

Laporan Tugas Besar 2 Pembelajaran Mesin

Convolutional Neural Network dan Recurrent Neural Network

Semester II Tahun 2024/2025



Oleh:

Muhamad Rafli Rasyiidin	13522088
Andhika Tanyo Anugrah	13522094
M. Hanief Fatkhan Nashrullah	13522100

**PROGRAM STUDI INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025**

Daftar Isi

Daftar Isi	2
Bab 1	
Deskripsi Persoalan	3
Bab 2	
Pembahasan	4
1. Penjelasan Implementasi	4
2. Hasil Pengujian	17
Bab 3	
Kesimpulan dan Saran	40
Bab 4	
Pembagian Tugas	41
Referensi	42

Bab 1

Deskripsi Persoalan

Pada tugas besar kali ini, terdapat tiga percobaan yang harus dilakukan. Percobaan tersebut terkait pengaruh parameter pada CNN, RNN, dan LSTM. Setelah melakukan percobaan dengan variasi parameter, terdapat pengujian implementasi *from scratch* dari ketiga jenis *neural network* tersebut.

Pada percobaan bagian CNN, akan dilakukan eksperimen dengan membandingkan pengaruh dari beberapa parameter dalam melakukan pembelajaran klasifikasi gambar pada dataset CIFAR-10 dengan membangun model CNN menggunakan keras. Parameter yang akan dibandingkan adalah jumlah layer konvolusi, banyak filter per layer konvolusi, ukuran filter per layer konvolusi, dan jenis pooling layer. Selain membandingkan hasil variasi parameter, dilakukan implementasi dan pengujian feedforward from scratch dengan menggunakan hasil pelatihan dari keras.

Pada percobaan bagian RNN, akan dilakukan eksperimen dengan membandingkan pengaruh dari beberapa parameter dalam melakukan pembelajaran klasifikasi teks pada dataset NusaX bahasa Indonesia dengan membangun model RNN menggunakan pustaka Keras. Parameter yang akan dibandingkan adalah jumlah layer RNN, banyak unit (cell) RNN, dan arah (unidirectional vs bidirectional). Selain membandingkan hasil variasi parameter, dilakukan implementasi dan pengujian feedforward from scratch dengan menggunakan hasil pelatihan dari keras.

Pada percobaan bagian LSTM, akan dilakukan eksperimen dengan membandingkan pengaruh beberapa parameter, seperti jumlah layer LSTM, jumlah *node* pada layer LSTM, dan jenis layer LSTM berdasarkan arah dalam melakukan pembelajaran klasifikasi teks pada dataset NusaX bahasa Indonesia. Model LSTM yang digunakan berasal dari *library* keras. Selain itu, terdapat model LSTM yang dikembangkan dari awal yang dapat menerima input bobot dari model keras yang telah dilatih sebelumnya. Namun, model ini hanya memiliki metode untuk melakukan inferensi terhadap data input yang diberikan. Hasil inferensi dari kedua model tersebut akan dibandingkan satu sama lain sesuai dengan parameter yang telah digunakan sebelumnya.

Bab 2

Pembahasan

1. Penjelasan Implementasi

a. Deskripsi kelas beserta deskripsi atribut dan methodnya

i. CNN

Class Conv2DLayer

Kelas ini merupakan kelas yang merepresentasikan layer konvolusi. Terdapat satu konstruktor dan dua method dalam layer ini. Berikut adalah penjelasan setiap method dan konstruktor Conv2DLayer

Konstruktor

```
def __init__(self, kernel, bias=None, padding='same',
activation=None):
    self.kernel = kernel
    self.bias = bias if bias is not None else
np.zeros(kernel.shape[-1])
    self.padding = padding
    self.activation = activation
```

Pada kelas ini terdapat empat atribut yang tersimpan, yaitu kernel, bias, padding, dan activation. Atribut kernel digunakan untuk menyimpan bobot kernel hasil pembelajaran. Tipe data yang diterima kernel adalah tensor empat dimensi berisi float. Atribut bias digunakan untuk menyimpan bias dari setiap kernel. Tipe data yang diterima oleh bias adalah tensor satu dimensi berisi float. Atribut padding digunakan untuk menentukan jenis padding yang akan digunakan pada layer ini. Input yang valid dari padding adalah 'same' dan 'valid'. 'same' berarti padding digunakan sedemikian sehingga bentuk dari output tidak berubah dari input. 'valid' berarti tidak menggunakan padding. Terakhir, activation digunakan untuk menyimpan string penentu jenis fungsi aktivasi yang akan digunakan. Input yang valid dari atribut ini adalah 'relu' untuk menggunakan relu dan 'softmax' untuk menggunakan softmax.

Apply Activation

```
def _apply_activation(self, x):
    if self.activation is None:
        return x
    elif self.activation == 'relu':
        return np.maximum(0, x)
    elif self.activation == 'softmax':
```

```

        e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
        return e_x / e_x.sum(axis=-1, keepdims=True)
    else:
        raise ValueError(f"Unsupported activation function:
{self.activation}")

```

Method ini berfungsi untuk mengembalikan nilai sesuai dengan fungsi aktivasi yang telah ditentukan pada konstruktor. Relu akan mengembalikan nilai asli atau nol jika nilai asli lebih kecil dari nol. Softmax akan mengembalikan nilai sesuai dengan softmax function. Perhitungan dari softmax dikurangi oleh angka maksimum terlebih dahulu agar output tidak terlalu besar. Selain relu dan softmax, error akan dithrow oleh method ini.

Feedforward

```

def forward(self, x):
    if x.ndim == 3:
        x = x[np.newaxis, ...]
    _, _, _, C = x.shape
    kh, kw, C_in, _ = self.kernel.shape
    assert C == C_in, "Input channels must match kernel"
    pad_h, pad_w = (kh // 2, kw // 2) if self.padding ==
'same' else (0, 0)
    x_padded = np.pad(x, ((0, 0), (pad_h, pad_h), (pad_w,
pad_w), (0, 0)), mode='constant')
    window = sliding_window_view(x_padded, window_shape=(kh,
kw), axis=(1, 2))
    window = window.transpose(0,1,2,4,5,3)
    out = np.einsum('bhwlk,klco->bhwo', window,
self.kernel)
    out += self.bias.reshape((1, 1, 1, -1))
    out = self._apply_activation(out)
    return out[0] if out.shape[0] == 1 else out

```

Method ini berfungsi untuk melakukan feedforward pada layer konvolusi. Method ini menerima parameter x yaitu input gambar berupa tensor 3 atau 4 dimensi tergantung banyaknya gambar. Jika gambar hanya satu (tensor 3 dimensi), maka akan x akan diubah menjadi tensor 4 dimensi dengan satu elemen pada salah satu axisnya. Kemudian ditambahkan padding yang akan mempertahankan panjang setiap dimensi pada output. Kemudian dibentuklah sebuah “window” untuk melakukan konvolusi. Window ini seperti sliding window yang memiliki ukuran kernel height dan kernel width yang akan digeser sebesar satu unit pada axis input height dan input width. Karena keluaran dari

sliding_window_view adalah tensor dengan format axis (batch input, height, width, kernel input, kernel width, kernel height) maka perlu dilakukan transpose agar axis memiliki format (batch input, height, width, kernel height, kernel width, kernel input). Lalu, dilakukan loop sebanyak 7 kali untuk mengalikan setiap elemen pada satu gambar dengan kernel yang ada. Untuk menyederhanakan penulisan digunakan einsum. Berikut adalah penulisan perkalian konvolusi tanpa menggunakan einsum.

```
for b in range(batch count):
    for h in range(input height):
        for w in range(input width):
            for o in range(output channels):
                acc = 0
                for k in range(kernel height):
                    for l in range(kernel width):
                        for c in range(input channels):
                            acc +=
window[b][h][w][k][l][c]*self.kernel[k][l][c][o]
                out[b][h][w][o] = acc
```

Pada dasarnya, untuk setiap gambar, pada titik x dan y, pada kernel output tertentu, terdapat perkalian elemen dari gambar pada titik x dan y pada channel tertentu dengan suatu posisi x dan y kernel untuk channel yang bersesuaian pada satu output tertentu. Kemudian hasil perkalian ini dijumlahkan dengan bias lalu dimasukkan ke dalam fungsi aktivasi. Method akan mengembalikan hasil feedforward berupa tensor 3 atau 4 dimensi tergantung banyaknya gambar.

Class Pooling2DLayer

Kelas ini merupakan representasi dari layer Average Pooling dan Max Pooling. Sama seperti class Conv2DLayer, kelas ini memiliki satu konstruktor dan dua method. Berikut adalah penjelasan setiap method dan konstruktor dari kelas ini.

Konstruktor

```
def __init__(self, pool_size=(2, 2), strides=(2, 2),
padding='valid', type="max"):
    self.pool_size = pool_size
    self.strides = strides
    self.padding = padding
    self.type = type
```

Pada kelas ini terdapat empat atribut yang tersimpan, yaitu pool size, strides, padding, dan type. Pool size merupakan atribut yang menentukan seberapa besar pooling size dari layer ini. Pool size menerima tuple dua angka yang merepresentasikan besar axis height dan width. Atribut strides merupakan besarnya strides pada layer, tipe data yang diterima adalah tuple dua angka yang merepresentasikan besar strides pada axis x dan y. Kemudian terdapat atribut padding yang memiliki definisi yang sama seperti padding pada kelas Conv2DLayer. Terakhir, terdapat atribut type yang menentukan apakah pooling akan menggunakan tipe max pooling atau average pooling. Input yang valid untuk atribut ini adalah 'max' dan avg'.

Pad Input

```
def _pad_input(self, x):
    is_batch = x.ndim == 4
    if not is_batch:
        x = x[np.newaxis, ...]
    _, H, W, _ = x.shape
    ph, pw = self.pool_size
    sh, sw = self.strides
    if self.padding == 'same':
        out_h = int(np.ceil(H / sh))
        out_w = int(np.ceil(W / sw))
        pad_h = max((out_h - 1) * sh + ph - H, 0)
        pad_w = max((out_w - 1) * sw + pw - W, 0)
        pad_top = pad_h // 2
        pad_bottom = pad_h - pad_top
        pad_left = pad_w // 2
        pad_right = pad_w - pad_left
        pad_val = -np.inf if self.type == "max" else 0.0
        x = np.pad(
            x,
            ((0, 0), (pad_top, pad_bottom), (pad_left,
            pad_right), (0, 0)),
            mode="constant",
            constant_values=pad_val
        )
    elif self.padding != 'valid':
        raise ValueError(f"Unsupported padding:
        {self.padding}")
    return x
```

Method pad input merupakan sebuah fungsi untuk menambahkan padding pada input yang diterima. Jika input berupa tensor 3 dimensi, maka akan

dinaikkan dimensinya menjadi 4 dimensi dengan makna tensor 4 dimensi dengan 1 gambar. Jika tipe paddingnya adalah 'same' maka digunakan padding yang akan menghasilkan output dengan dimensi yang sama dengan dimensi inputnya. Padding berupa negatif tak hingga untuk max pooling agar tidak padding tidak akan mungkin menjadi nilai max, dan nol untuk average agar tidak terlalu mempengaruhi nilai rata rata saat pooling. Setelah dilakukan padding kemudian dikembalikan nilai input yang telah ditambahkan padding.

Feedforward

```
def forward(self, x):
    is_batch = x.ndim == 4
    if not is_batch:
        x = x[np.newaxis, ...]
    x = self._pad_input(x)
    ph, pw = self.pool_size
    sh, sw = self.strides
    windows = sliding_window_view(x, window_shape=(ph, pw),
axis=(1, 2))
    windows = windows[:, ::sh, ::sw, :, :, :]
    windows = windows.transpose(0, 1, 2, 4, 5, 3)
    if self.type == "max":
        out = np.max(windows, axis=(3, 4))
    elif self.type == "avg":
        out = np.mean(windows, axis=(3, 4))
    else:
        raise ValueError("Unknown pooling type")
    return out[0] if not is_batch else out
```

Method ini merupakan sebuah fungsi representasi untuk melakukan feedforwarding pada layer pooling. Fungsi ini menerima input tensor 4 dimensi atau 3 dimensi jika hanya berisi 1 gambar. Setelah dimensi diperiksa, input ditambahkan padding dengan method pad input. Setelah itu, dibuat sebuah sliding window view dengan cara kerja yang sama dengan sliding window view pada layer konvolusi. Kemudian window tersebut diambil sebagian dengan mengambil setiap sh elemen pada axis h (axis ke-1) dan mengambil setiap sw elemen (axis ke-2) sebagai representasi pergeseran sebesar strides. Kemudian dilakukan transpose dengan tujuan yang sama seperti layer konvolusi. Setelah itu diperiksa apakah tipe layer ini adalah max atau average. Terakhir dikembalikan nilai hasil pooling.

Class FlattenLayer

Kelas ini merupakan sebuah kelas yang merepresentasikan layer yang melakukan flatten dari input yang diterima. Kelas ini memiliki sebuah konstruktor

tanpa parameter dan sebuah method feedforward. Berikut adalah penjelasan dari method feedforward pada layer ini.

Feedforward

```
def forward(self, x):
    if x.ndim == 4:
        return x.reshape(x.shape[0], -1)
    elif x.ndim == 3:
        return x.reshape(1, -1)
    else:
        raise ValueError(f"FlattenLayer expects 3D or 4D input, got shape {x.shape}")
```

Method ini menerima sebuah input x yaitu tensor yang akan di-flatten. Tipe data yang diterima adalah tensor berdimensi 3 atau tensor berdimensi 4 yang berisi float. Setelah tensor di-flatten, hasil tersebut akan di-return.

Class DenseLayer

Kelas ini merupakan representasi dari layer dense, yaitu layer yang bersifat fully connected. Layer ini sama seperti layer pada FFNN. Kelas ini memiliki sebuah konstruktor dan sebuah method. Berikut adalah penjelasan dari konstruktor dan method yang ada.

Konstruktor

```
def __init__(self, W, b, activation=None):
    self.W = W
    self.b = b
    self.activation = activation
```

Kelas ini memiliki tiga atribut, yaitu w sebagai weight, b sebagai bias, dan activation sebagai penentu tipe activation function. Atribut w menerima tensor dua dimensi berisi float. Atribut b menerima tensor satu dimensi berisi float. Atribut activation menerima string yang bernilai 'relu' atau 'softmax'.

Feedforward

```
def forward(self, x):
    if x.ndim == 1:
        x = x[np.newaxis, :]
    z = x @ self.W + self.b
    if self.activation is None:
        return z
```

```

elif self.activation == 'relu':
    return np.maximum(0, z)
elif self.activation == 'softmax':
    e_z = np.exp(z - np.max(z, axis=-1, keepdims=True))
    return e_z / e_z.sum(axis=-1, keepdims=True)
else:
    raise ValueError(f"Unsupported activation:
{self.activation}")

```

Method ini merupakan sebuah representasi dari feedforward pada dense layer. Cara kerja dari layer ini sama seperti dari layer pada FFNN. Dilakukan perkalian matrix antara x dengan weight, kemudian ditambahkan dengan bias. Setelah operasi tersebut dilakukan, kemudian tipe fungsi aktivasi diperiksa. Jika tanpa fungsi aktivasi, nilai feedforward langsung dikembalikan. Jika bertipe 'relu', nilai yang lebih kecil dari nol akan diganti menjadi nol kemudian dikembalikan. Jika bertipe 'softmax', nilai akan dimasukkan ke softmax function terlebih dahulu baru setelah itu dikembalikan.

ii. Simple RNN

Class RNNScratch

Kelas RNNScratch merupakan kelas yang merepresentasikan layer RNN seperti RNNScratch pada pustaka Keras. Kelas ini terdiri dari 1 konstruktor dan 1 metode. Berikut merupakan penjelasan dari konstruktor dan metode pada kelas RNNScratch:

Konstruktor

```

class RNNScratch:
    def __init__(self,
                  input_weight: NDArray[np.float64],
                  hidden_weight: NDArray[np.float64],
                  hidden_bias_weight: NDArray[np.float64],
                  activation: Callable[[NDArray[np.float64]],
NDArray[np.float64]] = np.tanh) -> None:
        self.input_weight = input_weight
        self.hidden_weight = hidden_weight
        self.bias_weight = hidden_bias_weight
        self.activation = activation

```

Pada kelas ini terdapat 4 atribut yang disimpan, yaitu input_weight, hidden_weight, hidden_bias_weight, dan fungsi aktivasi. Atribut input_weight, hidden_weight, dan hidden_bias_weight merupakan atribut yang menyimpan

bobot hasil pembelajaran model SimpleRNN dari pustaka Keras. Fungsi aktivasi defaultnya menggunakan fungsi tanh dari pustaka numpy, pengguna bisa menggantinya dengan fungsi yang bertipe sama dengan yang didefinisikan oleh saya, yaitu Callable[[NDArray[np.float64]] yang artinya variabel yang dapat dipanggil yang outputnya berupa float64 milik numpy.

Forward

```
def forward(self,
            input_feature: NDArray[np.float64],
            h_state_prev: NDArray[np.float64] | None = None) ->
NDArray[np.float64]:
    """
    ###  $h_t = f(U \cdot x_t + W \cdot h_{t-1} + b_{xh})$ 

    Legend:
    U : Input weight matrix
    W : Hidden weight matrix
    x_t : Feature vector (i features) at step t
    h_t : Hidden state
    f: activation function
    """

    batch_size, seq_length, _ = input_feature.shape
    hidden_dim = self.bias_weight.shape[0]

    if h_state_prev is None:
        h_state_prev = np.zeros((batch_size, hidden_dim))

    for t in range(seq_length):
        x_t = input_feature[:, t, :]

        #  $h_t = \tanh(x_t \cdot \text{input\_weight.T} + h_{\text{state\_prev}} \cdot$ 
        #  $\text{hidden\_weight.T} + \text{bias})$ 
        h_t = self.activation(
            np.matmul(x_t, self.input_weight.T) +
            np.matmul(h_state_prev, self.hidden_weight.T) + #
            self.bias_weight
        )
        type: ignore
```

```

h_state_prev = h_t

return h_state_prev # type: ignore

```

Method forward merupakan method utama untuk melakukan forward propagation pada kelas RNNScratch. Method ini menerima antara 1 atau 2 parameter, tergantung kebutuhan pengguna. Parameter pertama yang wajib ada adalah input_feature, yaitu data input yang akan diinferensi oleh fungsi ini. Data input sendiri terdiri dari 3 dimensi, yang batch size atau jumlah sampel, time step, dan jumlah atribut. Method ini akan melakukan looping sebanyak sample dan time step untuk menghitung seluruh nilai hidden state menggunakan perkalian matrix milik numpy.

iii. LSTM

Class LSTMScratch

Kelas LSTMScratch merupakan kelas yang merepresentasikan layer LSTM seperti kelas LSTM pada *library* keras. Kelas ini terdiri dari 1 konstruktor dan 3 metode. Berikut merupakan penjelasan dari konstruktor dan setiap metode pada kelas LSTMScratch:

Konstruktor

```

class LSTMScratch():
    def __init__(
        self,
        units: int, # neuron
        keras_weight: list[Variable],
        activation: Callable[[NDArray[np.float64]],
        NDArray[np.float64]] = np.tanh, # activation function
        return_sequences = False
    ):
        self.units = units
        self.kernel, self.recurrent_kernel, self.bias =
keras_weight
        self.h_t = []
        self.c_t = []
        self.h_t_total = []
        self.activation = activation
        self.return_sequences = return_sequences

```

Pada kelas ini terdapat 9 atribut yang disimpan, yaitu units, kernel, recurrent_kernel, bias, h_t, c_t, h_t_total, activation, dan return_sequences. Units merupakan atribut bertipe integer yang berfungsi untuk menyimpan jumlah *node* pada satu layer LSTM. Atribut kernel, recurrent_kernel, dan bias merupakan atribut yang menyimpan bobot hasil pelatihan model LSTM menggunakan *library* keras yang nantinya akan digunakan sebagai bobot inferensi. Kernel

merepresentasikan bobot dari input layer ke setiap *node* pada layer LSTM (biasanya ditulis sebagai W), *recurrent_kernel* merepresentasikan bobot dari *hidden state* sebelumnya pada layer LSTM (biasanya ditulis sebagai U), dan bias merepresentasikan nilai bias untuk setiap *gate* pada *node* LSTM. Atribut h_t dan h_{t_total} merupakan nilai *hidden state* sebelumnya, h_t merupakan nilai *hidden state* terakhir yang akan menjadi *output* ketika parameter *return_sequences* bernilai False, sedangkan h_{t_total} berisikan seluruh nilai *hidden* yang telah dihasilkan dan akan dikembalikan ketika *return_sequences* bernilai True. Atribut c_t merepresentasikan *cell state* pada layer LSTM dan *activation* berisikan fungsi aktivasi yang akan digunakan pada *candidate memory gate*.

Forward

```
def forward(self, input_feature):
    if isinstance(input_feature, tf.Tensor):
        input_feature = input_feature.numpy() # type:
    ignore
    sample = input_feature.shape[0]
    timestep = input_feature.shape[1]
    for i in range(sample):
        current_sample = input_feature[i]
        self.temp_h = [np.zeros(self.units)]
        self.temp_c = [np.zeros(self.units)]

        for j in range(timestep):
            h_t, c_t = self.calc(current_sample[j])
            self.temp_h.append(h_t)
            self.temp_c.append(c_t)

        self.h_t.append(self.temp_h[-1])
        self.h_t_total.append(self.temp_h)
        self.c_t.append(self.temp_c[-1])
    if self.return_sequences:
        h_sequences = [np.stack(h[1:], axis=0) for h in
self.h_t_total]
        return np.stack(h_sequences, axis=0)
    else:
        return np.array(self.h_t)
```

Method forward merupakan method utama untuk melakukan forward propagation pada kelas LSTMScratch. Method ini menerima 1 parameter yaitu data input yang akan diinferensi. Data input tersebut dapat bertipe Tensor ataupun numpy.ndarray. Jika data input bertipe Tensor, maka method tersebut akan mengkonversinya menjadi tipe numpy.ndarray. Data input sendiri terdiri dari 3 dimensi, yaitu batch size atau jumlah sampel, *time step*, dan jumlah atribut.

Method ini akan melakukan *looping* sebanyak sampel dan *time step* untuk menghitung seluruh nilai *hidden state* dan *cell state* menggunakan method `calc` (akan dijelaskan di bawah). Setelah seluruh *hidden state* dan *cell state* dihitung, mereka akan di-*append* ke atribut `h_t`, `c_t`, dan `h_t_total`. Jika `return_sequences` bernilai `True`, maka `h_t_total` yang telah dikelompokkan per batch akan menjadi *output*. Variabel `h_sequences` melepas elemen pertamanya karena elemen pertama berisi array 0 yang digunakan sebagai inisialisasi awal ketika *time step* 0. Jika `return_sequences` bernilai `False`, maka `h_t` (nilai *hidden state* terakhir yang akan menjadi *output*-nya).

Calc

```
def calc(self, x_t: NDArray[np.float64]):
    w_dot_x = x_t @ self.kernel
    u_dot_h = self.temp_h[-1] @ self.recurrent_kernel
    result = w_dot_x + u_dot_h + self.bias
    k_i, k_f, k_c, k_o = np.split(result, 4, axis=0)

    i_t, f_t, o_t = self.sigmoid(k_i), self.sigmoid(k_f),
self.sigmoid(k_o)

    c_tilde = self.activation(k_c)
    c_t = f_t * self.temp_c[-1] + i_t * c_tilde

    h_t = o_t * self.activation(c_t)

    return h_t, c_t
```

Method `calc` merupakan method yang digunakan untuk menghitung nilai keempat gate di seluruh *node* pada satu *time step* tertentu dan mengembalikan nilai dari *hidden state* dan *cell state*. `W_dot_x` merupakan hasil perkalian dot antara matriks input dengan bobot kernel. `U_dot_h` merupakan hasil perkalian antara matriks *hidden state* pada *time step* sebelumnya dengan bobot *recurrent kernel*. `Result` merupakan penjumlahan seluruh perkalian bobot dengan inputnya masing-masing dan ditambah dengan bias. Array `result` di-*split* per empat elemen karena hasil dari perkalian matriks dan penjumlahan bias masih terdiri dari 4 *gate* dan digabung menjadi 1 array untuk seluruh *node*. Oleh karena itu, `result` di-*split* menjadi `k_i`, `k_f`, `k_c`, dan `k_o` agar kita bisa mendapatkan hasil yang terpisah untuk setiap *gate* yang ada. Variabel `i_t`, `f_t`, dan `o_t` merepresentasikan nilai untuk *input gate*, *forget gate*, dan *output gate* yang diperoleh dengan menggunakan fungsi aktivasi (default sigmoid) pada nilai `k_i`, `k_f`, dan `k_c`. Variabel `C_tilde` merupakan nilai `k_c` yang dimasukkan ke dalam fungsi aktivasi (default tanh) yang nantinya akan digunakan untuk menghitung nilai dari *cell state*. Variabel `h_t` merepresentasikan nilai *hidden state*.

Sigmoid

```
def sigmoid(self, x):  
    return 1/(1+np.exp(-x))
```

Method ini merepresentasikan fungsi aktivasi sigmoid yang digunakan pada *input gate forget*, *gate*, dan *output gate*.

b. Penjelasan *forward propagation*

i. CNN

Forward propagation pada CNN dapat berbeda-beda tergantung model dan arsitektur yang dikembangkan. Namun, secara garis besar, setiap layer dengan jenis yang sama akan memiliki *forward propagation* yang sama. Pada tugas besar kali ini, akan digunakan layer konvolusi, layer pooling, layer flatten, dan layer dense untuk *forward propagation*.

Pada layer konvolusi, implementasi *forward propagation* dilakukan dengan menggunakan sliding window dan einsum yang disediakan oleh library numpy. Sliding window digunakan untuk membuat sebuah view seperti jendela yang digeser sebesar satu unit untuk axis width dan height. Setiap elemen dari setiap channel pada sliding window tersebut kemudian dikalikan dengan kernel dan filter pada posisi yang bersesuaian. Karena perkalian ini membutuhkan banyak loop, maka kami menggunakan einsum untuk mempermudah penulisan dan mengurangi for-loop yang dapat memperlambat performa *forward propagation*. Layer ini berfungsi untuk membentuk suatu abstraksi dan mendeteksi feature yang ada pada gambar input.

Pada layer pooling, implementasi *forward propagation* dilakukan dengan menggunakan sliding window. Hampir sama dengan layer konvolusi, layer ini juga memerlukan sebuah jendela yang bergeser untuk mendeteksi setiap elemen yang ada pada jendela tersebut. Perbedaannya, pooling tidak melakukan perkalian dengan kernel tetapi memilih suatu nilai berdasarkan nilai yang ada pada jendela tersebut sehingga einsum tidak diperlukan pada layer ini. Layer ini berfungsi untuk melakukan downsampling agar pendeteksian fitur tidak terlalu berat dan menekankan suatu elemen pada receptive field untuk lebih diperhitungkan.

Pada layer flatten, implementasi dibuat dengan cukup sederhana. Pada dasarnya layer ini mengubah tensor yang ada menjadi tensor satu dimensi. Layer ini akan digunakan sebagai input dari layer dense.

Pada layer dense, implementasi *forward propagation* dilakukan dengan metode yang sama pada FFNN. Seluruh elemen menjadi *fully connected* untuk dapat memprediksi hasil akhirnya. Implementasi dari layer ini hanya menggunakan perkalian matrix dan fungsi seperti max dan eksponen yang ada pada library numpy.

ii. Simple RNN

Tidak seperti Feed Forward Neural Network (FFNN) yang tidak memiliki konsep urutan waktu antar data, RNN memiliki konsep urutan waktu yang ditandai dengan adanya pengulangan sebanyak time step pada perhitungan hidden layer-nya. Layer RNN menggunakan forward propagation untuk memproses data berdasarkan time step tertentu. Data hasil time step sebelumnya (jika tidak ada maka di-set sebagai array dengan 0 sebagai elemen-elemennya) diproses dengan menghitungnya dengan formula:

$$h_t = f(U \cdot x_t + W \cdot h_{t-1} + b_{xh})$$

dengan

U : Input weight matrix

W : Hidden weight matrix

x_t : Feature vector (i features) at step t

h_t : Hidden state

f : activation function

Perhitungan ini akan dilakukan berulang-ulang sampai time step (t) selesai.

iii. LSTM

Forward propagation pada LSTM hampir sama dengan forward propagation pada RNN. Hal yang membedakan LSTM dengan RNN terdapat pada pemrosesan di setiap *node*-nya. RNN tidak memiliki *gate* apapun dan akan langsung menghasilkan nilai *hidden state* begitu data dimasukkan. LSTM memiliki 4 *gate* yang berfungsi untuk mengatur penambahan informasi baru dan penghapusan informasi lama dari *time step* sebelumnya. Nilai *hidden state* baru bisa didapat ketika kita telah menghitung seluruh nilai pada ketiga *gate* lainnya.

Pada tugas ini, LSTM digunakan untuk memprediksi kelas/label dari suatu kalimat. Sebelum memasuki layer LSTM, kalimat tersebut diproses oleh tokenizer. Tokenizer berfungsi untuk mengubah kata-kata pada kalimat menjadi suatu token tertentu dan mengasosiasikan setiap kata tersebut dengan suatu angka atau ID. Selanjutnya, teks tersebut akan berubah menjadi list of token id dan diproses oleh embedding layer.

Embedding layer berfungsi untuk mengubah setiap angka pada list of token id menjadi vektor berdimensi lebih tinggi yang dapat menangkap makna semantik dari data. Vektor-vektor tersebut akan memiliki nilai yang mirip untuk kata yang memiliki hubungan satu sama lain. Sebagai contoh, kata sendok dan garpu akan memiliki nilai vektor yang mirip karena mereka memiliki hubungan satu sama lain sebagai alat makan yang sering digunakan bersama. Setelah seluruh kata diubah menjadi vektor, data akan diteruskan ke layer LSTM.

Layer LSTM menggunakan forward propagation untuk memproses data berdasarkan *batch* dan *time step* tertentu. Data hasil embedding layer akan diproses di setiap *node* dengan memasukkannya ke setiap *gate* yang ada. Setiap *time step* akan memiliki nilai *hidden state* dan *cell state* masing-masing yang akan digunakan untuk memproses data pada *time step* berikutnya. Nilai *hidden state* pada *time step* terakhir merupakan *output* dari layer LSTM. Namun, jika

setelah layer LSTM terdapat layer LSTM lagi atau bukan layer Dense, maka *output* yang dikeluarkan adalah *hidden state* pada seluruh *time step*.

Setelah selesai diproses pada layer LSTM, data akan memasuki layer dense. Implementasi *forward propagation* dilakukan dengan metode yang sama pada FFNN. Seluruh elemen menjadi *fully connected* untuk dapat memprediksi hasil akhirnya. Implementasi dari layer ini hanya menggunakan perkalian matrix dan fungsi seperti max dan eksponen yang ada pada library numpy. Pada tugas ini, karena data NusaX merupakan multi-class, maka fungsi aktivasi yang digunakan adalah softmax.

2. Hasil Pengujian

a. CNN

Baseline

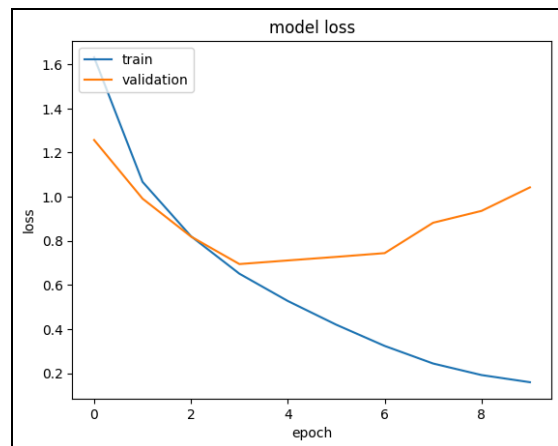
```
model1 = Sequential([
    Conv2D(64, (3, 3), padding='same', activation='relu',
input_shape=(32, 32, 3)),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(128, (3, 3), padding='same', activation='relu'),
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(256, (3, 3), padding='same', activation='relu'),
    Conv2D(256, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
F1Score: 0.1956
accuracy: 0.9499
loss: 0.1462
val_F1Score: 0.1956
val_accuracy: 0.7671
val_loss: 1.0423
```



Pengaruh jumlah layer konvolusi

Less convolutional layer (1 convolutional layer/pooling)

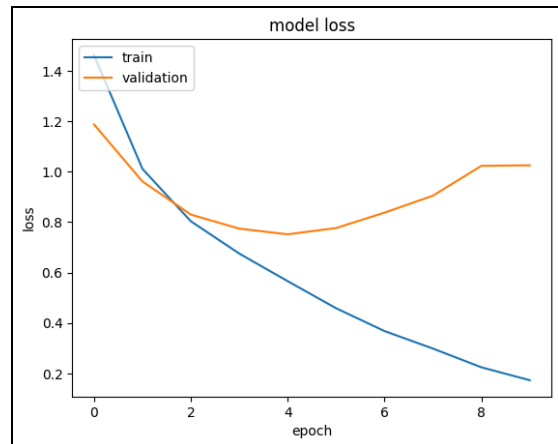
```
model2 = Sequential([
    Conv2D(64, (3, 3), padding='same', activation='relu',
input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),

    Conv2D(128, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(256, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
F1Score: 0.1957
accuracy: 0.9481
loss: 0.1538
val_F1Score: 0.1956
val_accuracy: 0.7437
val_loss: 1.0254
```



More convolutional layer (3 convolutional layer/pooling)

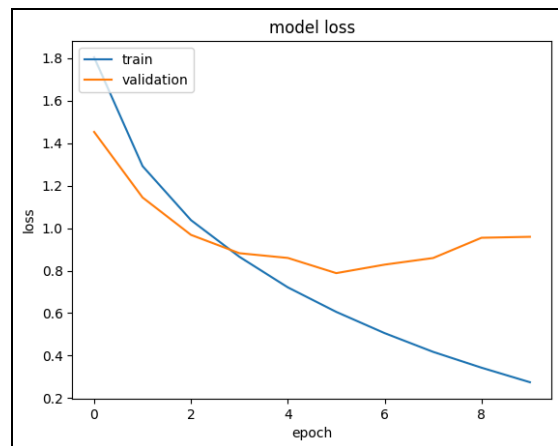
```
model3 = Sequential([
    Conv2D(64, (3, 3), padding='same', activation='relu',
input_shape=(32, 32, 3)),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(128, (3, 3), padding='same', activation='relu'),
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(256, (3, 3), padding='same', activation='relu'),
    Conv2D(256, (3, 3), padding='same', activation='relu'),
    Conv2D(256, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
F1Score: 0.1957
accuracy: 0.9094
loss: 0.2500
val_F1Score: 0.1956
val_accuracy: 0.7395
val_loss: 0.9591
```



Pengaruh banyak filter per layer konvolusi

More filter per layer (128->256->512)

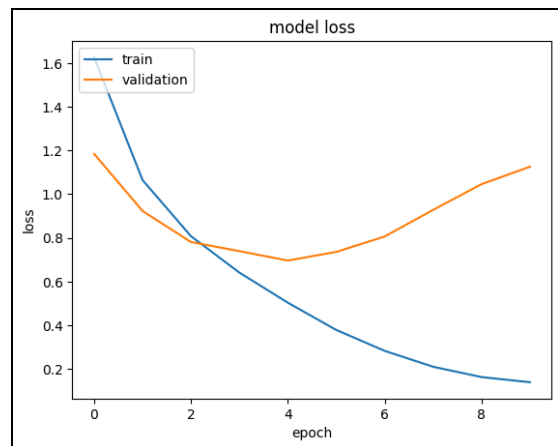
```
model4 = Sequential([
    Conv2D(128, (3, 3), padding='same', activation='relu',
input_shape=(32, 32, 3)),
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(256, (3, 3), padding='same', activation='relu'),
    Conv2D(256, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(512, (3, 3), padding='same', activation='relu'),
    Conv2D(512, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
F1Score: 0.1956
accuracy: 0.9588
loss: 0.1236
val_F1Score: 0.1955
val_accuracy: 0.7546
val_loss: 1.1257
```



Less filter per layer (32->64->128)

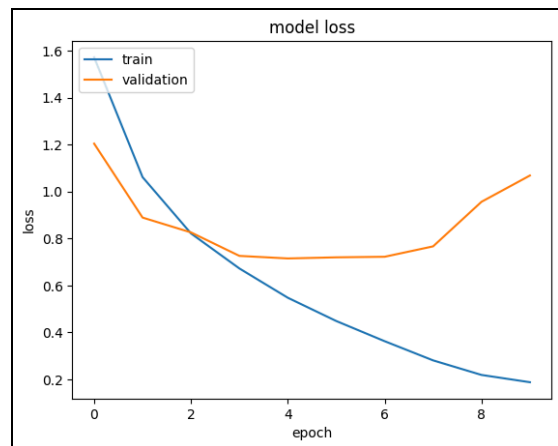
```
model5 = Sequential([
    Conv2D(32, (3, 3), padding='same', activation='relu',
input_shape=(32, 32, 3)),
    Conv2D(32, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), padding='same', activation='relu'),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(128, (3, 3), padding='same', activation='relu'),
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
F1Score: 0.1957
accuracy: 0.9428
loss: 0.1649
val_F1Score: 0.1957
val_accuracy: 0.7585
val_loss: 1.0681
```



Pengaruh ukuran filter per layer konvolusi

Bigger filter layer (5x5)

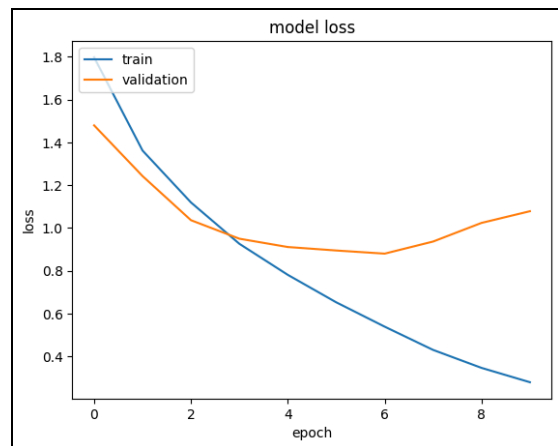
```
model6 = Sequential([
    Conv2D(64, (5, 5), padding='same', activation='relu',
input_shape=(32, 32, 3)),
    Conv2D(64, (5, 5), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(128, (5, 5), padding='same', activation='relu'),
    Conv2D(128, (5, 5), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(256, (5, 5), padding='same', activation='relu'),
    Conv2D(256, (5, 5), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
F1Score: 0.1956
accuracy: 0.9135
loss: 0.2500
val_F1Score: 0.1956
val_accuracy: 0.7103
val_loss: 1.0784
```



Smaller filter layer (2x2)

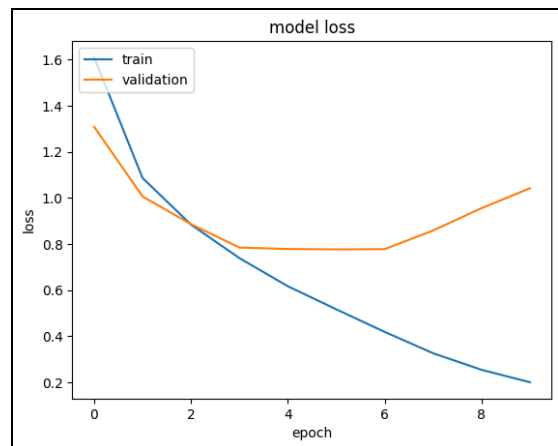
```
model7 = Sequential([
    Conv2D(64, (2, 2), padding='same', activation='relu',
input_shape=(32, 32, 3)),
    Conv2D(64, (2, 2), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(128, (2, 2), padding='same', activation='relu'),
    Conv2D(128, (2, 2), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Conv2D(256, (2, 2), padding='same', activation='relu'),
    Conv2D(256, (2, 2), padding='same', activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
F1Score: 0.1956
accuracy: 0.9334
loss: 0.1876
val_F1Score: 0.1956
val_accuracy: 0.7498
val_loss: 1.0416
```



Pengaruh jenis pooling layer

Average pooling

```
model8 = Sequential([
    Conv2D(64, (3, 3), padding='same', activation='relu',
input_shape=(32, 32, 3)),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    AveragePooling2D((2, 2)),

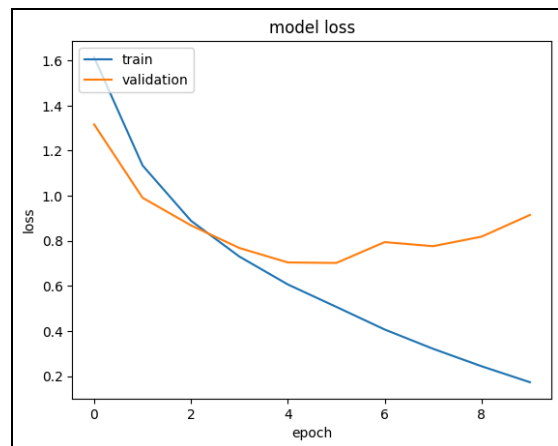
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    AveragePooling2D((2, 2)),

    Conv2D(256, (3, 3), padding='same', activation='relu'),
    Conv2D(256, (3, 3), padding='same', activation='relu'),
    AveragePooling2D((2, 2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```



```
F1Score: 0.1957
accuracy: 0.9467
loss: 0.1553
val_F1Score: 0.1956
val_accuracy: 0.7726
val_loss: 0.9149
```



Pengujian feedforward from scratch

```
# Feedforward Model from scratch
# Using Baseline Model (Model 1)
conv1_W, conv1_b = model.layers[0].get_weights()
conv2_W, conv2_b = model.layers[1].get_weights()
conv3_W, conv3_b = model.layers[3].get_weights()
conv4_W, conv4_b = model.layers[4].get_weights()
conv5_W, conv5_b = model.layers[6].get_weights()
conv6_W, conv6_b = model.layers[7].get_weights()
dense1_W, dense1_b = model.layers[10].get_weights()
dense2_W, dense2_b = model.layers[11].get_weights()

conv1 = Conv2DLayer(kernel=conv1_W, bias=conv1_b, activation='relu')
conv2 = Conv2DLayer(kernel=conv2_W, bias=conv2_b, activation='relu')

conv3 = Conv2DLayer(kernel=conv3_W, bias=conv3_b, activation='relu')
conv4 = Conv2DLayer(kernel=conv4_W, bias=conv4_b, activation='relu')

conv5 = Conv2DLayer(kernel=conv5_W, bias=conv5_b, activation='relu')
conv6 = Conv2DLayer(kernel=conv6_W, bias=conv6_b, activation='relu')

dense1 = DenseLayer(W=dense1_W, b=dense1_b, activation='relu')
dense2 = DenseLayer(W=dense2_W, b=dense2_b, activation='softmax')

pool1 = Pooling2DLayer(pool_size=(2, 2), strides=(2, 2), type='max')
pool2 = Pooling2DLayer(pool_size=(2, 2), strides=(2, 2), type='max')
pool3 = Pooling2DLayer(pool_size=(2, 2), strides=(2, 2), type='max')
```

```
flatten = FlattenLayer()
```

```
# Predict function
def predict_single(x):
    x = conv1.forward(x)
    x = conv2.forward(x)
    x = pool1.forward(x)
    x = conv3.forward(x)
    x = conv4.forward(x)
    x = pool2.forward(x)
    x = conv5.forward(x)
    x = conv6.forward(x)
    x = pool3.forward(x)
    x = flatten.forward(x)
    x = dense1.forward(x)
    output = dense2.forward(x)
    return output

def predict_thread(X):
    results = []
    with ThreadPoolExecutor() as executor:
        for res in tqdm(executor.map(predict_single, X), total=len(X),
desc="Predicting"):
            results.append(res)
    return results
```

Hasil feedforward library keras	Hasil feedforward from scratch
F1 Score (macro): 0.7612228830405996	F1 Score (macro): 0.7613187146552625
# true value vs predicted Total diff: 2410 out of 10000	# true value vs predicted Total diff: 2409 out of 10000

Pengujian feedforward from scratch

```
# Feedforward Model from scratch
rnn_scratch = RNNScratch(
    input_weight=input_weight,
```

```

        hidden_weight=hidden_weight,
        hidden_bias_weight=bias_weight
    )

    scratch_rnn_output = rnn_scratch.forward(input_feature=x_test_embedded)

```

```

# Scratch (Manual predict...)
dense_layer = lib_model.get_layer(index=-1)
dense_weights = dense_layer.get_weights()
W_dense = dense_weights[0]
b_dense = dense_weights[1]
out = np.matmul(scratch_rnn_output, W_dense) + b_dense
scratch_pred = softmax(out)
scratch_pred_label = np.argmax(scratch_pred, axis=1)
scratch_f1 = f1_score(y_test, scratch_pred_label, average='macro')

```

Hasil feedforward library keras	Hasil feedforward from scratch
F1 Score (macro): 0.3548	F1 Score (macro): 0.3761

Pembahasan

Pengaruh jumlah layer konvolusi

Jumlah layer konvolusi berpengaruh pada banyaknya langkah yang dilakukan untuk melakukan ekstraksi suatu feature. Semakin banyak layer, semakin banyak langkah dan berarti semakin banyak “simplifikasi” atau abstraksi dari gambar awal yang diterima sampai akhir dari proses feedforwarding. Nilai yang dihasilkan dari pengaruh jumlah layer konvolusi tidak terlalu berpengaruh dibandingkan dengan model baseline pada percobaan ini. Ketiga model memiliki F1 score sebesar 0.1956. Pada kasus ini, model baseline memiliki performa terbaik dari akurasi validasinya.

Pengaruh banyak filter per layer konvolusi

Banyak filter per layer konvolusi berpengaruh pada banyak feature yang diekstrak dari gambar. Semakin banyak filter, semakin banyak juga feature yang dideteksi dan diekstrak dari gambar. Pada percobaan kali ini, model less filter per memiliki F1 score yang terbaik yaitu sebesar 0.1957. Model baseline berada di posisi kedua dengan F1 score sebesar 0.1956 dan model more filter per layer dengan F1 score terburuk dengan nilai sebesar 0.1955. Filter yang terlalu banyak dapat menyebabkan terlalu banyak feature untuk dipertimbangkan pada hasil akhir yang dapat mengakibatkan overfitting. Tidak dapat ditarik kesimpulan secara linier bahwa semakin

banyak filter semakin baik atau buruk. Performa yang dihasilkan tergantung pada ukuran dari gambar input.

Pengaruh ukuran filter per layer konvolusi

Ukuran filter per layer konvolusi berpengaruh pada besar pada area yang dilihat sebagai satu fitur. Semakin besar ukuran filter, semakin besar receptive field sehingga area yang lebih besar dianggap sebagai suatu fitur. Sebaliknya, jika ukuran filter semakin kecil, maka receptive field akan semakin kecil sehingga semakin banyak feature dan semakin detail feature yang dipertimbangkan. Pada percobaan, model baseline, bigger filter, dan smaller filter memiliki F1 score yang sama yaitu 0.1956. Namun, baseline masih memiliki akurasi yang terbaik. Disusul oleh smaller filter dan terakhir bigger filter. Terdapat kemungkinan bahwa terlalu banyak feature yang tercampur dari ukuran 5x5 sehingga kehilangan beberapa detail feature dari gambar. Pada smaller feature, terlalu banyak feature tidak relevan yang dipertimbangkan sehingga menjadi “noise” saat melakukan prediksi.

Pengaruh jenis pooling layer

Jenis pooling layer menentukan pixel seperti apa yang akan lebih diperhitungkan. Max pooling akan lebih menekankan pada nilai yang dominan dari suatu receptive field. Average pooling mempertimbangkan seluruh elemen pada receptive field dan mengambil rata-ratanya. Pada percobaan, kedua model memiliki performa F1 score pada validation yang sama, yaitu 0.1956. Namun, model average pooling memiliki akurasi yang lebih baik, yaitu sebesar 0.7726. Hal ini kemungkinan besar terjadi karena gambar pada input merupakan objek dengan background yang halus dan tidak terlalu menonjol sehingga terdapat feature yang diabaikan oleh model baseline.

Feedforward from scratch

Pada percobaan ini, hanya dilakukan perbandingan prediksi antara feedforward dari library dengan feedforward from scratch. Pada percobaan, feedforward from scratch memiliki F1 score yang lebih baik dibandingkan dengan feedforward oleh library. Terdapat satu perbedaan prediksi dengan weight dan bias dari pelatihan yang sama. Hal ini bukan berarti feedforward from scratch lebih baik, tetapi karena perbedaan rounding error ketika melakukan perkalian elemen pada tensor.

b. Simple RNN

Fungsi untuk membuat model

```
embedding_dim = 100
max_tokens = len(vectorizer.get_vocabulary())
def create_rnn_model(n_layers=1, rnn_units=64, isBidirectional=False):
    model = Sequential()
```

```

model.add(Embedding(input_dim=max_tokens, output_dim=embedding_dim))

for i in range(n_layers):
    return_seq = True if i < n_layers - 1 else False
    if isBidirectional:
        model.add(Bidirectional(SimpleRNN(rnn_units,
return_sequences=return_seq)))
    else:
        model.add(SimpleRNN(rnn_units, return_sequences=return_seq))

model.add(Dropout(0.5))
model.add(Dense(3, activation='softmax'))

model.compile(loss='sparse_categorical_crossentropy',
optimizer=Adam(), metrics=['accuracy'])
return model

```

Fungsi untuk membuat variasi layer

```

histories_layers = {}
for layers in [1, 2, 3]:
    model_tmp = create_rnn_model(n_layers=layers, rnn_units=64,
isBidirectional=True)
    history = model_tmp.fit(x_train, y_train, validation_split=0.2,
epochs=5, batch_size=32)
    histories_layers[layers] = history

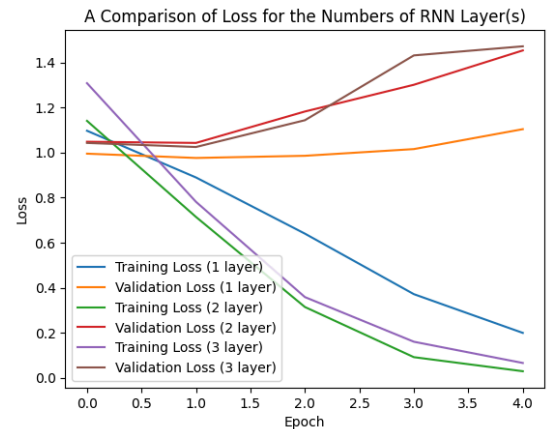
preds = model_tmp.predict(x_test)
preds_labels = np.argmax(preds, axis=1)
f1 = f1_score(y_test, preds_labels, average='macro')
print(f"Macro F1-score with {layers} layer(s) of RNN: {f1:.4f}")

```

Macro F1-score with 1 layer(s) of RNN: 0.4567

Macro F1-score with 2 layer(s) of RNN: 0.3968

Macro F1-score with 3 layer(s) of RNN: 0.4010



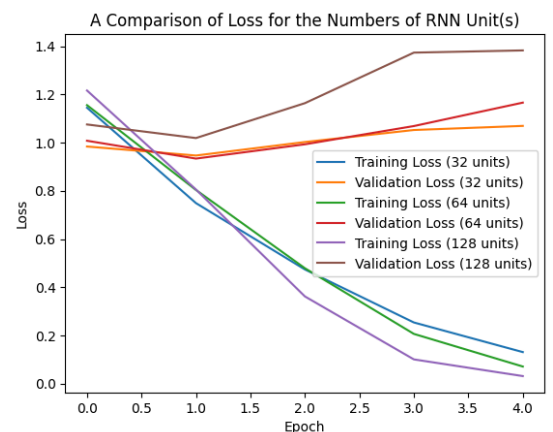
Fungsi untuk membuat variasi unit (cell) RNN

```
histories_units = {}
for units in [32, 64, 128]:
    model_tmp = create_rnn_model(n_layers=2, rnn_units=units,
isBidirectional=True)
    history = model_tmp.fit(x_train, y_train, validation_split=0.2,
epochs=5, batch_size=32)
    histories_units[units] = history
    preds = model_tmp.predict(x_test)
    preds_labels = np.argmax(preds, axis=1)
    f1 = f1_score(y_test, preds_labels, average='macro')
    print(f"Macro F1-score with {units} unit(s): {f1:.4f}")
```

Macro F1-score with 32 unit(s): 0.5121

Macro F1-score with 64 unit(s): 0.4439

Macro F1-score with 128 unit(s): 0.4405



Fungsi untuk membuat variasi arah

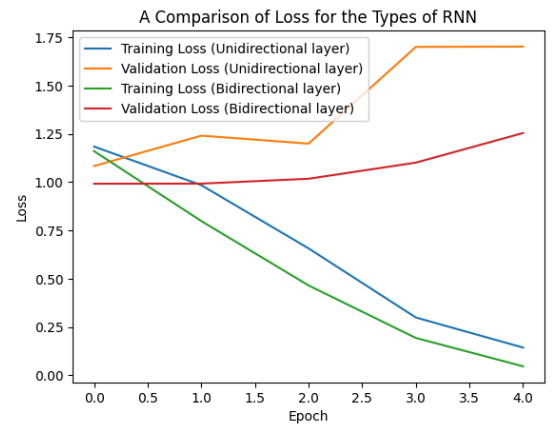
```

models_direction = {}
for direction in [False, True]:
    type = "Unidirectional" if not direction else "Bidirectional"
    model_tmp = create_rnn_model(n_layers=2, rnn_units=64,
isBidirectional=direction)
    history = model_tmp.fit(x_train, y_train, validation_split=0.2,
epochs=5, batch_size=32)
    models_direction[type] = (model_tmp, history)
    preds = model_tmp.predict(x_test)
    preds_labels = np.argmax(preds, axis=1)
    f1 = f1_score(y_test, preds_labels, average='macro')
    print(f"Macro F1-score with {type} RNN: {f1:.4f}")

```

Macro F1-score with Unidirectional
RNN: 0.3468

Macro F1-score with Bidirectional
RNN: 0.4916



Pembahasan

Pengaruh jumlah layer

Banyaknya layer berpengaruh pada banyaknya langkah yang dibutuhkan oleh model untuk menangkap pola kompleks (lebih banyak fitur yang diambil) dalam data terutama pada tugas yang membutuhkan pola data dalam jangka panjang. Sisi buruknya, penambahan layer yang berlebihan dapat menyebabkan vanishing gradient problem yang dapat menurunkan kinerja dalam pembelajaran oleh model dan masalah overfitting. Karena ketergantungannya terhadap timestep, semakin banyak layer, maka waktu yang dibutuhkan untuk inferensi dan proses belajar akan menjadi semakin lama.

Pengaruh jumlah unit (cell) RNN

Banyaknya unit (cell) RNN memungkinkan model menangkap lebih banyak informasi dari input sehingga dapat menyimpan lebih banyak data. Tetapi jika lebih banyak unit, kemungkinan overfitting menjadi lebih tinggi dan memperlambat proses inferensi dan belajar, sama seperti peningkatan jumlah layer.

Pengaruh arah (unidirectional vs bidirectional)

Unidirectional RNN memproses data dari time step ke-0 sampai ke-t. Jadi hanya dari kiri (masa lalu) ke kanan (masa depan). Bidirectional RNN memproses data secara 2 arah (bolak-balik). Bidirectional lebih cocok digunakan karena f1-score makro nya tinggi, jika direnungkan, kata-kata dalam kalimat berganti konteksnya bergantung pada kata sebelumnya atau sesudahnya sehingga hasil f1-score makro-nya sesuai dengan kenyataan.

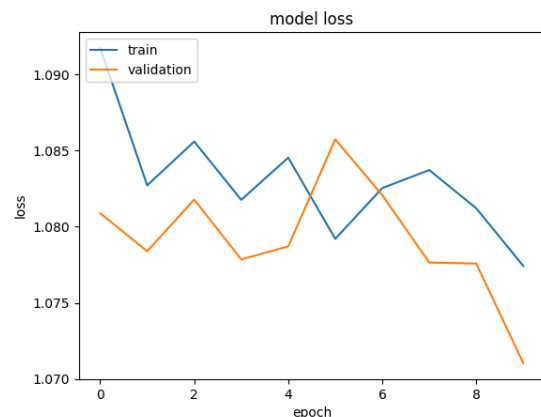
c. LSTM

Baseline

```
base = keras.models.Sequential([
    keras.layers.LSTM(256, return_sequences=True),
    keras.layers.LSTM(256, return_sequences=True),
    keras.layers.LSTM(256, return_sequences=True),
    keras.layers.LSTM(256),

    keras.layers.Dense(3, activation="softmax")
])
```

```
F1Score: 0.2093
accuracy: 0.3668
loss: 1.0768
val_accuracy: 0.38
val_loss: 1.071
```



- Pengaruh jumlah layer LSTM

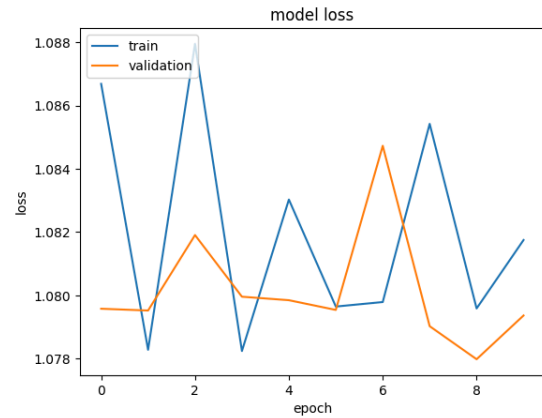
LSTM → LSTM → Dense (2 layer LSTM)

```
model1 = keras.models.Sequential([
    keras.layers.LSTM(256, return_sequences=True),
    keras.layers.LSTM(256),

    keras.layers.Dense(3, activation="softmax")
])
```


1)

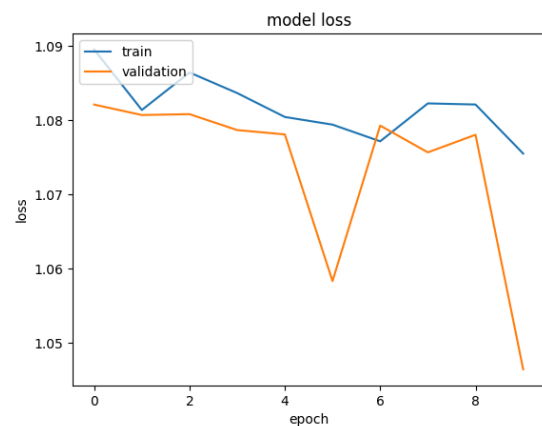
```
F1Score: 0.2258  
accuracy: 0.3869  
loss: 1.0691  
val_accuracy: 0.38  
val_loss: 1.0794
```



LSTM → LSTM → LSTM → Dense (3 layer LSTM)

```
model2 = keras.models.Sequential([  
    keras.layers.LSTM(256, return_sequences=True),  
    keras.layers.LSTM(256, return_sequences=True),  
    keras.layers.LSTM(256),  
  
    keras.layers.Dense(3, activation="softmax")  
])
```

```
F1Score: 0.4011  
accuracy: 0.3829  
loss: 1.0786  
val_accuracy: 0.47  
val_loss: 1.0464
```



LSTM → LSTM → LSTM → LSTM → LSTM → Dense (5 layer LSTM)

```
model3 = keras.models.Sequential([  
    keras.layers.LSTM(256, return_sequences=True),  
    keras.layers.LSTM(256, return_sequences=True),  
    keras.layers.LSTM(256, return_sequences=True),  
    keras.layers.LSTM(256, return_sequences=True),  
    keras.layers.LSTM(256),  
  
    keras.layers.Dense(3, activation="softmax")  
])
```

```

keras.layers.LSTM(256, return_sequences=True),
keras.layers.LSTM(256, return_sequences=True),
keras.layers.LSTM(256),

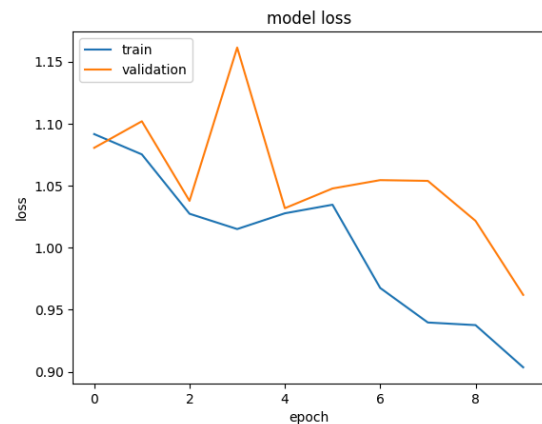
keras.layers.Dense(3, activation="softmax")
])

```

```

F1Score: 0.462
accuracy: 0.5899
loss: 0.8388
val_accuracy: 0.53
val_loss: 0.962

```



- Banyak cell LSTM per layer

LSTM(64) → LSTM(128) → LSTM(512) → LSTM(256) → Dense

```

model4 = keras.models.Sequential([
    keras.layers.LSTM(64, return_sequences=True),
    keras.layers.LSTM(128, return_sequences=True),
    keras.layers.LSTM(512, return_sequences=True),
    keras.layers.LSTM(256),

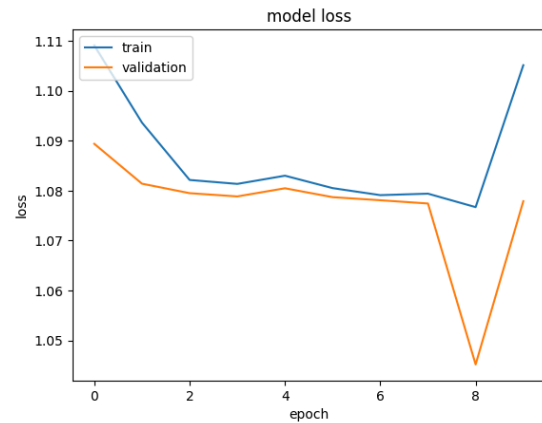
    keras.layers.Dense(3, activation="softmax")
])

```

```

F1Score: 0.2093
accuracy: 0.3566
loss: 1.1112
val_accuracy: 0.38
val_loss: 1.0779

```



LSTM(32) → LSTM(256) → LSTM(64) → LSTM(32) → Dense

```

model5 = keras.models.Sequential([
    keras.layers.LSTM(32, return_sequences=True),
    keras.layers.LSTM(256, return_sequences=True),
    keras.layers.LSTM(64, return_sequences=True),
    keras.layers.LSTM(32),

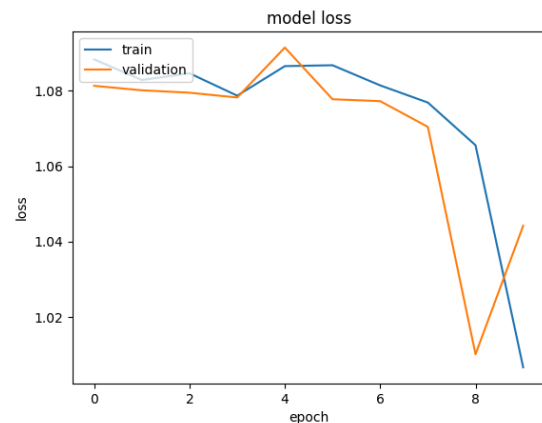
    keras.layers.Dense(3, activation="softmax")
])

```

```

F1Score: 0.4411
accuracy: 0.4449
loss: 1.0118
val_accuracy: 0.48
val_loss: 1.0442

```



LSTM(32) → LSTM(32) → LSTM(64) → LSTM(512) → Dense

```

model6 = keras.models.Sequential([
    keras.layers.LSTM(32, return_sequences=True),
    keras.layers.LSTM(32, return_sequences=True),
    keras.layers.LSTM(64, return_sequences=True),

```

```

keras.layers.LSTM(512),

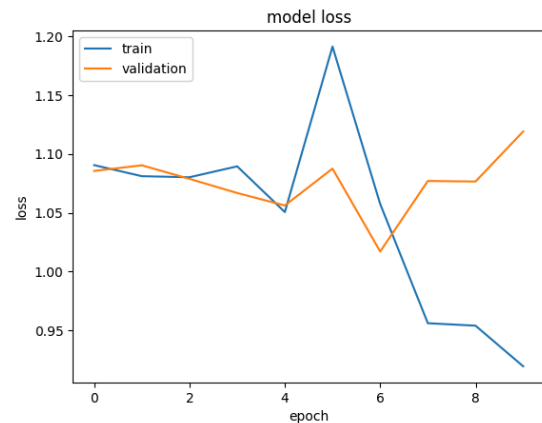
keras.layers.Dense(3, activation="softmax")
])

```

```

F1Score: 0.4142
accuracy: 0.523
loss: 0.9165
val_accuracy: 0.5
val_loss: 1.1191

```



- Jenis layer berdasarkan arah

Unidirectional

```

model7 = keras.models.Sequential([
    keras.layers.LSTM(128, return_sequences=True),
    keras.layers.LSTM(128, return_sequences=True),
    keras.layers.LSTM(128, return_sequences=True),
    keras.layers.LSTM(64),

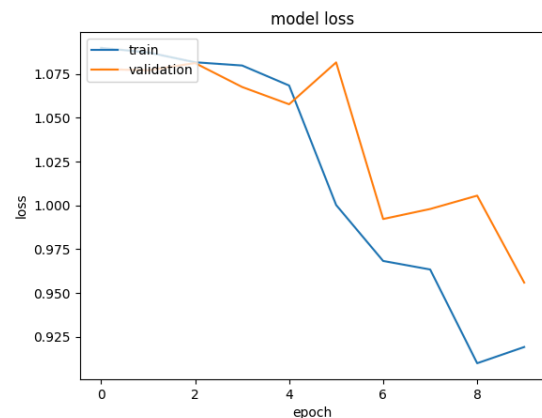
    keras.layers.Dense(3, activation="softmax")
])

```

```

F1Score: 0.4739
accuracy: 0.5417
loss: 0.9092
val_accuracy: 0.53
val_loss: 0.9559

```

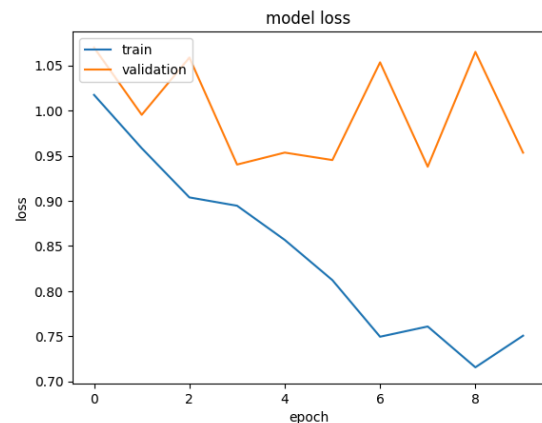


Bidirectional

```
model8 = keras.models.Sequential([
    keras.layers.Bidirectional(keras.layers.LSTM(128,
return_sequences=True)),
    keras.layers.Bidirectional(keras.layers.LSTM(128,
return_sequences=True)),
    keras.layers.Bidirectional(keras.layers.LSTM(128,
return_sequences=True)),
    keras.layers.Bidirectional(keras.layers.LSTM(64)),

    keras.layers.Dense(3, activation="softmax")
])
```

```
F1Score: 0.4752
accuracy: 0.6213
loss: 0.7586
val_accuracy: 0.52
val_loss: 0.9534
```



Pengujian feed forward from scratch

```
library_model = keras.models.Sequential([
    keras.layers.LSTM(256, return_sequences=True,
input_shape=(x_train_embed.shape[1], x_train_embed.shape[2])),
    keras.layers.LSTM(256, return_sequences=True),
    keras.layers.LSTM(256),
    keras.layers.Dropout(0.3),

    keras.layers.Dense(3, activation="softmax")
])

library_model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer="adam",
```

```

        metrics=['accuracy']
    )

    library_model.load_weights("./training_result/model2.weights.h5")

```

```

library_result = library_model.predict(x_test_embed)
scratch_result = predict_scratch(x_test_embed)

```

Hasil feedforward library keras	Hasil feedforward from scratch
F1 Score (macro): 0.18269812462189958	F1 Score (macro): 0.18269812462189958

Pembahasan

Pengaruh jumlah layer LSTM

Jumlah layer pada LSTM berfungsi untuk menangkap pola sekuensial yang terdapat pada data. Semakin banyak layer LSTM, semakin banyak juga pola sekuensial yang dapat ditemukan oleh model dan membuat kinerja model menjadi semakin baik. Hal ini dapat dilihat pada model 1, 2, dan 3. Model 1 memiliki layer yang lebih sedikit dibandingkan dengan model 2 dan 3, hasilnya model 1 memiliki nilai F1 score dan akurasi yang lebih kecil dibandingkan dengan kedua model tersebut. Model 3 dengan jumlah layer terbanyak, memiliki nilai F1 score dan akurasi yang paling tinggi. Oleh karena itu, semakin banyak jumlah layer LSTM, maka semakin baik juga kinerja model. Namun, hal tersebut tidak selalu terjadi. Model memiliki jumlah layer LSTM optimal dan jika melewati jumlah tersebut, maka model dapat mengalami overfitting dan menyebabkan kinerjanya menurun.

Pengaruh banyak cell LSTM per layer

Banyaknya cell LSTM per layer berpengaruh terhadap jumlah informasi yang dapat disimpan oleh satu layer. Semakin banyak cell LSTM, semakin banyak juga jumlah informasi yang dapat disimpan dalam 1 layer. Berdasarkan model 4, 5, dan 6, dapat dilihat bahwa semakin banyak cell LSTM, semakin baik juga kinerja model. Model 6 adalah model yang paling baik di antara ketiganya karena memiliki jumlah cell LSTM yang banyak, tetapi tidak berlebihan sehingga tidak menyebabkan overfitting. Model 4 adalah model dengan jumlah cell LSTM terbanyak, tetapi model tersebut merupakan model terburuk di antara ketiganya. Hal ini dapat terjadi karena jumlah cell LSTM terlalu banyak dan jumlah layer pun cukup banyak sehingga menyebabkan *gradient vanishing*.

Pengaruh jenis layer berdasarkan arah

Jenis layer berdasarkan arah dibedakan menjadi dua, yaitu unidirectional dan bidirectional. Unidirectional hanya memproses data dari masa lalu ke masa depan,

sedangkan bidirectional memproses data dari dua arah, yaitu masa lalu ke masa depan dan masa depan ke masa lalu. Bidirectional merupakan model yang lebih baik dibandingkan dengan unidirectional karena model ini memproses data dari dua arah dan mampu memperoleh informasi lebih banyak dari model unidirectional. Hal ini juga didukung oleh hasil dari model 7 (unidirectional) dan model 8 (bidirectional). Model 8 memiliki hasil yang lebih baik daripada model 7 sehingga dapat dibuktikan bahwa bidirectional lebih baik daripada unidirectional. Namun, waktu dan komputasi yang diperlukan oleh model bidirectional biasanya lebih banyak daripada unidirectional sehingga harus diperhatikan kembali penggunaannya.

Bab 3

Kesimpulan dan Saran

Pada CNN, hubungan antara parameter yang bersifat kuantitatif tidak dapat diprediksi secara linier terhadap performa yang dihasilkan. Semakin banyak jumlah layer, banyak filter, dan ukuran filter bukan berarti performa yang dihasilkan akan semakin baik. Pada percobaan ini, model ke-8 (Average pooling) merupakan model yang memiliki performa terbaik dari sisi F1 score dan akurasi. Hal ini dikarenakan model baseline yang sudah memiliki performa yang baik ditambah dengan modifikasi pada pooling dengan average pooling yang memiliki performa baik pada gambar dengan elemen yang tidak terlalu mendominasi elemen lainnya.

Pada implementasi CNN *from scratch*, hasil implementasi sudah berhasil melakukan *forward propagation* dan dapat menerima *weight* serta *bias* yang dihasilkan oleh *library* keras. Terdapat satu perbedaan prediksi antara CNN *from scratch* dengan CNN *library* keras yang disebabkan oleh rounding error saat melakukan perkalian *floating point* number.

Pada RNN, peningkatan salah satu parameter kuantitatif tidak dapat diprediksi secara linier terhadap performa yang dihasilkan. Dari variasi yang telah dilakukan pada RNN milik Keras, peningkatan jumlah layer ataupun unit tidak dapat ditentukan apakah meningkatkan performa atau tidak, hal ini bisa dilihat pada variasi layer, f1-score-nya berkurang ketika ditambah menjadi 2, tetapi bertambah kembali ketika layer-nya ditambah menjadi 3. Di antara semua variasi yang telah dilakukan, model ke-4 (2 layer, 32 unit, bidirectional) merupakan model dengan hasil f1-score macro tertinggi diantara 8 model yang telah diuji.

Pada implementasi RNN *from scratch*, hasil implementasi sudah berhasil melakukan *forward propagation* dan dapat menerima *weight* serta *bias* yang dihasilkan oleh pustaka Keras. f1-score yang dihasilkan oleh kedua model (pustaka dan *scratch*) mirip (0.3548 vs 0.3761). Terdapat perbedaan prediksi antara RNN *from scratch* dengan RNN pustaka Keras. Hal ini mungkin dikarenakan oleh RNN *from scratch* yang melakukan *forward propagation* menggunakan *weights* yang dimiliki oleh model pustaka.

Pada LSTM, jumlah layer atau jumlah cell LSTM yang lebih banyak dapat meningkatkan kinerja model. Namun, ada batasan jumlah yang optimal sehingga ketika batas tersebut terlewati, kinerja model akan menurun karena *overfitting*. Selain itu, jika kita hanya menggunakan layer LSTM saja tanpa layer Dropout atau layer lainnya, hasilnya tidak terlalu bagus, meskipun jumlah layer LSTM dan cell LSTM ditingkatkan.


Pada implementasi LSTM *from scratch*, hasil implementasi sudah berhasil melakukan *forward propagation* dan menerima *weight* serta *bias* yang dihasilkan oleh *library* keras. Hasil prediksi antara model LSTM dari keras dengan model LSTM *from scratch* hampir sama, hanya terdapat perbedaan yang sangat kecil setelah melewati layer dense (biasanya selisih 0.000001). Namun, jika nilai tersebut dibulatkan ataupun diubah menjadi label sebenarnya, hasilnya selalu sama antara LSTM keras dengan LSTM *from scratch*.

Bab 4

Pembagian Tugas

NIM	Nama	Kontribusi
13522088	Muhamad Rafli Rasyiidin	LSTM
13522094	Andhika Tanyo Anugrah	RNN
13522100	M. Hanief Fatkhan Nasrullah	CNN

Referensi

1.  Spesifikasi Tugas Besar 2 IF3270 Pembelajaran Mesin
2. [numpy.einsum - NumPy v2.2 Manual](#)
3. [numpy.lib.stride_tricks.sliding_window_view - NumPy v2.2 Manual](#)
4. [Softmax function - Wikipedia](#)
5. [What is the difference between 'SAME' and 'VALID' padding in tf.nn.max_pool of tensorflow?](#)