

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



Kelompok 65

13522079	Emery Fathan Zwageri
13522089	Abdul Rafi Radityo Hutomo
13522097	Ellijah Darrelshane Suryanegara

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025

DESKRIPSI PERSOALAN

A. Tujuan

Tugas Besar II pada kuliah IF3270 Pembelajaran Mesin agar peserta kuliah mendapatkan wawasan tentang bagaimana cara mengimplementasikan Convolutional Neural Network (CNN) dan Recurrent Neural Network. Pada tugas ini, peserta kuliah akan ditugaskan untuk mengimplementasikan modul forward propagation CNN dan RNN from scratch.

B. Deskripsi Persoalan

Pada Tugas Besar 2 IF3270 Pembelajaran Mesin, akan dilakukan implementasi dan analisis *Convolutional Neural Network* (CNN) dan *Recurrent Neural Network* (RNN), termasuk *Simple RNN* dan *Long-Short Term Memory* (LSTM).

Untuk setiap jenis jaringan (CNN, Simple RNN, dan LSTM), hal-hal berikut perlu dilakukan:

1. Pelatihan Model dengan Keras:

- Model klasifikasi dilatih menggunakan *library* Keras dengan dataset dan arsitektur *layer* minimal yang telah ditentukan untuk masing-masing jenis jaringan.
- *Sparse Categorical Crossentropy* digunakan sebagai *loss function* dan Adam sebagai *optimizer*.
- Bobot (*weights*) hasil pelatihan disimpan.

2. Analisis *Hyperparameter*:

- Serangkaian eksperimen dilakukan dengan memvariasikan *hyperparameter* spesifik untuk tiap jenis jaringan.
- Hasil prediksi dibandingkan menggunakan metrik *macro f1-score*.
- Grafik *training loss* dan *validation loss* dibandingkan untuk setiap variasi.
- Kesimpulan mengenai pengaruh tiap *hyperparameter* terhadap kinerja model diberikan.

3. Implementasi *Forward Propagation from Scratch*:

- Modul *forward propagation* dibuat secara modular (per *layer*) dari awal (tanpa *library high-level* seperti Keras, hanya *library* matematika seperti NumPy).

- Modul ini harus dapat membaca model dan bobot yang telah dilatih dengan Keras.
- Implementasi diuji dengan membandingkan hasilnya dengan *output* dari Keras pada data tes, menggunakan metrik *macro f1-score*.

Untuk masing-masing jaringan sendiri, terdapat spesifikasi berikut

Convolutional Neural Network (CNN)

- Dataset: CIFAR-10, yang perlu dibagi menjadi data latih, validasi, dan uji.
- Layer Minimal: Conv2D, Pooling (Max/Average), Flatten/Global Pooling, Dense.
- Analisis Hyperparameter:
 - Jumlah *layer* konvolusi (3 variasi).
 - Banyak *filter* per *layer* konvolusi (3 variasi).
 - Ukuran *filter* per *layer* konvolusi (3 variasi).
 - Jenis *pooling layer* (Max vs. Average, 2 variasi).

Simple Recurrent Neural Network (Simple RNN)

- Dataset: NusaX-Sentiment (Bahasa Indonesia).
- Preprocessing: Tokenisasi dengan TextVectorization Keras dan *Embedding layer* Keras.
- Layer: Embedding, Bidirectional/Unidirectional RNN, Dropout, Dense.
- Analisis Hyperparameter:
 - Jumlah *layer* RNN (3 variasi).
 - Banyak *cell* RNN per *layer* (3 variasi).
 - Jenis *layer* RNN berdasarkan arah (Bidirectional vs. Unidirectional, 2 variasi).

Long-Short Term Memory (LSTM) Network

- Dataset: NusaX-Sentiment (Bahasa Indonesia).
- Preprocessing: Tokenisasi dengan TextVectorization Keras dan *Embedding layer* Keras.
- Layer: Embedding, Bidirectional/Unidirectional LSTM, Dropout, Dense.
- Analisis Hyperparameter:
 - Jumlah *layer* LSTM (3 variasi).
 - Banyak *cell* LSTM per *layer* (3 variasi).
 - Jenis *layer* LSTM berdasarkan arah (Bidirectional vs. Unidirectional, 2 variasi).

PEMBAHASAN

A. Penjelasan Forward Propagation

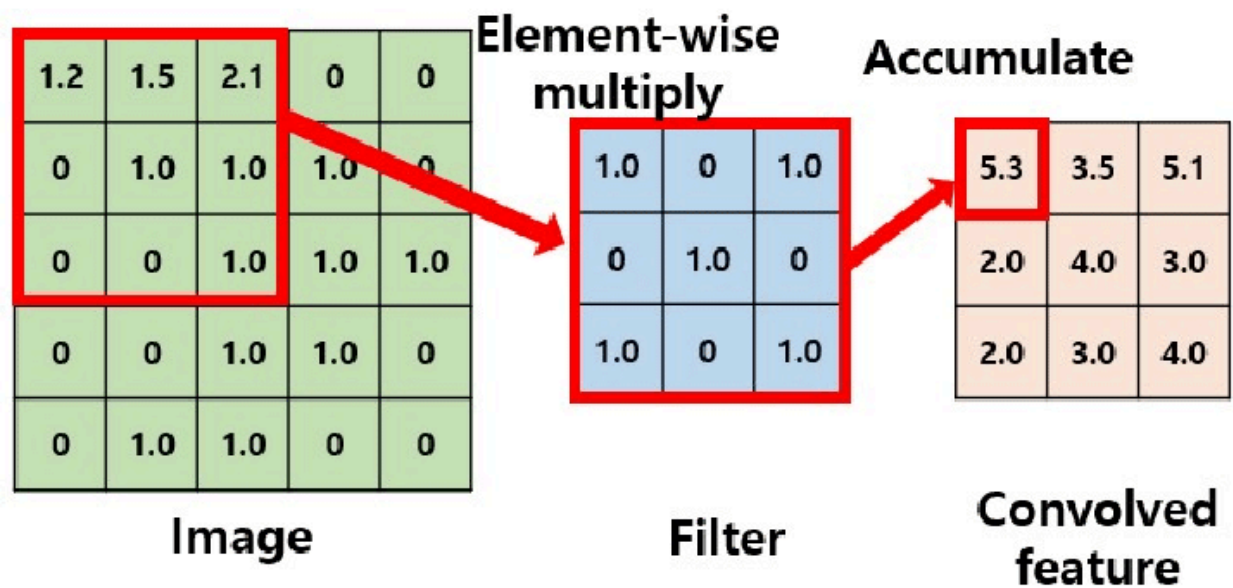
a. Penjelasan Forward Propagation CNN

Forward propagation merupakan proses utama dalam pelatihan dan inferensi pada jaringan saraf, termasuk pada arsitektur Convolutional Neural Network (CNN). Proses ini menggambarkan bagaimana input diproses dalam jaringan hingga menghasilkan output akhir, seperti prediksi kelas pada tugas klasifikasi. Pada CNN, forward propagation dilakukan melalui serangkaian layer yang dirancang untuk mengekstraksi fitur spasial dari input dan menerjemahkannya menjadi representasi yang berguna untuk pengambilan keputusan.

1. Input Layer

Proses forward propagation dimulai dari input layer, yang menerima data mentah. Input biasanya berupa gambar dalam bentuk tensor tiga dimensi dengan ukuran height x width x channel.

2. Convolutional Layer



Lapisan konvolusi merupakan komponen inti dari CNN. Pada tahap ini, sejumlah filter atau kernel diterapkan pada input untuk mendeteksi pola lokal seperti tepi, sudut, atau tekstur. Setiap filter memiliki ukuran tetap, misalnya 3×3 atau 5×5 , dan digeser sebanyak stride, melintasi seluruh area input. Pada setiap posisi, dilakukan operasi perkalian elemen-per-elemen (dot product) antara kernel dan area input yang ditutupi oleh kernel, kemudian dijumlahkan dan ditambahkan dengan bias. Operasi ini menghasilkan sebuah nilai output pada titik tersebut. Hasil dari keseluruhan proses ini adalah sebuah **feature map** (peta fitur).

3. Activation Function

Setelah hasil konvolusi diperoleh, fungsi aktivasi seperti ReLU (Rectified Linear Unit) diterapkan secara elemen-per-elemen. Tujuan dari fungsi aktivasi ini adalah untuk memperkenalkan non-linearitas ke dalam jaringan, sehingga jaringan dapat mempelajari representasi yang lebih kompleks dari data input.

4. Pooling Layer

Setelah aktivasi, CNN biasanya menggunakan layer pooling untuk mereduksi dimensi spasial dari feature map. Jenis pooling yang umum adalah max pooling, yang membagi

feature map ke dalam region kecil (misalnya 2×2) dan hanya mengambil nilai maksimum dari tiap region tersebut. Jenis pooling lain yang umum juga adalah average pooling yang mengambil nilai average dari suatu region pada data. Dengan demikian, pooling mengurangi jumlah parameter dan komputasi, sekaligus membantu jaringan menjadi lebih tahan terhadap translasi kecil dalam input.

5. Flatten Layer

Setelah semua layer konvolusi dan pooling selesai, output akhir yang berbentuk tensor tiga dimensi akan diubah menjadi vektor satu dimensi melalui proses yang disebut flattening. Proses ini diperlukan karena layer-layer berikutnya (fully connected) memerlukan input dalam bentuk vektor, seperti pada jaringan saraf konvensional (fully connected neural networks).

6. Fully Connected Layer

Layer dense menerima vektor hasil flatten dan menghubungkannya ke sejumlah neuron. Setiap neuron pada layer ini terhubung ke seluruh input, dan menghasilkan aktivasi berdasarkan kombinasi linier dari input yang diterima, kemudian diteruskan ke fungsi aktivasi. Layer ini bertugas untuk menggabungkan fitur-fitur lokal yang diekstraksi oleh layer konvolusi menjadi representasi global untuk klasifikasi.

7. Output Layer (Softmax)

$$\frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Pada tugas klasifikasi, output layer biasanya menggunakan fungsi aktivasi softmax, yang mengubah output neuron menjadi probabilitas untuk masing-masing kelas. Fungsi softmax menghitung eksponensial dari setiap nilai output dan membaginya dengan

jumlah seluruh eksponensial dari nilai-nilai tersebut, sehingga menghasilkan vektor probabilitas yang berjumlah total 1. Output ini menunjukkan keyakinan model terhadap masing-masing kelas.

b. Penjelasan Forward Propagation Simple RNN

Proses forward propagation pada Simple RNN dapat dijelaskan melalui serangkaian langkah yang terjadi secara rekursif di sepanjang urutan waktu. Misalkan kita memiliki input sekuens $x = \{x_1, x_2, x_3, \dots, x_T\}$, di mana T adalah panjang sekuens. Pada setiap langkah waktu t , jaringan menerima input x_t dan hidden state sebelumnya h_{t-1} , lalu menghasilkan hidden state baru h_t yang merepresentasikan ingatan atau informasi gabungan dari input saat ini dan masa lalu. Hidden state ini juga dapat digunakan untuk menghasilkan output y_t jika diperlukan.

Secara matematis, forward propagation pada Simple RNN di setiap timestep didefinisikan sebagai:

$$h_t = \tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b_h)$$

$$y_t = W_{hy} \cdot h_t + b_y$$

Dengan:

- x_t adalah input pada waktu t ,
- h_{t-1} adalah hidden state dari waktu sebelumnya,
- W_{xh} adalah bobot antara input dan hidden layer,
- W_{hh} adalah bobot antara hidden state sebelumnya dan hidden state saat ini,
- W_{hy} adalah bobot dari hidden state ke output (jika output dihasilkan setiap langkah),

- b_h dan b_y adalah bias,
- \tanh adalah fungsi aktivasi non-linear yang menjaga nilai hidden state tetap dalam rentang $[-1, 1]$.

c. Penjelasan Forward Propagation LSTM

Forward Propagation pada LSTM mirip dengan RNN, namun LSTM menggunakan 4 gate yaitu forget, input, cell dan output gate. Input gate berfungsi sebagai penentu banyaknya informasi masuk dari input yang akan ditambahkan pada cell *untuk timestep* sekarang, forget gate menentukan seberapa banyak cell sebelumnya dilupakan, output gate menentukan apa yang keluar dari hidden state.

$$\begin{aligned}
 i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
 f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
 g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
 o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
 c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
 h_t &= o_t \odot \tanh(c_t)
 \end{aligned}$$

Di mana \mathbf{h}_t adalah *hidden state* pada waktu ke t , \mathbf{c}_t adalah *cell state* pada waktu ke t , \mathbf{x}_t adalah input saat waktu ke t , \mathbf{h}_{t-1} adalah *hidden state* lapisan pada waktu ke $t-1$ atau hidden state awal saat $t-1$ adalah 0, dan \mathbf{i}_t , \mathbf{f}_t , \mathbf{g}_t , \mathbf{o}_t adalah input, forget, cell, and output gates, masing masing. σ adalah *sigmoid function* dan \odot adalah *Hadamard product*.

B. Deskripsi Kelas dan Implementasi

a. CNN

1. Conv2D

Deskripsi	Layer implementasi stage konvolusi
-----------	------------------------------------

Nama Fungsi	set_weights(self, weights: np.ndarray, biases: np.ndarray)
-------------	--

Deskripsi Fungsi	Mengset nilai weight dan bias
Source Code	<pre>def set_weights(self, weights: np.ndarray, biases: np.ndarray): """Set weights from trained Keras model""" self.weights = weights # Shape: (kernel_h, kernel_w, input_channels, output_channels) self.biases = biases # Shape: (output_channels,)</pre>

Nama Fungsi	<code>__call__(self, x: np.ndarray)</code>
Deskripsi Fungsi	Melakukan forward propagation untuk stage konvolusi
Source Code	<pre>def __call__(self, x: np.ndarray) -> np.ndarray: """Forward pass through convolution layer""" batch_size, height, width, channels = x.shape kernel_h, kernel_w, in_channels, out_channels = self.weights.shape # Calculate output dimensions if self.padding == 'same': out_h = height // self.strides out_w = width // self.strides pad_h = max(0, (out_h - 1) * self.strides + kernel_h - height) pad_w = max(0, (out_w - 1) * self.strides + kernel_w - width) pad_top = pad_h // 2 pad_bottom = pad_h - pad_top pad_left = pad_w // 2 pad_right = pad_w - pad_left x_padded = np.pad(x, ((0, 0), (pad_top, pad_bottom), (pad_left, pad_right), (0, 0)), mode='constant') else: out_h = (height - kernel_h) // self.strides + 1 out_w = (width - kernel_w) // self.strides + 1 x_padded = x output = np.zeros((batch_size, out_h, out_w, out_channels)) # Perform convolution for i in range(out_h): for j in range(out_w): h_start = i * self.strides h_end = h_start + kernel_h w_start = j * self.strides w_end = w_start + kernel_w x_slice = x_padded[:, h_start:h_end, w_start:w_end, :] for k in range(out_channels): output[:, i, j, k] = np.sum(x_slice * self.weights[:, :, :, k], axis=(1, 2, 3)) + self.biases[k] return output</pre>

2. MaxPool2D

Deskripsi	Layer implementasi max pooling
-----------	--------------------------------

Nama Fungsi	<code>__call__(self, x: np.ndarray)</code>
Deskripsi Fungsi	Melakukan max pool dari data input x

Source Code	<pre> def __call__(self, x: np.ndarray) -> np.ndarray: batch_size, height, width, channels = x.shape out_h = (height - self.pool_size) // self.stride + 1 out_w = (width - self.pool_size) // self.stride + 1 output = np.zeros((batch_size, out_h, out_w, channels)) for i in range(out_h): for j in range(out_w): h_start = i * self.stride h_end = h_start + self.pool_size w_start = j * self.stride w_end = w_start + self.pool_size output[:, i, j, :] = np.max(x[:, h_start:h_end, w_start:w_end, :], axis=(1, 2)) return output </pre>
-------------	---

3. AvgPool2D

Deskripsi	Layer implementasi average pool
-----------	---------------------------------

Nama Fungsi	<code>__call__(self, x: np.ndarray)</code>
Deskripsi Fungsi	Melakukan average pool dari data input x
Source Code	<pre> def __call__(self, x: np.ndarray) -> np.ndarray: batch_size, height, width, channels = x.shape out_h = (height - self.pool_size) // self.stride + 1 out_w = (width - self.pool_size) // self.stride + 1 output = np.zeros((batch_size, out_h, out_w, channels)) for i in range(out_h): for j in range(out_w): h_start = i * self.stride h_end = h_start + self.pool_size w_start = j * self.stride w_end = w_start + self.pool_size output[:, i, j, :] = np.mean(x[:, h_start:h_end, w_start:w_end, :], axis=(1, 2)) return output </pre>

4. Flatten

Deskripsi	Layer implementasi flatten
-----------	----------------------------

Nama Fungsi	<code>__call__(self, x: np.ndarray)</code>
Deskripsi Fungsi	Melakukan flatten dari data input x

Source Code	<pre>class Flatten: """Flatten layer implementation""" def __call__(self, x: np.ndarray) -> np.ndarray: batch_size = x.shape[0] return x.reshape(batch_size, -1)</pre>
-------------	---

5. ReLU

Deskripsi	Layer implementasi ReLU
Nama Fungsi	<code>__call__(self, x: np.ndarray)</code>
Deskripsi Fungsi	Melakukan ReLU untuk setiap data pada x
Source Code	<pre>class ReLU: """ReLU activation function""" def __call__(self, x: np.ndarray) -> np.ndarray: return np.maximum(0, x)</pre>

6. CNNFromScratch

Deskripsi	Layer implementasi CNN from scratch
Nama Fungsi	<code>add_layer(self, layer)</code>
Deskripsi Fungsi	menaambah layer ke model CNN tersebut
Source Code	<pre>def add_layer(self, layer): self.layers.append(layer)</pre>
Nama Fungsi	<code>load_keras_weights(self, keras_model)</code>
Deskripsi Fungsi	Melakukan load weight dari model keras

Source Code	<pre> def load_keras_weights(self, keras_model): """Load weights from trained Keras model""" keras_layers = [layer for layer in keras_model.layers if len(layer.get_weights()) > 0] # Separate different layer types scratch_conv_layers = [layer for layer in self.layers if isinstance(layer, Conv2D)] scratch_dense_layers = [layer for layer in self.layers if hasattr(layer, 'set_weights') and not isinstance(layer, Conv2D)] conv_idx = 0 dense_idx = 0 for keras_layer in keras_layers: weights = keras_layer.get_weights() if isinstance(keras_layer, tf.keras.layers.Conv2D): if conv_idx < len(scratch_conv_layers): scratch_conv_layers[conv_idx].set_weights(weights[0], weights[1]) conv_idx += 1 elif isinstance(keras_layer, tf.keras.layers.Dense): if dense_idx < len(scratch_dense_layers): scratch_dense_layers[dense_idx].set_weights(weights[0], weights[1]) dense_idx += 1 </pre>
-------------	--

Nama Fungsi	predict(self, x: np.ndarray)
Deskripsi Fungsi	Melakukan forward propagation terhadap semua layer
Source Code	<pre> def predict(self, x: np.ndarray) -> np.ndarray: """Forward propagation through the network""" current_input = x is_value_object = False # Track whether we're working with Value objects for i, layer in enumerate(self.layers): if isinstance(layer, Conv2D): # Conv2D layers work with numpy arrays if is_value_object: # Convert Value back to numpy array current_input = current_input.data is_value_object = False current_input = layer(current_input) # Apply ReLU after convolution current_input = np.maximum(0, current_input) elif hasattr(layer, '__call__') and hasattr(layer, 'W'): # expects Value objects if not is_value_object: # Convert numpy array to Value for Dense layer computation current_input = Value(current_input) is_value_object = True current_input = layer(current_input) # current_input is now a Value object else: # Regular layers (pooling, flatten, etc.) work with numpy arrays if is_value_object: # Convert Value back to numpy array current_input = current_input.data is_value_object = False current_input = layer(current_input) # Ensure to return a numpy array if is_value_object: return current_input.data else: return current_input </pre>

Nama Fungsi	<code>calculate_flattened_size(self, input_shape, conv_layers, kernel_sizes, pooling_type)</code>
Deskripsi Fungsi	Menghitung ukuran output setelah flatten
Source Code	<pre>def calculate_flattened_size(self, input_shape, conv_layers, kernel_sizes, pooling_type): """Calculate the flattened size after conv and pooling layers""" height, width, channels = input_shape for filters, kernel_size in zip(conv_layers, kernel_sizes): # Conv layer with 'same' padding doesn't change spatial dimensions # Pooling layer reduces by factor of 2 height = height // 2 width = width // 2 channels = filters return height * width * channels</pre>

b. Simple RNN

1. RNN

Nama Fungsi	<code>__init__(self, input_size: int, hidden_size: int, activation: Callable[[Value], Value] = tanh, weight_init_func=None)</code>
Deskripsi Fungsi	Melakukan inisiasi instance RNN
Source Code	<pre>def __init__(self, input_size: int, hidden_size: int, activation: Callable[[Value], Value] = tanh, weight_init_func=None): """ Initializes a Simple RNN Layer. Args: input_size (int): The number of expected features in the input x. hidden_size (int): The number of features in the hidden state h. activation (Callable[[Value], Value]): The activation function to use. Defaults to tanh. weight_init_func (Callable, optional): Function to initialize weights. Should take (n_inputs, n_outputs) and return np.ndarray. Defaults to a standard random initialization if None. """ self.input_size = input_size self.hidden_size = hidden_size self.activation = activation if weight_init_func is None: # Default initialization (glorot/xavier uniform) limit_xh = np.sqrt(6.0 / (input_size + hidden_size)) limit_hh = np.sqrt(6.0 / (hidden_size + hidden_size)) self.W_xh = Value(np.random.uniform(-limit_xh, limit_xh, (input_size, hidden_size))) # Weights for input x self.W_hh = Value(np.random.uniform(-limit_hh, limit_hh, (hidden_size, hidden_size))) # Weights for hidden state h self.b_h = Value(np.zeros((1, hidden_size))) # Bias for hidden state else: self.W_xh = Value(weight_init_func(input_size, hidden_size)) self.W_hh = Value(weight_init_func(hidden_size, hidden_size)) self.b_h = Value(np.zeros((1, hidden_size)))</pre>

Nama Fungsi	<code>parameters(self) -> List[Value]</code>
-------------	---

Deskripsi Fungsi	Mengembalikan daftar parameter pada layer
Source Code	<pre>def parameters(self) -> List[Value]: """Returns the list of parameters of the layer.""" return [self.W_xh, self.W_hh, self.b_h]</pre>
Nama Fungsi	set_weights(self, W_xh: np.ndarray, W_hh: np.ndarray, b_h: np.ndarray)
Deskripsi Fungsi	Menset weight model agar sesuai dengan weight yang ingin dimuat.
Source Code	<pre>def set_weights(self, W_xh: np.ndarray, W_hh: np.ndarray, b_h: np.ndarray): """ Sets the weights for the SimpleRNN layer. Keras kernel (W_xh) has shape (input_dim, units). Keras recurrent_kernel (W_hh) has shape (units, units). Keras bias (b_h) has shape (units,). Our W_xh shape: (input_size, hidden_size) Our W_hh shape: (hidden_size, hidden_size) Our b_h shape: (1, hidden_size) """ if W_xh.shape != self.W_xh.data.shape: raise ValueError(f"Expected W_xh shape {self.W_xh.data.shape} but got {W_xh.shape}") if W_hh.shape != self.W_hh.data.shape: raise ValueError(f"Expected W_hh shape {self.W_hh.data.shape} but got {W_hh.shape}") if b_h.shape != (self.b_h.data.shape[-1],): # Keras bias is 1D raise ValueError(f"Expected b_h shape {(self.b_h.data.shape[-1],)} but got {b_h.shape}") self.W_xh.data = W_xh self.W_hh.data = W_hh self.b_h.data = b_h.reshape(self.b_h.data.shape)</pre>
Nama Fungsi	__call__(self, x_sequence: Value, h_prev: Optional[Value] = None) -> Tuple[Value, Value]
Deskripsi Fungsi	<i>Forward propagation</i> RNN. Mengembalikan hidden state tiap time step, hidden state terakhir.

Source Code	<pre> def __call__(self, x_sequence: Value, h_prev: Optional[Value] = None) -> Tuple[Value, Value]: """ Performs the forward pass for a sequence of inputs. Args: x_sequence (Value): Input sequence. Data shape (batch_size, sequence_length, input_size). h_prev (Value, optional): Initial hidden state. Data shape (batch_size, hidden_size). Defaults to zeros if None. Returns: Tuple[Value, Value]: - outputs_value (Value): Hidden states for each time step. Data shape (batch_size, sequence_length, hidden_size). - h_t (Value): The last hidden state. Data shape (batch_size, hidden_size). """ batch_size, sequence_length, _ = x_sequence.data.shape if h_prev is None: h_prev = Value(np.zeros((batch_size, self.hidden_size))) outputs_list = [] for t in range(sequence_length): # Get input for the current time step: x_t # x_t data shape: (batch_size, input_size) x_t_data = x_sequence.data[:, t, :] if x_t_data.ndim == 1: # Ensure x_t_data is 2D for single feature input x_t_data = np.atleast_2d(x_t_data) if x_t_data.shape[0] != batch_size: # If batch_size is 1 and previous step made it (feature_size,) x_t_data = x_t_data.reshape(batch_size, -1) x_t = Value(x_t_data) # Hidden state calculation: h_t = activation(x_t @ W_xh + h_prev @ W_hh + b_h) term_xh = x_t.matmul(self.W_xh) term_hh = h_prev.matmul(self.W_hh) h_t = self.activation(term_xh + term_hh + self.b_h) outputs_list.append(h_t.data) # Store numpy data for stacking h_prev = h_t # Update h_prev for the next time step # Stack outputs along the sequence_length dimension # outputs_list contains numpy arrays of shape (batch_size, hidden_size) stacked_outputs_data = np.stack(outputs_list, axis=1) # (batch_size, sequence_length, hidden_size) outputs_value = Value(stacked_outputs_data) return outputs_value, h_t # h_t is the last hidden state Value object </pre>
-------------	---

c. LSTM

1. LSTM

Nama Fungsi	<code>__init__(self, input_size: int, hidden_size: int, weight_init_func=None)</code>
Deskripsi Fungsi	Melakukan inisiasi instance LSTM

Source Code	<pre> def __init__(self, input_size: int, hidden_size: int, weight_init_func=None): """ Initializes an LSTM Layer. Args: input_size (int): The number of expected features in the input x. hidden_size (int): The number of features in the hidden state h and cell state c. weight_init_func (Callable, optional): Function to initialize weights. Should take (n_inputs, n_outputs) and return np.ndarray. Defaults to a standard random initialization if None. """ self.input_size = input_size self.hidden_size = hidden_size if weight_init_func is None: # Default initialization (glorot/xavier uniform like) def init_weights(n_in, n_out): limit = np.sqrt(6.0 / (n_in + n_out)) return np.random.uniform(-limit, limit, (n_in, n_out)) self.W_f = Value(init_weights(input_size, hidden_size)) # Forget gate input weights self.U_f = Value(init_weights(hidden_size, hidden_size)) # Forget gate hidden weights self.b_f = Value(np.zeros((1, hidden_size))) # Forget gate bias self.W_i = Value(init_weights(input_size, hidden_size)) # Input gate input weights self.U_i = Value(init_weights(hidden_size, hidden_size)) # Input gate hidden weights self.b_i = Value(np.zeros((1, hidden_size))) # Input gate bias self.W_c = Value(init_weights(input_size, hidden_size)) # Cell state candidate input weights self.U_c = Value(init_weights(hidden_size, hidden_size)) # Cell state candidate hidden weights self.b_c = Value(np.zeros((1, hidden_size))) # Cell state candidate bias self.W_o = Value(init_weights(input_size, hidden_size)) # Output gate input weights self.U_o = Value(init_weights(hidden_size, hidden_size)) # Output gate hidden weights self.b_o = Value(np.zeros((1, hidden_size))) # Output gate bias else: self.W_f = Value(weight_init_func(input_size, hidden_size)) self.U_f = Value(weight_init_func(hidden_size, hidden_size)) self.b_f = Value(np.zeros((1, hidden_size))) self.W_i = Value(weight_init_func(input_size, hidden_size)) self.U_i = Value(weight_init_func(hidden_size, hidden_size)) self.b_i = Value(np.zeros((1, hidden_size))) self.W_c = Value(weight_init_func(input_size, hidden_size)) self.U_c = Value(weight_init_func(hidden_size, hidden_size)) self.b_c = Value(np.zeros((1, hidden_size))) self.W_o = Value(weight_init_func(input_size, hidden_size)) self.U_o = Value(weight_init_func(hidden_size, hidden_size)) self.b_o = Value(np.zeros((1, hidden_size))) </pre>
-------------	--

Nama Fungsi	parameters(self) -> List[Value]
Deskripsi Fungsi	Mengembalikan daftar parameter pada layer
Source Code	<pre> def parameters(self) -> List[Value]: """Returns the list of parameters of the layer.""" return [self.W_f, self.U_f, self.b_f, self.W_i, self.U_i, self.b_i, self.W_c, self.U_c, self.b_c, self.W_o, self.U_o, self.b_o,] </pre>
Nama Fungsi	set_weights(self, W_all: np.ndarray, U_all: np.ndarray, b_all: np.ndarray)

Deskripsi Fungsi	Set semua weight ke dalam LSTM
Source Code	<pre> def set_weights(self, W_all: np.ndarray, U_all: np.ndarray, b_all: np.ndarray): """ Sets the weights for the LSTM layer from Keras format. Keras kernel (W_all) has shape (input_dim, 4 * units). Keras recurrent_kernel (U_all) has shape (units, 4 * units). Keras bias (b_all) has shape (4 * units,). Order of gates in Keras is typically i, f, c, o. Our W_gate shape: (input_size, hidden_size) Our U_gate shape: (hidden_size, hidden_size) Our b_gate shape: (1, hidden_size) """ hidden_size = self.hidden_size input_size = self.input_size # from __init__ expected_W_all_shape = (input_size, 4 * hidden_size) if W_all.shape != expected_W_all_shape: raise ValueError(f"Expected W_all shape {expected_W_all_shape} but got {W_all.shape}") W_i_k, W_f_k, W_c_k, W_o_k = np.split(W_all, 4, axis=1) self.W_i.data = W_i_k self.W_f.data = W_f_k self.W_c.data = W_c_k self.W_o.data = W_o_k expected_U_all_shape = (hidden_size, 4 * hidden_size) if U_all.shape != expected_U_all_shape: raise ValueError(f"Expected U_all shape {expected_U_all_shape} but got {U_all.shape}") U_i_k, U_f_k, U_c_k, U_o_k = np.split(U_all, 4, axis=1) self.U_i.data = U_i_k self.U_f.data = U_f_k self.U_c.data = U_c_k self.U_o.data = U_o_k expected_b_all_shape = (4 * hidden_size,) if b_all.shape != expected_b_all_shape: raise ValueError(f"Expected b_all shape {expected_b_all_shape} but got {b_all.shape}") b_i_k, b_f_k, b_c_k, b_o_k = np.split(b_all, 4, axis=0) self.b_i.data = b_i_k.reshape(self.b_i.data.shape) self.b_f.data = b_f_k.reshape(self.b_f.data.shape) self.b_c.data = b_c_k.reshape(self.b_c.data.shape) self.b_o.data = b_o_k.reshape(self.b_o.data.shape) </pre>
Nama Fungsi	<code>__call__(self, x_sequence: Value, initial_states: Optional[Tuple[Value, Value]] = None) -> Tuple[Value, Tuple[Value, Value]]</code>
Deskripsi Fungsi	<i>Forward propagation</i> LSTM. Mengembalikan hidden state tiap time step, hidden state terakhir, dan cell state terakhir.

Source Code

```
def __call__(self, x_sequence: Value, initial_states: Optional[Tuple[Value, Value]] = None) -> Tuple[Value, Tuple[Value, Value]]:
    """
    Performs the forward pass for a sequence of inputs.

    Args:
        x_sequence (Value): Input sequence. Data shape (batch_size, sequence_length, input_size).
        initial_states (Tuple[Value, Value], optional): Tuple of (h_prev, c_prev).
            h_prev: Initial hidden state, data shape (batch_size, hidden_size).
            c_prev: Initial cell state, data shape (batch_size, hidden_size).
            Defaults to zeros if None.

    Returns:
        Tuple[Value, Tuple[Value, Value]]:
            - outputs_value (Value): Hidden states for each time step. Data shape (batch_size, sequence_length, hidden_size).
            - (h_t, c_t) (Tuple[Value, Value]): The last hidden state and cell state.
    """
    batch_size, sequence_length, _ = x_sequence.data.shape

    if initial_states is None:
        h_prev = Value(np.zeros((batch_size, self.hidden_size)))
        c_prev = Value(np.zeros((batch_size, self.hidden_size)))
    else:
        h_prev, c_prev = initial_states

    outputs_list = [] # To store numpy arrays of h_t for each time step

    for t in range(sequence_length):
        # Get input for the current time step: x_t
        # x_t data shape: (batch_size, input_size)
        x_t_data = x_sequence.data[:, t, :]
        if x_t_data.ndim == 1: # Ensure x_t_data is 2D
            x_t_data = np.atleast_2d(x_t_data)
        if x_t_data.shape[0] != batch_size: # if batch_size is 1 and previous step made it (feature_size,)
            x_t_data = x_t_data.reshape(batch_size, -1)
        x_t = Value(x_t_data)

        # Forget gate: f_t = sigmoid(x_t @ W_f + h_prev @ U_f + b_f)
        f_t = sigmoid(x_t.matmul(self.W_f) + h_prev.matmul(self.U_f) + self.b_f)

        # Input gate: i_t = sigmoid(x_t @ W_i + h_prev @ U_i + b_i)
        i_t = sigmoid(x_t.matmul(self.W_i) + h_prev.matmul(self.U_i) + self.b_i)

        # Candidate cell state: c_tilde_t = tanh(x_t @ W_c + h_prev @ U_c + b_c)
        c_tilde_t = tanh(x_t.matmul(self.W_c) + h_prev.matmul(self.U_c) + self.b_c)

        # Cell state: c_t = f_t * c_prev + i_t * c_tilde_t
        c_t = f_t * c_prev + i_t * c_tilde_t

        # Output gate: o_t = sigmoid(x_t @ W_o + h_prev @ U_o + b_o)
        o_t = sigmoid(x_t.matmul(self.W_o) + h_prev.matmul(self.U_o) + self.b_o)

        # Hidden state: h_t = o_t * tanh(c_t)
        h_t = o_t * tanh(c_t)

        outputs_list.append(h_t.data) # Store numpy data
        h_prev = h_t
        c_prev = c_t

    # Stack outputs along the sequence_length dimension
    stacked_outputs_data = np.stack(outputs_list, axis=1) # (batch_size, sequence_length, hidden_size)
    outputs_value = Value(stacked_outputs_data)

    return outputs_value, (h_t, c_t) # h_t and c_t are the last states (Value objects)
```

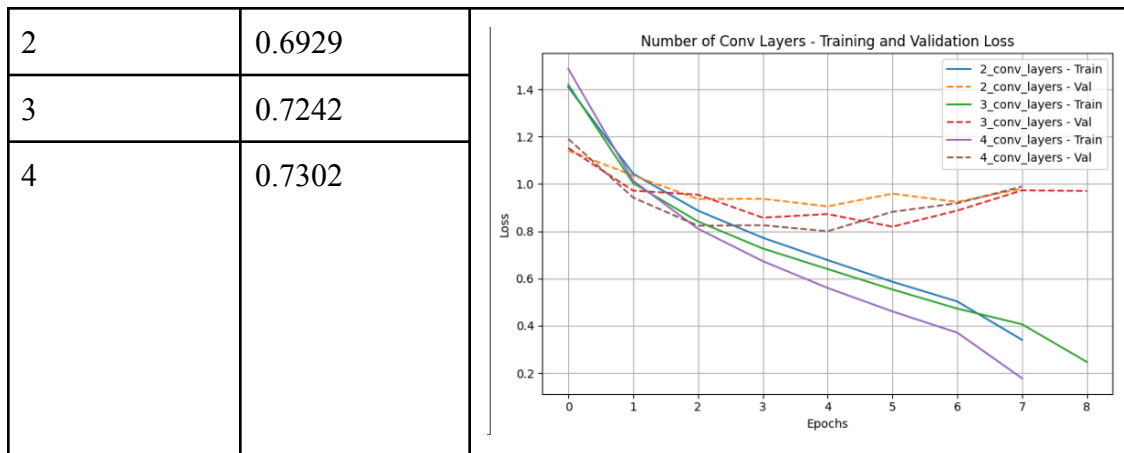
**SISA DARI CLASS DAN FUNGSI MENGGUNAKAN HASIL DARI TUBES 1
SEPERTI SOFTMAX, VALUE CLASS, DLL.**

C. Hasil Pengujian dan Analisis

Pengujian CNN menggunakan dataset CIFAR10

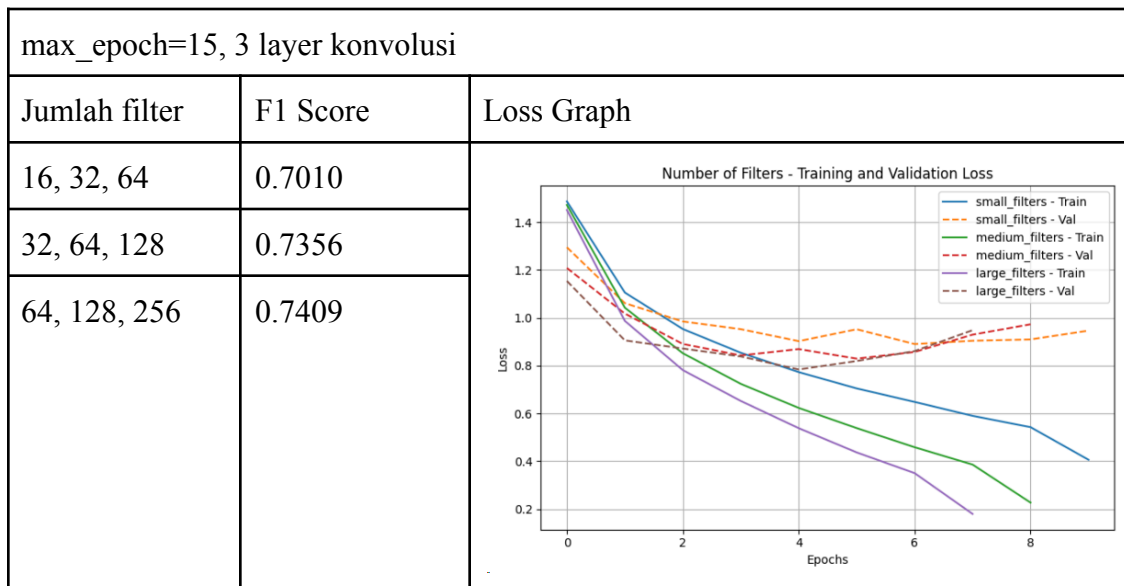
1. Pengaruh Jumlah Layer Konvolusi

max_epoch=15		
Jumlah Layer Konvolusi	F1 Score	Loss Graph



Berdasarkan hasil percobaan dengan variasi jumlah layer konvolusi dan parameter pelatihan tetap (max_epoch = 15), terlihat bahwa peningkatan jumlah layer konvolusi memberikan pengaruh positif terhadap performa model. F1 Score mengalami peningkatan bertahap dari 0.6929 pada 2 layer menjadi 0.7242 pada 3 layer, dan mencapai 0.7302 pada 4 layer. Hal ini menunjukkan bahwa dengan menambahkan layer konvolusi, model mampu mengekstraksi fitur yang lebih kompleks dan representatif dari data, sehingga menghasilkan prediksi yang lebih akurat.

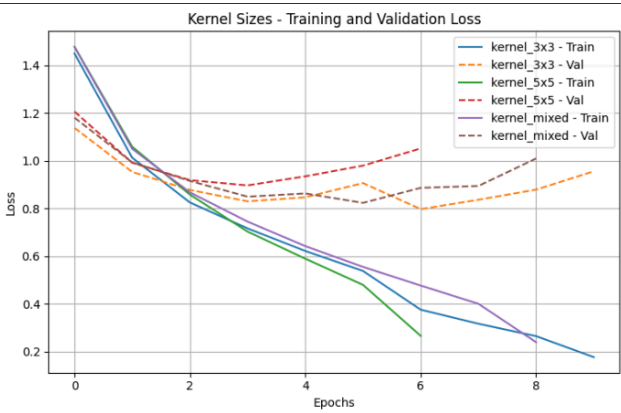
2. Pengaruh banyak filter per layer konvolusi



Dari hasil eksperimen dengan konfigurasi tiga layer konvolusi dan jumlah epoch tetap (max_epoch = 15), dapat disimpulkan bahwa peningkatan jumlah filter pada setiap layer secara konsisten meningkatkan performa model. Saat jumlah filter dinaikkan dari (16, 32,

64) menjadi (32, 64, 128), F1 Score meningkat signifikan dari 0.7010 menjadi 0.7356. Peningkatan lebih lanjut ke (64, 128, 256) kembali memberikan kenaikan F1 Score menjadi 0.7409. Hal ini menunjukkan bahwa jumlah filter yang lebih besar memungkinkan model untuk menangkap lebih banyak variasi dan kompleksitas fitur pada setiap level konvolusi, sehingga meningkatkan kemampuan generalisasi terhadap data.

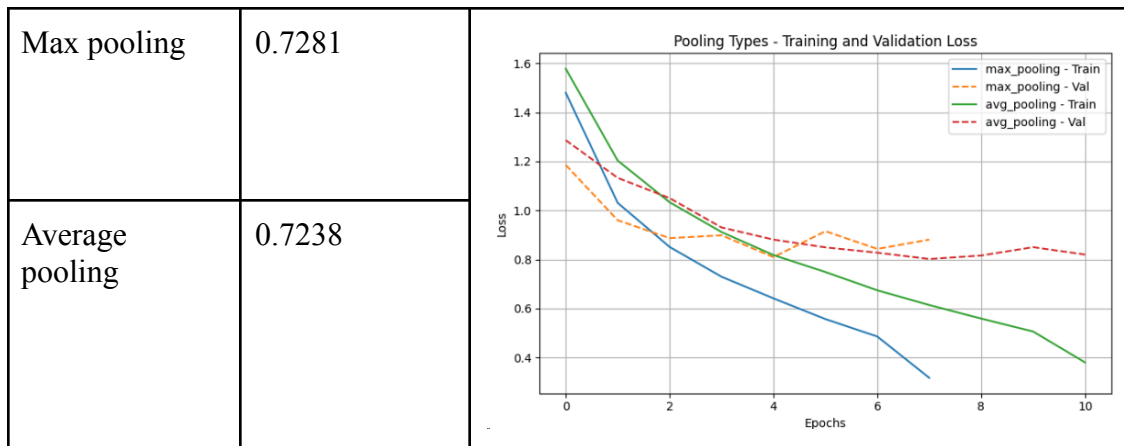
3. Pengaruh ukuran filter per layer konvolusi

max_epoch=15, 3 layer konvolusi		
Ukuran filter	F1 Score	Loss Graph
3x3	0.7485	
5x5	0.6906	
3x3, 5x5, 3x3	0.7244	

Berdasarkan eksperimen dengan tiga layer konvolusi dan jumlah epoch tetap (max_epoch = 15), dapat dilihat bahwa ukuran filter memiliki pengaruh signifikan terhadap performa model. Penggunaan filter berukuran 3×3 secara konsisten menghasilkan F1 Score tertinggi, yaitu 0.7485, dibandingkan dengan filter 5×5 yang justru menyebabkan penurunan performa ke 0.6906. Kombinasi ukuran filter (3×3, 5×5, 3×3) menghasilkan F1 Score menengah sebesar 0.7244. Hal ini menunjukkan bahwa filter 3×3 lebih efektif dalam menangkap fitur lokal yang halus dan mempertahankan informasi spasial pada setiap layer. Sebaliknya, filter yang lebih besar seperti 5×5 cenderung mengaburkan detail lokal, sehingga menurunkan akurasi prediksi.

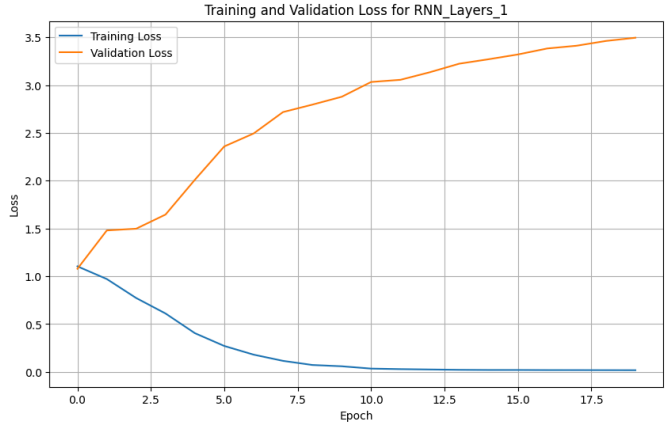
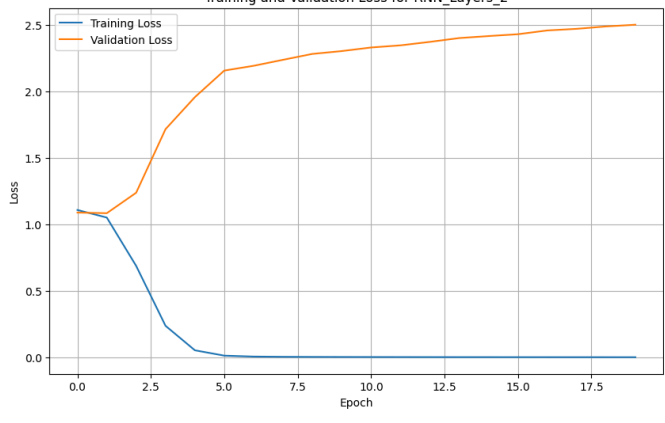
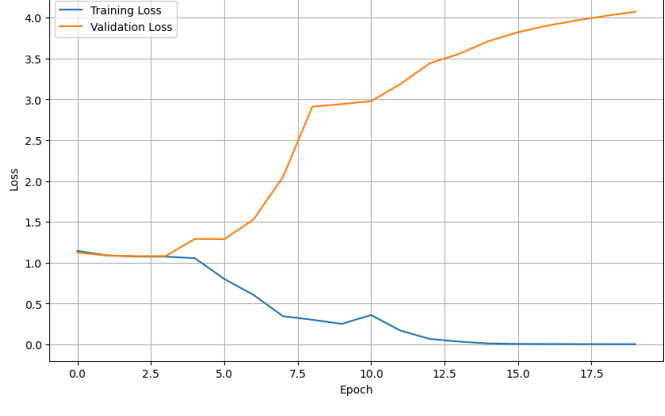
4. Pengaruh jenis pooling layer yang digunakan

max_epoch=15, 3 layer konvolusi		
Jenis pooling	F1 Score	Loss Graph



Dari hasil eksperimen dengan tiga layer konvolusi dan jumlah epoch tetap (max_epoch = 15), terlihat bahwa jenis pooling layer yang digunakan memberikan pengaruh terhadap performa model. Penggunaan max pooling menghasilkan F1 Score sebesar 0.7281, sedikit lebih tinggi dibandingkan dengan average pooling yang mencatat F1 Score sebesar 0.7238. Hal ini menunjukkan bahwa max pooling lebih efektif dalam mengekstraksi fitur penting karena mempertahankan nilai aktivasi tertinggi dari setiap region, sehingga lebih mampu menangkap ciri-ciri dominan dalam data. Sebaliknya, average pooling merata-ratakan nilai dalam region, yang dapat menyebabkan hilangnya informasi penting, terutama pada fitur dengan aktivasi kuat. Meskipun perbedaannya tidak terlalu besar, hasil ini mendukung kecenderungan umum dalam arsitektur CNN modern yang lebih mengandalkan max pooling untuk mempertahankan fitur yang lebih informatif selama proses downsampling.

1. Pengaruh jumlah layer RNN

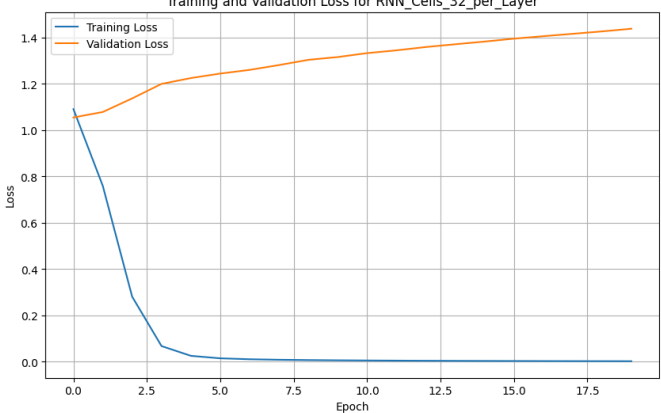
Jumlah layer	F1 Score	Loss Graph																																																												
1	0.3875	 <p>Training and Validation Loss for RNN_Layers_1</p> <table border="1"> <thead> <tr> <th>Epoch</th> <th>Training Loss</th> <th>Validation Loss</th> </tr> </thead> <tbody> <tr><td>0.0</td><td>1.1</td><td>1.1</td></tr> <tr><td>1.0</td><td>1.0</td><td>1.5</td></tr> <tr><td>2.0</td><td>0.8</td><td>1.6</td></tr> <tr><td>3.0</td><td>0.6</td><td>1.7</td></tr> <tr><td>4.0</td><td>0.4</td><td>2.1</td></tr> <tr><td>5.0</td><td>0.3</td><td>2.4</td></tr> <tr><td>6.0</td><td>0.2</td><td>2.7</td></tr> <tr><td>7.0</td><td>0.1</td><td>2.8</td></tr> <tr><td>8.0</td><td>0.1</td><td>2.9</td></tr> <tr><td>9.0</td><td>0.1</td><td>3.0</td></tr> <tr><td>10.0</td><td>0.1</td><td>3.1</td></tr> <tr><td>11.0</td><td>0.1</td><td>3.2</td></tr> <tr><td>12.0</td><td>0.1</td><td>3.3</td></tr> <tr><td>13.0</td><td>0.1</td><td>3.4</td></tr> <tr><td>14.0</td><td>0.1</td><td>3.4</td></tr> <tr><td>15.0</td><td>0.1</td><td>3.5</td></tr> <tr><td>16.0</td><td>0.1</td><td>3.5</td></tr> <tr><td>17.0</td><td>0.1</td><td>3.5</td></tr> <tr><td>18.0</td><td>0.1</td><td>3.5</td></tr> </tbody> </table>	Epoch	Training Loss	Validation Loss	0.0	1.1	1.1	1.0	1.0	1.5	2.0	0.8	1.6	3.0	0.6	1.7	4.0	0.4	2.1	5.0	0.3	2.4	6.0	0.2	2.7	7.0	0.1	2.8	8.0	0.1	2.9	9.0	0.1	3.0	10.0	0.1	3.1	11.0	0.1	3.2	12.0	0.1	3.3	13.0	0.1	3.4	14.0	0.1	3.4	15.0	0.1	3.5	16.0	0.1	3.5	17.0	0.1	3.5	18.0	0.1	3.5
Epoch	Training Loss	Validation Loss																																																												
0.0	1.1	1.1																																																												
1.0	1.0	1.5																																																												
2.0	0.8	1.6																																																												
3.0	0.6	1.7																																																												
4.0	0.4	2.1																																																												
5.0	0.3	2.4																																																												
6.0	0.2	2.7																																																												
7.0	0.1	2.8																																																												
8.0	0.1	2.9																																																												
9.0	0.1	3.0																																																												
10.0	0.1	3.1																																																												
11.0	0.1	3.2																																																												
12.0	0.1	3.3																																																												
13.0	0.1	3.4																																																												
14.0	0.1	3.4																																																												
15.0	0.1	3.5																																																												
16.0	0.1	3.5																																																												
17.0	0.1	3.5																																																												
18.0	0.1	3.5																																																												
2	0.3750	 <p>Training and Validation Loss for RNN_Layers_2</p> <table border="1"> <thead> <tr> <th>Epoch</th> <th>Training Loss</th> <th>Validation Loss</th> </tr> </thead> <tbody> <tr><td>0.0</td><td>1.1</td><td>1.1</td></tr> <tr><td>1.0</td><td>1.0</td><td>1.1</td></tr> <tr><td>2.0</td><td>0.8</td><td>1.2</td></tr> <tr><td>3.0</td><td>0.4</td><td>1.7</td></tr> <tr><td>4.0</td><td>0.2</td><td>2.0</td></tr> <tr><td>5.0</td><td>0.1</td><td>2.2</td></tr> <tr><td>6.0</td><td>0.1</td><td>2.3</td></tr> <tr><td>7.0</td><td>0.1</td><td>2.3</td></tr> <tr><td>8.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>9.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>10.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>11.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>12.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>13.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>14.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>15.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>16.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>17.0</td><td>0.1</td><td>2.4</td></tr> <tr><td>18.0</td><td>0.1</td><td>2.4</td></tr> </tbody> </table>	Epoch	Training Loss	Validation Loss	0.0	1.1	1.1	1.0	1.0	1.1	2.0	0.8	1.2	3.0	0.4	1.7	4.0	0.2	2.0	5.0	0.1	2.2	6.0	0.1	2.3	7.0	0.1	2.3	8.0	0.1	2.4	9.0	0.1	2.4	10.0	0.1	2.4	11.0	0.1	2.4	12.0	0.1	2.4	13.0	0.1	2.4	14.0	0.1	2.4	15.0	0.1	2.4	16.0	0.1	2.4	17.0	0.1	2.4	18.0	0.1	2.4
Epoch	Training Loss	Validation Loss																																																												
0.0	1.1	1.1																																																												
1.0	1.0	1.1																																																												
2.0	0.8	1.2																																																												
3.0	0.4	1.7																																																												
4.0	0.2	2.0																																																												
5.0	0.1	2.2																																																												
6.0	0.1	2.3																																																												
7.0	0.1	2.3																																																												
8.0	0.1	2.4																																																												
9.0	0.1	2.4																																																												
10.0	0.1	2.4																																																												
11.0	0.1	2.4																																																												
12.0	0.1	2.4																																																												
13.0	0.1	2.4																																																												
14.0	0.1	2.4																																																												
15.0	0.1	2.4																																																												
16.0	0.1	2.4																																																												
17.0	0.1	2.4																																																												
18.0	0.1	2.4																																																												
3	0.3375	 <p>Training and Validation Loss for RNN_Layers_3</p> <table border="1"> <thead> <tr> <th>Epoch</th> <th>Training Loss</th> <th>Validation Loss</th> </tr> </thead> <tbody> <tr><td>0.0</td><td>1.1</td><td>1.1</td></tr> <tr><td>1.0</td><td>1.1</td><td>1.1</td></tr> <tr><td>2.0</td><td>1.1</td><td>1.1</td></tr> <tr><td>3.0</td><td>1.1</td><td>1.1</td></tr> <tr><td>4.0</td><td>1.1</td><td>1.3</td></tr> <tr><td>5.0</td><td>0.8</td><td>1.3</td></tr> <tr><td>6.0</td><td>0.6</td><td>1.5</td></tr> <tr><td>7.0</td><td>0.4</td><td>2.0</td></tr> <tr><td>8.0</td><td>0.3</td><td>2.9</td></tr> <tr><td>9.0</td><td>0.3</td><td>3.0</td></tr> <tr><td>10.0</td><td>0.4</td><td>3.0</td></tr> <tr><td>11.0</td><td>0.3</td><td>3.4</td></tr> <tr><td>12.0</td><td>0.1</td><td>3.5</td></tr> <tr><td>13.0</td><td>0.1</td><td>3.7</td></tr> <tr><td>14.0</td><td>0.1</td><td>3.8</td></tr> <tr><td>15.0</td><td>0.1</td><td>3.9</td></tr> <tr><td>16.0</td><td>0.1</td><td>4.0</td></tr> <tr><td>17.0</td><td>0.1</td><td>4.0</td></tr> <tr><td>18.0</td><td>0.1</td><td>4.0</td></tr> </tbody> </table>	Epoch	Training Loss	Validation Loss	0.0	1.1	1.1	1.0	1.1	1.1	2.0	1.1	1.1	3.0	1.1	1.1	4.0	1.1	1.3	5.0	0.8	1.3	6.0	0.6	1.5	7.0	0.4	2.0	8.0	0.3	2.9	9.0	0.3	3.0	10.0	0.4	3.0	11.0	0.3	3.4	12.0	0.1	3.5	13.0	0.1	3.7	14.0	0.1	3.8	15.0	0.1	3.9	16.0	0.1	4.0	17.0	0.1	4.0	18.0	0.1	4.0
Epoch	Training Loss	Validation Loss																																																												
0.0	1.1	1.1																																																												
1.0	1.1	1.1																																																												
2.0	1.1	1.1																																																												
3.0	1.1	1.1																																																												
4.0	1.1	1.3																																																												
5.0	0.8	1.3																																																												
6.0	0.6	1.5																																																												
7.0	0.4	2.0																																																												
8.0	0.3	2.9																																																												
9.0	0.3	3.0																																																												
10.0	0.4	3.0																																																												
11.0	0.3	3.4																																																												
12.0	0.1	3.5																																																												
13.0	0.1	3.7																																																												
14.0	0.1	3.8																																																												
15.0	0.1	3.9																																																												
16.0	0.1	4.0																																																												
17.0	0.1	4.0																																																												
18.0	0.1	4.0																																																												

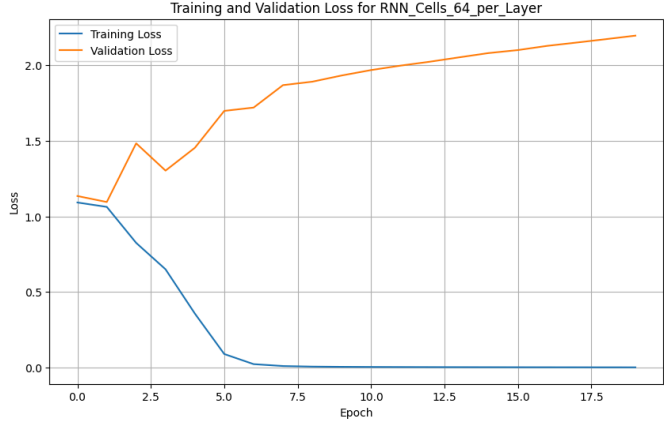
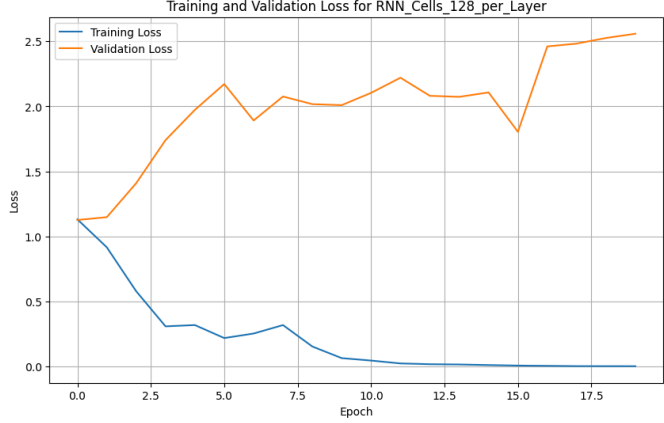
Semakin banyak layer, model memiliki kapasitas yang lebih besar untuk mempelajari representasi fitur yang lebih kompleks dari data sekuensial. Namun, terlalu banyak

layer dapat menyebabkan overfitting, kesulitan pelatihan (vanishing/exploding gradients), dan peningkatan waktu komputasi.

Pada model SimpleRNN, diamati bahwa terjadi *vanishing gradient* yang diamati pada training loss mendekati 0. *Vanishing gradient* terjadi karena gradien yang digunakan untuk memperbarui bobot model akan menyusut secara eksponensial saat mereka menyebar mundur melalui banyak langkah waktu dalam sekuens. Ini berarti, koneksi yang bertanggung jawab untuk mempelajari dependensi jangka panjang dari awal sekuens menerima gradien yang sangat kecil, sehingga pembaruannya menjadi tidak efektif. Meskipun *training loss* mungkin mendekati nol, ini seringkali mengindikasikan bahwa SimpleRNN tersebut berhasil belajar dan mengeksploitasi pola-pola jangka pendek atau korelasi lokal yang sangat kuat dalam data pelatihan. Model menjadi sangat baik dalam "menghafal" atau menyesuaikan diri dengan pola-pola terdekat, menyebabkan penurunan loss yang signifikan. Namun, pada saat yang sama, ia kesulitan untuk menangkap dependensi yang lebih kompleks atau konteks jangka panjang, karena gradien dari kesalahan yang terkait dengan informasi yang lebih jauh telah "lenyap" sebelum dapat mempengaruhi bobot awal secara berarti, membatasi kemampuan model untuk melakukan generalisasi yang efektif meskipun loss pelatihan rendah.

2. Pengaruh banyak cell RNN per layer

Cell per layer	F1 Score	Loss Graph																											
32	0.4025	<div><p>Training and Validation Loss for RNN_Cells_32_per_Layer</p><table border="1"><caption>Approximate data points from the Loss Graph</caption><thead><tr><th>Epoch</th><th>Training Loss</th><th>Validation Loss</th></tr></thead><tbody><tr><td>0.0</td><td>1.1</td><td>1.1</td></tr><tr><td>2.5</td><td>0.3</td><td>1.2</td></tr><tr><td>5.0</td><td>0.05</td><td>1.25</td></tr><tr><td>7.5</td><td>0.02</td><td>1.3</td></tr><tr><td>10.0</td><td>0.01</td><td>1.35</td></tr><tr><td>12.5</td><td>0.01</td><td>1.4</td></tr><tr><td>15.0</td><td>0.01</td><td>1.42</td></tr><tr><td>17.5</td><td>0.01</td><td>1.45</td></tr></tbody></table></div>	Epoch	Training Loss	Validation Loss	0.0	1.1	1.1	2.5	0.3	1.2	5.0	0.05	1.25	7.5	0.02	1.3	10.0	0.01	1.35	12.5	0.01	1.4	15.0	0.01	1.42	17.5	0.01	1.45
Epoch	Training Loss	Validation Loss																											
0.0	1.1	1.1																											
2.5	0.3	1.2																											
5.0	0.05	1.25																											
7.5	0.02	1.3																											
10.0	0.01	1.35																											
12.5	0.01	1.4																											
15.0	0.01	1.42																											
17.5	0.01	1.45																											

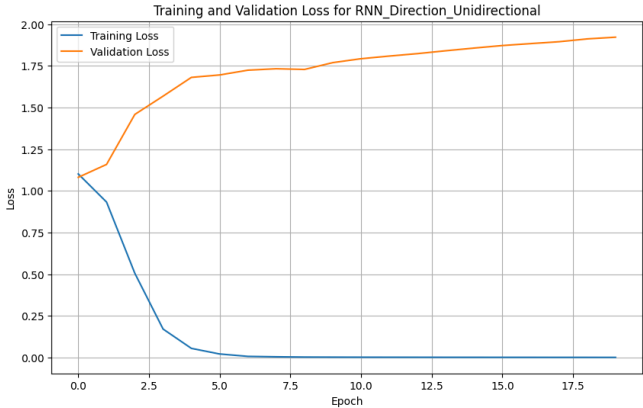
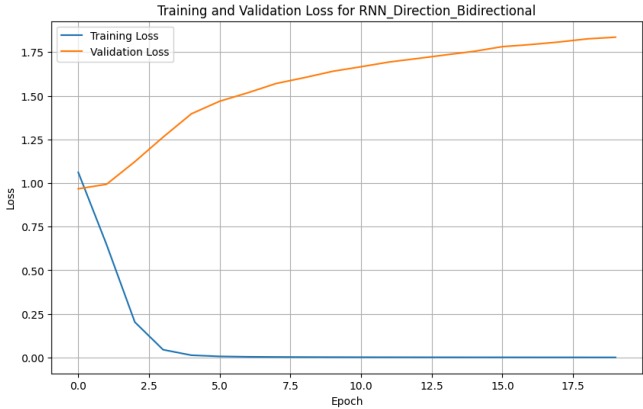
64	0.4000	
128	0.5225	

Jumlah cell (unit) per layer menentukan kapasitas representasional dari layer tersebut. Jumlah cell yang lebih banyak memungkinkan layer untuk mempelajari pola yang lebih kompleks. Namun, terlalu banyak cell dapat menyebabkan overfitting, membutuhkan lebih banyak data pelatihan, dan meningkatkan waktu komputasi serta penggunaan memori.

Pada model SimpleRNN, diamati bahwa terjadi *vanishing gradient* yang diamati pada training loss mendekati 0. *Vanishing gradient* terjadi karena gradien yang digunakan untuk memperbarui bobot model akan menyusut secara eksponensial saat mereka menyebar mundur melalui banyak langkah waktu dalam sekuens. Ini berarti, koneksi yang bertanggung jawab untuk mempelajari dependensi jangka panjang dari awal sekuens menerima gradien yang sangat kecil, sehingga pembaruannya menjadi tidak efektif. Meskipun *training loss* mungkin mendekati nol, ini seringkali mengindikasikan bahwa SimpleRNN tersebut berhasil belajar dan mengeksploitasi

pola-pola jangka pendek atau korelasi lokal yang sangat kuat dalam data pelatihan. Model menjadi sangat baik dalam "menghafal" atau menyesuaikan diri dengan pola-pola terdekat, menyebabkan penurunan loss yang signifikan. Namun, pada saat yang sama, ia kesulitan untuk menangkap dependensi yang lebih kompleks atau konteks jangka panjang, karena gradien dari kesalahan yang terkait dengan informasi yang lebih jauh telah "lenyap" sebelum dapat mempengaruhi bobot awal secara berarti, membatasi kemampuan model untuk melakukan generalisasi yang efektif meskipun loss pelatihan rendah.

3. Pengaruh jenis layer RNN berdasarkan arah

Direction	F1 Score	Loss Graph
Unidirectional	0.4225	
Bidirectional	0.4775	

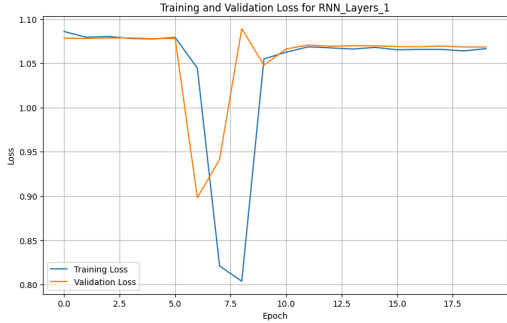
Unidirectional RNN memproses sekuens dari awal hingga akhir (masa lalu ke masa depan). Bidirectional RNN sekuens dari dua arah: dari awal ke akhir dan dari akhir ke awal. Ini memungkinkan model untuk menangkap konteks dari masa lalu dan masa

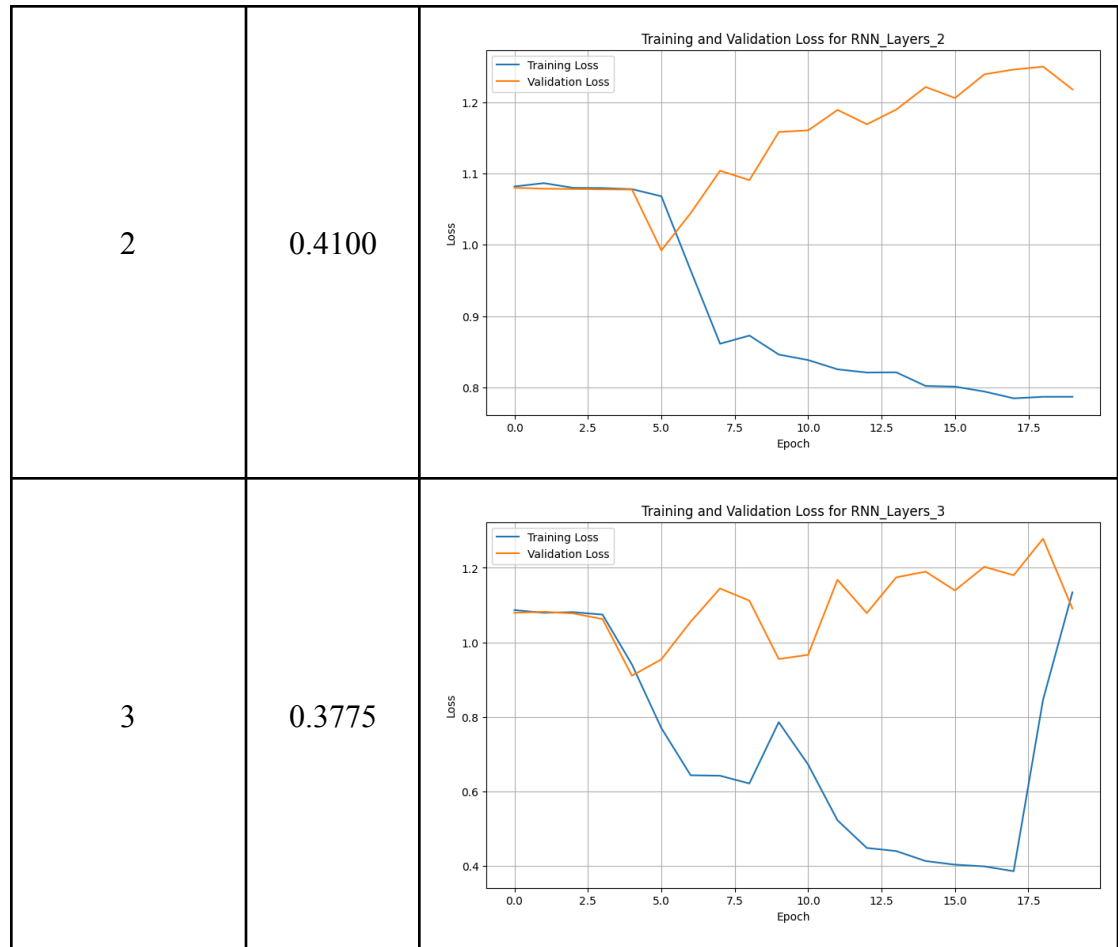
depan pada setiap titik waktu. Bidirectional RNN seringkali lebih baik untuk tugas-tugas di mana konteks dari kedua arah penting (misalnya, pemrosesan bahasa alami).

Pada model SimpleRNN, diamati bahwa terjadi *vanishing gradient* yang diamati pada training loss mendekati 0. *Vanishing gradient* terjadi karena gradien yang digunakan untuk memperbarui bobot model akan menyusut secara eksponensial saat mereka menyebar mundur melalui banyak langkah waktu dalam sekuens. Ini berarti, koneksi yang bertanggung jawab untuk mempelajari dependensi jangka panjang dari awal sekuens menerima gradien yang sangat kecil, sehingga pembaruannya menjadi tidak efektif. Meskipun *training loss* mungkin mendekati nol, ini seringkali mengindikasikan bahwa SimpleRNN tersebut berhasil belajar dan mengeksploitasi pola-pola jangka pendek atau korelasi lokal yang sangat kuat dalam data pelatihan. Model menjadi sangat baik dalam "menghafal" atau menyesuaikan diri dengan pola-pola terdekat, menyebabkan penurunan loss yang signifikan. Namun, pada saat yang sama, ia kesulitan untuk menangkap dependensi yang lebih kompleks atau konteks jangka panjang, karena gradien dari kesalahan yang terkait dengan informasi yang lebih jauh telah "lenyap" sebelum dapat mempengaruhi bobot awal secara berarti, membatasi kemampuan model untuk melakukan generalisasi yang efektif meskipun loss pelatihan rendah.

Pengujian LSTM menggunakan dataset NusaX

1. Pengaruh jumlah layer LSTM

Jumlah layer	F1 Score	Loss Graph
1	0.3850	

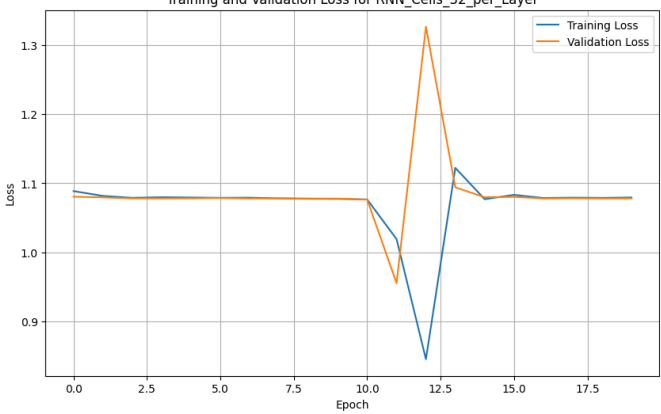


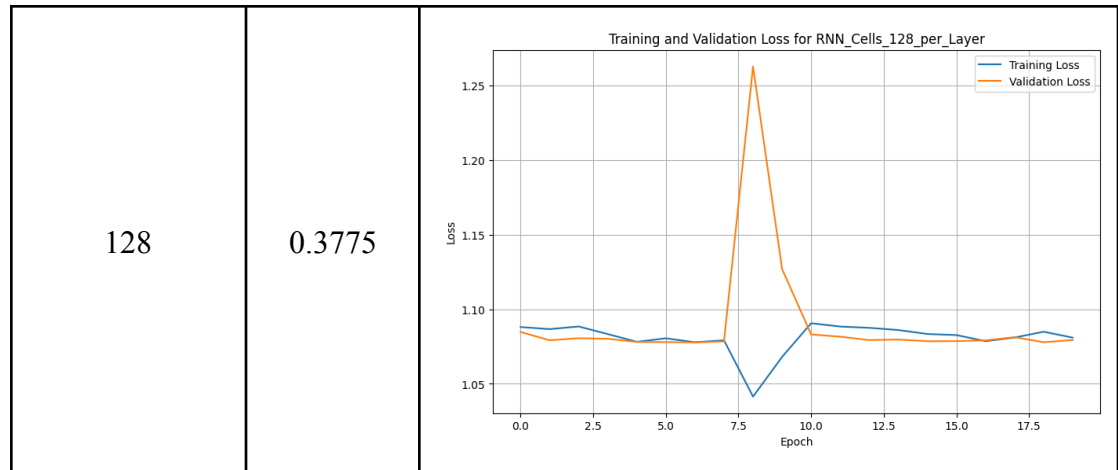
Semakin banyak layer, model memiliki kapasitas yang lebih besar untuk mempelajari representasi fitur yang lebih kompleks dari data sekuensial. Namun, terlalu banyak layer dapat menyebabkan overfitting, kesulitan pelatihan (vanishing/exploding gradients), dan peningkatan waktu komputasi.

Berbeda dengan model SimpleRNN, diamati bahwa ada fluktuasi training loss pada model LSTM. Fluktuasi training loss yang lebih jelas pada model LSTM dibandingkan dengan SimpleRNN utamanya disebabkan oleh arsitektur internal LSTM yang lebih kompleks dan kemampuannya yang lebih baik dalam mengatasi masalah vanishing atau exploding gradients. SimpleRNN seringkali kesulitan untuk mempertahankan informasi dari langkah waktu yang jauh, yang bisa menyebabkan gradien menjadi sangat kecil (vanishing) atau sangat besar (exploding), sehingga pelatihan cenderung stagnan atau tidak stabil secara drastis di awal. Sebaliknya, gerbang input, forget, dan output pada LSTM memungkinkan model untuk secara selektif mengingat atau

melupakan informasi, sehingga gradien dapat mengalir lebih efektif melalui waktu. Kemampuan adaptif ini membuat LSTM lebih sensitif terhadap setiap batch data dan secara aktif menyesuaikan bobotnya di seluruh ruang parameter yang lebih kompleks, yang secara alami dapat menghasilkan oscillation atau fluktuasi pada training loss seiring model terus belajar dan beradaptasi secara dinamis di setiap epoch, dibandingkan dengan kurva loss SimpleRNN yang mungkin terlihat lebih "diam" karena kesulitan pembelajaran internalnya.

2. Pengaruh banyak cell LSTM per layer

Cell per layer	F1 Score	Loss Graph
32	0.3825	<div>Training and Validation Loss for RNN_Cells_32_per_Layer</div> 



Jumlah cell (unit) per layer menentukan kapasitas representasional dari layer tersebut. Jumlah cell yang lebih banyak memungkinkan layer untuk mempelajari pola yang lebih kompleks. Namun, terlalu banyak cell dapat menyebabkan overfitting, membutuhkan lebih banyak data pelatihan, dan meningkatkan waktu komputasi serta penggunaan memori.

Berbeda dengan model SimpleRNN, diamati bahwa ada fluktuasi training loss pada model LSTM. Fluktuasi training loss yang lebih jelas pada model LSTM dibandingkan dengan SimpleRNN utamanya disebabkan oleh arsitektur internal LSTM yang lebih kompleks dan kemampuannya yang lebih baik dalam mengatasi masalah vanishing atau exploding gradients. SimpleRNN seringkali kesulitan untuk mempertahankan informasi dari langkah waktu yang jauh, yang bisa menyebabkan gradien menjadi sangat kecil (vanishing) atau sangat besar (exploding), sehingga pelatihan cenderung stagnan atau tidak stabil secara drastis di awal. Sebaliknya, gerbang input, forget, dan output pada LSTM memungkinkan model untuk secara selektif mengingat atau melupakan informasi, sehingga gradien dapat mengalir lebih efektif melalui waktu. Kemampuan adaptif ini membuat LSTM lebih sensitif terhadap setiap batch data dan secara aktif menyesuaikan bobotnya di seluruh ruang parameter yang lebih kompleks, yang secara alami dapat menghasilkan oscillation atau fluktuasi pada training loss seiring model terus belajar dan beradaptasi secara dinamis di setiap epoch, dibandingkan dengan kurva loss SimpleRNN yang mungkin terlihat lebih "diam" karena kesulitan pembelajaran internalnya.

3. Pengaruh jenis layer LSTM berdasarkan arah

Direction	F1 Score	Loss Graph
Unidirectional	0.5250	<p>Training and Validation Loss for RNN_Direction_Unidirectional</p>
Bidirectional	0.7225	<p>Training and Validation Loss for RNN_Direction_Bidirectional</p>

Unidirectional LSTM memproses sekuens dari awal hingga akhir (masa lalu ke masa depan). Bidirectional LSTM memproses sekuens dari dua arah: dari awal ke akhir dan dari akhir ke awal. Ini memungkinkan model untuk menangkap konteks dari masa lalu dan masa depan pada setiap titik waktu. Bidirectional LSTM seringkali lebih baik untuk tugas-tugas di mana konteks dari kedua arah penting (misalnya, pemrosesan bahasa alami).

Berbeda dengan model SimpleRNN, diamati bahwa ada fluktuasi training loss pada model LSTM. Fluktuasi training loss yang lebih jelas pada model LSTM dibandingkan dengan SimpleRNN utamanya disebabkan oleh arsitektur internal LSTM yang lebih kompleks dan kemampuannya yang lebih baik dalam mengatasi masalah vanishing atau exploding gradients. SimpleRNN seringkali kesulitan untuk mempertahankan informasi dari langkah waktu yang jauh, yang bisa menyebabkan gradien menjadi sangat kecil (vanishing) atau sangat besar (exploding), sehingga pelatihan cenderung

stagnan atau tidak stabil secara drastis di awal. Sebaliknya, gerbang input, forget, dan output pada LSTM memungkinkan model untuk secara selektif mengingat atau melupakan informasi, sehingga gradien dapat mengalir lebih efektif melalui waktu. Kemampuan adaptif ini membuat LSTM lebih sensitif terhadap setiap batch data dan secara aktif menyesuaikan bobotnya di seluruh ruang parameter yang lebih kompleks, yang secara alami dapat menghasilkan oscillation atau fluktuasi pada training loss seiring model terus belajar dan beradaptasi secara dinamis di setiap epoch, dibandingkan dengan kurva loss SimpleRNN yang mungkin terlihat lebih "diam" karena kesulitan pembelajaran internalnya.

KESIMPULAN DAN SARAN

A. Kesimpulan

Dari hasil uji inferensi yang kami lakukan sesuai spesifikasi tugas besar didapat model - model from scratch kami dapat memuat bobot dari keras dan hasil inferensinya sama persis yang artinya implementasi tersebut sudah benar.

CNN, RNN, Dan LSTM dapat diimplementasikan di numpy from scratch tetapi memang kecepatannya tidak secepat menggunakan framework seperti keras ataupun pytorch karena framework tersebut sudah *optimized* dan kebanyakan operasinya dilakukan di bahasa c ataupun c++.

B. Saran

Sebaiknya Tubes tubes from scratch gini gak usah diadakan lagi karena buang buang waktu, mendingan kayak 1 tubes tapi bikin meaningful project yang ada produknya. Karena di Stanford, MIT, dll pun begitu (<https://cs230.stanford.edu/past-projects/>). Implementasi from scratch sebaiknya diganti ke pr, tetapi tidak usah dibanding bandingkan dengan yang keras, hanya buat notebook fill function gitu, dan disetiap sebelum functionnya dijelasin functionnya mau ngapain(kaya di coursera andrew ng.). Praktikum juga mending ganti tutorial pake pytorch atau tutorial keras.

PEMBAGIAN KERJA

NIM	Pembagian Kerja
13522079	RNN, LSTM
13522089	CNN
13522097	RNN, LSTM

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>