitung dimana pada max pooling gradien hanya didistribusikan ke posisi nilai maksimum yang tersimpan dalam cache, sedangkan pada average pooling gradien dibagi rata ke seluruh elemen dalam receptive field sesuai ukuran pool window.

```
def forward(self, input_tensor):
    self.input_tensor = input_tensor
    batch_size, i_h, i_w, i_c = input_tensor.shape

    o_h = (i_h - self.pool_h) // self.stride_h + 1
    o_w = (i_w - self.pool_w) // self.stride_w + 1

    output_tensor = np.zeros((batch_size, o_h, o_w,
i_c))
    self.cache = {}

    for b in range(batch_size):
        for c_idx in range(i_c):
            for h_out in range(o_h):
                for w_out in range(o_w):
                    h_start = h_out * self.stride_h
                    h_end = h_start + self.pool_h
                    w_start = w_out * self.stride_w
                    w_end = w_start + self.pool_w

                    receptive_field = input_tensor[b,
```

```

    h_start:h_end, w_start:w_end, c_idx]

        if self.mode == 'max':
            output_tensor[b, h_out, w_out,
c_idx] = np.max(receptive_field)
            max_idx_local =
np.unravel_index(np.argmax(receptive_field),
receptive_field.shape)
            self.cache[(b, h_out, w_out,
c_idx)] = (h_start + max_idx_local[0], w_start +
max_idx_local[1])
        elif self.mode == 'average':
            output_tensor[b, h_out, w_out,
c_idx] = np.mean(receptive_field)
        else:
            raise ValueError(f"Unsupported
pooling mode: {self.mode}. Choose 'max' or 'average'.")
        self.output = output_tensor
    return self.output

```

h. Kelas ReLULayer

Kelas ini merupakan fungsi aktivasi yang akan mengembalikan nilai minimal 0 pada fungsi forward dan pada fungsi backward akan menghitung gradien dengan mengalikan output_gradient dengan mask boolean (`self.input_tensor > 0`) yang bernilai 1 untuk input positif dan 0 untuk input negatif atau nol.

```

def forward(self, input_tensor):
    self.input_tensor = input_tensor
    self.output = np.maximum(0, input_tensor)
    return self.output

```

i. SoftmaxLayer

Kelas SoftmaxLayer merupakan kelas yang akan menjadi fungsi aktivasi pada output layer

```

def forward(self, input_tensor):
    self.input_tensor = input_tensor
    exp_values = np.exp(input_tensor -
np.max(input_tensor, axis=1, keepdims=True))
    self.output = exp_values / np.sum(exp_values,
axis=1, keepdims=True) # Probabilities
    return self.output

```

2.1.2. RNN

1. Implementasi RNN Mandiri
 - a. Kelas EmbeddingLayer

EmbeddingLayer berfungsi untuk mengkonversi token integer menjadi representasi vektor dense dengan dimensi tertentu. Layer ini merupakan tahap awal dalam pemrosesan data teks untuk model RNN.

Forward propagation pada Embedding Layer yaitu dengan melakukan lookup operasi sederhana. Ketika menerima input berupa indeks token (integer), layer ini menggunakan indeks tersebut untuk mengambil baris yang sesuai dari matriks weights. Operasi ini mengkonversi representasi sparse (indeks) menjadi representasi dense (vektor embedding) dengan dimensi yang telah ditentukan.

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

b. Kelas DenseLayer

DenseLayer adalah fully connected layer yang melakukan transformasi linear pada input dan menerapkan fungsi aktivasi. Layer ini umumnya digunakan sebagai layer output dalam model klasifikasi. Terdapat method `_apply_activation(z)` untuk menerapkan fungsi aktivasi.

Forward propagation pada DenseLayer dimulai dengan perhitungan transformasi linear yaitu perkalian matriks antara input dengan weights, kemudian ditambahkan dengan bias jika tersedia. Hasil perhitungan linear ini kemudian masuk ke fungsi aktivasi yang telah ditentukan. Implementasi rumus $z = x \cdot W + b$, lalu menerapkan fungsi aktivasi pada z untuk menghasilkan output akhir.

```
def forward(self, x):
    z = np.dot(x, self.weights)
    if self.bias is not None:
        z += self.bias

    return self._apply_activation(z)
```

c. Kelas SimpleRNNCell

SimpleRNNCell adalah unit dasar dari Simple RNN yang memproses satu timestep input. Cell ini mengimplementasikan perhitungan hidden state baru berdasarkan input saat ini dan hidden state sebelumnya.

Forward propagation pada SimpleRNNCell mengimplementasikan rumus dasar RNN yaitu $h_t = \tanh(x_t \cdot W_{\text{input}} + h_{\text{prev}} \cdot W_{\text{recurrent}} + \text{bias})$. Cell ini menerima input saat ini (x_t) dan hidden state sebelumnya (h_{prev}), kemudian melakukan perhitungan linear dengan masing-masing matriks bobotnya. Hasil penjumlahan kedua transformasi linear tersebut

ditambah bias, lalu masuk fungsi aktivasi tanh untuk menghasilkan hidden state baru.

```
def forward_step(self, x_t, h_prev):
    # h_t = tanh(x_t @ W_input + h_prev @ W_recurrent
    + bias)
    linear = np.dot(x_t, self.kernel) + np.dot(h_prev,
    self.recurrent_kernel)
    if self.use_bias and self.bias is not None:
        linear += self.bias
    h_t = np.tanh(linear)
    return h_t
```

d. Kelas SimpleRNNLayer

SimpleRNNLayer adalah layer lengkap yang menggunakan SimpleRNNCell untuk memproses sequence input. Layer ini dapat dikonfigurasi untuk mengembalikan seluruh sequence atau hanya output terakhir.

Forward propagation pada SimpleRNNLayer bekerja secara iteratif untuk memproses sequence input. Pertama, melakukan reshaping input menjadi format 3D (batch_size, sequence_length, input_dim), kemudian menginisialisasi hidden state awal dengan nilai nol. Untuk setiap timestep dalam sequence, layer menggunakan SimpleRNNCell untuk menghitung hidden state baru berdasarkan input saat ini dan hidden state sebelumnya. Bergantung pada parameter return_sequences, layer akan mengembalikan seluruh hidden states atau hanya hidden state terakhir.

```
def forward(self, x):
    original_shape = x.shape

    if len(x.shape) == 1:
        x = x.reshape(1, 1, -1)
    elif len(x.shape) == 2:
        x = x.reshape(x.shape[0], 1, x.shape[1])
    elif len(x.shape) == 3:
        pass
    else:
        raise ValueError(f"Input must have 1, 2, or 3
dimensions, got {len(x.shape)} dimensions with shape
{x.shape}")

    batch_size, sequence_length, input_dim = x.shape

    if self.go_backwards:
        x = x[::-1, ::-1, :]

    hidden_states = []
    h_t = np.zeros((batch_size, self.cell.units))

    for t in range(sequence_length):
```

```

        x_t = x[:, t, :]
        h_t = self.cell.forward_step(x_t, h_t)
        hidden_states.append(h_t.copy())

    if self.return_sequences:
        output = np.stack(hidden_states, axis=1)
        if self.go_backwards:
            output = output[:, ::-1, :]
        return output
    else:
        if len(original_shape) == 1:
            return hidden_states[-1].squeeze(0)
        else:
            return hidden_states[-1]

```

e. Kelas BidirectionalRNNLayer

BidirectionalRNNLayer mengimplementasikan RNN bidirectional yang memproses sequence dalam dua arah (forward dan backward) secara bersamaan. Layer ini dapat menangkap informasi kontekstual dari kedua arah dalam sequence.

Forward propagation pada BidirectionalRNNLayer bekerja dengan memproses input sequence dalam dua arah secara bersamaan. Layer ini menggunakan dua SimpleRNNLayer terpisah. Satu untuk arah forward dan satu untuk arah backward. Setelah kedua RNN memproses sequence, layer mengambil output terakhir dari forward RNN dan output pertama dari backward RNN (yang merepresentasikan informasi dari seluruh sequence). Kedua output ini kemudian digabungkan sesuai dengan mode yang ditentukan, baik melalui concatenation, penjumlahan, atau rata-rata untuk menghasilkan representasi final yang menangkap konteks dari kedua arah.

```

def forward(self, x):
    forward_output = self.forward_rnn.forward(x)
    backward_output = self.backward_rnn.forward(x)

    forward_last = forward_output[:, -1, :]
    backward_last = backward_output[:, 0, :]

    if self.merge_mode == 'concat':
        output = np.concatenate([forward_last,
backward_last], axis=1)
    elif self.merge_mode == 'sum':
        output = forward_last + backward_last
    elif self.merge_mode == 'avg':
        output = (forward_last + backward_last) / 2
    else:
        raise ValueError(f"Unsupported merge_mode:
{self.merge_mode}")

```

```
    return output
```

f. Kelas SimpleRNNModelFromScratch

SimpleRNNModelFromScratch merupakan kelas yang utama yang bertugas merekonstruksi model Simple RNN dari model Keras yang telah dilatih, menjadi implementasi forward propagation secara manual (from scratch). Kelas ini dirancang untuk membaca bobot-bobot dari model Keras dan membangun ulang arsitektur jaringan menggunakan layer-layer yang telah diimplementasikan sendiri.

Pada tahap inisialisasi, kelas ini akan memuat model Keras dari path yang diberikan, lalu secara otomatis membangun layer-layer yang sesuai melalui metode `_build_layers()`.

Fungsi `_get_activation_name()` berfungsi sebagai utility function untuk mengekstrak nama fungsi aktivasi dari layer Keras.

Fungsi `_build_layers()` merupakan inti dari proses rekonstruksi model. Metode ini bekerja dengan mengiterasi setiap layer dalam model Keras dan mengkonversinya ke dalam bentuk implementasi manual yang sesuai: Fungsi `forward()` mengimplementasikan proses forward propagation secara menyeluruh dengan menerima input, lalu melewatkannya secara berurutan ke setiap layer yang telah dibangun.

Fungsi `predict()` melengkapi fungsionalitas model dengan mengubah output probabilitas menjadi prediksi kelas melalui operasi `argmax`.

Fungsi `debug_model_structure()` disediakan sebagai alat bantu analisis struktur model Keras.

Fungsi `compare_implementation()` berfungsi untuk melakukan komparasi antara hasil dari keras dan hasil dari scratch.

```
class SimpleRNNModelFromScratch:  
    def __init__(self, keras_model_path):  
        self.model =  
        keras.models.load_model(keras_model_path)  
        self.layers = []  
        self._build_layers()  
  
    def _get_activation_name(self, activation_func):  
        if hasattr(activation_func, '__name__'): # Python 3  
            return activation_func.__name__  
        elif hasattr(activation_func, 'name'): # TensorFlow 1.x  
            return activation_func.name  
        else:  
            return  
        str(activation_func).split('.')[ -1].split(' ')[0]
```

```

def _build_layers(self):
    print("Building layers from Keras model...")
    print(f"Model has {len(self.model.layers)} layers")

    layer_idx = 0
    for i, layer in enumerate(self.model.layers):
        layer_type = layer.__class__.__name__
        print(f"Layer {i}: {layer_type} - {layer.name}")

        if layer_type == 'Embedding':
            weights = layer.get_weights()
            if len(weights) > 0:
                embedding_weights = weights[0]
                print(f" Embedding weights shape: {embedding_weights.shape}")

            self.layers.append(EmbeddingLayer(embedding_weights))
            layer_idx += 1

        elif layer_type == 'Bidirectional':
            print(f" Processing Bidirectional layer...")
            all_weights = layer.get_weights()
            print(f" Total weights in bidirectional layer: {len(all_weights)}")

            if len(all_weights) == 6:
                forward_weights = all_weights[:3]
                backward_weights = all_weights[3:6]
            elif len(all_weights) == 4:
                forward_weights = all_weights[:2]
                backward_weights = all_weights[2:4]
            else:
                raise ValueError(f"Unexpected number of weights in bidirectional layer: {len(all_weights)}")

            print(f" Forward weights shapes: {[w.shape for w in forward_weights]}")
            print(f" Backward weights shapes: {[w.shape for w in backward_weights]}")

            merge_mode = 'concat'
            if hasattr(layer, 'merge_mode'):
                merge_mode = layer.merge_mode

            self.layers.append(BidirectionalRNNLayer(forward_weights,
                                         backward_weights, merge_mode))
            layer_idx += 1

        elif layer_type == 'SimpleRNN':
            weights = layer.get_weights()
            print(f" SimpleRNN weights shapes: "

```

```

{[w.shape for w in weights]}")  
  

        return_seq = getattr(layer,  

'return_sequences', False)  

        go_back = getattr(layer, 'go_backwards',  

False)  
  

        self.layers.append(SimpleRNNLayer(weights,  

return_sequences=return_seq, go_backwards=go_back))  

        layer_idx += 1  
  

    elif layer_type == 'Dense':  

        weights = layer.get_weights()  

        if len(weights) == 0:  

            print(f"  Warning: Dense layer has no  

weights!")  

            continue  
  

        dense_weights = weights[0]  

        dense_bias = weights[1] if len(weights) >  

1 else None  
  

        print(f"  Dense weights shape:  

{dense_weights.shape}")  

        if dense_bias is not None:  

            print(f"  Dense bias shape:  

{dense_bias.shape}")  
  

        # Get activation function  

        activation =  

self._get_activation_name(layer.activation)  

        print(f"  Dense activation: {activation}")  
  

self.layers.append(DenseLayer(dense_weights, dense_bias,  

activation=activation))  

        layer_idx += 1  
  

    elif layer_type == 'Dropout':  

        print("  Skipping Dropout layer (inference  

mode)")  

        continue  
  

    else:  

        print(f"  Warning: Unknown layer type  

{layer_type}, skipping...")  

        continue  
  

        print(f"Built {len(self.layers)} functional  

layers")  
  

def forward(self, x):  

    print(f"Input shape: {x.shape}")  

    current_output = x  
  

    for i, layer in enumerate(self.layers):
```

```

        print(f"Processing layer {i}:
{type(layer).__name__}")
        current_output = layer.forward(current_output)
        print(f"  Output shape:
{current_output.shape}")

    return current_output

def predict(self, x):
    probabilities = self.forward(x)
    return np.argmax(probabilities, axis=1)

def debug_model_structure(keras_model_path):
    print("==== MODEL STRUCTURE DEBUG ===")

    model = keras.models.load_model(keras_model_path)
    print(f"Model type: {type(model)}")
    print(f"Number of layers: {len(model.layers)}")
    print()

    for i, layer in enumerate(model.layers):
        print(f"Layer {i}: {layer.__class__.__name__}")
        print(f"  Name: {layer.name}")
        print(f"  Input shape: {layer.input_shape if
hasattr(layer, 'input_shape') else 'N/A'}")
        print(f"  Output shape: {layer.output_shape if
hasattr(layer, 'output_shape') else 'N/A'}")

        weights = layer.get_weights()
        print(f"  Weights count: {len(weights)}")
        for j, w in enumerate(weights):
            print(f"    Weight {j} shape: {w.shape}")

        if hasattr(layer, 'activation'):
            print(f"    Activation: {layer.activation}")
        if hasattr(layer, 'return_sequences'):
            print(f"    Return sequences:
{layer.return_sequences}")
            if hasattr(layer, 'go_backwards'):
                print(f"      Go backwards: {layer.go_backwards}")
            if hasattr(layer, 'merge_mode'):
                print(f"      Merge mode: {layer.merge_mode}")

    print()

```

2.1.3. LSTM

1. Implementasi LSTM Mandiri

a. Kelas EmbeddingLayer

Kelas EmbeddingLayer mengonversi token integer menjadi representasi vektor dense. Kelas menerima parameter weights, yang merupakan hasil

dari model Keras yang telah dilatih sebelumnya, berupa matriks embedding dengan dimensi (vocab_size, embedding_dim). Kemudian, EmbeddingLayer akan mengambil vektor embedding yang bersesuaian. EmbeddingLayer dapat menghandle berbagai dimensi input.

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

b. Kelas DenseLayer

DenseLayer adalah fully connected layer yang melakukan transformasi linear pada input dan menerapkan fungsi aktivasi. Layer ini umumnya digunakan sebagai layer output dalam model klasifikasi. Terdapat method `_apply_activation(z)` untuk menerapkan fungsi aktivasi.

Forward propagation pada DenseLayer dimulai dengan perhitungan transformasi linear yaitu perkalian matriks antara input dengan weights, kemudian ditambahkan dengan bias jika tersedia. Hasil perhitungan linear ini kemudian masuk ke fungsi aktivasi yang telah ditentukan. Implementasi rumus $z = x \cdot W + b$, lalu menerapkan fungsi aktivasi pada z untuk menghasilkan output akhir.

```
def forward(self, x):
    z = np.dot(x, self.weights)
    if self.bias is not None:
        z += self.bias

    return self._apply_activation(z)
```

c. Kelas LSTMCell

LSTMCell adalah unit dasar dari LSTM yang memproses satu timestep input. Cell ini mengimplementasikan perhitungan hidden state baru berdasarkan input saat ini dan hidden state juga cell state sebelumnya.

Forward propagation pada LSTMCell mengimplementasikan mekanisme kalkulasi nilai untuk input gate, output gate, forget gate, candidate cell state, cell state, hingga menghitung nilai akhir untuk hidden state terkait.

Input Gate

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

Candidate Cell State

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

Forget Gate

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Output Gate

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

Cell State

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

Hidden State

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

Kelas LSTMCell akan menerima input, serta states yang terdiri dari nilai hidden state dan cell state sebelumnya. Setelah melakukan kalkulasi input gate, forget gate, output gate, dan candidate cell state, akan dilakukan perhitungan nilai cell state. Cell state dihitung nilainya dengan menjumlahkan hasil dot product antara forget gate dan cell state sebelumnya dengan hasil dot product input gate dengan nilai candidate cell state. Kemudian, akan dihasilkan nilai hidden state dengan melakukan dot product output gate dengan hiperbolik tangen dari nilai cell state yang didapat. LSTMCell akan mengembalikan hasil akhir perhitungan hidden state dan cell state.

```

def forward_step(self, x_t, states):
    h_prev, c_prev = states

    i_t = np.dot(x_t, self.W_i) + np.dot(h_prev,
self.U_i)
    if self.b_i is not None:
        i_t += self.b_i
    i_t = self._sigmoid(i_t)

    f_t = np.dot(x_t, self.W_f) + np.dot(h_prev,
self.U_f)
    if self.b_f is not None:
        f_t += self.b_f
    f_t = self._sigmoid(f_t)

```

```

        candidate_t = np.dot(x_t, self.W_c) +
np.dot(h_prev, self.U_c)
        if self.b_c is not None:
            candidate_t += self.b_c
candidate_t = np.tanh(candidate_t)

c_t = f_t * c_prev + i_t * candidate_t

o_t = np.dot(x_t, self.W_o) + np.dot(h_prev,
self.U_o)
        if self.b_o is not None:
            o_t += self.b_o
o_t = self._sigmoid(o_t)

h_t = o_t * np.tanh(c_t)

return h_t, c_t

```

d. Kelas LSTMCell

LSTM Layer dapat menangani pemrosesan sekuensial dengan memori jangka panjang dan pendek. Layer ini memiliki kemampuan untuk memproses input secara forward maupun backward dan dapat mengembalikan seluruh sequence atau hanya output terakhir. Layer ini menggunakan LSTMCell untuk menangani operasi forget gate, input gate, candidate cell state, output gate, cell state, dan hidden state pada setiap timestep.

Proses forward dimulai dengan reshaping input untuk memastikan format yang tepat (batch_size, sequence_length, input_dim). Jika parameter go_backwards diaktifkan, urutan input akan dibalik. Kemudian, hidden state (h_t) dan cell state (c_t) diinisialisasi dengan nilai nol. Untuk setiap timestep dalam sequence, input x_t diteruskan ke LSTMCell yang menghitung hidden state dan cell state baru menggunakan operasi gate LSTM. Jika return_sequences True, semua hidden states disimpan dan dikembalikan sebagai output; jika False, hanya hidden state terakhir yang dikembalikan.

```

def forward(self, x):
    original_shape = x.shape

    if len(x.shape) == 1:
        x = x.reshape(1, 1, -1)
    elif len(x.shape) == 2:
        x = x.reshape(x.shape[0], 1, x.shape[1])
    elif len(x.shape) == 3:
        pass
    else:
        raise ValueError(f"Input must have 1, 2, or 3")

```

```

dimensions, got {len(x.shape)} dimensions with shape
{x.shape}")

batch_size, sequence_length, input_dim = x.shape

if self.go_backwards:
    x = x[:, ::-1, :]

h_t = np.zeros((batch_size, self.cell.units))
c_t = np.zeros((batch_size, self.cell.units))

hidden_states = []

for t in range(sequence_length):
    x_t = x[:, t, :]
    h_t, c_t = self.cell.forward_step(x_t, (h_t,
c_t))
    hidden_states.append(h_t.copy())

if self.return_sequences:
    output = np.stack(hidden_states, axis=1)

    if self.go_backwards:
        output = output[:, ::-1, :]
    return output
else :
    if len(original_shape) == 1:
        return hidden_states[-1].squeeze(0)
    else:
        return hidden_states[-1]

```

e. Kelas BidirectionalLSTM

BidirectionalLSTM menggabungkan dua layer LSTM yang memproses input dalam arah yang berlawanan. Layer ini terdiri dari forward LSTM yang memproses dari awal ke akhir sequence dan backward LSTM yang memproses dari akhir ke awal sequence. Hasil dari kedua LSTM kemudian digabungkan menggunakan metode yang ditentukan oleh parameter merge_mode.

Proses forward melibatkan pemrosesan input melalui dua LSTM terpisah. Forward LSTM memproses input secara normal dari timestep pertama hingga terakhir, sedangkan backward LSTM memproses input dalam urutan terbalik. Setelah kedua LSTM selesai memproses, output terakhir dari forward LSTM (timestep terakhir) dan output terakhir dari backward LSTM (timestep pertama dari urutan terbalik) diambil. Kedua output ini kemudian digabungkan berdasarkan merge_mode

```

def forward(self, x):
    forward_output = self.forward_lstm.forward(x)

```

```

backward_output = self.backward_lstm.forward(x)

forward_last = forward_output[:, -1, :]
backward_last = backward_output[:, 0, :]

if self.merge_mode == 'concat':
    output = np.concatenate([forward_last,
backward_last], axis=1)
elif self.merge_mode == 'sum':
    output = forward_last + backward_last
elif self.merge_mode == 'avg':
    output = (forward_last + backward_last) / 2
else:
    raise ValueError(f"Unsupported merge_mode:
{self.merge_mode}")

return output

```

f. Kelas LSTMModelFromScratch

LSTMModelFromScratch merupakan kelas utama dalam implementasi mandiri LSTM yang bertugas merekonstruksi model LSTM dari model Keras yang telah dilatih, menjadi implementasi forward propagation manual. Kelas ini dirancang untuk membaca bobot-bobot dari model yang telah dilatih dengan *library* Keras dan membangun ulang arsitektur jaringan menggunakan layer-layer yang telah diimplementasikan sendiri.

Pada tahap inisialisasi, kelas ini akan memuat model Keras dari path yang diberikan, lalu secara otomatis membangun layer-layer yang sesuai melalui metode `_build_layers()`.

Fungsi `_get_activation_name()` berfungsi sebagai utility function untuk mengekstrak nama fungsi aktivasi dari layer Keras.

Fungsi `_build_layers()` merupakan inti dari proses rekonstruksi model. Metode ini bekerja dengan mengiterasi setiap layer dalam model Keras dan mengkonversinya ke dalam bentuk implementasi manual yang sesuai: Fungsi `forward()` mengimplementasikan proses forward propagation secara menyeluruh dengan menerima input, lalu melewatkannya secara berurutan ke setiap layer yang telah dibangun.

Fungsi `predict()` melengkapi fungsionalitas model dengan mengubah output probabilitas menjadi prediksi kelas melalui operasi argmax.

Fungsi `debug_model_structure()` disediakan sebagai alat bantu analisis struktur model Keras.

Fungsi `compare_implementation()` berfungsi untuk melakukan komparasi antara hasil dari keras dan hasil dari scratch

```

class LSTMModelFromScratch:
    def __init__(self, keras_model_path):
        self.model =
keras.models.load_model(keras_model_path)
        self.layers = []
        self._build_layers()

    def _get_activation_name(self, activation_func):
        if hasattr(activation_func, '__name__'):
            return activation_func.__name__
        elif hasattr(activation_func, 'name'):
            return activation_func.name
        else:
            return
        str(activation_func).split('.')[ -1].split(' ') [0]

    def _build_layers(self):
        print("Building LSTM layers from Keras model...")
        print(f"Model has {len(self.model.layers)} "
layers")

        layer_idx = 0
        for i, layer in enumerate(self.model.layers):
            layer_type = layer.__class__.__name__
            print(f"Layer {i}: {layer_type} - "
{layer.name}")

            if layer_type == 'Embedding':
                weights = layer.get_weights()
                if len(weights) > 0:
                    embedding_weights = weights[0]
                    print(f" Embedding weights shape: "
{embedding_weights.shape})

                self.layers.append(EmbeddingLayer(embedding_weights))
                layer_idx += 1

            elif layer_type == 'Bidirectional':
                print(f" Processing Bidirectional
layer...")
                all_weights = layer.get_weights()
                print(f" Total weights in bidirectional
layer: {len(all_weights)}")

                if len(all_weights) == 6:
                    forward_weights = all_weights[:3]
                    backward_weights = all_weights[3:6]
                elif len(all_weights) == 4:
                    forward_weights = all_weights[:2]
                    backward_weights = all_weights[2:4]
                else:
                    raise ValueError(f"Unexpected number
of weights in bidirectional layer: {len(all_weights)}")

                print(f" Forward weights shapes:
{[w.shape for w in forward_weights]}")

```

```

        print(f" Backward weights shapes:
{[w.shape for w in backward_weights]})")

        merge_mode = 'concat'
        if hasattr(layer, 'merge_mode'):
            merge_mode = layer.merge_mode

    self.layers.append(BidirectionalLSTMLayer(forward_weights,
backward_weights, merge_mode))
    layer_idx += 1

    elif layer_type == 'LSTM':
        weights = layer.get_weights()
        print(f" LSTM weights shapes: {[w.shape
for w in weights]})"

        return_seq = getattr(layer,
'return_sequences', False)
        go_back = getattr(layer, 'go_backwards',
False)

        self.layers.append(LSTMLayer(weights,
return_sequences=return_seq, go_backwards=go_back))
        layer_idx += 1

    elif layer_type == 'Dense':
        weights = layer.get_weights()
        if len(weights) == 0:
            print(f" Warning: Dense layer has no
weights!")
            continue

        dense_weights = weights[0]
        dense_bias = weights[1] if len(weights) >
1 else None

        print(f" Dense weights shape:
{dense_weights.shape}")
        if dense_bias is not None:
            print(f" Dense bias shape:
{dense_bias.shape}")

        activation =
self._get_activation_name(layer.activation)
        print(f" Dense activation: {activation}")

    self.layers.append(DenseLayer(dense_weights, dense_bias,
activation=activation))
    layer_idx += 1

    elif layer_type == 'Dropout':
        print(" Skipping Dropout layer (inference
mode)")
        continue

```

```

        else:
            print(f" Warning: Unknown layer type
{layer_type}, skipping...")
            continue

        print(f"Built {len(self.layers)} functional
layers")

    def forward(self, x):
        print(f"Input shape: {x.shape}")
        current_output = x

        for i, layer in enumerate(self.layers):
            print(f"Processing layer {i}:
{type(layer).__name__}")
            current_output = layer.forward(current_output)
            print(f" Output shape:
{current_output.shape}")

        return current_output

    def predict(self, x):
        probabilities = self.forward(x)
        return np.argmax(probabilities, axis=1)

def debug_lstm_model_structure(keras_model_path):
    print("==== LSTM MODEL STRUCTURE DEBUG ====")

    model = keras.models.load_model(keras_model_path)
    print(f"Model type: {type(model)}")
    print(f"Number of layers: {len(model.layers)}")
    print()

    for i, layer in enumerate(model.layers):
        print(f"Layer {i}: {layer.__class__.__name__}")
        print(f" Name: {layer.name}")
        print(f" Input shape: {layer.input_shape if
hasattr(layer, 'input_shape') else 'N/A'}")
        print(f" Output shape: {layer.output_shape if
hasattr(layer, 'output_shape') else 'N/A'}")

        weights = layer.get_weights()
        print(f" Weights count: {len(weights)}")
        for j, w in enumerate(weights):
            print(f"     Weight {j} shape: {w.shape}")

        # Print layer-specific attributes
        if hasattr(layer, 'activation'):
            print(f" Activation: {layer.activation}")
        if hasattr(layer, 'return_sequences'):
            print(f" Return sequences:
{layer.return_sequences}")
            if hasattr(layer, 'go_backwards'):
                print(f" Go backwards: {layer.go_backwards}")

```

```

        if hasattr(layer, 'merge_mode'):
            print(f"  Merge mode: {layer.merge_mode}")
        if hasattr(layer, 'units'):
            print(f"  Units: {layer.units}")

    print()

def compare_lstm_implementations(keras_model_path,
test_data, test_labels, max_samples=100):
    print("==== LSTM IMPLEMENTATION COMPARISON ===")

    if hasattr(test_data, 'numpy'):
        test_data = test_data.numpy()
    if hasattr(test_labels, 'numpy'):
        test_labels = test_labels.numpy()

    test_data = np.array(test_data)
    test_labels = np.array(test_labels)

    if len(test_data) > max_samples:
        indices = np.random.choice(len(test_data),
max_samples, replace=False)
        test_data_subset = test_data[indices]
        test_labels_subset = test_labels[indices]
    else:
        test_data_subset = test_data
        test_labels_subset = test_labels

    print(f"Using {len(test_data_subset)} samples for
comparison")

    keras_model =
keras.models.load_model(keras_model_path)
scratch_model = LSTMModelFromScratch(keras_model_path)

    print("\nGetting Keras predictions...")
    keras_predictions =
keras_model.predict(test_data_subset, verbose=0)
    keras_classes = np.argmax(keras_predictions, axis=1)

    print("\nGetting from-scratch predictions...")
    batch_size = 8
    scratch_predictions_list = []

    for i in range(0, len(test_data_subset), batch_size):
        end_idx = min(i + batch_size,
len(test_data_subset))
        batch = test_data_subset[i:end_idx]
        print(f"  Batch {i//batch_size +
1}/{(len(test_data_subset)-1)//batch_size + 1}")

    import sys
    from io import StringIO
    old_stdout = sys.stdout
    sys.stdout = StringIO()

```

```

try:
    batch_pred = scratch_model.forward(batch)
    scratch_predictions_list.append(batch_pred)
finally:
    sys.stdout = old_stdout

scratch_predictions =
np.vstack(scratch_predictions_list)
scratch_classes = np.argmax(scratch_predictions,
axis=1)

print(f"\nKeras predictions shape:
{keras_predictions.shape}")
print(f"Scratch predictions shape:
{scratch_predictions.shape}")

from sklearn.metrics import f1_score, accuracy_score

keras_f1 = f1_score(test_labels_subset, keras_classes,
average='macro')
scratch_f1 = f1_score(test_labels_subset,
scratch_classes, average='macro')

keras_acc = accuracy_score(test_labels_subset,
keras_classes)
scratch_acc = accuracy_score(test_labels_subset,
scratch_classes)

prediction_agreement = np.mean(keras_classes ==
scratch_classes)
max_diff = np.max(np.abs(keras_predictions -
scratch_predictions))
mean_diff = np.mean(np.abs(keras_predictions -
scratch_predictions))

print(f"\n===== LSTM RESULTS
=====")
print(f"Keras Model:")
print(f"  Accuracy: {keras_acc:.4f}")
print(f"  Macro F1: {keras_f1:.4f}")
print(f"\nFrom-Scratch Model:")
print(f"  Accuracy: {scratch_acc:.4f}")
print(f"  Macro F1: {scratch_f1:.4f}")
print(f"\nPrediction Agreement:
{prediction_agreement:.4f}")
print(f"Max Absolute Difference: {max_diff:.6f}")
print(f"Mean Absolute Difference: {mean_diff:.6f}")

return {
'keras_f1': keras_f1,
'scratch_f1': scratch_f1,
'prediction_agreement': prediction_agreement,
'max_diff': max_diff,
'mean_diff': mean_diff
}

```

2.2. Hasil Pengujian

2.2.1. Hasil Pengujian CNN

2.2.1.1. Compare model keras vs manual

Compare antara CNN keras dan CNN manual dilakukan dengan membuat model keras lalu disimpan dalam format .keras. Setelah itu file .keras ini akan dimuat lagi untuk diambil weightnya, setelah itu dilakukan proses forward menggunakan CNN manual / from scratch dengan batch size data (200). Lalu bandingkan nilai f1 score nya

Konfigurasi model CNN keras nya yaitu:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
re_lu (ReLU)	(None, 32, 32, 32)	0
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
re_lu_1 (ReLU)	(None, 32, 32, 32)	0
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
re_lu_2 (ReLU)	(None, 16, 16, 64)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
re_lu_3 (ReLU)	(None, 16, 16, 64)	0
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
re_lu_4 (ReLU)	(None, 8, 8, 128)	0
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147,584
re_lu_5 (ReLU)	(None, 8, 8, 128)	0
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 521)	1,067,529
re_lu_6 (ReLU)	(None, 521)	0
dense_1 (Dense)	(None, 10)	5,220

Lalu didapatkan hasil seperti ini

```
Keras (on subset 200) - Accuracy: 0.8650, Macro F1: 0.8601
Manual Model Prediction Time for Subset: 219.0255 seconds
Manual (on subset 200, loaded weights) - Accuracy: 0.8650, Macro F1: 0.8601
```

Dapat disimpulkan bahwa untuk simple model LSTM, menghasilkan nilai f1-score yang sangat akurat

2.2.1.2. Pengaruh jumlah layer konvolusi

Parameter kontrol yang digunakan adalah :

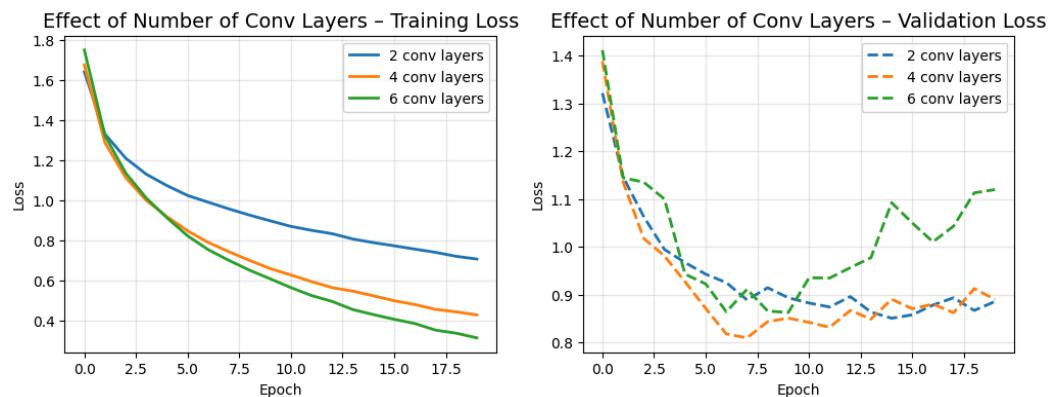
Inisialisasi Bobot : Xavier Uniform Initialization

Fungsi Aktivasi : ReLU

Fungsi loss : Categorical Cross Entropy

Batch size : 64
 Learning rate : 0.001
 Epoch maksimal : 20
 Fungsi Aktivasi untuk layer terakhir adalah softmax

Pada percobaan ini, pengujian dilakukan sebanyak 3 kali dengan variasi 2, 4, dan 6 convolution layer. Berikut merupakan grafik training loss dan validation loss hasil pengujian.



Gambar 2.2.1.1.1 Grafik Training Loss dan Validation Loss Eksperimen Jumlah Layer

Hasil F1-Scores:

1. 2 convolution layer: 0.7101
2. 4 convolution layer: 0.7429
3. 6 convolution layer: 0.7265

Pada percobaan ini terlihat bahwa penambahan jumlah convolution layer dari 2 ke 4 dan 6 secara konsisten menurunkan training loss lebih cepat dan mencapai nilai lebih rendah (6 layer terbaik), namun untuk validation loss model 6 layer cenderung mengalami kenaikan setelah epoch ke-6 menunjukkan kemungkinan overfitting, sedangkan model 4 layer memberikan keseimbangan terbaik dengan validation loss terendah dan paling stabil di sekitar epoch 7–19 dibandingkan model 2 layer yang relatif lebih tinggi dan fluktuatif.

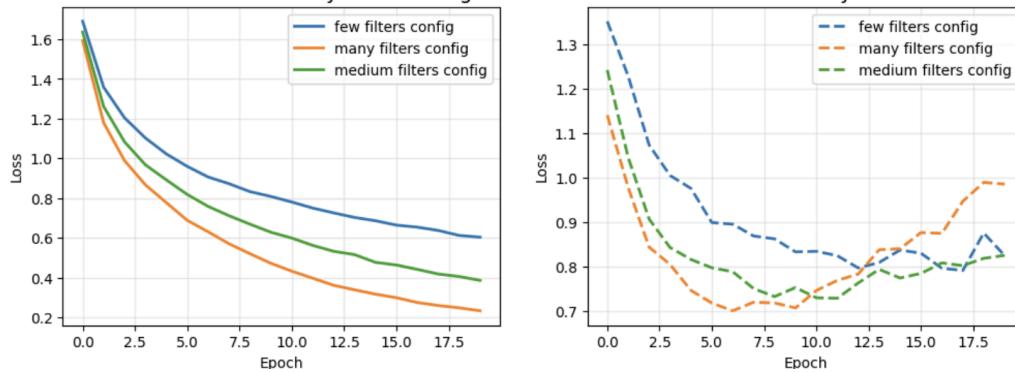
2.2.1.3. Pengaruh jumlah filter

Parameter kontrol yang digunakan adalah :

Inisialisasi Bobot : Xavier Uniform Initialization
 Fungsi Aktivasi : ReLU
 Fungsi loss : Categorical Cross Entropy
 Batch size : 64
 Learning rate : 0.001
 Epoch maksimal : 20

Fungsi Aktivasi untuk layer terakhir adalah softmax

Pada percobaan ini, pengujian dilakukan sebanyak 3 kali dengan variasi sedikit filter (16, 32, 64), medium filter (32, 64, 128), dan banyak filter (64, 128, 256). Berikut merupakan grafik training loss dan validation loss hasil pengujian.



Gambar 2.2.1.2.1 Grafik Training Loss dan Validation Loss Eksperimen Jumlah Filter

Hasil F1-Scores:

1. Sedikit filter: 0.7336
2. Medium filter: 0.7560
3. Banyak filter: 0.7710

Grafik training loss menunjukkan bahwa konfigurasi banyak filters menurunkan loss paling cepat dan mencapai nilai terendah, diikuti medium dan sedikit. Pada validation loss, sedikit filter tetap tinggi dan fluktuatif dan banyak filter mulai naik kembali sekitar epoch ke-8–10 (tanda overfitting ringan), sedangkan medium filter paling stabil dengan loss terendah sebelum sedikit naik setelah epoch 10.

2.2.1.4. Pengaruh ukuran filter

Parameter kontrol yang digunakan adalah :

Inisialisasi Bobot : Xavier Uniform Initialization

Fungsi Aktivasi : ReLU

Fungsi loss : Categorical Cross Entropy

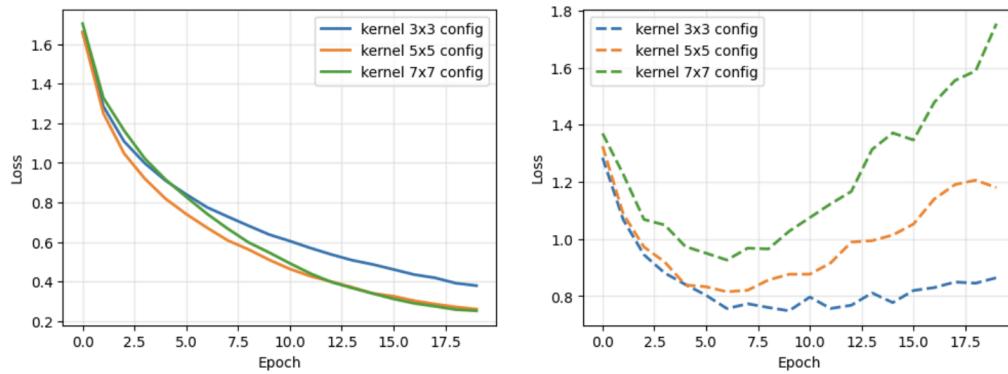
Batch size : 64

Learning rate : 0.001

Epoch maksimal : 20

Fungsi Aktivasi untuk layer terakhir adalah softmax

Pada percobaan ini, pengujian dilakukan sebanyak 3 kali dengan variasi 3×3 filter, 5×5 filter, dan 7×7 filter. Berikut merupakan grafik training loss dan validation loss hasil pengujian.



Gambar 2.2.1.3.1 Grafik Training Loss dan Validation Loss Eksperimen Ukuran Filter

Hasil F1-Scores:

1. Filter 3×3 : 0.7531
2. Filter 5×5 : 0.7256
3. Filter 7×7 : 0.6796

Grafik training loss menunjukkan bahwa kernel 5×5 dan 7×7 sedikit lebih cepat menurunkan loss dibanding 3×3 , tetapi pada validation loss kernel 3×3 memberikan nilai terendah dan paling stabil setelah epoch ke-5, kernel 5×5 berada di tengah dengan adanya kenaikan setelah epoch ke-7, sedangkan kernel 7×7 mulai overfitting menyebabkan validation loss naik drastis.

2.2.1.5. Pengaruh jenis pooling

Parameter kontrol yang digunakan adalah :

Inisialisasi Bobot : Xavier Uniform Initialization

Fungsi Aktivasi : ReLU

Fungsi loss : Categorical Cross Entropy

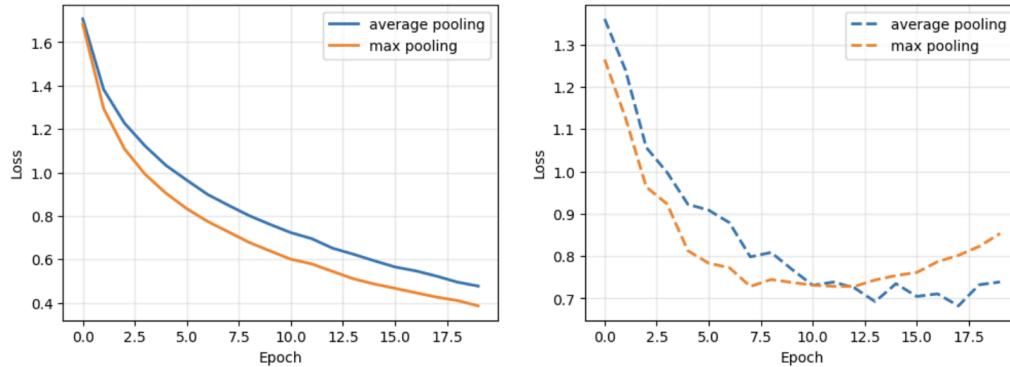
Batch size : 64

Learning rate : 0.001

Epoch maksimal : 20

Fungsi Aktivasi untuk layer terakhir adalah softmax

Pada percobaan ini, pengujian dilakukan sebanyak 2 kali dengan variasi max pooling dan average pooling. Berikut merupakan grafik training loss dan validation loss hasil pengujian.



Gambar 2.2.1.4.1 Grafik Training Loss dan Validation Loss Eksperimen Jenis Pooling

Hasil F1-Scores:

1. Max pooling: 0.7494
2. Average pooling: 0.7681

Pada percobaan ini model dengan max pooling menurunkan training loss lebih cepat dan mencapai nilai lebih rendah dibanding average pooling, serta pada validation loss max pooling juga memberikan kurva yang lebih rendah dan stabil—tidak ada lonjakan overfitting signifikan—sedangkan average pooling menunjukkan loss yang lebih tinggi dan sedikit fluktuasi setelah epoch ke-10, mengindikasikan bahwa max pooling lebih efektif dalam mengekstrak fitur dan menjaga generalisasi.

2.2.2. Hasil Pengujian RNN

2.2.2.1. Compare antara model keras dan manual

Compare antara RNN keras dan RNN manual dilakukan dengan membuat model keras lalu disimpan dalam format .h5. Setelah itu file .h5 ini akan dimuat lagi untuk diambil weightnya, setelah itu dilakukan proses forward menggunakan RNN manual / from scratch dengan batch size data (400). Lalu bandingkan nilai f1 score nya

Konfigurasi model RNN keras nya yaitu:

1 Embedding layer, 1 bidirectional layer, 1 dropout layer, 1 dense layer dengan fungsi aktivasi softmax

```

-----  

Layer (type)           Output Shape        Param #  

-----  

embedding (Embedding)  (None, None, 128)    363008  

bidirectional (Bidirection (None, 128)    24704  

al)  

dropout (Dropout)      (None, 128)         0  

dense (Dense)          (None, 3)            387  

-----  

Total params: 388099 (1.48 MB)  

Trainable params: 388099 (1.48 MB)  

Non-trainable params: 0 (0.00 Byte)
-----
```

Lalu didapatkan hasil seperti ini

```

===== RESULTS =====
Keras Model:  

  Accuracy: 0.5175  

  Macro F1: 0.4942  

From-Scratch Model:  

  Accuracy: 0.5175  

  Macro F1: 0.4942
```

Dapat disimpulkan bahwa untuk simple model RNN, menghasilkan nilai f1-score yang sangat akurat

2.2.2.2 Pengaruh jumlah layer RNN

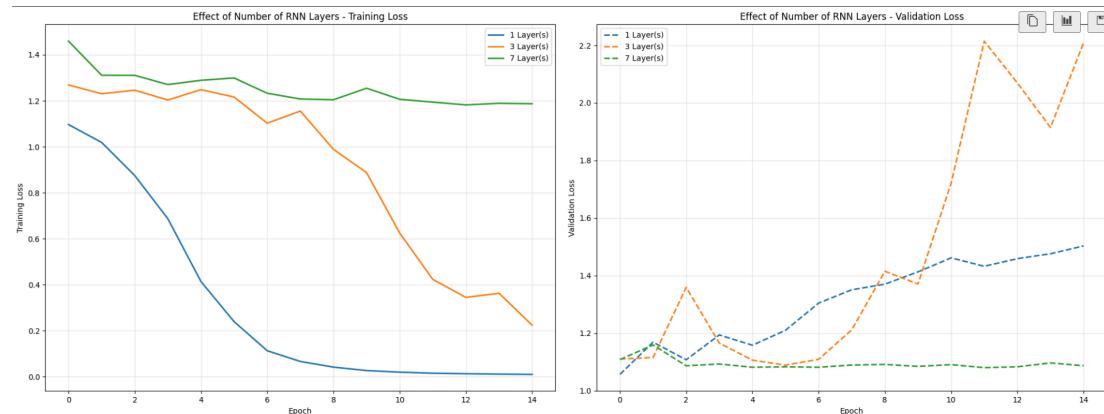
Cell per layer: 64

Bidirectional: False

Aktivasi Dense layer: Softmax

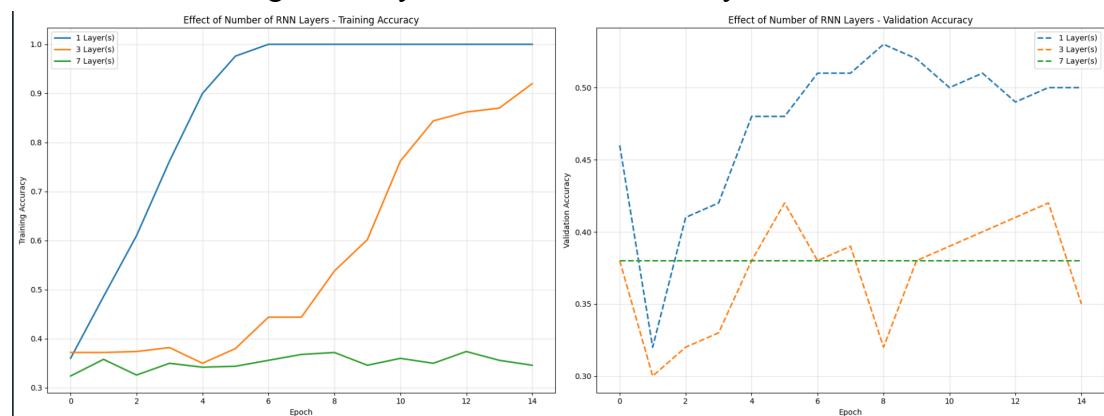
Pengujian dilakukan 3 kali dengan variasi 1 layer, 3 layer, dan 7 layer. Dimana 1 layer direpresentasikan dengan warna biru, 3 layer direpresentasikan dengan warna orange, dan 7 layer direpresentasikan dengan warna hijau.

Hasil Grafik Training Loss dan Validation Loss



Gambar 2.2.2.1.1 Grafik Training Loss dan Validation Loss Eksperimen layer

Hasil Grafik Training Accuracy dan Validation Accuracy



Gambar 2.2.2.1.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen layer

Hasil F1 Score:

- 1 Layer : 0.4092
- 3 Layer : 0.3571
- 7 Layer : 0.1827

Eksperimen menunjukkan bahwa peningkatan jumlah layer RNN malah menurunkan performa model. Bisa dilihat dari model satu layer mencapai F1 score tertinggi (0.4092), diikuti tiga layer (0.3571), dan tujuh layer dengan skor terendah (0.1827). Hal ini mengindikasikan bahwa arsitektur dengan layer yang makin banyak tidak meningkatkan kemampuan generalisasi, bahkan memperburuk performa.

Grafik training dan validation loss menunjukkan bahwa model satu layer mengalami overfitting training loss mendekati nol, sementara validation loss naik.

Model tiga layer menunjukkan ketidakstabilan tinggi dimana validation lossnya sangat tinggi bahkan paling tinggi dari antara yang lainnya, dan model tujuh layer tidak banyak berubah, menandakan adanya vanishing gradient problem.

Pada grafik accuracy, model satu layer memiliki training accuracy mendekati 100%, namun validation accuracy naik turun sehingga ada indikasi overfitting. Model tiga layer menunjukkan ketidakstabilan pada validation accuracy, sedangkan model tujuh layer menunjukkan performa stuck di sekitar 37% untuk validation accuracy. Hal ini menunjukan bahwa model dengan ayer yang lebih banyak mengalami kesulitan dalam propagasi gradien yang efektif, sehingga tidak mampu belajar secara optimal.

Kesimpulan:

Eksperimen ini menunjukkan bahwa pada klasifikasi teks dengan dataset yang digunakan, arsitektur RNN sederhana dengan satu layer memberikan hasil terbaik dibandingkan model dengan arsitektur yang memiliki layer lebih banyak. Penurunan performa disebabkan oleh dua faktor utama yaitu overfitting pada model yang lebih sederhana, dan vanishing gradient problem pada model yang lebih kompleks. Meskipun mengalami overfitting, model satu layer masih mampu memberikan hasil yang efektif. Kompleksitas model yang berlebihan tidak selalu menghasilkan performa yang lebih baik.

2.2.2.3. Pengaruh banyak cell RNN per layer

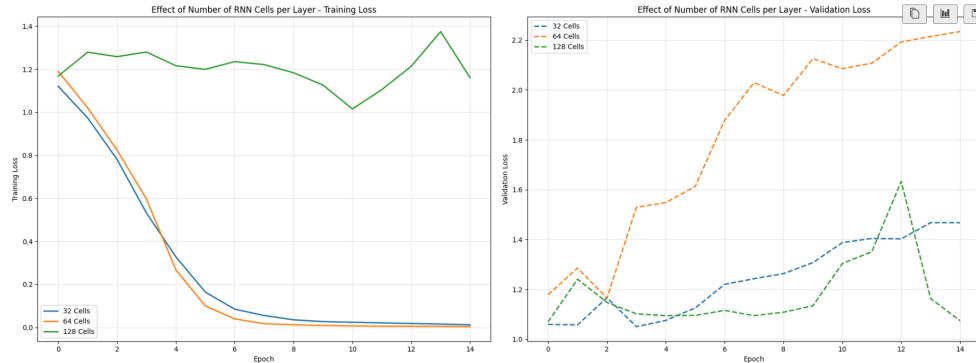
Jumlah Layer: 1

Bidirectional: False

Aktivasi Dense layer: Softmax

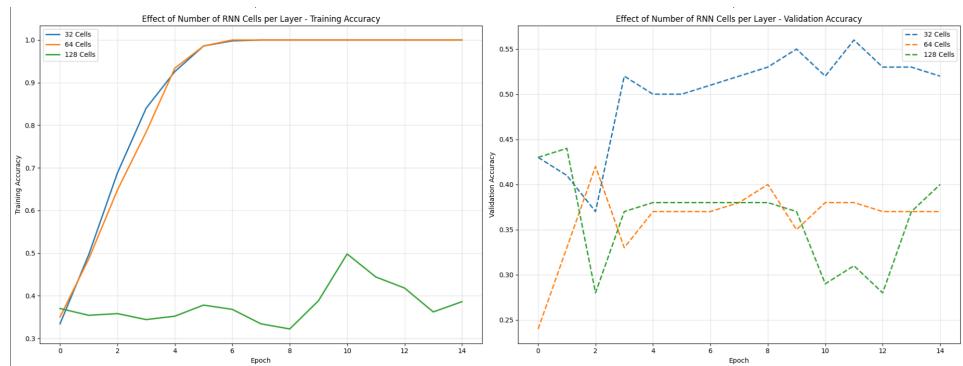
Pengujian dilakukan 3 kali dengan variasi cell sebanyak 32 cell, 64 cell, dan 128 cell. Dimana 32 cell direpresentasikan dengan warna biru, 64 cell direpresentasikan dengan warna orange, dan 128 cell direpresentasikan dengan warna hijau.

Hasil Grafik Training Loss dan Validation Loss



Gambar 2.2.2.2.1 Grafik Training Loss dan Validation Loss Eksperimen Arah

Hasil Grafik Training Accuracy dan Validation Accuracy



Gambar 2.2.2.2.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Arah

Hasil F1 Score:

- 32 Cells : 0.4937
- 64 Cells : 0.4614
- 128 Cells : 0.2657

Berdasarkan hasil eksperimen pengaruh jumlah cell RNN, Model dengan 32 cell memberikan nilai F1-score sebesar 0.4937, model dengan 64 cells memberikan nilai F1-score sebesar 0.4614, dan model dengan 128 cells memberikan nilai F1-score 0.2657. Hasil eksperimen menunjukkan adanya penurunan performa secara konsisten seiring dengan peningkatan jumlah cell. Dapat ditarik informasi bahwa peningkatan jumlah cell tidak selalu berbanding lurus dengan peningkatan kinerja, khususnya pada dataset.

Model dengan 32 cells dan 64 cells menunjukkan pola pembelajaran yang lebih stabil dibandingkan 128 cells. Penurunan training loss model 32 cells dan 64 cells yang cepat menuju 0 menandakan kemampuan belajar yang baik. Model dengan 128 cell menunjukkan pola pembelajaran yang tidak stabil ditandai dengan nilai training loss nya yang naik turun hingga tidak bisa mencapai konvergensi yang

baik. Hal ini menunjukkan bahwa model 128 cells kompleksitas arsitekturnya terlalu tinggi untuk diuji dengan dataset .

Dapat dilihat juga bahwa model dengan jumlah cell terlalu besar terlihat mengalami kesulitan untuk menghasilkan hasil yang efektif, ditandai dengan training accuracy model dengan 128 cell yang stagnan dan loss yang fluktuatif dengan aneh dan tidak menentu.

Kesimpulan

Model dengan 32 cells terbukti memberikan performa yang paling baik, model ini mampu untuk menggeneralisasi pola dari data. Sementara itu, peningkatan jumlah cell hingga 64 menyebabkan overfitting yang lebih parah, dan peningkatan hingga 128 justru mengarah pada underfitting. Model yang terlalu sederhana mungkin gagal menangkap pola kompleks, namun model yang terlalu besar juga berisiko overfit.

2.2.2.4. Pengaruh jenis layer RNN berdasarkan arah

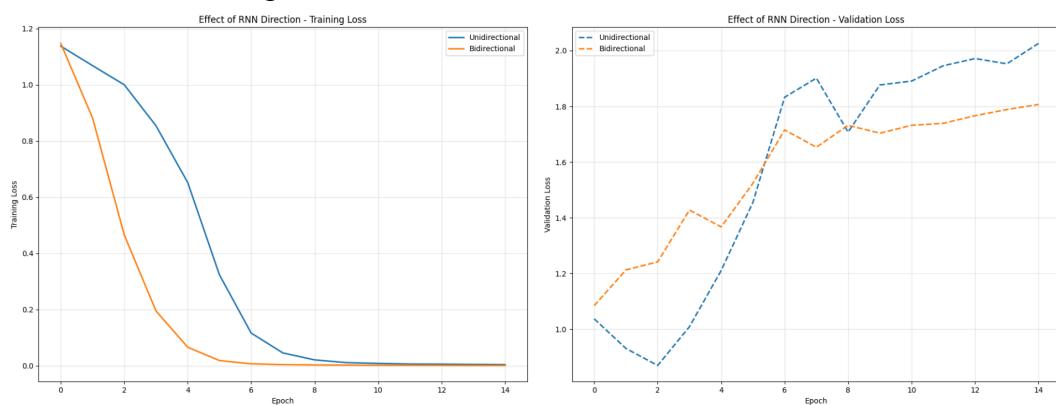
Jumlah Layer: 1

Cell per layer: 64

Aktivasi Dense layer: Softmax

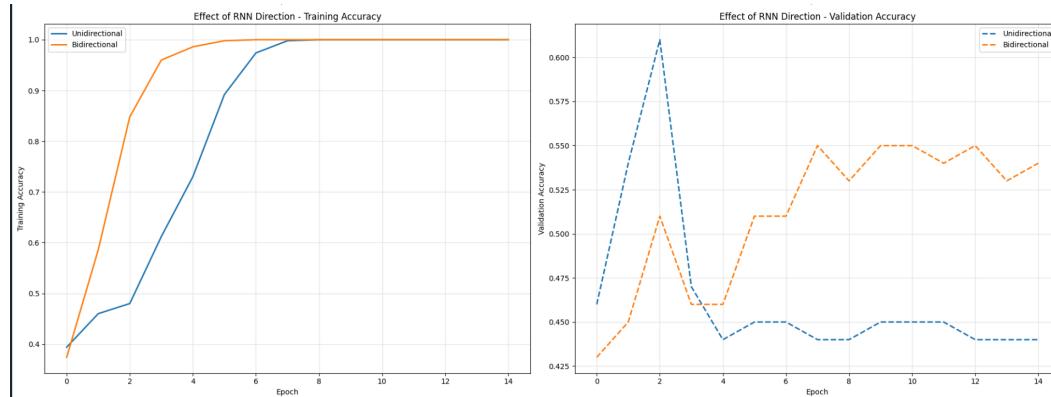
Pengujian dilakukan 2 kali dengan variasi bidirectional dan unidirectional. Dimana unidirectional direpresentasikan dengan warna biru, dan bidirectional direpresentasikan dengan warna orange.

Hasil Grafik Training Loss dan Validation Loss



Gambar 2.2.2.3.1 Grafik Training Loss dan Validation Loss Eksperimen Arah

Hasil Grafik Training Accuracy dan Validation Accuracy



Gambar 2.2.2.3.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Arah Hasil F1 Score:

- Unidirectional: 0.4963
- Bidirectional: 0.5270

Hasil menunjukkan bahwa bidirectional RNN mencapai F1 score 0.5270, sedikit lebih tinggi dibandingkan unidirectional RNN dengan F1 score 0.4963. Selisih skor sebesar 0.0307 meskipun tidak besar, tapi menunjukkan bahwa bidirectional lebih menghasilkan hasil yang baik

Keduanya mengalami overfitting setelah epoch ke 6, namun model unidirectional ada peningkatan validation loss yang lebih drastis hingga sekitar 2.0, sedangkan model bidirectional hanya meningkat secara bertahap hingga sekitar 1.8. Hal ini menunjukkan bahwa meskipun bidirectional RNN memiliki kompleksitas lebih tinggi, ia mampu mengontrol overfitting dengan lebih baik.

Pola accuracy juga menunjukkan perbedaan yang signifikan. Model bidirectional mencapai training accuracy 100% lebih cepat di sekitar epoch ke-3, sementara unidirectional mencapainya di sekitar epoch ke-6. Pada validation accuracy, model unidirectional menunjukkan kenaikan ekstrim pada awal epoch sampai 60%.

Kesimpulan

Dapat dilihat bahwa bidirectional RNN lebih baik dibandingkan unidirectional, baik dalam hal performa akhir maupun stabilitas training nya. Kemampuan bidirectional dalam menangkap informasi dari dua arah memberikan hasil yang lebih baik. Meskipun peningkatan performa tidak signifikan secara kuantitatif, model bidirectional RNN tetap menjadi pilihan yang lebih praktis untuk tugas klasifikasi teks.

2.2.3. Hasil Pengujian LSTM

2.2.3.1. Compare antara model keras dan manual

Compare antara LSTM keras dan LSTM manual dilakukan dengan membuat model keras lalu disimpan dalam format .h5. Setelah itu file .h5 ini akan dimuat lagi untuk diambil weightnya, setelah itu dilakukan proses forward menggunakan LSTM manual / from scratch dengan batch size data (400). Lalu bandingkan nilai f1 score nya

Konfigurasi model LSTM keras nya yaitu:

1 Embedding layer, 1 bidirectional layer, 1 dropout layer, 1 dense layer dengan fungsi aktivasi softmax

Layer (type)	Output Shape	Param #
<hr/>		
embedding_layer (Embedding	(None, None, 50)	141800
)		
bidirectional (Bidirection	(None, 64)	21248
al)		
dropout_layer (Dropout)	(None, 64)	0
output_layer (Dense)	(None, 3)	195
<hr/>		
Total params: 163243 (637.67 KB)		
Trainable params: 163243 (637.67 KB)		
Non-trainable params: 0 (0.00 Byte)		
<hr/>		

Lalu didapatkan hasil seperti ini

===== LSTM RESULTS =====	
Keras Model:	
Accuracy: 0.7200	
Macro F1: 0.6988	
From-Scratch Model:	
Accuracy: 0.7200	
Macro F1: 0.6988	

Dapat disimpulkan bahwa untuk simple model LSTM, menghasilkan nilai f1-score yang sangat akurat

2.2.3.2. Pengaruh jumlah layer LSTM

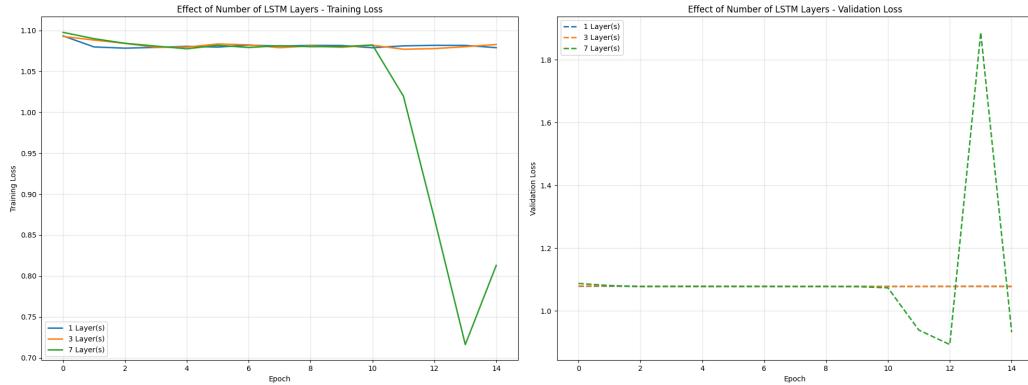
Cell per layer: 64

Bidirectional: False

Aktivasi Dense layer: Softmax

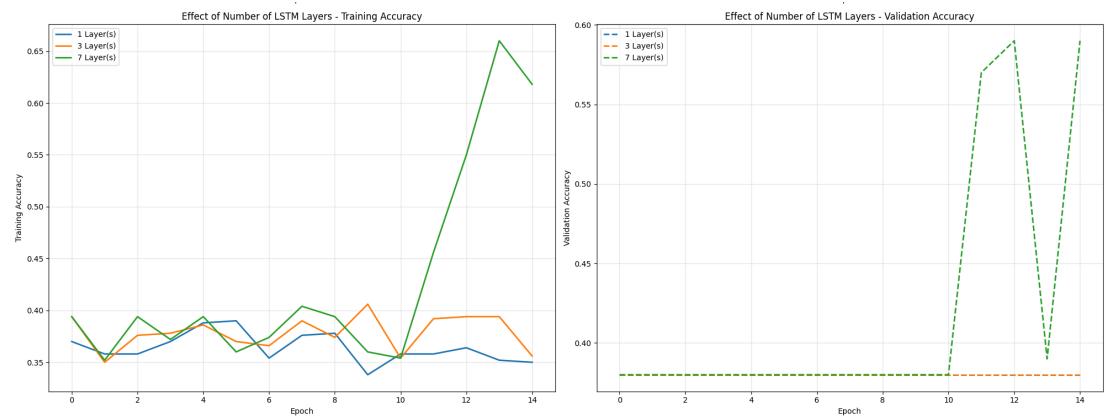
Pengujian dilakukan 3 kali dengan variasi 1 layer, 3 layer, dan 7 layer. Dimana 1 layer direpresentasikan dengan warna biru, 3 layer direpresentasikan dengan warna orange, dan 7 layer direpresentasikan dengan warna hijau.

Hasil Grafik Training Loss dan Validation Loss



Gambar 2.2.3.1.1 Grafik Training Accuracy dan Validation Accuracy Eksperimen Layer

Hasil Grafik Training Accuracy dan Validation Accuracy



Gambar 2.2.3.1.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Layer

Hasil F1 Score:

- 1 Layer : 0.1844
- 3 Layer : 0.1827
- 7 Layer : 0.4261

Berdasarkan hasil eksperimen pengaruh jumlah layer LSTM, Model dengan 1 layer memberikan nilai F1-score sebesar 0.1844, model dengan 3 layer memberikan nilai F1-score sebesar 0.1827, dan model dengan 7 layer memberikan nilai F1-score 0.4261.

Model dengan 1 layer dan 3 layer tidak memiliki perbedaan yang begitu berarti. Hal ini menandakan penambahan hingga 3 layer tidak memberikan peningkatan akurasi model yang dilatih dengan dataset NusaX ini. Model dengan 7 layer LSTM menunjukkan pola yang sangat berbeda. Model dengan 7 layer menunjukkan tanda-tanda overfitting yang jelas setelah epoch ke-10 karena nilai training loss turun drastis dari sekitar 1.1 menjadi sekitar 0.7, sementara validation loss justru melonjak hingga di atas 1.8.

Penambahan jumlah layer LSTM tidak selalu menghasilkan peningkatan performa yang konsisten. Model dengan 1-3 layer menunjukkan stabilitas dan generalisasi yang lebih baik, sementara model dengan 7 layer meskipun memiliki F1-score lebih tinggi, mengalami overfitting yang signifikan.

2.2.3.3. Pengaruh banyak cell LSTM per layer

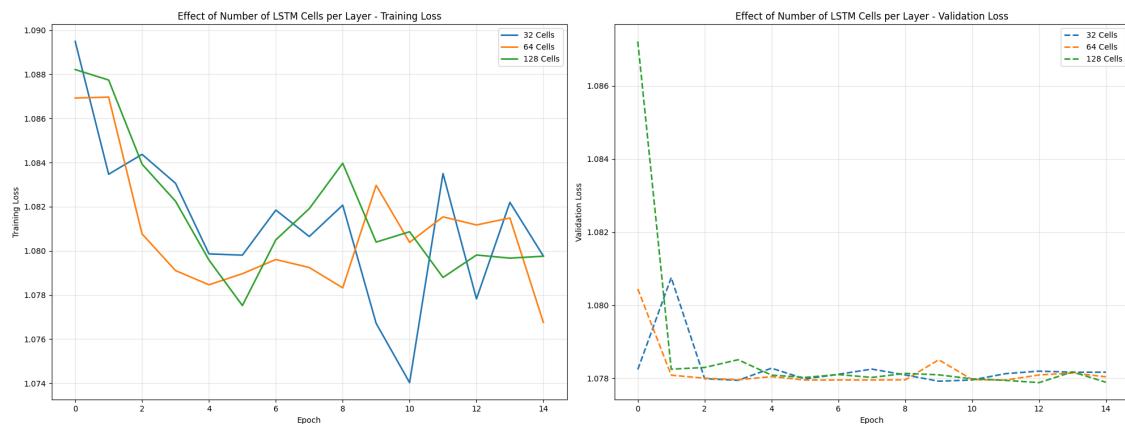
Jumlah Layer: 1

Bidirectional: False

Aktivasi Dense layer: Softmax

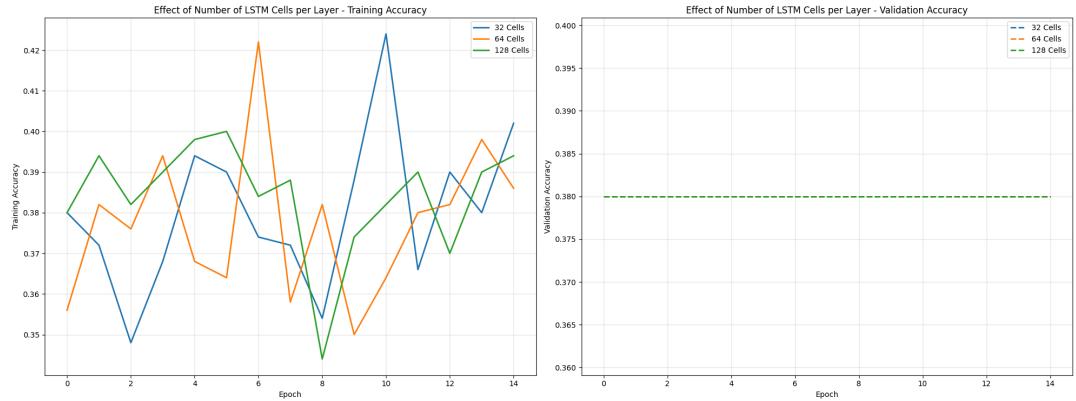
Pengujian dilakukan 3 kali dengan variasi cell sebanyak 32 cell, 64 cell, dan 128 cell. Dimana 32 cell direpresentasikan dengan warna biru, 64 cell direpresentasikan dengan warna orange, dan 128 cell direpresentasikan dengan warna hijau.

Hasil Grafik Training Loss dan Validation Loss



Gambar 2.2.3.2.1 Grafik Training Accuracy dan Validation Accuracy Eksperimen Cell

Hasil Grafik Training Accuracy dan Validation Accuracy



Gambar 2.2.3.2.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Cell

Hasil F1 Score:

- 32 Cells: 0.1844
- 64 Cells: 0.1844
- 128 Cells: 0.1844

Berdasarkan hasil eksperimen pengaruh banyak cell LSTM per layer, dapat diamati bahwa penambahan jumlah cell tidak memberikan pengaruh yang signifikan pada akurasi model. Bahkan didapati bahwa training loss sangat fluktuatif yang mengindikasikan ketidakseimbangan pelatihan model. Oleh karena itu, untuk parameter kontrol dan dataset yang digunakan, konfigurasi 32 cell lebih efisien untuk pelatihan model.

2.2.3.4. Pengaruh jenis layer LSTM berdasarkan arah

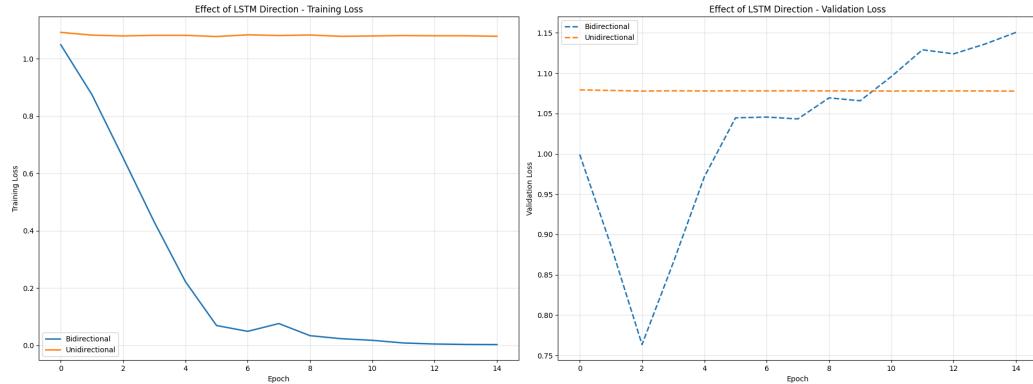
Jumlah Layer: 1

Cell per layer: 64

Aktivasi Dense layer: Softmax

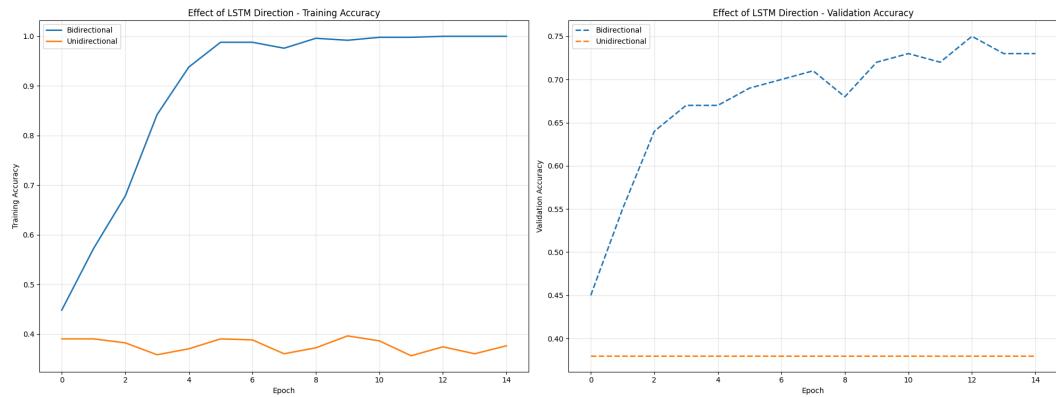
Pengujian dilakukan 2 kali dengan variasi bidirectional dan unidirectional. Dimana unidirectional direpresentasikan dengan warna biru, dan bidirectional direpresentasikan dengan warna orange.

Hasil Grafik Training Loss dan Validation Loss



Gambar 2.2.3.3.1 Grafik Training Accuracy dan Validation Accuracy Eksperimen Arah

Hasil Grafik Training Accuracy dan Validation Accuracy



Gambar 2.2.3.3.2 Grafik Training Accuracy dan Validation Accuracy Eksperimen Arah

Hasil F1 Score:

- Bidirectional: 0.7401
- Unidirectional: 0.1844

Berdasarkan hasil eksperimen pengaruh arah layer LSTM , terdapat perbedaan performa yang sangat signifikan antara kedua konfigurasi. Model LSTM bidirectional memiliki F1-score sebesar 0.7401. Perbedaan nilainya signifikan bila dibandingkan dengan F1-score unidirectional yang bernilai 0.1844.

Bidirectional LSTM yang mampu memproses informasi dari dua arah membuat model dapat menangkap konteks yang lebih komprehensif dibandingkan dengan model unidirectional.

BAB III

Kesimpulan dan Saran

3.1. Kesimpulan

Berdasarkan hasil pengujian yang telah dilakukan pada tiga arsitektur neural network yaitu CNN, RNN, dan LSTM, dapat disimpulkan beberapa temuan penting terkait pengaruh hyperparameter terhadap performa model. Pada eksperimen CNN dengan dataset CIFAR-10, arsitektur dengan 4 layer konvolusi terbukti memberikan keseimbangan terbaik, dimana model dengan 2 layer kurang kompleks untuk menangkap fitur yang memadai sementara 6 layer cenderung mengalami overfitting. Konfigurasi jumlah filter menunjukkan bahwa meskipun banyak filter menghasilkan F1-score tertinggi, ukuran filter 3×3 terbukti paling efektif dibandingkan 5×5 dan 7×7 karena memberikan stabilitas yang lebih baik dan mengurangi risiko overfitting. Dari segi pooling, average pooling memberikan hasil yang sedikit lebih baik dibandingkan max pooling, meskipun max pooling menunjukkan konvergensi yang lebih cepat.

Eksperimen pada Simple RNN menunjukkan bahwa arsitektur sederhana justru memberikan performa yang lebih baik. Model dengan 1 layer RNN mencapai F1-score tertinggi (0.4092), dimana penambahan layer justru menurunkan performa karena vanishing gradient problem dan overfitting yang semakin parah. Konfigurasi dengan 32 cell menghasilkan hasil terbaik, menunjukkan bahwa kompleksitas yang berlebihan tidak selalu menghasilkan peningkatan performa. Bidirectional RNN menunjukkan keunggulan dibandingkan unidirectional yang menunjukkan pentingnya kemampuan model untuk menangkap konteks dari dua arah.

Pada eksperimen LSTM, ditemukan bahwa model dengan 7 layer memberikan F1-score tertinggi (0.4261) namun mengalami overfitting yang signifikan. Ketiga variasi jumlah cell (32, 64, 128) menghasilkan F1-score yang identik (0.1844), menunjukkan bahwa pada dataset ini penambahan cell tidak memberikan dampak signifikan terhadap performa dengan unidirectional. Bidirectional LSTM menunjukkan keunggulan yang sangat signifikan dibandingkan unidirectional. Ini menunjukkan pentingnya konteks dua arah dalam analisis sentimen.

Secara keseluruhan, implementasi forward propagation from scratch untuk ketiga arsitektur berhasil menghasilkan akurasi yang sangat tinggi dan konsisten dengan implementasi Keras, membuktikan keberhasilan pemahaman dan implementasi algoritma. Perbandingan antar arsitektur menunjukkan bahwa untuk tugas klasifikasi gambar, CNN memberikan performa yang sangat baik dengan F1-score mencapai 0.77, sementara untuk klasifikasi teks, LSTM bidirectional memberikan hasil terbaik dengan F1-score 0.74, diikuti RNN bidirectional dengan 0.53.

3.2. Saran

Untuk pengembangan lebih lanjut model, beberapa saran yang penulis berikan untuk mengoptimasi lebih lanjut hasil model yang dibangun adalah :

- Lakukan analisis data yang lebih mendalam untuk memahami karakteristik dataset, implementasikan teknik preprocessing yang lebih canggih seperti word embedding pre trained, dan pertimbangkan balancing dataset untuk mengatasi class imbalance.
- Perkuat evaluasi dengan penggunaan cross-validation untuk evaluasi yang lebih robust, implementasikan multiple metrics selain F1-score (precision, recall, accuracy), dan lakukan analisis error untuk memahami kegagalan model.
- Meningkatkan optimasi melalui eksperimen dengan optimizer lain seperti RMSprop, implementasikan learning rate finder untuk menentukan learning rate optimal, dan gunakan ensemble methods untuk meningkatkan performa.

LAMPIRAN

Pranala GitHub: https://github.com/salsbiila/Tubes2ML_G40_SBB

Pembagian Tugas

Nama	Tugas
Andhita Naura Hariyanto	LSTM
Salsabiila	CNN
Keanu Amadius Gonza Wrahatno	RNN

REFERENSI

https://d2l.ai/chapter_recurrent-modern/lstm.html
https://d2l.ai/chapter_recurrent-modern/deep-rnn.html
https://d2l.ai/chapter_recurrent-modern/bi-rnn.html
https://d2l.ai/chapter_recurrent-neural-networks/index.html
https://d2l.ai/chapter_convolutional-neural-networks/index.html
<https://numpy.org/doc/2.1/reference/generated/numpy.einsum.html>