LAPORAN TUGAS BESAR
IF3270 - PEMBELAJARAN MESIN
Convolution Neural Network
Recurrent Neural Network
Long Short Term Memory

Disusun Oleh

Fahrian Afdholi 13521031

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024/2025

BAB I
PENDAHULUAN

1.1 Latar Belakang

Convolution Neural Network (CNN) adalah salah satu algoritma pada *neural network* yang biasa digunakan untuk melakukan pembelajaran pada *image* ataupun *computer vision*.

Recurrent Neural Network (RNN) adalah salah satu algoritma pada neural network yang digunakan untuk melakukan pembelajaran pada data yang *sequential* dan memiliki output yang spesifik.

Long Short Term Memory (LSTM) adalah algoritma versi terbaik dari RNN karena pada algoritma ini dapat menyelesaikan masalah *vanishing gradient* pada RNN dengan menambahkan beberapa *gate* saat pembelajaran seperti *output gate*, *input gate, forget gate*, dan *cell state*.

1.2 Tujuan

1. Memahami konsep dasar dan komponen dari CNN, RNN, dan LSTM
2. Melakukan eksperimen dengan berbagai hyperparameter
3. Melakukan perbandingan kinerja dengan library pada python

BAB II
PEMBAHASAN

2.1 Penjelasan Implementasi

Bagian ini merupakan penjelasan implementasi dari kelas CNN, RNN, dan LSTM. Pada tiap *method* terdapat beberapa deskripsi seperti parameter dan kegunaan dari *method* tersebut. Berikut adalah kelas dan masing masing *method* yang diimplementasi pada masing masing kelas:

2.1.1 CNN

Tabel 2.1 Kelas CNN

```
import os
import pickle
from typing import Dict, Tuple, Optional

import numpy as np
from numpy.lib.stride_tricks import as_strided


class CNNFromScratch:
    def __init__(self, model_path: str):
        """
        Initialize CNNFromScratch with model path.
        :param model_path: Path to the model pkl file
        """

        self.model_path = model_path
        self.weights = self._load_model()
        self.layers = []

    def _load_model(self) -> Dict[str, np.ndarray]:
        """
        Load model weights from a pickle file.
        :return: Dictionary of weights
        """

        if not os.path.exists(self.model_path):
            raise FileNotFoundError(f"Model file {self.model_path} not found")

        try:
            with open(self.model_path, 'rb') as f:
                return pickle.load(f)
        except Exception as e:
```

```
            raise Exception(f"Failed to load model: {str(e)}")

    def add_layer(self, layer_type: str, **kwargs) -> 'CNNFromScratch':
        """
        This method is used to add a layer to the CNN model.
        Supported layer types: dense, conv2d, maxpooling2d, flatten.
        :param layer_type: str
        :param kwargs: Additional parameters for the layer
        :return: self
        """

        layer_type = layer_type.lower()
        layer_map = {
            'dense': {
                'type': 'dense',
                'name': kwargs.get('name', f'dense_{len(self.layers)}'),
                'activation': kwargs.get('activation', 'linear'),
                'units': kwargs.get('units')
            },
            'conv2d': {
                'type': 'conv2d',
                'name': kwargs.get('name', f'conv2d_{len(self.layers)}'),
                'filters': kwargs.get('filters'),
                'kernel_size': kwargs.get('kernel_size', (3, 3)),
                'strides': kwargs.get('strides', (1, 1)),
                'padding': kwargs.get('padding', 'valid'),
                'activation': kwargs.get('activation', 'linear')
            },
            'maxpooling2d': {
                'type': 'maxpooling2d',
                'name': kwargs.get('name', f'maxpool_{len(self.layers)}'),
                'pool_size': kwargs.get('pool_size', (2, 2)),
                'strides': kwargs.get('strides')
            },
            'flatten': {
                'type': 'flatten',
                'name': kwargs.get('name', f'flatten_{len(self.layers)}')
            }
        }

        if layer_type not in layer_map:
            raise ValueError(f"Unsupported layer type: {layer_type}")

        self.layers.append(layer_map[layer_type])
        return self

    def _activation(self, x: np.ndarray, activation: str) -> np.ndarray:
        """
        Activation function to apply non-linearity to the data.
        Supported activations: relu, sigmoid, tanh, softmax, linear.
        :param x: Input data
        :param activation: Activation function name
        :return: Activated data
        """

        activations = {
```

```
            'relu': lambda x: np.maximum(0, x),
            'sigmoid': lambda x: 1 / (1 + np.exp(-x)),
            'tanh': np.tanh,
            'softmax': lambda x: np.exp(x - np.max(x, axis=-1, keepdims=True))
/
                                    np.sum(np.exp(x - np.max(x, axis=-1,
keepdims=True)),
                                           axis=-1, keepdims=True),
            'linear': lambda x: x
        }

        return activations.get(activation.lower(), lambda x: x)(x)

    def _conv2d(self, input_data: np.ndarray, weights: np.ndarray,
                bias: np.ndarray, kernel_size: Tuple[int, int],
                strides: Tuple[int, int], padding: str) -> np.ndarray:
        """
        Conv 2D is a method to perform a 2D convolution operation.
        this method operation using strided view for optimized performance.
        :param input_data:
        :param weights:
        :param bias:
        :param kernel_size:
        :param strides:
        :param padding:
        :return:
        """

        batch_size, in_height, in_width, in_channels = input_data.shape
        kernel_h, kernel_w = kernel_size
        stride_h, stride_w = strides

        # Calculate output dimensions based on padding
        if padding.lower() == 'same':
            out_height = (in_height + stride_h - 1) // stride_h
            out_width = (in_width + stride_w - 1) // stride_w
            pad_h = max((out_height - 1) * stride_h + kernel_h - in_height, 0)
            pad_w = max((out_width - 1) * stride_w + kernel_w - in_width, 0)
            pad_top = pad_h // 2
            pad_bottom = pad_h - pad_top
            pad_left = pad_w // 2
            pad_right = pad_w - pad_left
            padded_input = np.pad(input_data,
                                  ((0, 0), (pad_top, pad_bottom),
                                   (pad_left, pad_right), (0, 0)),
                                  mode='constant')
        else:  # 'valid' padding
            padded_input = input_data
            out_height = (in_height - kernel_h) // stride_h + 1
            out_width = (in_width - kernel_w) // stride_w + 1

        # Create strided view of the padded input
        shape = (batch_size, out_height, out_width, kernel_h, kernel_w,
in_channels)
        strides = (padded_input.strides[0],
                   padded_input.strides[1] * stride_h,
```

```
                padded_input.strides[2] * stride_w,
                padded_input.strides[1],
                padded_input.strides[2],
                padded_input.strides[3])

        # Create a strided view of the padded input
        strided_input = as_strided(padded_input, shape=shape, strides=strides,
writeable=False)

        """
        Perform convolution using einsum for optimized performance.
        The einsum operation computes the dot product between the strided
input and the weights.
        The output shape will be (batch_size, out_height, out_width, filters).
        """

        output = np.einsum('bhwijc,ijcf->bhwf', strided_input, weights)
        output += bias  # Add bias

        return output

    def _maxpool2d(self, input_data: np.ndarray, pool_size: Tuple[int, int],
                   strides: Optional[Tuple[int, int]] = None) -> np.ndarray:
        """
        Max pooling operation to downsample the input data.
        this method operation using strided view for optimized performance.
        :param input_data:
        :param pool_size:
        :param strides:
        :return:
        """

        batch_size, in_height, in_width, channels = input_data.shape
        pool_h, pool_w = pool_size
        stride_h, stride_w = strides if strides is not None else pool_size

        out_height = (in_height - pool_h) // stride_h + 1
        out_width = (in_width - pool_w) // stride_w + 1

        # Create strided view
        shape = (batch_size, out_height, out_width, pool_h, pool_w, channels)
        strides = (input_data.strides[0],
                   input_data.strides[1] * stride_h,
                   input_data.strides[2] * stride_w,
                   input_data.strides[1],
                   input_data.strides[2],
                   input_data.strides[3])

        strided_input = as_strided(input_data, shape=shape, strides=strides,
writeable=False)

        # Max pooling along the spatial dimensions
        return np.max(strided_input, axis=(3, 4))

    def _dense(self, input_data: np.ndarray, weights: np.ndarray, bias:
np.ndarray) -> np.ndarray:
```

```
        """
        Dense layer operation to perform a fully connected layer.
        this operation using matrix multiplication numpy.
        :param input_data:
        :param weights:
        :param bias:
        :return:
        """

        return np.dot(input_data, weights) + bias

    def _flatten(self, input_data: np.ndarray) -> np.ndarray:
        """
        Flatten operation to convert input data to 2D.
        :param input_data:
        :return:
        """

        return input_data.reshape(input_data.shape[0], -1)

    def _forward(self, input_data: np.ndarray) -> np.ndarray:
        """
        Forward Propagation method to process the input data through the
network.
        :param input_data:
        :return:
        """

        x = input_data

        for layer in self.layers:
            layer_type = layer['type']
            layer_name = layer['name']

            weights = self.weights.get(f"{layer_name}/kernel")
            bias = self.weights.get(f"{layer_name}/bias")

            if layer_type == 'dense':
                if weights is None or bias is None:
                    raise ValueError(f"Missing weights/bias for layer
{layer_name}")
                x = self._dense(x, weights, bias)
                x = self._activation(x, layer['activation'])

            elif layer_type == 'conv2d':
                if weights is None or bias is None:
                    raise ValueError(f"Missing weights/bias for layer
{layer_name}")
                x = self._conv2d(x, weights, bias,
                                 layer['kernel_size'],
                                 layer['strides'],
                                 layer['padding'])
                x = self._activation(x, layer['activation'])

            elif layer_type == 'maxpooling2d':
                x = self._maxpool2d(x, layer['pool_size'], layer['strides'])
```

```
            elif layer_type == 'flatten':
                x = self._flatten(x)

        return x

    def predict(self, input_data: np.ndarray) -> np.ndarray:
        """
        Predict method to handle both single sample and batch inputs.
        :param input_data:
        :return:
        """

        if len(input_data.shape) == 3:
            input_data = np.expand_dims(input_data, axis=0)
        elif len(input_data.shape) != 4:
            raise ValueError("Input must be 3D (single sample) or 4D (batch)
array")

        return self._forward(input_data)
```

2.1.2 RNN dan LSTM

Tabel 2.2 Kelas RNN dan LSTM

```
import pickle
import numpy as np


class RNNLSTMFromScratch:
    def __init__(self, model_path):
        """
        Initialize RNNForwardProp with model path.

        :param model_path: Path to the pickled Keras model
        """

        self.model_path = model_path
        self.weights = {}
        self.model_config = {}
        self.load_model()

    def load_model(self):
        """Load weights and biases from Keras model pickle file"""

        try:
            with open(self.model_path, 'rb') as f:
                model = pickle.load(f)

            # Get model architecture info
            self.model_config['layers'] = []
```

```
            for layer in model.layers:
                layer_info = {
                    'name': layer.name,
                    'type': type(layer).__name__,
                    'config': layer.get_config()
                }
                self.model_config['layers'].append(layer_info)

            # Extract weights from the model
            weights = model.get_weights()
            self.parse_weights(weights, model)

        except Exception as e:
            print(f"Error loading model: {e}")
            raise

    def parse_weights(self, weights, model):
        """
        Parse weights from the model and assign them to the weights
dictionary.

        :param weights: List of weight arrays from the model
        :param model: Keras model instance
        :return: None
        """

        weight_idx = 0

        for i, layer in enumerate(model.layers):
            layer_type = type(layer).__name__
            layer_name = layer.name

            if layer_type == 'Embedding':
                # Embedding layer has one weight matrix
                if weight_idx < len(weights):
                    self.weights[f'{layer_name}_embeddings'] =
weights[weight_idx]
                    weight_idx += 1

            elif layer_type == 'SimpleRNN':
                # SimpleRNN has 3 weight matrices in Keras
                if weight_idx + 2 < len(weights):
                    self.weights[f'{layer_name}_W_input'] =
weights[weight_idx]
                    self.weights[f'{layer_name}_W_recurrent'] =
weights[weight_idx + 1]
                    self.weights[f'{layer_name}_bias'] = weights[weight_idx +
2]
                    weight_idx += 3

            elif layer_type == 'LSTM':
                # LSTM has 3 weight matrices
                if weight_idx + 2 < len(weights):
                    self.weights[f'{layer_name}_W_input'] =
weights[weight_idx]
                    self.weights[f'{layer_name}_W_recurrent'] =
```

```
weights[weight_idx + 1]
                    self.weights[f'{layer_name}_bias'] = weights[weight_idx +
2]
                    weight_idx += 3

            elif layer_type == 'Dense':
                # Dense layer has weight matrix and bias
                if weight_idx + 1 < len(weights):
                    self.weights[f'{layer_name}_W'] = weights[weight_idx]
                    self.weights[f'{layer_name}_bias'] = weights[weight_idx +
1]
                    weight_idx += 2

    def _activation(self, x: np.array, activation: str):
        """
        Apply activation function to the input array.
        This method supports ReLU, Sigmoid, Tanh, Softmax, and Linear
activations.

        :param x:
        :param activation:
        :return:
        """

        # Clip values to prevent overflow
        x = np.clip(x, -500, 500)

        if activation.lower() == 'relu':
            return np.maximum(0, x)
        elif activation.lower() == 'sigmoid':
            return 1 / (1 + np.exp(-x))
        elif activation.lower() == 'tanh':
            return np.tanh(x)
        elif activation.lower() == 'softmax':
            return self._softmax(x)
        elif activation.lower() == 'linear':
            return x
        else:
            return x

    def _softmax(self, x):
        """
        Softmax is a function that converts logits to probabilities.
        It is numerically stable by subtracting the max value.
        This implementation supports both 1D and 2D inputs.

        :param x:
        :return:
        """

        # Handle both 1D and 2D inputs
        if x.ndim == 1:
            x_max = np.max(x)
            exp_x = np.exp(x - x_max)
            return exp_x / np.sum(exp_x)
        else:
```

```
            x_max = np.max(x, axis=-1, keepdims=True)
            exp_x = np.exp(x - x_max)
            return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

    def _get_dropout_rate(self, layer_name):
        """
        Dropout rate is a hyperparameter that controls the fraction of input
units to drop.
        This method retrieves the dropout rate for a given layer.

        :param layer_name:
        :return:
        """

        for layer_info in self.model_config['layers']:
            if layer_info['name'] == layer_name and layer_info['type'] ==
'Dropout':
                return layer_info['config'].get('rate', 0.0)
        return 0.0

    def _dropout(self, X, layer_name, training=False):
        """
        Dropout is a regularization technique to prevent overfitting.
        It randomly sets a fraction of input units to zero during training.

        :param X:
        :param layer_name:
        :param training:
        :return:
        """

        rate = self._get_dropout_rate(layer_name)

        if training:
            # During training, randomly set elements to zero and scale
            mask = np.random.binomial(1, 1 - rate, size=X.shape) / (1 - rate)
            return X * mask
        else:
            # During inference, no dropout is applied (Keras automatically
handles scaling)
            return X

    def _create_mask(self, X, mask_value=0):
        """
        Mask is used to ignore certain values in the input data.
        This method creates a mask for the input data where the mask_value is
considered as padding.
        Masking is useful for handling variable-length sequences in RNNs.

        :param X:
        :param mask_value:
        :return:
        """

        return X != mask_value
```

```python
    def embedding(self, X, layer_name):
        """
        Embedding layer converts integer indices to dense vectors.
        This method retrieves the embedding weights and applies them to the
input indices.

        :param X:
        :param layer_name:
        :return:
        """

        embeddings = self.weights[f'{layer_name}_embeddings']

        # Convert to numpy if TensorFlow tensor
        if hasattr(X, 'numpy'):
            X = X.numpy()

        # Clip indices to valid range
        X = np.clip(X.astype(int), 0, embeddings.shape[0] - 1)

        # Apply embedding lookup
        embedded = embeddings[X]

        # Create mask for zero-padded positions
        mask = self._create_mask(X, mask_value=0)

        return embedded, mask

    def simple_rnn(self, X, layer_name, mask=None):
        """
        Simple RNN is a basic recurrent layer that processes sequences.
        This method performs forward propagation through a SimpleRNN layer.

        :param X:
        :param layer_name:
        :param mask:
        :return:
        """

        W_input = self.weights[f'{layer_name}_W_input']
        W_recurrent = self.weights[f'{layer_name}_W_recurrent']
        bias = self.weights[f'{layer_name}_bias']

        batch_size, seq_len, input_dim = X.shape
        hidden_dim = W_recurrent.shape[0]

        # Initialize hidden state
        h = np.zeros((batch_size, hidden_dim))

        # Process each timestep
        for t in range(seq_len):
            # Get current input (batch_size, input_dim)
            x_t = X[:, t, :]

            # Compute new hidden state
            # h_t = tanh(W_input * x_t + W_recurrent * h + bias)
```

```
            h_new = np.tanh(
                np.dot(x_t, W_input) +
                np.dot(h, W_recurrent) +
                bias
            )

            # Apply mask if provided
            if mask is not None:
                mask_t = mask[:, t].reshape(-1, 1)
                h = h_new * mask_t + h * (1 - mask_t)
            else:
                h = h_new

        return h

    def lstm(self, X, layer_name, mask=None):
        """
        LSTM (Long Short-Term Memory) is a type of RNN that can learn
long-term dependencies.
        This method performs forward propagation through an LSTM layer.

        :param X:
        :param layer_name:
        :param mask:
        :return:
        """

        W_input = self.weights[f'{layer_name}_W_input']
        W_recurrent = self.weights[f'{layer_name}_W_recurrent']
        bias = self.weights[f'{layer_name}_bias']

        batch_size, seq_len, input_size = X.shape
        hidden_size = W_recurrent.shape[0]

        # Initialize hidden and cell states
        h = np.zeros((batch_size, hidden_size))
        c = np.zeros((batch_size, hidden_size))

        for t in range(seq_len):
            # Compute all gates at once
            # gates = W_input * x_t + W_recurrent * h + bias
            gates = np.dot(X[:, t, :], W_input) + np.dot(h, W_recurrent) +
bias

            # Split gates (input, forget, candidate, output) - Keras order
            i_gate = self._activation(gates[:, :hidden_size], 'sigmoid')
            f_gate = self._activation(gates[:, hidden_size:2 * hidden_size],
'sigmoid')
            c_candidate = self._activation(gates[:, 2 * hidden_size:3 *
hidden_size], 'tanh')
            o_gate = self._activation(gates[:, 3 * hidden_size:], 'sigmoid')

            # Update cell state
            # c_new = f_gate * c + i_gate * c_candidate
            c_new = f_gate * c + i_gate * c_candidate
```

```
            # Update hidden state
            # h_new = o_gate * tanh(c_new)
            h_new = o_gate * self._activation(c_new, 'tanh')

            # Apply mask
            if mask is not None:
                mask_t = mask[:, t].reshape(-1, 1)
                h = h_new * mask_t + h * (1 - mask_t)
                c = c_new * mask_t + c * (1 - mask_t)
            else:
                h = h_new
                c = c_new

        return h

    def dense(self, X, layer_name, activation='linear'):
        """
        Dense layer performs a linear transformation followed by an activation function.
        This method applies the weights and bias of a Dense layer to the input data.

        :param X:
        :param layer_name:
        :param activation:
        :return:
        """

        W = self.weights[f'{layer_name}_W']
        bias = self.weights[f'{layer_name}_bias']

        # Linear transformation
        output = np.dot(X, W) + bias

        # Apply activation
        output = self._activation(output, activation)
        return output

    def predict(self, X, training=False):
        """
        This method performs forward propagation through the RNN/LSTM model.
        It processes the input data through each layer in sequence, applying the appropriate operations.

        :param X: Input data (integer sequences for embedding)
        :param training: Whether to apply dropout
        :return: Output predictions
        """

        current_output = X
        mask = None

        # Process each layer in sequence
        for layer_info in self.model_config['layers']:
            layer_name = layer_info['name']
            layer_type = layer_info['type']
```

```
            layer_config = layer_info['config']

            if layer_type == 'Embedding':
                # Check if masking is enabled
                mask_zero = layer_config.get('mask_zero', False)
                if mask_zero:
                    current_output, mask = self.embedding(current_output,
layer_name)
                else:
                    current_output = self.embedding(current_output,
layer_name)
                    if isinstance(current_output, tuple):
                        current_output = current_output[0]  # If masking was
returned anyway

            elif layer_type == 'SimpleRNN':
                current_output = self.simple_rnn(current_output, layer_name,
mask)
                # After SimpleRNN, we no longer need the mask for subsequent
layers
                mask = None

            elif layer_type == 'LSTM':
                current_output = self.lstm(current_output, layer_name, mask)
                # After LSTM, we no longer need the mask for subsequent layers
                mask = None

            elif layer_type == 'Dense':
                activation = layer_config.get('activation', 'linear')
                current_output = self.dense(current_output, layer_name,
activation)

            elif layer_type == 'Dropout':
                current_output = self._dropout(current_output, layer_name,
training)

        return current_output

    def predict_classes(self, X):
        """
        Predict class labels for the input data.

        :param X:
        :return:
        """

        predictions = self.predict(X)
        return np.argmax(predictions, axis=1)
```

2.2 Pengujian

Pada bagian ini akan dilakukan pengujian pada model CNN, RNN, dan LSTM Keras dan *Custom Forward Propagation* yang sudah di implementasi. Berikut adalah pengujian yang akan dilakukan:

1. Pengaruh jumlah *convolution layers* pada model keras CNN
2. Pengaruh jumlah *filters* pada model keras CNN
3. Pengaruh jumlah besarnya *filters* pada model CNN
4. Perbandingan CNN *Forward Propagation* dengan CNN model library keras
5. Pengaruh jumlah *layers* pada model RNN dan LSTM keras
6. Pengaruh jumlah *cells* pada model RNN dan LSTM keras
7. Pengaruh tipe dari RNN dan LSTM (*Unidirectional* dan *Bidirectional*)
8. Perbandingan RNN dan LSTM *Forward Propagation Custom* dengan library keras

Pada pengujiannya, kami menggunakan *dataset* cifar10 untuk CNN dan NusaX untuk RNN dan LSTM.

2.2.1 CNN

Pada pengujian kali ini akan dilakukan pengujian jumlah *convolution layers*, jumlah *filters*, besar *filters*, dan perbandingan CNN model keras dan CNN *custom*.

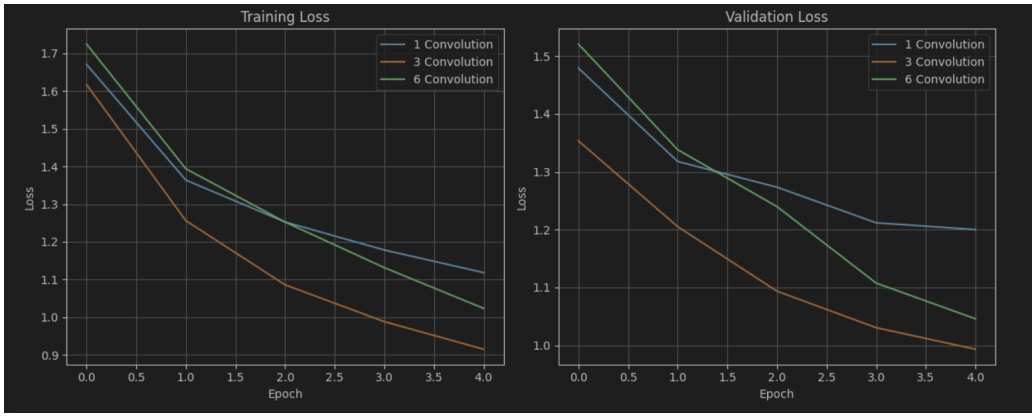
2.2.1.1 Jumlah Convolution Layers

Pada pengujian ini terdapat 3 variasi jumlah *layers* yang akan diujikan yaitu 1, 3, dan 5 jumlah *convolution layers*. Berikut adalah variabel dan jumlah layer yang akan diujikan.

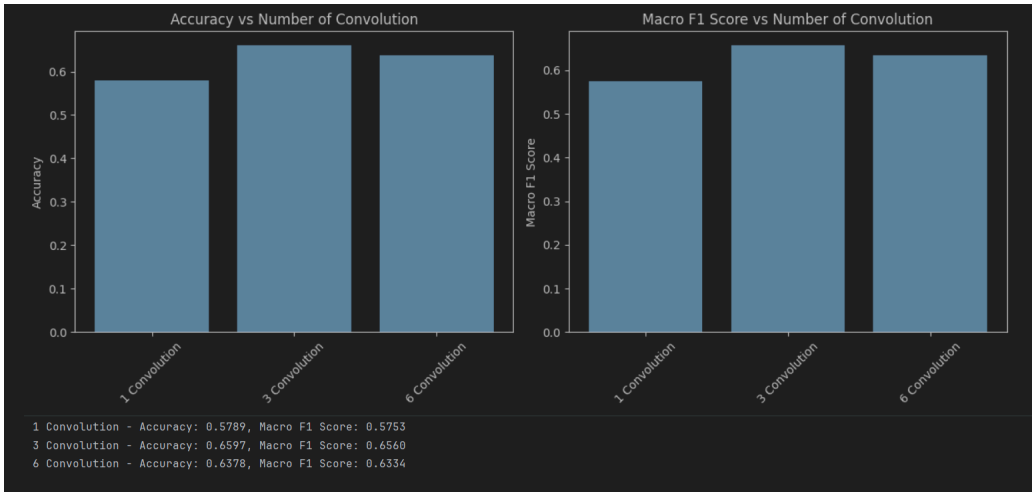
```
batch_size = 128
epochs = 5

conv = {
   "1 Convolution" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
```

```
        layers.Dense(num_classes, activation="softmax")
    ],
    "3 Convolution" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")
    ],
    "6 Convolution" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")

    ]
}
```

Berikut adalah hasil dari pengujian masing masing jumlah layer yang ditampilkan melalui plot.



Gambar 2.1 *training loss* dan *validation loss*



Gambar 2.2 *accuracy* dan *f1-score*

Pada Gambar 2.1 *loss* masing masing *layers* cenderung menurun stabil kecuali pada jumlah 1 *convolution layers* yang memiliki penurunan yang tidak stabil pada *epoch* 1 dan 3 yang menyebabkan hasil dari *loss* pada *validation* dan *training* tidak lebih baik dibandingkan dengan jumlah layer lainnya. Jika diurutkan *training* dan *validation loss* yang paling dari yang terbaik pada pengujian ini adalah 3 *convolution*, 6 *convolution*, dan 1 *convolution*.

Pada Gambar 2.2 akurasi yang diuji menggunakan *accuracy_score* dan *f1_score* juga memiliki urutan dari yang terbaik sama dengan urutan pada *validation* dan *training loss*.

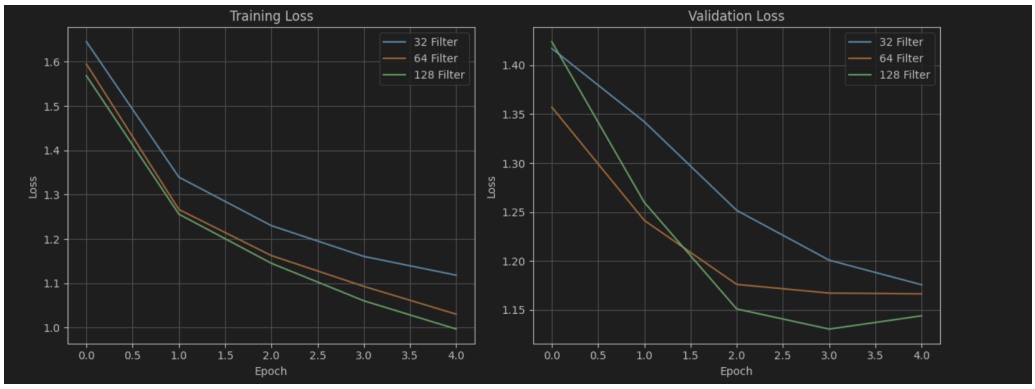
Dari hasil pada Gambar 2.1 dan 2.2 bisa disimpulkan jika 1 *convolution layer* masih terlalu sederhana untuk menangkap pola dari data yang dipelajari sehingga memiliki akurasi yang lebih kecil dan *loss* yang lebih banyak dibandingkan yang lain. Pada jumlah *convolution layer* 3 dan 6 memiliki hasil yang terbaik pada 3 layer yang dimana seharusnya 6 layer lebih baik dalam mempelajari pola yang ada karena memiliki parameter yang lebih banyak sehingga dapat mempelajari pola data lebih baik dibandingkan 3 *layer*, hal ini mungkin terjadi karena *epoch* yang dipakai sangat kecil (5 *epoch*) dan perlu diperbanyak karena jika dilihat pada Gambar 2.1 dan 2.2 tidak terdapat tanda tanda *overfitting* pada 6 *layer*.

2.2.1.2 Jumlah Filters

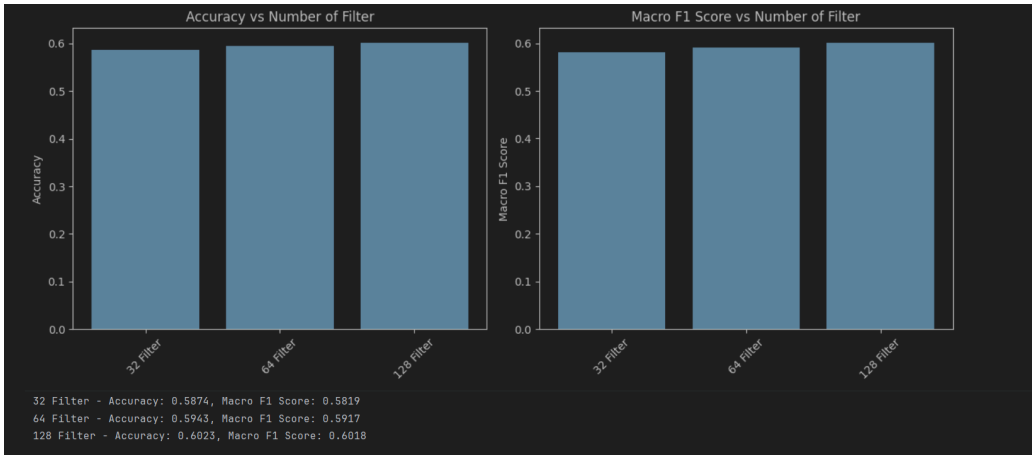
Pada pengujian ini akan dilakukan perbandingan 3 variasi jumlah *filters* yaitu 32 *filter*, 64 *filter*, dan 128 *filter.* Berikut adalah konfigurasi yang dilakukan saat melakukan pengujian ini.

```
batch_size = 128
epochs = 5

num_of_fiter = {
    "32 Filter" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")
    ],
    "64 Filter" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")
    ],
    "128 Filter" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")
    ]
}
```

Berikut adalah hasil pengujian *validation* dan *training loss* serta *accuracy score* dan *f1 score* masing masing jumlah *filters*.



Gambar 2.3 *validation* dan *training loss*



Gambar 2.4 *accuracy* dan *f1-score*

Pada Gambar 2.3 masing masing *validation* dan *training loss* jumlah *filters* turun secara stabil dengan urutan dari yang terbaik adalah 128, 64, dan 32 *filters*.

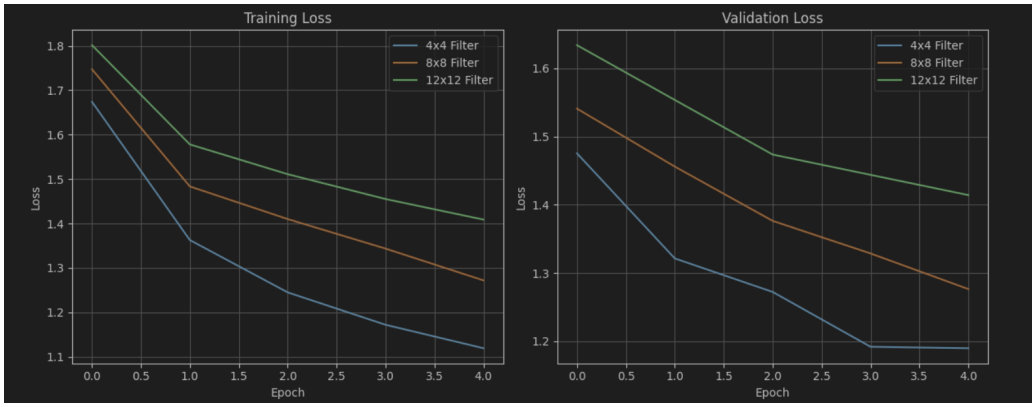
Pada Gambar 2.4 menampilkan akurasi dari masing masing *filters* dengan urutan yang terbaik sama dengan *loss* tetapi hasil akurasi masing masing dari pengujian tidak terlalu meningkat signifikan hal ini mungkin disebabkan karena 32 *filters* mungkin sudah cukup untuk dilakukan dan tidak perlu menambah jumlah *filters* yang besar pada dataset CIFAR10.

2.2.1.3 Besar Filters

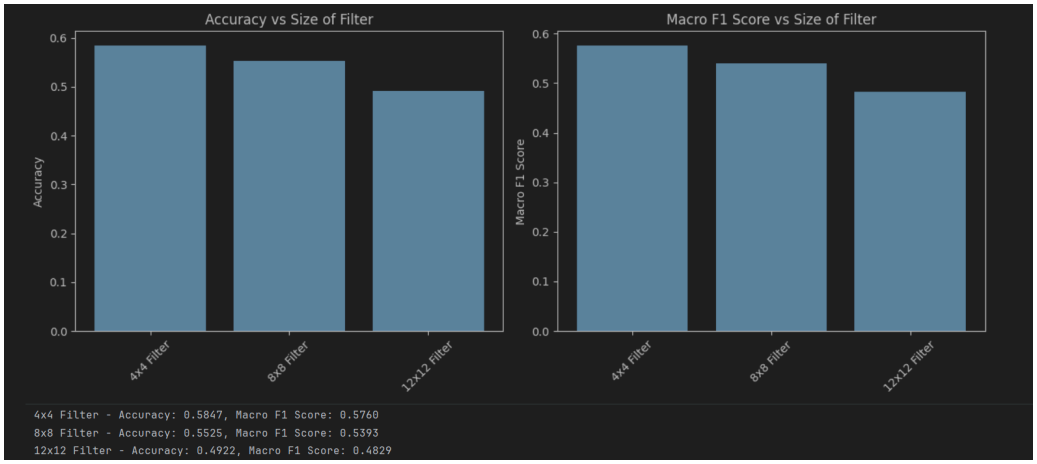
Pada pengujian ini terdapat 3 variasi besar filter yang diuji di antaranya 4x4 *filters*, 8x8 *filters*, dan 12x12 *filters*. Berikut adalah konfigurasi saat melakukan pengujian ini.

```
batch_size = 128
epochs = 5

size_of_fiter = {
    "4x4 Filter" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(4, 4), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")
    ],
    "8x8 Filter" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(8, 8), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")
    ],
    "12x12 Filter" : [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(12, 12),
activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")
    ]
}
```

Berikut adalah hasil pengujian *validation* dan *training loss* beserta *accuracy* dan *f1 score* pada masing masing besar *filters*.



Gambar 2.5 *validation* dan *training loss*



Gambar 2.6 *accuracy* dan *f1 score*

Pada Gambar 2.5 terlihat sangat jelas gap masing masing besar dari *filters*, semakin kecil *filters* semakin baik dalam meng-*handle loss*. Penurunan *loss* pada masing masing *filters* juga stabil dan tidak ada *spike* pada *epoch* tertentu.

Pada Gambar 2.6 menampilkan *accuracy score* dan *f1 score* dan memperlihatkan jika semakin besar *filter* yang diuji akurasi semakin rendah bahkan kita bisa melihat penurunan yang sangat signifikan dari filter 8x8 ke 12x12. Hal ini mungkin terjadi karena kernel 3x3 dapat mempelajari detail yang lebih kecil dibandingkan 8x8 ataupun 12x12 yang menyebabkan akurasi yang dihasilkan lebih baik dibandingkan dengan keduanya.

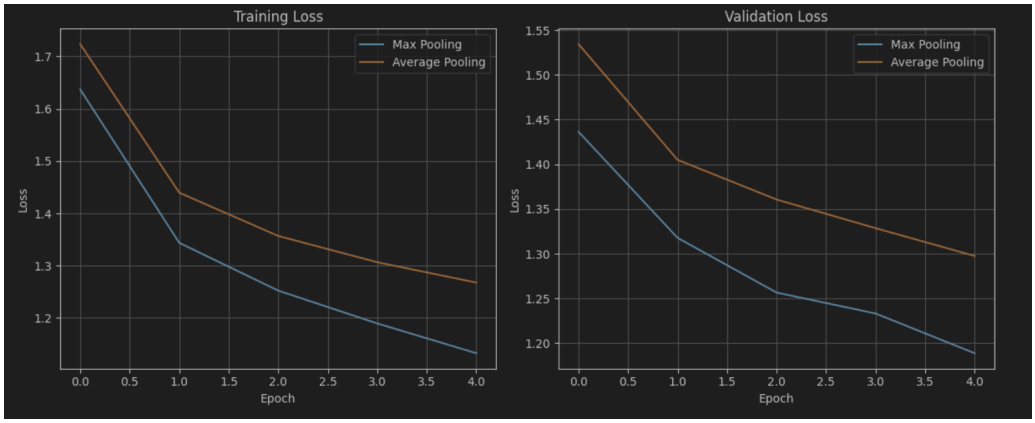
2.2.1.4 Variasi Pooling

Pada pengujian ini akan diujikan variasi dari *pooling* yang ada di CNN berupa *average pooling* dan *max pooling*. Berikut adalah konfigurasi pada pengujian ini.

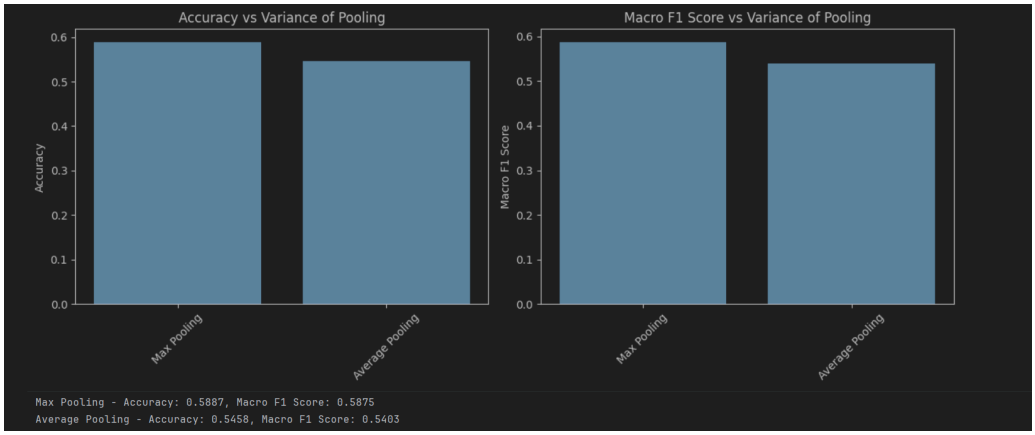
```
batch_size = 128
epochs = 5

variance_of_pooling = {
   "Max Pooling" : [
       keras.Input(shape=input_shape),
       layers.Conv2D(32, kernel_size=(4, 4), activation="relu"),
       layers.MaxPooling2D(pool_size=(2, 2)),
       layers.Flatten(),
       layers.Dense(num_classes, activation="softmax")
   ],
   "Average Pooling" : [
       keras.Input(shape=input_shape),
       layers.Conv2D(32, kernel_size=(4, 4), activation="relu"),
       layers.AveragePooling2D(pool_size=(2, 2)),
```

```
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax")
    ]
}
```

Berikut adalah hasil pengujian *validation* dan *training loss* beserta *accuracy* dan *f1 score* pada masing masing *pooling*.



Gambar 2.7 *validation* dan *training loss*



Gambar 2.8 *accuracy* dan *f1 score*

Pada Gambar 2.7 dapat dilihat jika *loss* pada masing masing variasi *pooling* turun secara stabil. Gap *loss* pada kedua *pooling* tersebut terlihat cukup besar yang dimana *max pooling* unggul dengan *loss* yang paling kecil dibandingkan dengan *average pooling*.

Pada Gambar 2.8 juga menampilkan jika akurasi (baik *f1 score* maupun *accuracy score*) *max pooling* lebih baik dibandingkan dengan *average pooling* hal ini bisa

disebabkan karena *max pooling* hanya mengambil fitur paling penting saja untuk melakukan pembelajaran dibandingkan dengan *average pooling* yang mencampur semua data baik itu yang lemah maupun yang kuat sehingga data yang diambil mungkin terdapat *noise* dan menghasilkan akurasi yang lebih kecil dibandingkan dengan *max pooling*.

2.2.1.5 Custom CNN Forward Propagation

Pada pengujian ini akan dilakukan perbandingan model CNN *custom* yang diimplementasi dengan CNN *library* Keras. Berikut adalah konfigurasi model yang akan disimpan sebagai “cnn.pkl” dan digunakan oleh CNN *custom*.

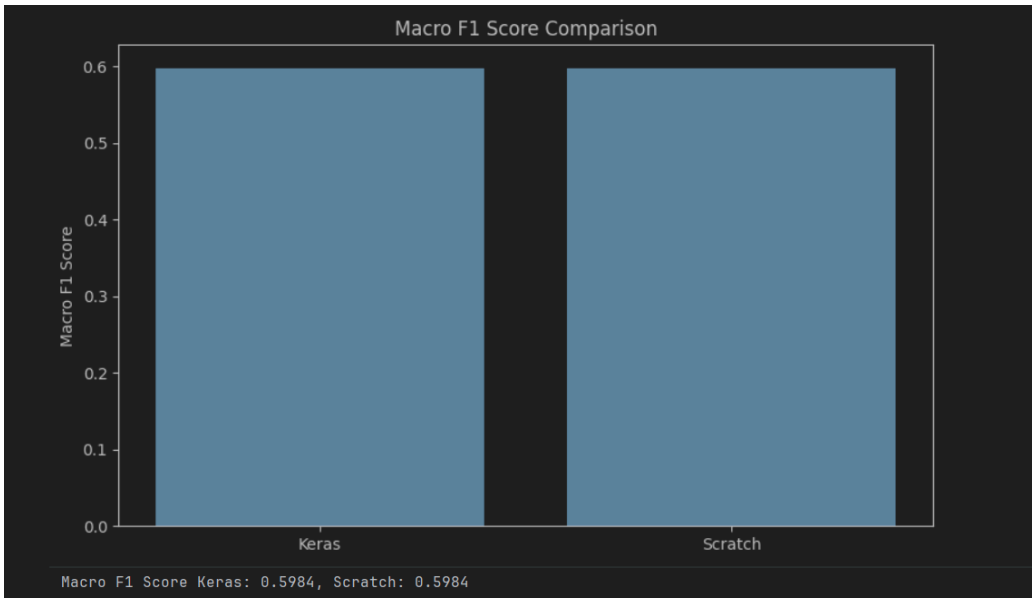
```
batch_size = 128
epochs = 5

model = Sequential([
    layers.Input(shape=input_shape),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu",
name="conv2d"),
    layers.MaxPooling2D(pool_size=(2, 2), name="maxpooling2d"),
    layers.Flatten(name="flatten"),
    layers.Dense(num_classes, activation="softmax", name="dense")
])

model.compile(
    loss="sparse_categorical_crossentropy",
    optimizer="adam",
    metrics=["accuracy"]
)
history = model.fit(x_train, y_train,
                    batch_size=batch_size, epochs=epochs,
                    validation_data=(x_val, y_val)
                    )
```

Model ini akan dilakukan perbandingan hasil prediksi dengan model yang diimplementasi pada Tabel 2.1. Berikut adalah konfigurasi kelas CNN *custom* sebelum melakukan prediksi.

```
# Initialize and configure CNNFromScratch make layers same as
Keras model
CNNCustom = CNNFromScratch("../model/cnn.pkl")

# Make sure the layers are added in the same order as Keras model
CNNCustom.add_layer("conv2d", name="conv2d", filters=32,
kernel_size=(3, 3), activation="relu")
CNNCustom.add_layer("maxpooling2d", name="maxpooling2d",
pool_size=(2, 2))
CNNCustom.add_layer("flatten", name="flatten")
CNNCustom.add_layer("dense", name="dense", units=num_classes,
```

```
activation="softmax")
```

Dengan memastikan *layers* yang ditambahkan sesuai dengan model Keras, hasil prediksi dari model Keras dan model *custom* dengan *f1 score* adalah sebagai berikut.



Gambar 2.9 *f1 score*

Pada Gambar 2.9 menampilkan jika akurasi dari model Keras dan model *custom* menghasilkan akurasi yang sama yaitu 0.5948. Dari hasil tersebut dapat disimpulkan jika model *forward propagation* pada Tabel 2.1 sesuai model Keras.

2.2.2 RNN

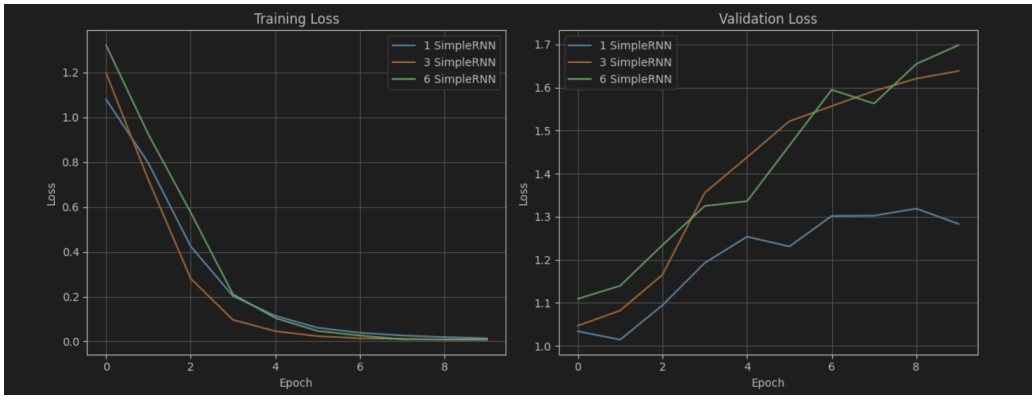
Pada pengujian kali ini akan dilakukan pengujian pada model RNN keras berupa jumlah *layers*, jumlah *cells*, tipe RNN (*Unidirectional* dan *Bidirectional*) dan dilakukan perbandingan antara RNN *forward propagation custom* dengan model RNN keras. Berikut adalah hasil dari pengujiannya.

2.2.2.1 Jumlah Layers RNN

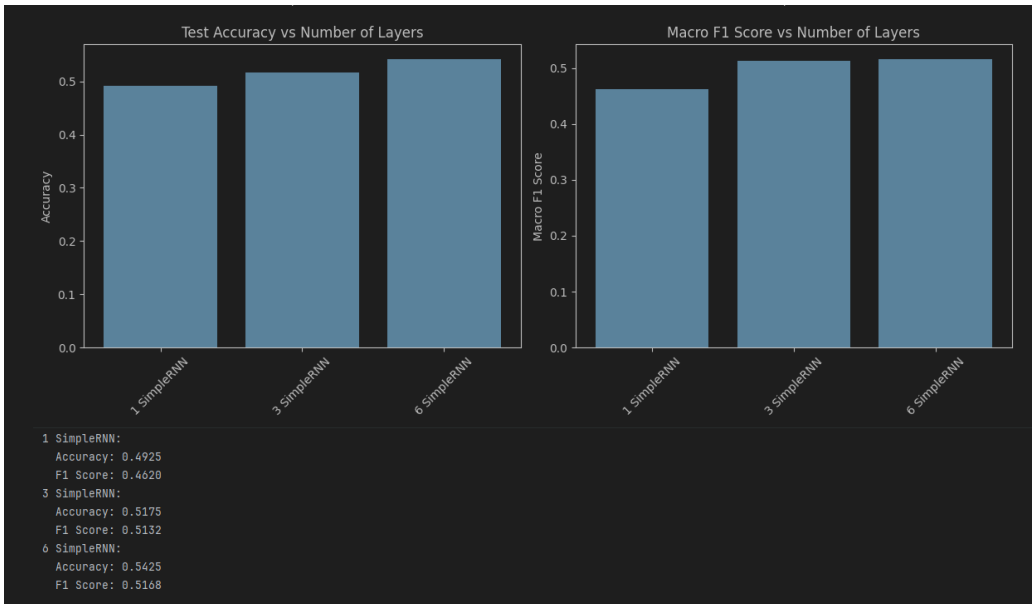
Pada pengujian kali ini akan dilakukan perbandingan *validation* dan *training loss* serta *accuracy* dan *f1 score* pada 3 variasi jumlah *layers* RNN. berikut adalah konfigurasi dari pengujian kali ini.

```
epochs = 10
batch_size = 32

lay = {
   "1 SimpleRNN": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       SimpleRNN(64),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ],
   "3 SimpleRNN": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       SimpleRNN(64, return_sequences=True),
       SimpleRNN(64, return_sequences=True),
       SimpleRNN(64),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ],
   "6 SimpleRNN": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       SimpleRNN(64, return_sequences=True),
       SimpleRNN(64, return_sequences=True),
       SimpleRNN(64, return_sequences=True),
       SimpleRNN(64, return_sequences=True),
       SimpleRNN(64, return_sequences=True),
       SimpleRNN(64),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ]
}
```

Berikut adalah hasil dari pengujian *validation* dan *training loss* serta *accuracy* dan *f1 score* dari masing masing jumlah *layers* yang diuji.



Gambar 2.10 *validation* dan *training loss*



Gambar 2.11 *accuracy* dan *f1 score*

Pada Gambar 2.10 menampilkan *training loss* pada masing masing jumlah *layer* yang menurun secara stabil dan memiliki hasil *loss* yang hampir mirip, sedangkan pada *validation loss* menampilkan hasil yang sangat *random* dan menaik, hal ini mungkin terjadi karena data validasi yang diberikan sangat sedikit sehingga *validation loss* yang dihasilkan tidak menurun stabil.

Pada Gambar 2.11 menampilkan hasil akurasi pada masing masing jumlah *layer* yang jika disimpulkan semakin banyak *layer* yang diuji semakin baik akurasinya.

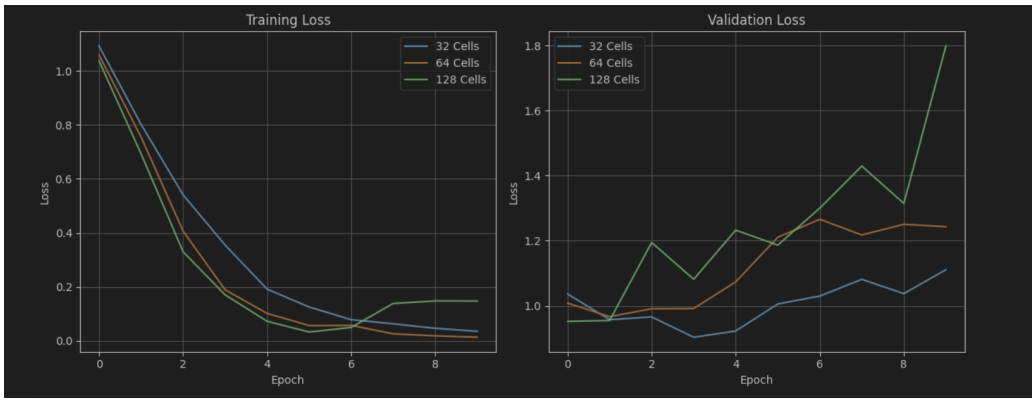
Walaupun begitu terdapat kenaikan yang tidak terlalu signifikan dari 3 *layers* ke 6 *layers* yang bisa menjadi tanda akan adanya *overfitting*. Jadi bisa disimpulkan jika 3 *layers* saja sudah cukup dalam mempelajari pola data yang ada karena dengan menambahkan *layers* lebih dari 3 akan mengalami kenaikan akurasi yang tidak terlalu signifikan.

2.2.2.2 Jumlah Cells

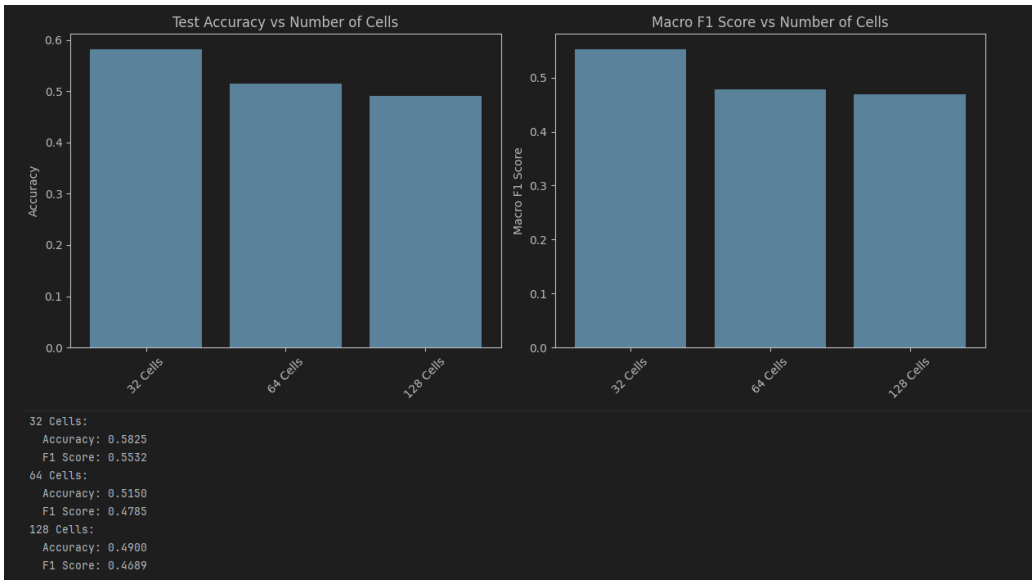
Pada pengujian ini, akan dilakukan pengujian dengan 3 variasi jumlah *cells* diantaranya 32, 64, dan 128 *cells*. Berikut adalah konfigurasi model saat melakukan pengujian.

```
epochs = 10
batch_size = 32

cells = {
   "32 Cells": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       SimpleRNN(32),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ],
   "64 Cells": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       SimpleRNN(64),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ],
   "128 Cells": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       SimpleRNN(128),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ]
}
```

Berikut adalah hasil dari *validation loss* dan *train loss* serta *accuracy* dan *f1 score* selama pengujian variasi jumlah *cells*.



Gambar 2.12 *validation* dan *training loss*



Gambar 2.13 *accuracy* dan *f1 score*

Pada Gambar 2.12 menampilkan *training loss* masing masing *cells* turun secara stabil kecuali pada 128 *cells* yang memiliki kenaikan pada *epoch* 5-8 yang bisa jadi penyebabnya adalah mulai munculnya kondisi *overfitting* pada *epoch* tersebut. Untuk mengatasi hal ini dapat dilakukan pengurangan *epoch* ke 5 *epoch*.

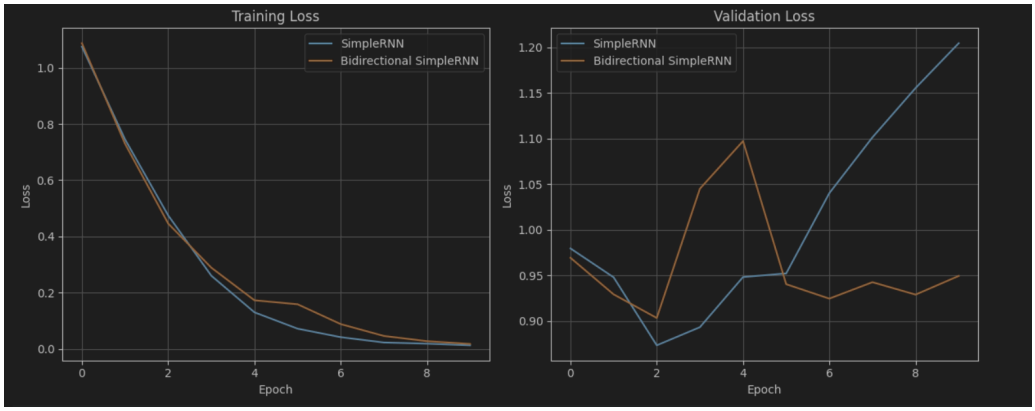
Pada Gambar 2.13 menampilkan hasil akurasi yang semakin menurun seiring banyaknya *cells* yang diuji. Hal ini bisa disebabkan salah satunya karena *vanishing gradient* yang menyebabkan *weight* yang ada pada model akan sedikit berubah.

2.2.2.3 Tipe RNN

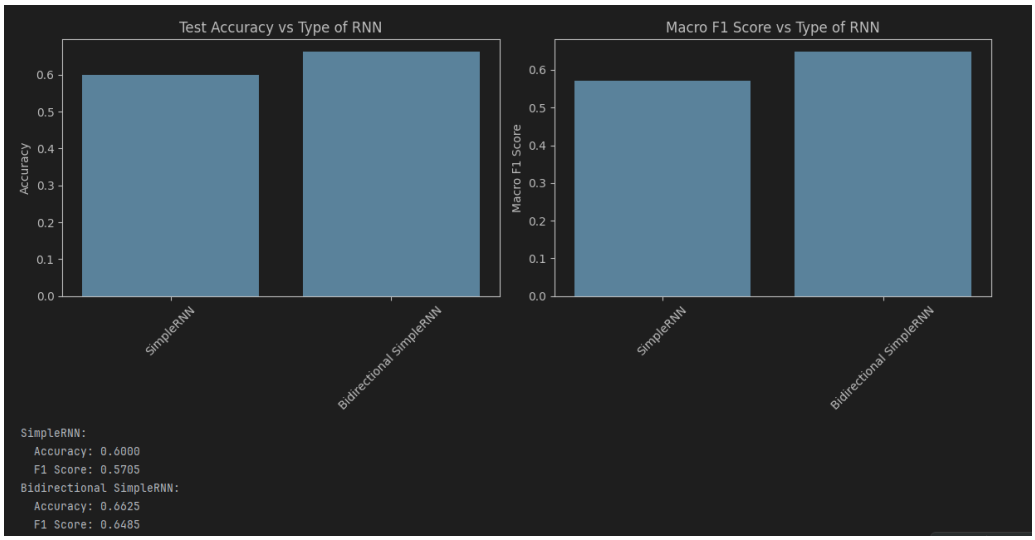
Pada tahap ini, akan dilakukan pengujian pada model *Unidirectional* RNN dan *Bidirectional* RNN. Berikut adalah konfigurasi model pada masing masing tipe RNN tersebut.

```
epochs = 10
batch_size = 32

rnn_types = {
    "SimpleRNN": [
        vectorizer,
        Embedding(input_dim=vocab_size + 1,
                  output_dim=embedding_dim,
                  mask_zero=True,
                  embeddings_initializer='uniform'),
        SimpleRNN(64),
        Dropout(0.5, seed=SEED),
        Dense(len(label_map),
              activation='softmax')
    ],
    "Bidirectional SimpleRNN": [
        vectorizer,
        Embedding(input_dim=vocab_size + 1,
                  output_dim=embedding_dim,
                  mask_zero=True,
                  embeddings_initializer='uniform'),
        Bidirectional(SimpleRNN(64)),
        Dropout(0.5, seed=SEED),
        Dense(len(label_map),
              activation='softmax')
    ]
}
```

Berikut adalah hasil dari *validation* dan *training loss* serta *accuracy* dan *f1 score* masing masing dari tipe RNN yang diuji.



Gambar 2.14 *validation* dan *training loss*



Gambar 2.15 *accuracy* dan *f1 score*

Pada Gambar 2.14 menampilkan hasil dari *training loss* yang menurun secara stabil dan memiliki hasil akhir yang hampir mirip, sedangkan pada *validation loss* memiliki hasil yang random dan hal ini terjadi kemungkinan mirip dengan hasil pengujian *validation loss* pada RNN sebelumnya yaitu dataset yang kurang.

Pada Gambar 2.15 menampilkan hasil dari akurasi kedua variasi yang diuji. Dari hasil akurasi tersebut dapat ditarik kesimpulan jika hasil akurasi *Bidirectional* lebih baik dari *Unidirectional* hal ini terjadi karena dalam melakukan pembelajarannya *Unidirectional* hanya melakukan pembelajaran satu arah (kiri ke kanan) sedangkan *Bidirectional* melakukan pembelajaran secara bolak balik sehingga *Bidirectional* dapat lebih mengetahui konteks dari urutan data yang diuji.

2.2.2.4 Custom RNN Forward Propagation

Pada pengujian ini akan dilakukan perbandingan hasil dari akurasi model RNN *custom forward propagation* yang diimplementasi pada Tabel 2.2 dengan RNN model keras. Berikut adalah konfigurasi dari model keras RNN yang akan dipakai oleh model RNN custom.

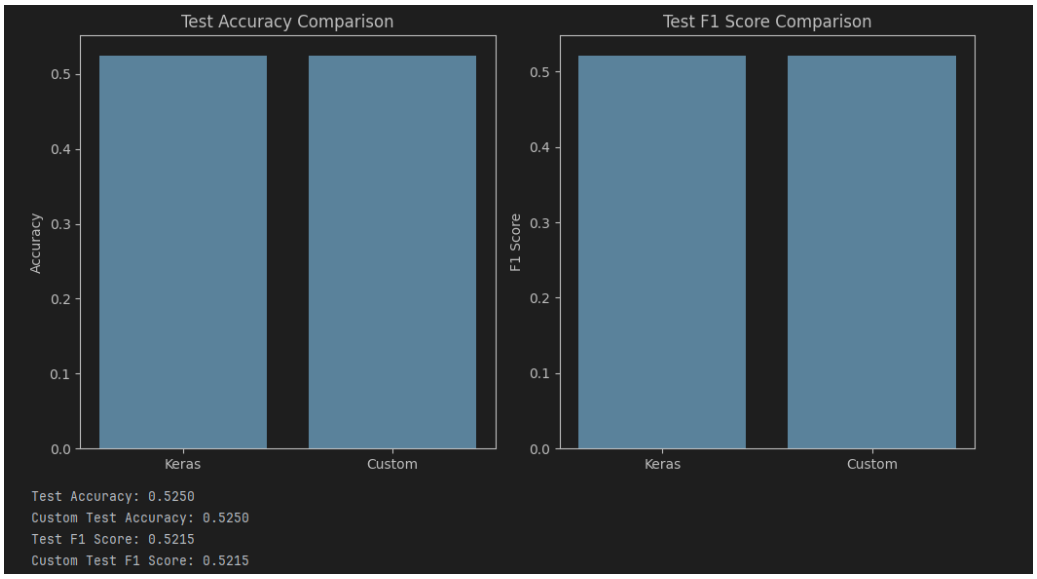
```
epochs = 10
batch_size = 32

accuracy = []
train_loss = {}
```

```
val_loss = {}
f1_scores = []

tf.random.set_seed(SEED)
np.random.seed(SEED)

model = Sequential([
    vectorizer,
    Embedding(input_dim=vocab_size + 1,
              output_dim=embedding_dim,
              mask_zero=True,
              embeddings_initializer='uniform'),
    SimpleRNN(64),
    Dropout(0.5, seed=SEED),
    Dense(len(label_map),
          activation='softmax')
])

model.compile(
    loss="sparse_categorical_crossentropy",
    optimizer='adam',
    metrics=["accuracy"]
)

history = model.fit(
    np.array(train_texts, dtype=object),
    y_train,
    epochs=epochs,
    batch_size=batch_size,
    validation_data=(np.array(val_texts, dtype=object), y_val),
    verbose=1,
    shuffle=True
)
```

Berikut adalah konfigurasi dari model RNN *custom* yang diimplementasi pada Tabel 2.2 dengan melakukan *load weight* dari model RNN keras dengan format .pkl.

```
custom_model = RNNLSTMFromScratch('../model/rnn.pkl')
```

Berikut adalah hasil dari perbandingan *accuracy* dan *f1 score* dari RNN *custom* dengan *library* keras.



Gambar 2.16 *accuracy* dan *f1 score*

Pada Gambar 2.16 menampilkan kedua akurasi baik itu *accuracy score* maupun *f1 score* memiliki hasil yang sama. Hal ini dapat disimpulkan jika implementasi RNN *custom forward propagation* berjalan sesuai dengan model keras dan dapat mengoptimalkan *weight* yang diberikan.

2.2.3 LSTM

Pada tahap ini akan dilakukan pengujian untuk model LSTM pada keras berupa jumlah *layers*, jumlah *cells*, tipe LSTM (*Unidirectional* dan *Bidirectional*), dan melakukan perbandingan model LSTM keras dengan model LSTM *custom forward propagation* berupa hasil akurasi. Berikut adalah hasil pengujiannya.

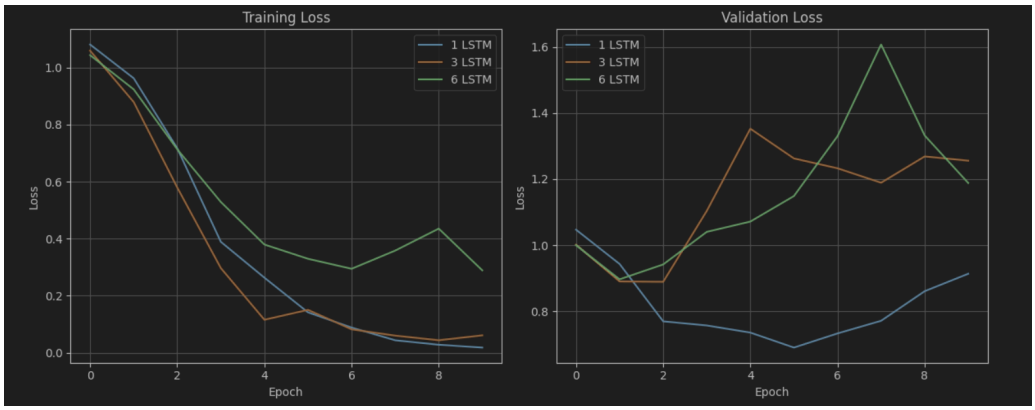
2.2.3.1 Jumlah Layers LSTM

Pada tahap ini akan dilakukan pengujian dengan 3 variasi jumlah *layers* LSTM yaitu 1, 3, dan 6 *layers*. Berikut ini adalah konfigurasi saat melakukan pengujian LSTM.

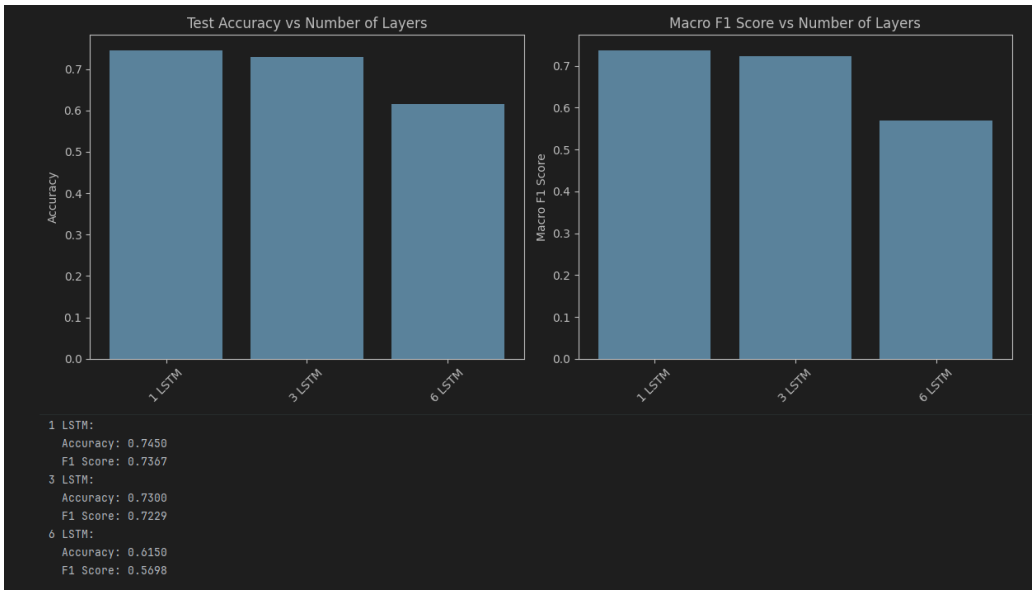
```
epochs = 10
batch_size = 32

lay = {
   "1 LSTM": [
```

```
        vectorizer,
        Embedding(input_dim=vocab_size + 1,
                  output_dim=embedding_dim,
                  mask_zero=True,
                  embeddings_initializer='uniform'),
        LSTM(64),
        Dropout(0.5, seed=SEED),
        Dense(len(label_map),
              activation='softmax')
    ],
    "3 LSTM": [
        vectorizer,
        Embedding(input_dim=vocab_size + 1,
                  output_dim=embedding_dim,
                  mask_zero=True,
                  embeddings_initializer='uniform'),
        LSTM(64, return_sequences=True),
        LSTM(64, return_sequences=True),
        LSTM(64),
        Dropout(0.5, seed=SEED),
        Dense(len(label_map),
              activation='softmax')
    ],
    "6 LSTM": [
        vectorizer,
        Embedding(input_dim=vocab_size + 1,
                  output_dim=embedding_dim,
                  mask_zero=True,
                  embeddings_initializer='uniform'),
        LSTM(64, return_sequences=True),
        LSTM(64, return_sequences=True),
        LSTM(64, return_sequences=True),
        LSTM(64, return_sequences=True),
        LSTM(64, return_sequences=True),
        LSTM(64),
        Dropout(0.5, seed=SEED),
        Dense(len(label_map),
              activation='softmax')
    ]
}
```

Berikut adalah hasil dari *validation* dan *training loss* serta *accuracy* dan *f1 score* dari hasil tahap pengujian ini.



Gambar 2.17 *validation* dan *training loss*'



Gambar 2.18 *accuracy* dan *training loss*

Pada Gambar 2.17 dapat dilihat jika beberapa jumlah *layers* yang diuji mengalami *overfitting*, mari kita fokus pada *training loss* karena pada *validation loss* hasil *loss* tiap *epoch* terlihat *random* karena *validation set* yang terlalu kecil. 3 *layers* mengalami *overfitting* saat *epoch* ke-4 dan ke-8 karena ada kenaikan *loss* dan 6 *layers* juga mengalami lonjakan *loss* pada *epoch* ke-6.

Pada Gambar 2.18 dapat dilihat jika hasil akurasi model akan semakin menurun jika menambahkan jumlah *layers* dan terdapat penurunan yang cukup signifikan dari 3 *layers* ke 6 *layers*.

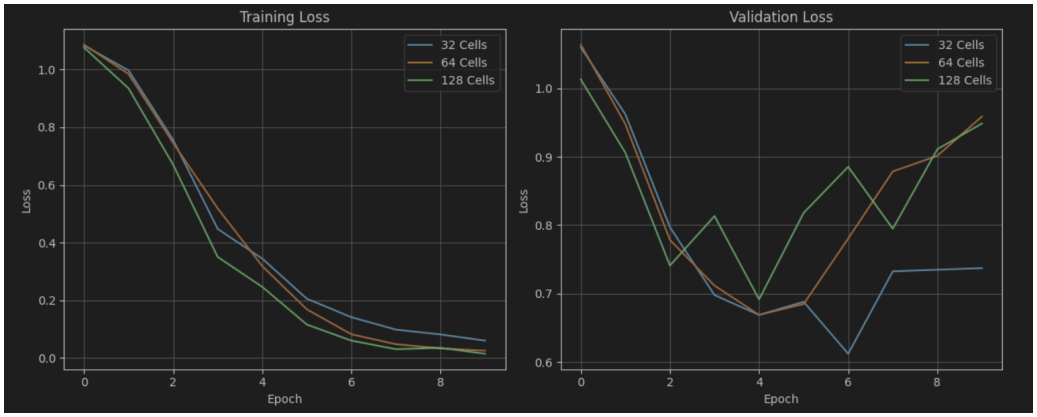
Dari pengujian ini dapat disimpulkan jika dengan menambah jumlah *layers* akan membuat model semakin rawan untuk *overfitting* yang berdampak pada penurunan akurasi.

2.2.3.2 Jumlah Cells

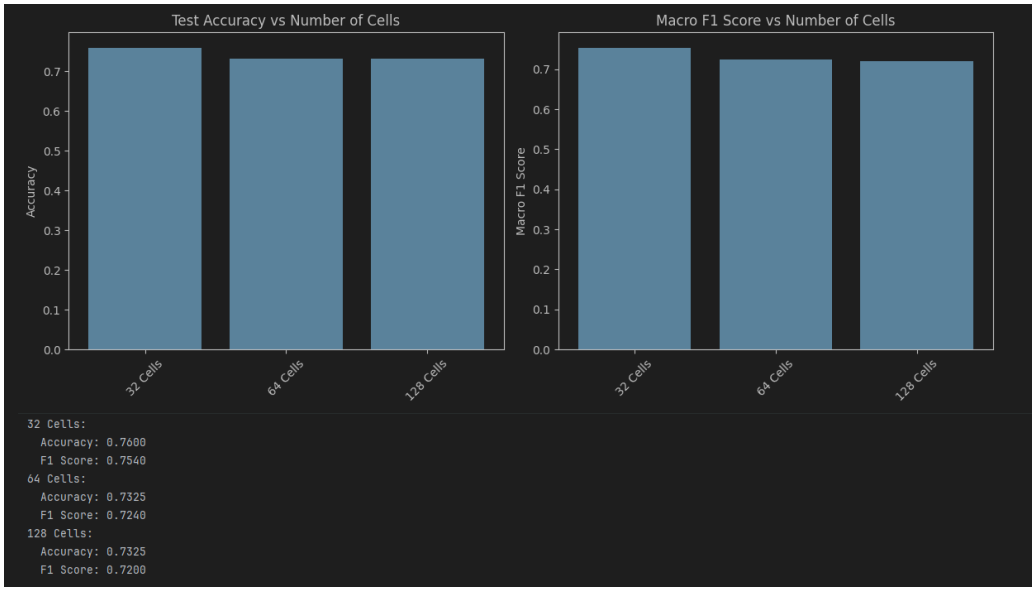
Pada tahap ini akan diujikan 3 variasi jumlah *cells* pada model LSTM keras diantaranya adalah 32, 64, dan 128 *cells*. Berikut adalah konfigurasi yang ada pada model yang diujikan.

```
epochs = 10
batch_size = 32


cells = {
   "32 Cells": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       LSTM(32),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ],
   "64 Cells": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       LSTM(64),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ],
   "128 Cells": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       LSTM(128),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ]
}
```

Berikut adalah hasil dari *training* dan *validation loss* serta *accuracy* dan *f1 score* pada pengujian tahap ini.



Gambar 2.19 *validation loss* dan *training loss*



Gambar 2.20 *accuracy* dan *f1 score*

Pada Gambar 2.19 menampilkan hasil dari *train loss* yang cukup stabil dan *validation loss* yang sangat *random*, untuk tahap ini akan difokuskan untuk analisis *training loss* karena data dari *validation* yang terlalu kecil sehingga mendapatkan hasil seperti di Gambar 2.19. Pada masing masing *cells* terlihat penurunan yang cukup stabil, tetapi pada 128 *cells* terdapat kenaikan *loss* yang tidak terlalu signifikan pada *epoch* ke-7 bisa jadi hal ini merupakan tanda model pada *cells* tersebut *overfitting*.

Pada Gambar 2.20 menampilkan hasil dari akurasi *cells* yang diuji. Dapat dilihat jika semakin banyaknya *cells* yang diuji hasil akurasinya semakin menurun. Penurunan yang cukup signifikan terjadi dari 32 *cells* ke 64 *cells* setelahnya ia tidak terlalu menurun signifikan.

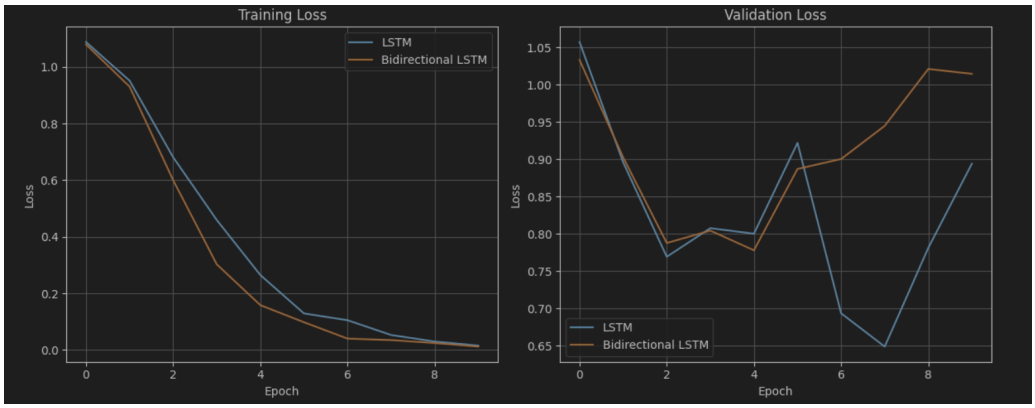
Pada pengujian ini dapat disimpulkan jika menambah jumlah *cells* pada LSTM dapat mengurangi akurasi walaupun tidak terlalu signifikan dan menimbulkan *overfitting*, jika ingin melihat hasilnya secara jelas mungkin bisa dilakukan penambahan *epoch* pada model.

2.2.3.3 Tipe LSTM

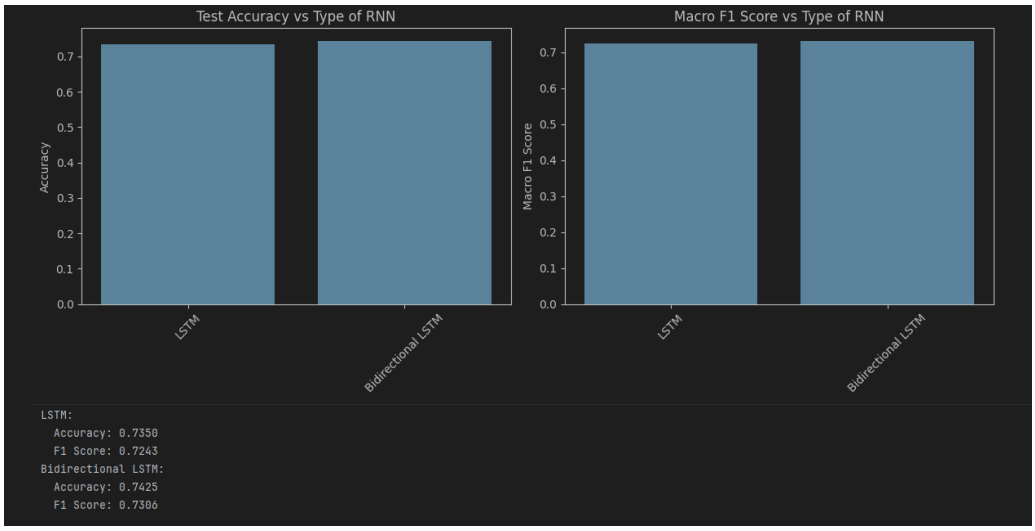
Pada tahap ini akan dilakukan pengujian dengan model LSTM *Unidirectional* dan LSTM *Bidirectional*. Berikut adalah konfigurasi yang ada pada masing masing model yang diuji.

```
epochs = 10
batch_size = 32

rnn_types = {
   "LSTM": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       LSTM(64),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ],
   "Bidirectional LSTM": [
       vectorizer,
       Embedding(input_dim=vocab_size + 1,
                 output_dim=embedding_dim,
                 mask_zero=True,
                 embeddings_initializer='uniform'),
       Bidirectional(LSTM(64)),
       Dropout(0.5, seed=SEED),
       Dense(len(label_map),
             activation='softmax')
   ]
}
```

Berikut adalah hasil dari *validation* dan *training loss* serta *accuracy* dan *f1 score* pada masing masing tipe LSTM.



Gambar 2.21 *validation* dan *training loss*



Gambar 2.22 *accuracy* dan *f1 score*

Pada Gambar 2.21 menampilkan hasil dari *train loss* yang cukup stabil menurun dan *validation loss* yang menghasilkan *loss* yang random tiap *epoch*-nya, permasalahannya sama seperti *validation loss* pada pengujian sebelumnya. Hasil akhir *epoch* menampilkan jika *loss* pada keduanya hampir mirip.

Pada Gambar 2.22 menampilkan hasil dari akurasi pada kedua tipe LSTM yang dimana *Bidirectional* memiliki akurasi paling tinggi dibandingkan dengan *Unidirectional* tetapi tidak terlalu signifikan perbedaanya.

2.2.3.4 Custom LSTM Forward Propagation

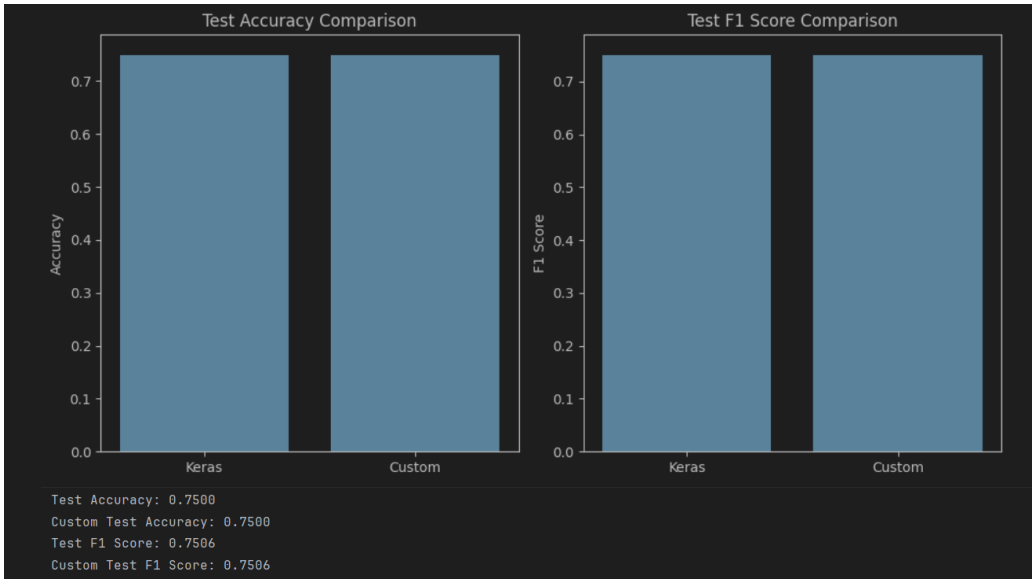
Pada pengujian ini akan dilakukan perbandingan hasil dari akurasi model LSTM *custom forward propagation* yang diimplementasi pada Tabel 2.2 dengan model LSTM keras. Berikut adalah konfigurasi dari model LSTM keras yang akan dipakai oleh LSTM *custom*.

```
epochs = 10
batch_size = 32

accuracy = []
train_loss = {}
val_loss = {}
f1_scores = []

tf.random.set_seed(SEED)
np.random.seed(SEED)

model = Sequential([
    vectorizer,
    Embedding(input_dim=vocab_size + 1,
              output_dim=embedding_dim,
              mask_zero=True,
              embeddings_initializer='uniform'),
    LSTM(64),
    Dropout(0.5, seed=SEED),
    Dense(len(label_map),
          activation='softmax')
])

model.compile(
    loss="sparse_categorical_crossentropy",
    optimizer='adam',
    metrics=["accuracy"]
)

history = model.fit(
    np.array(train_texts, dtype=object),
    y_train,
    epochs=epochs,
    batch_size=batch_size,
    validation_data=(np.array(val_texts, dtype=object), y_val),
    verbose=1,
    shuffle=True
)
```

Setelah melakukan pelatihan, model akan disimpan dalam bentuk *pickle file* dengan nama “lstm.pkl” dan akan di-*load* oleh LSTM *custom* sebagai berikut

```
custom_model = RNNLSTMFromScratch('../model/lstm.pkl')
```

Berikut adalah hasil dari perbandingan akurasi LSTM model keras dengan LSTM model *custom forward propagation* yang diimplementasi.



Gambar 2.23 *accuracy* dan *f1 score*

Pada Gambar 2.23 menampilkan hasil akurasi dari masing masing model dan dapat disimpulkan jika model *forward propagation* LSTM yang diimplementasi sudah sesuai dengan LSTM model keras.

BAB III
PENUTUP

3.1 Kesimpulan

Berdasarkan seluruh rangkaian pengujian yang dilakukan terhadap model CNN, RNN, dan LSTM, dapat disimpulkan bahwa kinerja model sangat dipengaruhi oleh konfigurasi arsitektur dan parameter yang digunakan. Berikut adalah kesimpulan dari setiap pengujian,

- Penambahan *convolution layers* pada CNN dapat meningkatkan akurasi dan mengurangi *loss* dengan catatan parameter pembelajaran harus cukup dengan dataset yang ada agar tidak terjadi *overfitting*.
- Penambahan jumlah *filters* pada CNN dapat meningkatkan akurasi, tetapi tidak terlalu signifikan kenaikannya.
- Semakin besar *filters* pada CNN hasil akurasinya akan semakin menurun dan *loss* akan semakin besar, hal ini bisa terjadi karena CNN dengan *filters* kecil dapat mempelajari detail detail kecil yang ada pada data sehingga akurasinya akan lebih baik dibandingkan dengan *filters* yang besar.
- *Max Pooling* memiliki akurasi yang lebih tinggi dan *loss* yang lebih rendah dibandingkan dengan *Average Pooling*, hal ini bisa terjadi karena *Max Pooling* hanya mempelajari data yang paling menonjol saja dan tidak seperti *Average Pooling* yang mempelajari semuanya dan malah menimbulkan *noise*.
- Penambahan *layers Simple RNN* pada pengujian RNN berdampak pada akurasi yang semakin menaik walaupun terdapat kenaikan yang tidak terlalu signifikan pada 3-6 *layers*.
- Penambahan *cells* pada RNN dapat mengurangi akurasi dan menimbulkan *overfitting* pada model, hal ini mungkin terjadi karena adanya *vanishing gradient* saat melakukan pembelajaran.
- *Bidirectional* RNN lebih baik dalam melakukan pembelajaran dataset NusaX dibandingkan dengan *Unidirectional* RNN secara akurasi.

- Penambahan *layers* LSTM pada pengujian ini berdampak pada penurunan akurasi dan naiknya *loss*, hal ini mungkin terjadi karena kompleksnya pembelajaran yang terjadi pada LSTM dan menyebabkan *overfitting* dan penurunan akurasi.
- Penambahan *cells* pada *layer* LSTM dapat mengurangi akurasi, hal ini sama seperti halnya penambahan *cells* pada RNN.
- *Bidirectional* LSTM lebih baik dibandingkan dengan *Unidirectional* LSTM, tetapi tidak memberikan peningkatan akurasi yang signifikan seperti RNN.

3.2 Saran

Sebagai saran dari pengujian FFNN ini, terdapat beberapa saran yang dapat dilakukan untuk pengembangan pengujian ini.

1. Untuk mengatasi *overfitting* yang mungkin terjadi pada model yang lebih kompleks, disarankan untuk menambahkan regularisasi seperti L2 *regularization*.
2. Implementasi strategi *early stopping* dan penyesuaian *learning rate* selama *training* bisa meningkatkan efisiensi *training*.

3.3 Pembagian Tugas

NIM	Nama	Tugas
13521031	Fahrian Afdholi	Semua