

Laporan Tugas Besar 1

IF3270 - Pembelajaran Mesin



Oleh kelompok 22:

Nama	NIM
Renaldy Arief Susanto	13522022
Kristo Anugrah	13522024
Nyoman Ganadipa Narayana	13522066

1. Deskripsi Persoalan

Pada tugas ini, mahasiswa diminta untuk mengimplementasikan modul untuk algoritma **forward propagation** pada CNN (*Convolutional Neural Network*) dan RNN (*Recurrent Neural Network*) *from scratch*, sebagaimana yang telah dipelajari di perkuliahan. Bobot untuk *forward propagation* diperoleh dari *training* yang dilakukan dengan library `keras`. Algoritma diimplementasikan dalam bahasa Python dan akan diuji dengan variasi parameter tertentu (seperti jumlah layer, jenis layer, dan lainnya).

2. Pembahasan

2.1 Penjelasan Implementasi

Pada subbab ini akan dijelaskan tentang implementasi model FFNN.

2.1.1 Deskripsi Kelas, Atribut dan Metode

Berikut adalah struktur proyek. Catatan: ada direktori atau *file* yang tidak ditampilkan karena tidak relevan, misalnya *script* untuk *testing* atau folder penyimpanan *weights*.

```
.
├── __init__.py
├── activation_functions
│   ├── __init__.py
│   ├── base.py
│   ├── relu.py
│   ├── sigmoid.py
│   ├── softmax.py
│   └── tanh.py
├── layers
│   ├── __init__.py
│   ├── avg_pooling2d.py
│   ├── base.py
│   ├── bidirectional_rnn.py
│   ├── conv2d.py
│   ├── dense.py
│   ├── dropout.py
│   ├── embedding.py
│   ├── flatten.py
│   ├── lstm.py
│   ├── max_pooling2d.py
│   └── simple_rnn.py
├── models
│   ├── __init__.py
│   ├── cnn.py
│   ├── lstm.py
│   └── rnn.py
├── playgrounds
│   ├── gana_rnn.ipynb
│   ├── kristo_cnn.ipynb
│   ├── lstm.ipynb
│   └── starter.py
└── utils
    ├── __init__.py
    ├── data_loader.py
    ├── keras_weight_loader.py
    ├── metrics_calculator.py
    └── text_preprocessor.py
```

Intinya, model-model *from scratch* disimpan di modul `models`, dan pada model-model tersebut melakukan import dari modul `layers` serta modul `activation_functions`. Kemudian, ada folder `utils` yang digunakan untuk memuat dan memproses data, memuat bobot, serta melakukan kalkulasi metrik.

Masing-masing model mempunyai dua metode yang utama, yaitu `forward` untuk melakukan *forward propagation*, serta `set_weights` atau `load_weights` untuk memuat *weights*. Khusus untuk CNN, bisa dihitung *loss*, dilakukan *training*, dan *back propagation*.

Modul `layers` mengandung seluruh definisi jenis layer yang akan digunakan, dan pada masing-masing kelas layer mempunyai metode `forward` untuk *forward propagation*, dan juga `set_weights` atau `load_weights` untuk dipanggil secara iteratif oleh model *neural network*. Tentunya ada detail implementasi lain yang spesifik ke masing-masing jenis layer. Satu contoh saja, pada layer LSTM didefinisikan sebuah kelas `LSTMCell` yang mempunyai metode `forward_step` dan ini berguna untuk melakukan satu *time step* pada sel LSTM.

Untuk *activation functions*, tidak dibahas karena merupakan jangkauan tugas besar 1.

2.1.2 Penjelasan Forward Propagation

Penjelasan-penjelasan pada bagian ini tidak mengulang penjelasan lengkap *forward propagation* dari awal, melainkan hanya ulasan singkat kemudian cuplikan implementasi berbentuk *pseudocode* (atau kode aslinya).

CNN

CNN adalah sebuah *feedforward neural network* dengan satu atau lebih ***convolution layer***. Convolution layer adalah pembeda CNN dengan FFNN biasa, layer ini didesain untuk mengatasi permasalahan ***spatial locality*** (menginterpretasi data dengan tambahan konteks posisi relatifnya terhadap data lain) dan juga menerima **input data dengan dimensi lebih**.

Cara kerja CNN adalah sebagai berikut. Bobot yang pada FFNN bentuknya adalah vektor kolom kini berupa kumpulan matriks, disebut dengan **kernel**. Perhatikan gambar berikut.

$$\begin{array}{|c|c|c|} \hline \text{Input} & & \\ \hline 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ \hline \end{array} * \begin{array}{|c|c|} \hline \text{Kernel} & \\ \hline 0 & 1 \\ 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline \text{Output} & \\ \hline 19 & 25 \\ 37 & 43 \\ \hline \end{array}$$

Pada gambar ini, ada satu input berdimensi $1 \times 3 \times 3$, dan satu kernel berukuran $1 \times 2 \times 2$. Output sebuah sel diperoleh dengan **jumlah dari component wise multiplication** pada elemen-elemen di matriks kernel dan matriks input. Jadi, untuk contoh di atas, nilai 19 diperoleh dari $(0 \times 0) + (1 \times 1) + (3 \times 2) + (4 \times 3) = 19$.

Satu kernel diadakan untuk masing-masing pasangan input matrix dan output matrix. Jadi, misal ada input berukuran 7×7 dengan 4 channel dan *batch size* 3, kemudian ingin dipetakan menjadi 2 matriks output. Artinya, akan dipelajari kernel sebanyak:

$$(4 \text{ channel}) \times (2 \text{ output count}) = 8 \text{ kernel}$$

Kemudian ukuran matriksnya bergantung pada *stride* dan *padding*, tetapi kami tidak mengimplementasi *stride* dan *padding* sehingga ukuran matriks ditentukan langsung oleh kami.

Note: penggunaan istilah **kernel** ini sebenarnya salah jika mengacu pada materi kuliah. Rupanya istilah **filter** lebih tepat, sedangkan kernel mengacu ke sekumpulan filter untuk satu proses konvolusi ke satu output. Jadi, pada contoh sebelumnya, ada **2 kernel**. Namun, pada kenyataannya mayoritas penggunaan kedua istilah ini bersifat *interchangeable* dan tidak begitu diperhatikan.

Berikut cuplikan kode kami untuk proses *forward propagation* pada sebuah *convolutional layer*.

```
for b in range(batch_size):
    for f in range(self.filters):
        for i in range(out_height):
            for j in range(out_width):

                # Extract partial input matrix
                patch = x[b, i:i+self.kernel_size[0], j:j+self.kernel_size[1], :]

                # Convolution operation
                output[b,i,j,f] = np.sum(patch*self.weights[:, :, :, f]) + self.biases[f]
```

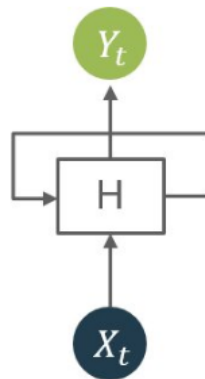
RNN

Recurrent Neural Network merupakan ekstensi dari FFNN biasa dengan ciri khas **memiliki memori internal**. Setelah menghasilkan output, output tersebut disalin dan dikirim kembali ke recurrent network. Kemudian untuk membuat sebuah keputusan, RNN mempertimbangkan input saat ini dan output yang telah dipelajari dari input sebelumnya.

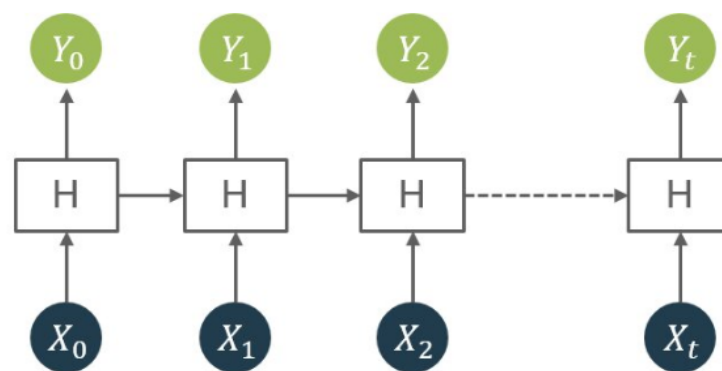
Desain ini ditujukan untuk menangani data deret **waktu** atau data yang melibatkan **urutan**. FFNN biasa hanya dimaksudkan untuk titik data yang independen satu sama lain.

Namun, jika kita memiliki data dalam urutan sedemikian rupa sehingga satu titik data bergantung pada titik data sebelumnya, kita perlu memodifikasi *neural network* untuk menggabungkan ketergantungan antara titik data ini.

Cara kerja RNN adalah sebagai berikut. Perhatikan *neural network* berikut yang telah disimplifikasi tampilannya.



Pada gambar di atas, X_t adalah input, H adalah output pada hidden layer (layaknya FFNN biasa), dan Y_t adalah output. Perhatikan bahwa ada **panah dari H ke dirinya sendiri**. Ini menunjukkan bahwa pada setiap iterasi, output H_t akan digunakan untuk memengaruhi H_{t+1} . Implikasinya, akan ada **bobot** yang dipelajari untuk memetakan pengaruh output yang sudah dihitung ke output-output berikutnya. Seringkali pada RNN, dilakukan istilah *unroll* untuk ilustrasi konsep ini secara lebih nyata.



Berikut cuplikan kode kami untuk proses *forward propagation* pada sebuah RNN layer.

```
def forward(self, inputs: np.ndarray) -> np.ndarray:
    if self.rnn_cell is None:
        self.build(inputs.shape)
```

```

batch_size, seq_length, input_size = inputs.shape

# Initialize hidden state
hidden_state = np.zeros((batch_size, self.hidden_size))

if self.return_sequences:
    outputs = np.zeros((batch_size, seq_length, self.hidden_size))
    for t in range(seq_length):
        hidden_state = self.rnn_cell.forward_step(inputs[:, t, :], hidden_state)
        outputs[:, t, :] = hidden_state

    return outputs

else:
    # Only return the last hidden state
    for t in range(seq_length):
        hidden_state = self.rnn_cell.forward_step(inputs[:, t, :], hidden_state)

    return hidden_state

```

Kemudian fungsi `forward_step` pada satu sel RNN adalah seperti berikut.

```

def forward_step(self, input_t: np.ndarray, hidden_t_prev: np.ndarray) ->
np.ndarray:
    #  $h_t = \text{activation}(W_{ih} @ x_t + W_{hh} @ h_{t-1} + b_h)$ 
    linear_out = (
        np.dot(input_t, self.W_ih.T) +
        np.dot(hidden_t_prev, self.W_hh.T) +
        self.b_h
    )
    return self.activation.forward(linear_out)

```

LSTM

Lorem lorem.

2.2 Hasil Pengujian

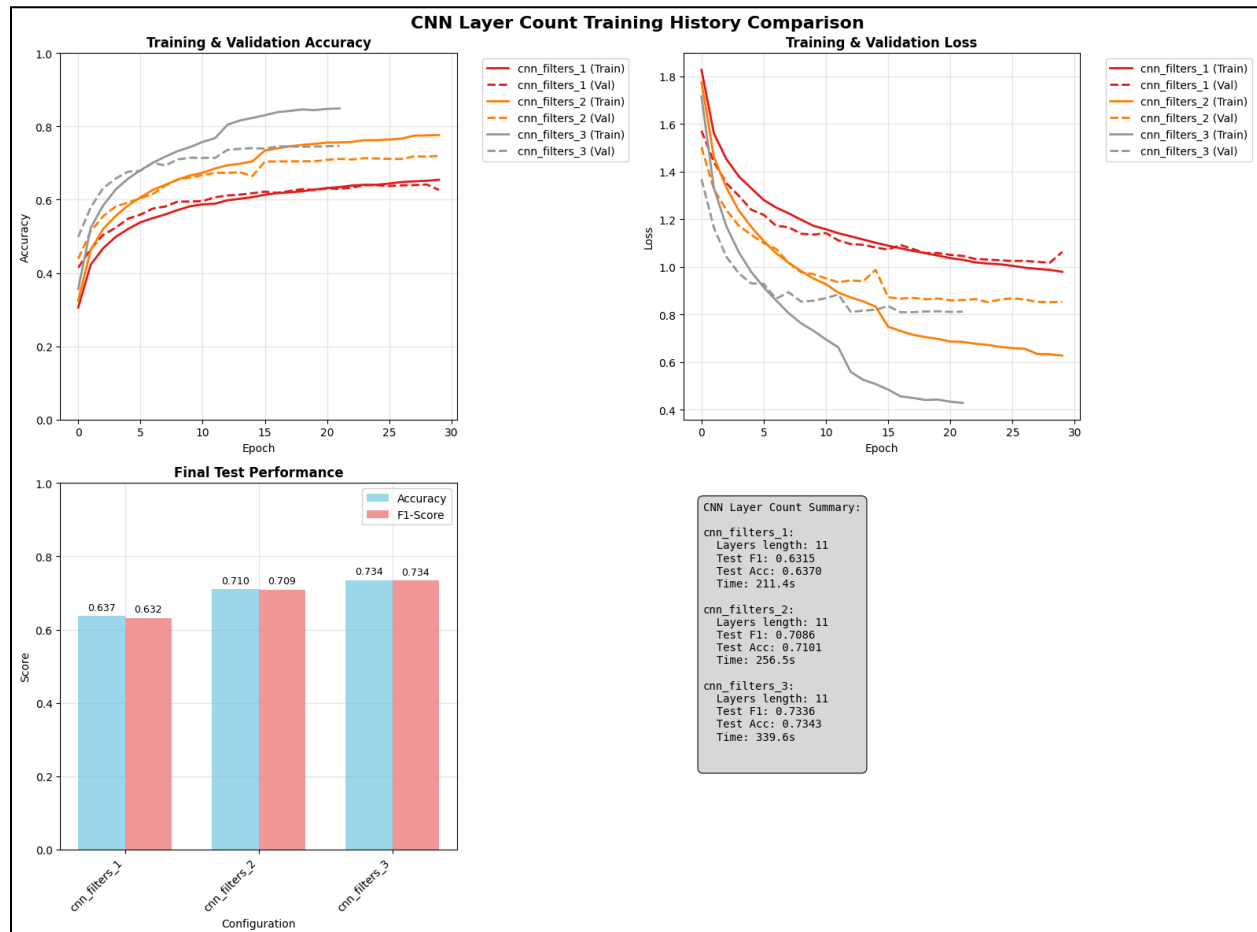
Akan dijelaskan hasil pengujian yang telah dilakukan sebagaimana diminta di spesifikasi tugas. Pengujian secara umum dilakukan di 3 *file ipynb* yang terletak pada direktori `src/playgrounds/`.

2.2.1 CNN

Berikut adalah hasil eksperimen dengan beberapa variasi parameter.

Jumlah Layer

Eksperimen dilakukan dengan 3 variasi jumlah layer, yaitu 1, 2, dan 3 layer. Hasil eksperimen dapat dilihat pada 3 grafik di bawah: F1-Score pada grafik kiri bawah, grafik *training & validation loss* pada kanan atas, serta akurasi pada grafik kiri atas.

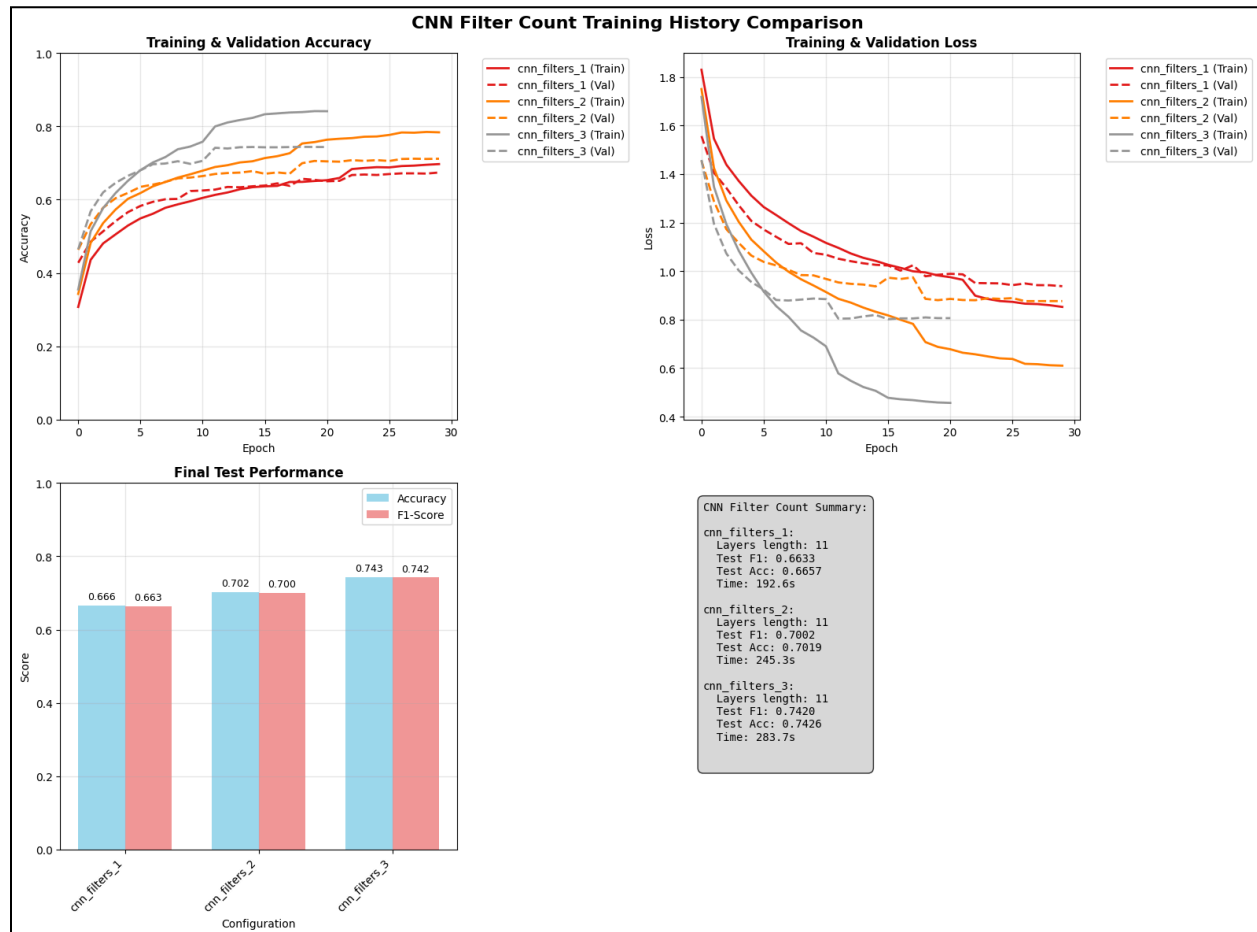


Agar memperjelas, warna **merah** untuk **1 layer**, warna **orange** untuk **2 layer**, dan **abu-abu** adalah untuk **3 layer**. Kemudian itu berturut-turut adalah pilar **kiri**, **tengah**, dan **kanan** di grafik performa. Dapat dilihat bahwa ada korelasi positif antara jumlah layer dan akurasi/performa.

Disimpulkan bahwa jumlah layer pada kasus ini berpengaruh **positif** pada performa model.

Jumlah Filter

Eksperimen dilakukan dengan 3 variasi jumlah layer, yaitu 8, 16, dan 32 filter. Hasil eksperimen dapat dilihat pada 3 grafik di bawah: F1-Score pada grafik kiri bawah, grafik *training & validation loss* pada kanan atas, serta akurasi pada grafik kiri atas.

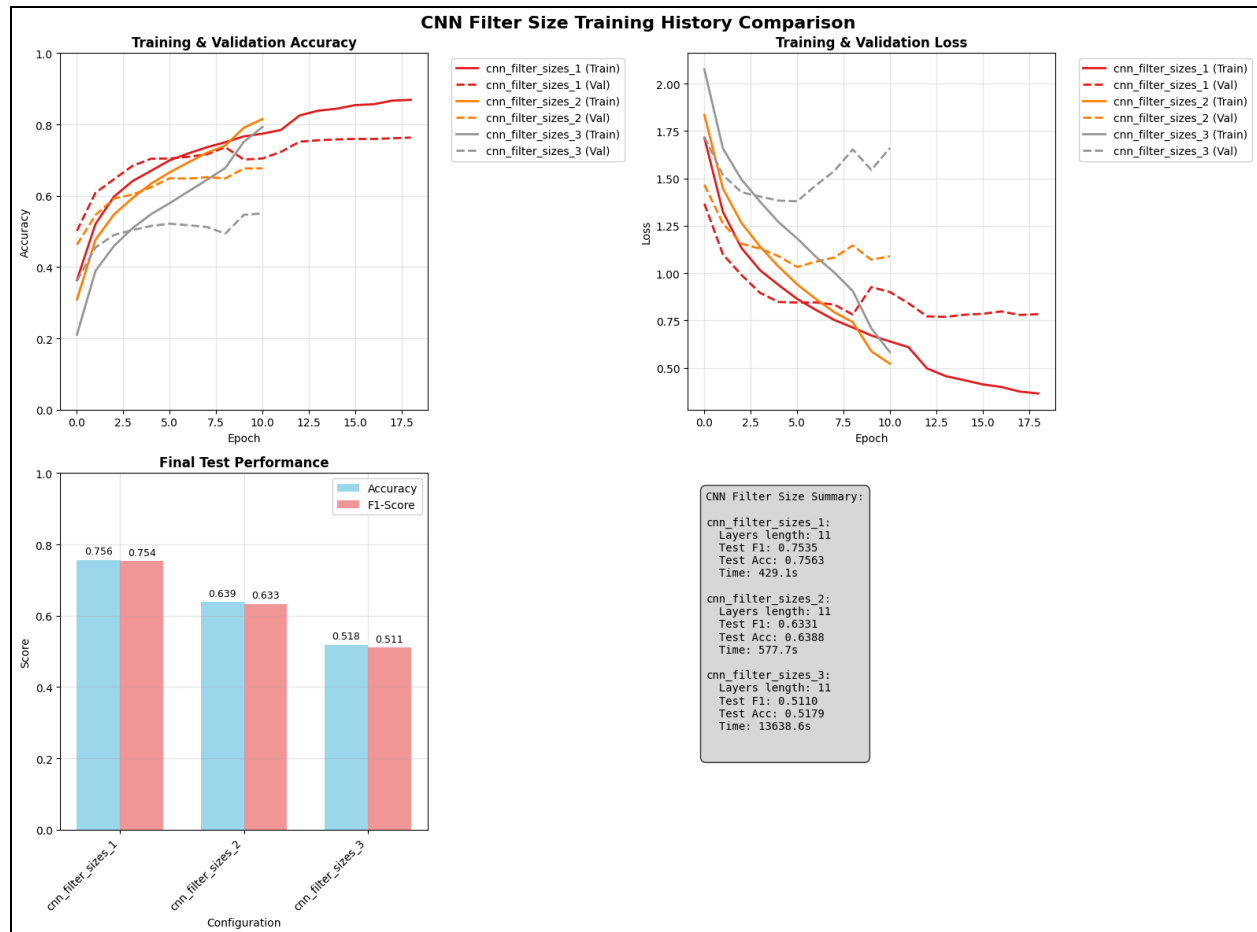


Agar memperjelas, warna **merah** untuk **8 filter**, warna **orange** untuk **16 filter**, dan **abu-abu** adalah untuk **32 layer**. Kemudian itu berturut-turut adalah pilar **kiri**, **tengah**, dan **kanan** di grafik performa. Dapat dilihat bahwa ada korelasi positif antara jumlah filter dan akurasi/performa.

Disimpulkan bahwa jumlah filter pada kasus ini berpengaruh **positif** pada performa model.

Ukuran Filter

Eksperimen dilakukan dengan 3 variasi ukuran filter, yaitu 3×3, 5×5, dan 7×7. Hasil eksperimen dapat dilihat pada 3 grafik di bawah: F1-Score pada grafik kiri bawah, grafik *training & validation loss* pada kanan atas, serta akurasi pada grafik kiri atas.

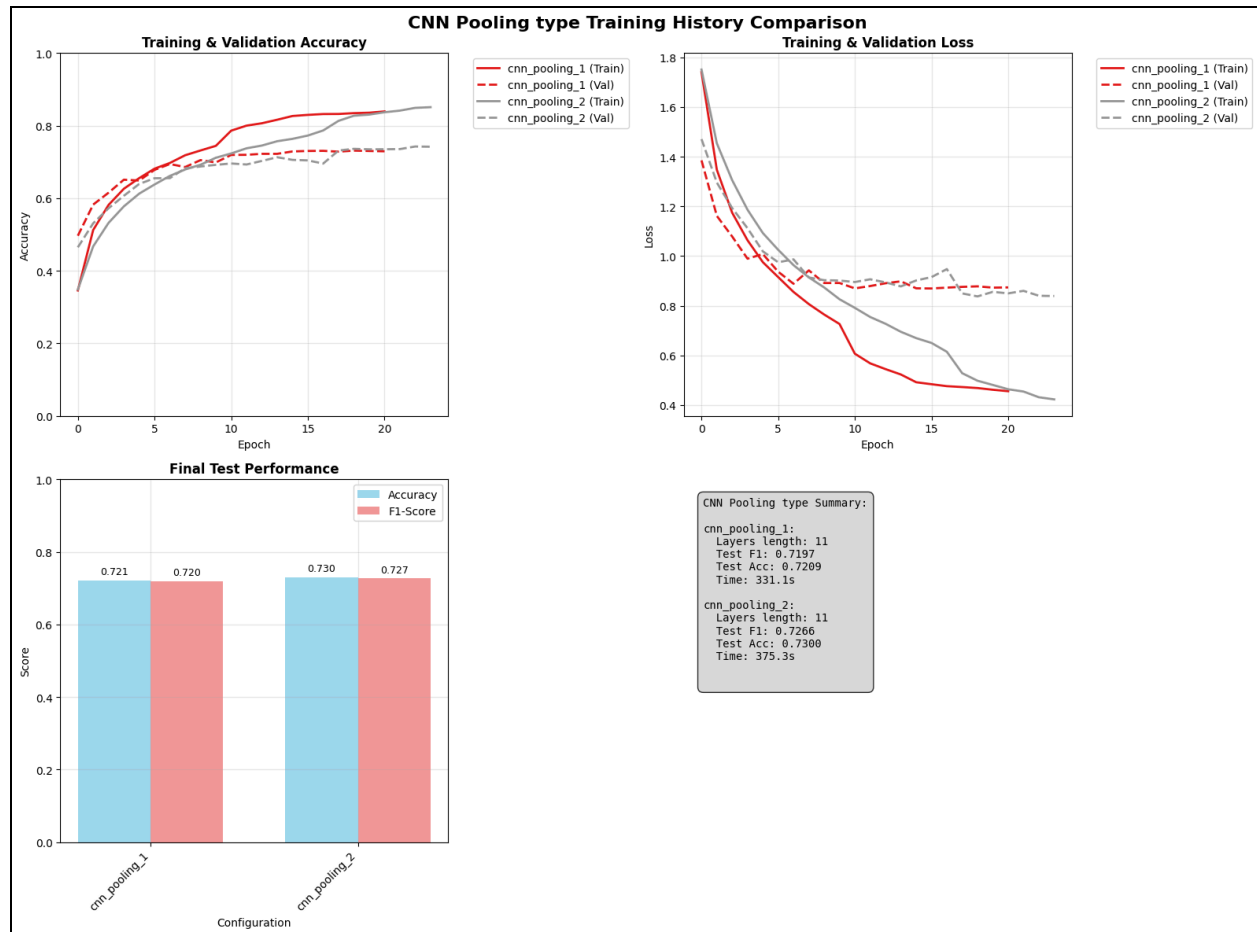


Agar memperjelas, warna **merah** untuk **filter 3×3**, warna **orange** untuk **filter 5×5**, dan **abu-abu** adalah untuk **layer 7×7**. Kemudian itu berturut-turut adalah pilar **kiri**, **tengah**, dan **kanan** di grafik performa. Dapat dilihat bahwa ada korelasi negatif antara ukuran filter dan akurasi/performa.

Disimpulkan bahwa jumlah filter pada kasus ini berpengaruh **negatif** pada performa model.

Jenis Pooling

Eksperimen dilakukan dengan 2 jenis pooling, yaitu *max pooling* dan *average pooling*. Hasil eksperimen dapat dilihat pada 3 grafik di bawah: F1-Score pada grafik kiri bawah, grafik *training & validation loss* pada kanan atas, serta akurasi pada grafik kiri atas.



Agar memperjelas, warna **merah** untuk **max pooling**, dan **abu-abu** adalah untuk **average pooling**. Kemudian itu berturut-turut adalah pilar **kiri** dan **kanan** di grafik performa. Dapat dilihat bahwa jenis pooling tidak terlalu memengaruhi akurasi/performa.

Disimpulkan bahwa jenis pooling pada kasus ini tidak memengaruhi akurasi/performa model.

Perbandingan dengan Keras

Model *from scratch* berhasil menyamakan output dengan *forward propagation* menggunakan keras. Perbandingan lebih detail dapat dilihat pada sel terakhir pada *file* pengujian `src/playgrounds/kristo_cnn.ipynb`.

2.2.2 RNN

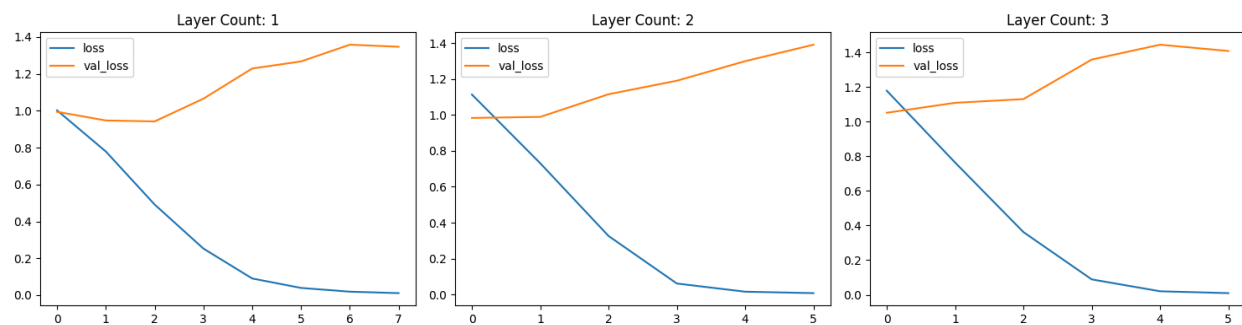
Berikut adalah hasil eksperimen dengan beberapa variasi parameter.

Jumlah Layer

Eksperimen dilakukan dengan variasi jumlah layer 1, 2, dan 3 layer. Berikut adalah performa prediksinya.

Jumlah Layer	Akurasi	F1-Score
1	0.6350	0.6284
2	0.5225	0.5006
3	0.5325	0.5059

Kemudian berikut adalah grafik *loss* dan *validation loss* terhadap jumlah epoch untuk masing-masing jumlah layer.



Pada gambar di atas, terlihat bahwa jumlah layer tidak jauh berbeda satu sama lain, tetapi dari F1-Score-nya, jumlah layer 1 merupakan yang terbaik. Kemungkinan ini ada hubungannya dengan jumlah layer sedikit lebih baik apabila klasifikasi bersifat lebih simpel atau tidak banyak fitur untuk diekstrak dari dataset.

Disimpulkan bahwa jumlah layer yang terlalu besar mungkin terlalu mengekstrak fitur-fitur tertentu sehingga terjadi *overfitting*.

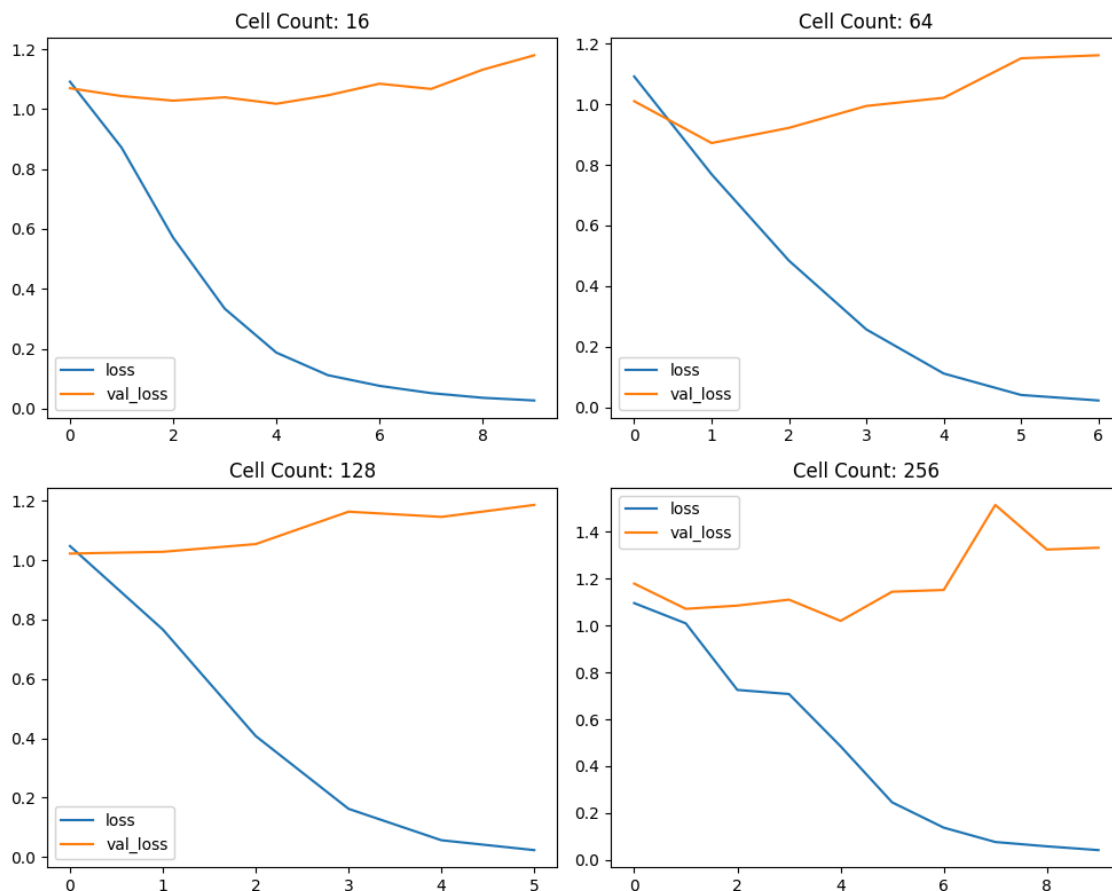
Jumlah Unit/Sel

Eksperimen dilakukan dengan variasi jumlah unit 16, 64, 128, dan 256 unit. Berikut adalah performa prediksinya.

Jumlah Unit	Akurasi	F1-Score
16	0.5200	0.4895
64	0.5800	0.4636

128	0.5000	0.5005
256	0.4775	0.3891

Kemudian berikut adalah grafik *loss* dan *validation loss* terhadap jumlah epoch untuk masing-masing jumlah unit/sel.



Pada gambar di atas, terlihat bahwa masing-masing jumlah unit stabil pada *validation loss*-nya, kecuali jumlah unit 256 yang mempunyai *spike* di akhir. Selain itu, hanya jumlah unit 256 yang F1-Score-nya relatif lebih jelek dari yang lain.

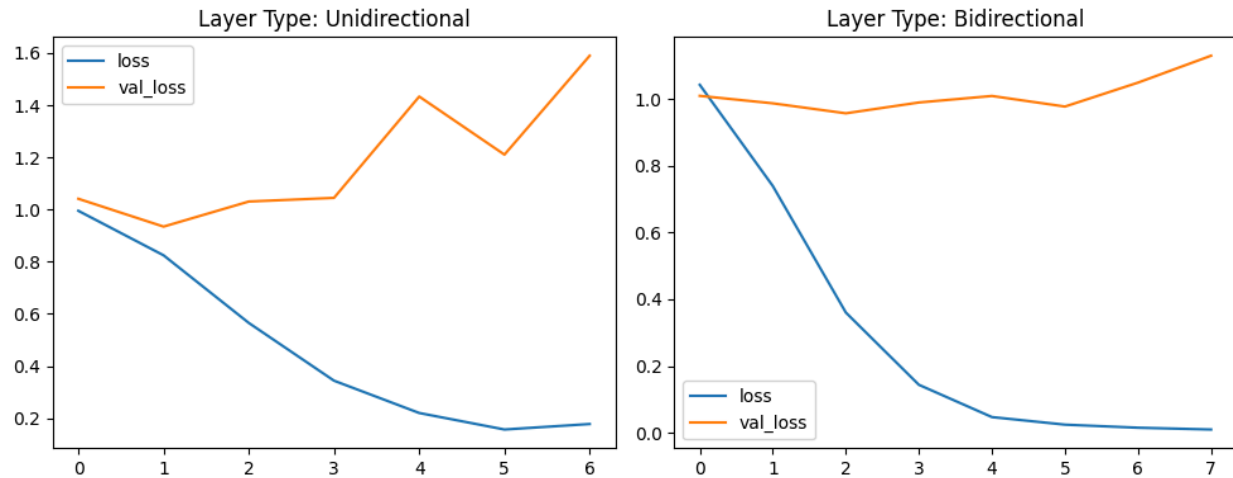
Disimpulkan bahwa jumlah unit yang terlalu banyak rentan terhadap *overfitting*.

Jenis Layer

Eksperimen dilakukan dengan variasi layer *bidirectional* dan *unidirectional*. Berikut adalah performa prediksinya.

Jenis Layer	Akurasi	F1-Score
Bidirectional	0.7550	0.7479
Unidirectional	0.5950	0.4574

Kemudian berikut adalah grafik *loss* dan *validation loss* terhadap jumlah epoch untuk masing-masing jenis.



Pada gambar di atas, terlihat bahwa layer bidirectional mempunyai *loss* dan *validation loss* yang lebih stabil secara signifikan. Hal ini diduga karena datanya bersifat kalimat yang kontekstual dan kata-kata yang saling memengaruhi dapat mempunyai urutan yang tidak tentu (bisa kiri ke kanan maupun kanan ke kiri).

Oleh karena itu, disimpulkan *bidirectional layer* memang lebih bagus untuk kasus ini.

Perbandingan dengan Keras

Model *from scratch* berhasil menyamakan output dengan *forward propagation* menggunakan keras. Perbandingan lebih detail dapat dilihat pada sel terakhir pada *file* pengujian [src/playgrounds/gana_rnn.ipynb](#).

2.2.3 LSTM

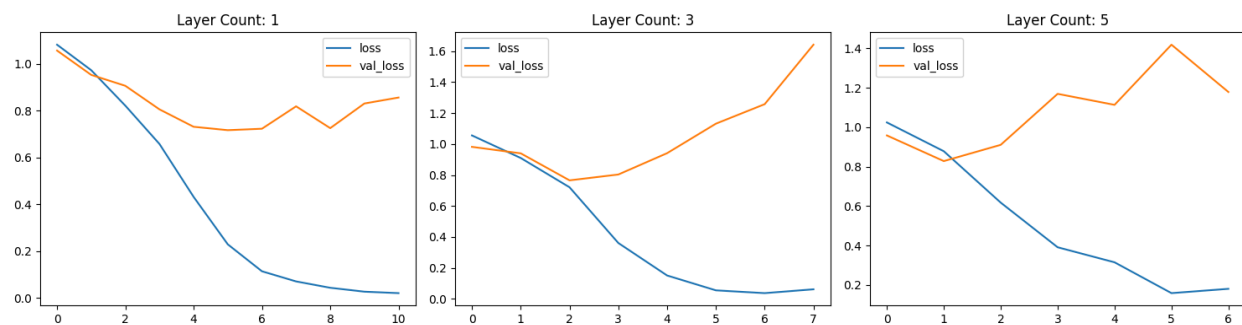
Berikut adalah hasil eksperimen dengan beberapa variasi parameter. **Note:** karena LSTM pada dasarnya mirip dengan RNN, beberapa penjelasan pada subbagian ini memang sama persis dengan [subbagian 2.2.2](#).

Jumlah Layer

Eksperimen dilakukan dengan variasi jumlah layer 1, 3, dan 5 layer. Berikut adalah performa prediksinya.

Jumlah Layer	Akurasi	F1-Score
1	0.7675	0.7593
3	0.6250	0.4951
5	0.6225	0.4734

Kemudian berikut adalah grafik *loss* dan *validation loss* terhadap jumlah epoch untuk masing-masing jumlah layer.



Pada gambar di atas, terlihat bahwa jumlah layer 1 mempunyai grafik *validation loss* yang paling stabil, dan juga mempunyai F1-Score terbesar. Di samping itu, jumlah layer 3 dan 5 mempunyai *spike* mendekati akhir pada grafik *loss* dan *validation loss*-nya. Namun begitu, seluruhnya mempunyai *loss* yang kecil.

Disimpulkan bahwa jumlah layer yang terlalu besar rentan terhadap *overfitting*.

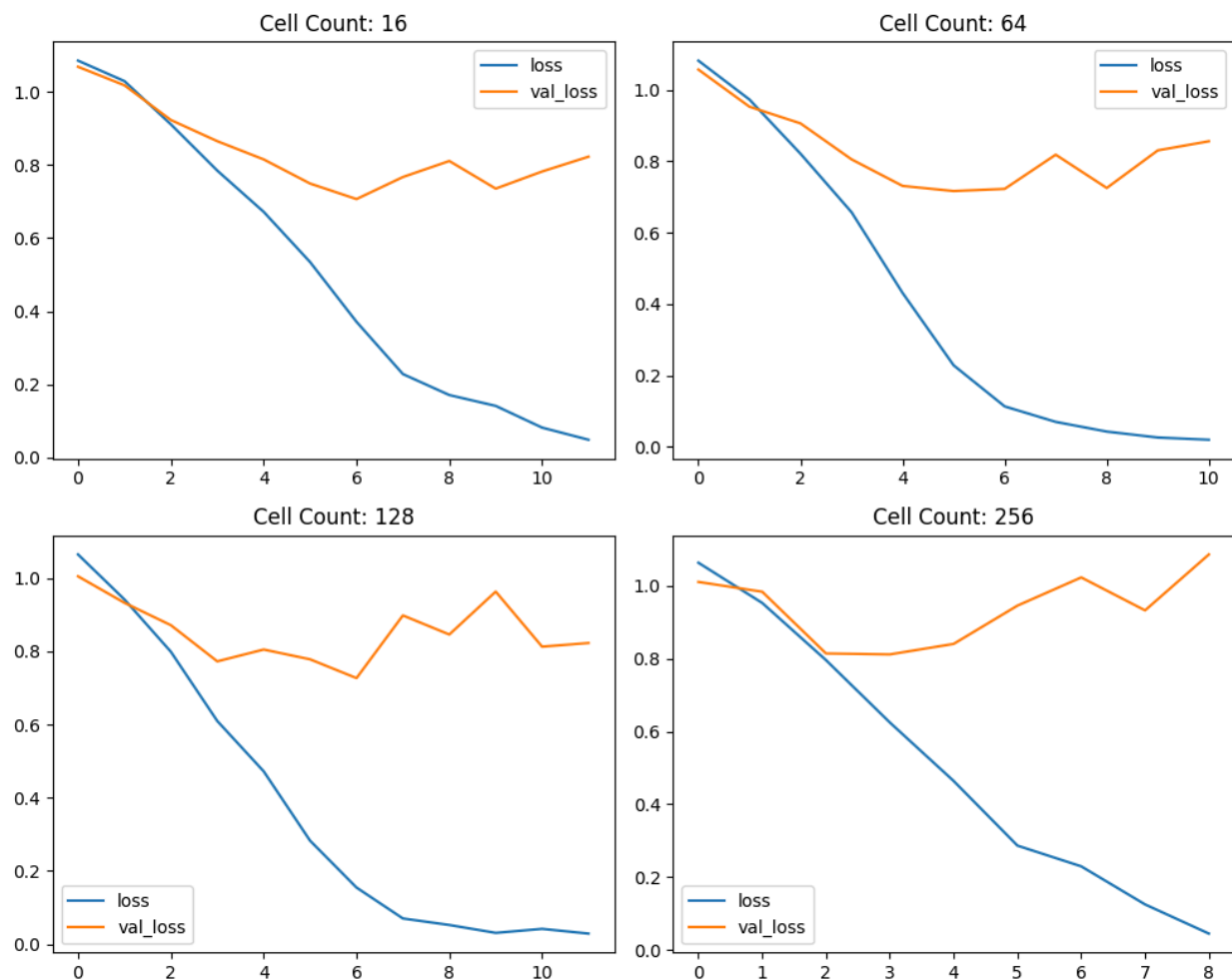
Jumlah Unit/Sel

Eksperimen dilakukan dengan variasi jumlah unit 16, 64, 128, dan 256 unit. Berikut adalah performa prediksinya.

Jumlah Unit	Akurasi	F1-Score
16	0.7550	0.7479
64	0.7675	0.7593

128	0.7375	0.7214
256	0.5925	0.4984

Kemudian berikut adalah grafik *loss* dan *validation loss* terhadap jumlah epoch untuk masing-masing jumlah unit/sel.



Pada gambar di atas, terlihat bahwa layer bidirectional mempunyai *loss* dan *validation loss* yang lebih stabil secara signifikan. Hal ini diduga karena datanya bersifat kalimat yang kontekstual dan kata-kata yang saling memengaruhi dapat mempunyai urutan yang tidak tentu (bisa kiri ke kanan maupun kanan ke kiri).

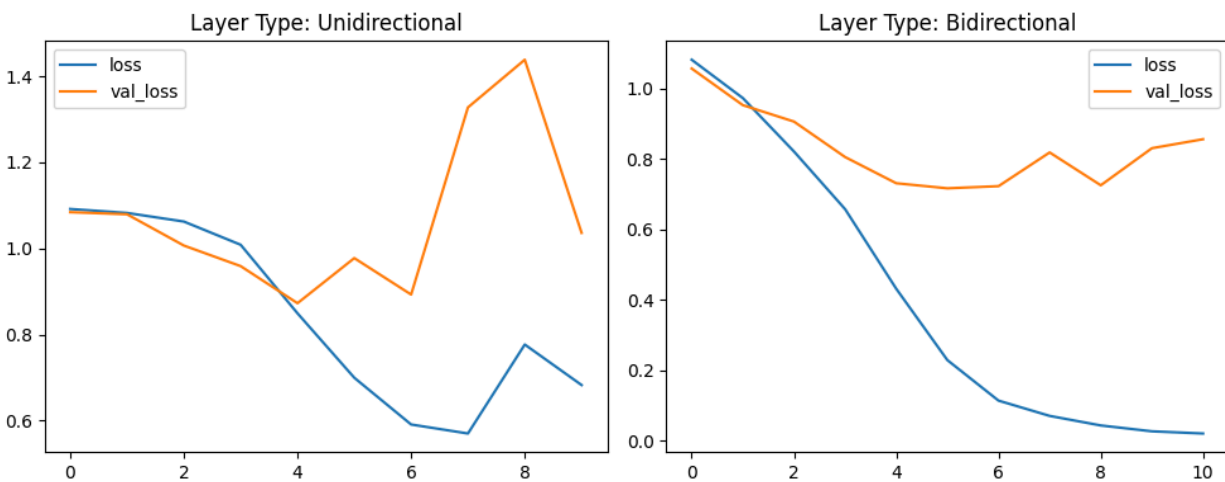
Oleh karena itu, disimpulkan *bidirectional layer* memang lebih bagus untuk kasus ini.

Jenis Layer

Eksperimen dilakukan dengan variasi layer *bidirectional* dan *unidirectional*. Berikut adalah performa prediksinya.

Jenis Layer	Akurasi	F1-Score
Bidirectional	0.7550	0.7479
Unidirectional	0.5950	0.4574

Kemudian berikut adalah grafik *loss* dan *validation loss* terhadap jumlah epoch untuk masing-masing jenis.



Pada gambar di atas, terlihat bahwa layer *bidirectional* mempunyai *loss* dan *validation loss* yang lebih stabil secara signifikan.

Disimpulkan bahwa karena datanya bersifat kalimat yang kontekstual dan kata-kata yang saling memengaruhi dapat mempunyai urutan yang tidak tentu (bisa kiri ke kanan maupun kanan ke kiri), *bidirectional layer* memang lebih bagus untuk kasus ini.

Perbandingan dengan Keras

Model *from scratch* berhasil menyamakan output dengan *forward propagation* menggunakan keras. Perbandingan lebih detail dapat dilihat pada sel terakhir pada *file* pengujian `src/playgrounds/lstm.ipynb`.

3. Kesimpulan dan Saran

4. Pembagian Tugas Tiap Anggota Kelompok

Berikut adalah tabel pembagian tugas tiap anggota kelompok untuk tugas ini.

Nama	NIM	Pembagian tugas
Renaldy Arief Susanto	13522022	<ul style="list-style-type: none">- Implementasi model Keras untuk LSTM- Pengujian untuk LSTM- Laporan bagian RNN, LSTM, dan implementasi
Kristo Anugrah	13522066	<ul style="list-style-type: none">- Implementasi <i>forward propagation</i> untuk CNN- Pengujian untuk CNN- Laporan bagian CNN
Nyoman Ganadipa Narayana	13522114	<ul style="list-style-type: none">- Implementasi forward propagation untuk RNN dan LSTM- Pengujian untuk RNN

5. Referensi

https://d2l.ai/chapter_convolutional-neural-networks/why-conv.html

<https://www.shiksha.com/online-courses/articles/introduction-to-recurrent-neural-network/>

<https://stackoverflow.com/questions/48714407/rnn-regularization-which-component-to-regularize/58868383#58868383>