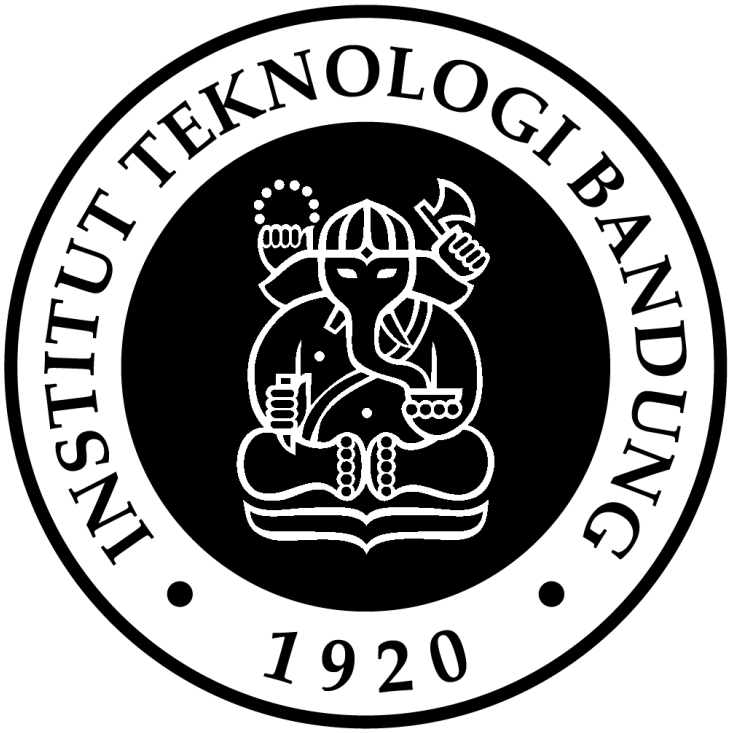
Laporan Tugas Besar 2 IF3270 Pembelajaran Mesin

Semester II tahun 2024/2025

Convolutional Neural Network dan Recurrent Neural Network



Oleh:

Jimly Nur Arif	(13522123)
~~Yosef Rafael Joshua~~	~~(13522133)~~
Rayhan Ridhar Rahman	(13522160)

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

Daftar Isi

Bab 1 Deskripsi Persoalan

Persoalan yang dihadapi pada Tugas Besar ini adalah pengimplementasian fungsi forward propagation untuk beberapa arsitektur dalam Machine Learning, yaitu: *Convolutional Neural Network*, *Recurrent Neural Network*, dan *Long Short Term Memory*. Untuk pelaksanaan training akan menggunakan bantuan dari library *Keras* dari *TensorFlow*. Kemudian dilakukan pengambilan terhadap bobot dari hasil training tersebut untuk digunakan model fungsi forward propagation yang disusun dari awal. Kemudian setelah menghasilkan prediksi, kesuksesan akan diukur menggunakan model f1-score.

Pada CNN, akan dilakukan klasifikasi gambar berdasarkan dataset CIFAR-10. Dataset tersebut berisi 60000 data gambar. 50000 untuk training dan sisanya untuk pengujian. Setiap gambar memiliki ukuran 32×32×3. Terdiri dari 3 *channel* untuk masing-masing warna. Terpenting dari model ini adalah pembuatan fungsi konvolusinya. Maka beberapa faktor yang perlu diperhatikan adalah, jumlah layer konvolusi, ukuran kernel dari filter, dan banyaknya filter. Kemudian terdapat layer pooling antara MaxPooling dan AveragePooling.

Pada RNN, neural network tersebut dikenal baik untuk melakukan prediksi berdasarkan suatu urutan. Dataset yang digunakan adalah NusaX-Sentiment yang merupakan dataset untuk melakukan prediksi kata. Langkah pertama adalah melakukan preprocessing data melaluii tokenisasi. Kemudian model yang dihasilkan library akan terdiri dari beberapa layer yaitu: embedding layer, directional layer, dropout layer, dan dense layer. Dengan Loss function Sparse Categorical Crossentropy. Kemudian bobot yang dihasilkan akan dimasukkan ke model buatan untuk memprediksi kata selanjutnya.

Pada LSTM, neural network yang dibentuk akan serupa dengan RNN. Dimulai dengan preprocessing data melalui tokenisasi, kemudian disalurkan ke layer-layer model. Prebedaannya adalah terdapat memori jangka panjang dan jangka pendek. Memori jangka panjang akan menyimpan nilai yang serupa dapat dilupakan beberapa porsinya dan memori jangka pendek akan menyimpan nilai dari prediksi sebelumnya. Setelah dilakukan training oleh model Keras, weight akan disimpan dan digunakan pada model buatan sendiri.

Bab 2 Pembahasan

2.1. Penjelasan Implementasi

2.1.1. Convolutional Neural Network

Menggunakan library akan dibuat suatu model CNN. Kemudian data training dimasukkan ke model tersebut dan diambil bobot dan bias yang dihasilkan dari training untuk diprediksi oleh model buatan.

Layer Konvolusi direpresentasikan dengan kelas berikut dalam python:

```
class Conv2DScratch:
    def __init__(self, weight, bias, stride=1, padding=0):
        self.weight = weight
        self.bias = bias
        self.stride = stride
        self.padding = padding

    def pad_input(self, x):
        if self.padding == 0:
            return x
        return np.pad(x, ((0, 0), (self.padding, self.padding),
(self.padding, self.padding)), mode='constant')

    def forward(self, x):
        C_out, C_in, kH, kW = self.weight.shape
        x_padded = self.pad_input(x)
        _, H_in, W_in = x.shape
        H_out = (H_in + 2*self.padding - kH) // self.stride + 1
        W_out = (W_in + 2*self.padding - kW) // self.stride + 1

        out = np.zeros((C_out, H_out, W_out))

        for oc in range(C_out):
            for i in range(H_out):
                for j in range(W_out):
                    for ic in range(C_in):
                        h_start = i * self.stride
                        w_start = j * self.stride
                        patch = x_padded[ic, h_start:h_start+kH,
w_start:w_start+kW]
                        out[oc, i, j] += np.sum(patch *
self.weight[oc, ic])
                    out[oc, i, j] += self.bias[oc]
        return out
```

Menerima hasil dari weight training dalam format (N,C,H,W) dengan N adalah jumlah kernel, C adalah dimensi channel, H adalah tinggi dari kernel, dan W adalah lebar dari Kernel. Mengingat hal tersebut, format dalam tensor tidak seperti itu dan merepresentasikan bentuk (H,W,

n_input, n_output), sehingga perlu dilakukan transpose dengan weight.transpose(3, 2, 0, 1) sebelum dimasukkan ke layer konvolusi yang dibuat. Kemudian kelas tersebut menerima bias untuk masing-masing kernel.

Dalam layer konvolusi akan diperlukan padding agar hasil memiliki bentuk yang serupa ketika input diberikan. Ini dilakukan karena dalam model library yang digunakan menggunakan padding='same' yang menyebabkan bentuk input dan output harus sama. Sehingga ukuran padding = (ukuran_kernel - 1) / 2.

Setelah setiap proses konvolusi, akan menggunakan aktivasi ReLU sehingga tidak ada nilai yang negatif.

Kemudian layer pooling direpresentasikan berdasarkan kelas berikut dalam python:

```
class MaxPool2DScratch:
    def forward(self, x):
        # x: (C, H, W)
        C, H, W = x.shape
        out = np.zeros((C, H // 2, W // 2))
        for c in range(C):
            for i in range(0, H, 2):
                for j in range(0, W, 2):
                    out[c, i//2, j//2] = np.max(x[c, i:i+2, j:j+2])
        return out

class AveragePool2DScratch:
    def forward(self, x):
        # x: (C, H, W)
        C, H, W = x.shape
        out = np.zeros((C, H // 2, W // 2))
        for c in range(C):
            for i in range(0, H, 2):
                for j in range(0, W, 2):
                    out[c, i//2, j//2] = np.mean(x[c, i:i+2,
j:j+2])
        return out
```

Pooling dilakukan berdasarkan kernel 2×2 dengan 2 stride. Pooling akan menghasilkan bentuk yang setengah dari inputnya. Max pooling akan mengambil nilai tertinggi dari kernel yang dituju dan average pooling akan mengambil rata-rata nilai.

Layer Dense digunakan untuk menyimpulkan input menjadi sejumlah output. Kemudian akan dilakukan aktivasi dengan softmax untuk training. Jika untuk memprediksi akan digunakan fungsi argmax.

```
class FlattenScratch:
    def forward(self, x):
        return x.flatten()

class DenseScratch:
    def __init__(self, weight, bias):
        self.weight = weight
```

```
        self.bias = bias

    def forward(self, x):
        return self.weight @ x + self.bias

class SoftmaxScratch:
    def forward(self, x):
        x_shifted = x - np.max(x)
        exp_x = np.exp(x_shifted)
        return exp_x / np.sum(exp_x)
```

2.1.2. Recurrent Neural Network

Training akan dilakukan oleh library bawaan dari Keras.

Model buatan sendiri kemudian akan menyimpan referensi terhadap model library tersebut dan mengambil berat dan bias yang dihasilkan dari training. Begitu juga dengan info dari masing-masing layer seperti jenis, dimensi, nama, dan indeks. Setiap layer disusun sebagai suatu fungsi di dalam kelas SimpleRNNFromScratch. Berikut adalah potongan kode untuk mengambil bobot dan bias serta identitas layer dari model.

```
class SimpleRNNFromScratch:
    """Implementasi forward propagation Simple RNN from scratch"""

    def __init__(self, model, vectorizer):
        self.model = model
        self.vectorizer = vectorizer
        self.weights = self.extract_weights()
        self.layer_info = self.get_layer_info()

    def extract_weights(self):
        """Extract weights dari trained Keras model"""
        weights = {}

        print("\n=== EXTRACTING MODEL WEIGHTS ===")

        for i, layer in enumerate(self.model.layers):
            layer_name = f"layer_{i}_{layer.__class__.__name__}"
            layer_type = layer.__class__.__name__

            if hasattr(layer, 'get_weights') and
layer.get_weights():
                layer_weights = layer.get_weights()
                weights[layer_name] = layer_weights

                print(f"Layer {i} ({layer_type}):")
                for j, w in enumerate(layer_weights):
                    print(f"  Weight {j}: {w.shape}")
```

```
        return weights

    def get_layer_info(self):
        """Get layer information"""
        layer_info = []

        for i, layer in enumerate(self.model.layers):
            layer_info.append({
                'index': i,
                'name': f"layer_{i}_{layer.__class__.__name__}",
                'type': layer.__class__.__name__,
                'layer': layer
            })

        return layer_info
```

Terdapat juga fungsi aktivasi yang digunakan

```
    def tanh(self, x):
        """Tanh activation function"""
        return np.tanh(x)

    def softmax(self, x):
        """Softmax activation function dengan numerical
stability"""
        if x.ndim == 1:
            x = x.reshape(1, -1)

        # Subtract max for numerical stability
        x_stable = x - np.max(x, axis=-1, keepdims=True)
        exp_x = np.exp(x_stable)
        return exp_x / np.sum(exp_x, axis=-1, keepdims=True)
```

Kemudian layer embedding bisa direpresentasikan oleh fungsi berikut ini:

```
    def embedding_forward(self, input_ids, embedding_weights):
        """Forward pass untuk embedding layer"""
        if input_ids.ndim == 1:
            input_ids = input_ids.reshape(1, -1)

        return embedding_weights[input_ids]
```

Kemudian ada juga fungsi yang digunakan untuk melakukan forward pass dari satu layer ke yang selanjutnya

```
    def simple_rnn_cell_forward(self, x, h_prev, weights):
        """Forward pass untuk single Simple RNN cell"""
        # Simple RNN: h_t = tanh(W_x * x_t + W_h * h_{t-1} + b)

        W_x, W_h, b = weights[0], weights[1], weights[2]

        # Ensure proper shapes
        if x.ndim == 1:
            x = x.reshape(1, -1)
        if h_prev.ndim == 1:
            h_prev = h_prev.reshape(1, -1)

        # Simple RNN computation
        h_t = self.tanh(np.dot(x, W_x) + np.dot(h_prev, W_h) + b)

        return h_t

    def simple_rnn_forward(self, x, weights,
return_sequences=False):
        """Forward pass untuk Simple RNN layer"""
        if x.ndim == 2:  # Add batch dimension if needed
            x = x.reshape(1, x.shape[0], x.shape[1])

        batch_size, seq_len, input_dim = x.shape

        # Get hidden dimension from weights
        hidden_dim = weights[1].shape[0]  # From recurrent weights
W_h

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

        if return_sequences:
            outputs = np.zeros((batch_size, seq_len, hidden_dim))
            for t in range(seq_len):
                h = self.simple_rnn_cell_forward(x[:, t, :], h,
weights)
                outputs[:, t, :] = h
            return outputs
        else:
            for t in range(seq_len):
                h = self.simple_rnn_cell_forward(x[:, t, :], h,
weights)
            return h

    def bidirectional_simple_rnn_forward(self, x, weights):
        """Forward pass untuk Bidirectional Simple RNN layer"""
        # Ensure x is 3D
        if x.ndim == 2:
            x = x.reshape(1, x.shape[0], x.shape[1])
```

```
        # Split weights untuk forward dan backward
        num_weights_per_direction = len(weights) // 2
        forward_weights = weights[:num_weights_per_direction]
        backward_weights = weights[num_weights_per_direction:]

        # Forward direction
        forward_out = self.simple_rnn_forward(x, forward_weights,
return_sequences=False)

        # Backward direction (reverse sequence)
        x_reversed = x[:, ::-1, :]  # Now x is guaranteed to be 3D
        backward_out = self.simple_rnn_forward(x_reversed,
backward_weights, return_sequences=False)

        # Concatenate outputs
        if forward_out.ndim == 1:
            forward_out = forward_out.reshape(1, -1)
        if backward_out.ndim == 1:
            backward_out = backward_out.reshape(1, -1)

        combined_out = np.concatenate([forward_out, backward_out],
axis=1)
        return combined_out

    def dense_forward(self, x, weights):
        """Forward pass untuk dense layer"""
        W, b = weights[0], weights[1]

        if x.ndim == 1:
            x = x.reshape(1, -1)

        return np.dot(x, W) + b
```

2.1.3. Long Short Term Memory

Mulanya data di load, dilakukan TextVectorization, encoding label, membuat lstm menggunakan keras, melakukan analisis hyperparameter layer lstm, jumlah unit, directional, kemudian membuat lstm from scratch, membandingkan lstm from scratch dengan keras. menghitung loss.

cara kerja lstm from scratch adalah mengambil weight dari model yang sudah di train sebelumnya. kemudian

```
class LSTMFromScratch:
    """Implementasi forward propagation LSTM from scratch"""
```

```
    def __init__(self, model, vectorizer):
        self.model = model
        self.vectorizer = vectorizer
        self.weights = self.extract_weights()
        self.layer_info = self.get_layer_info()

    def extract_weights(self):
        """Extract weights dari trained Keras model"""
        weights = {}

        print("\n=== EXTRACTING MODEL WEIGHTS ===")

        for i, layer in enumerate(self.model.layers):
            layer_name = f"layer_{i}_{layer.__class__.__name__}"
            layer_type = layer.__class__.__name__

            if hasattr(layer, 'get_weights') and
layer.get_weights():
                layer_weights = layer.get_weights()
                weights[layer_name] = layer_weights

                print(f"Layer {i} ({layer_type}):")
                for j, w in enumerate(layer_weights):
                    print(f"  Weight {j}: {w.shape}")

        return weights

    def get_layer_info(self):
        """Get layer information"""
        layer_info = []

        for i, layer in enumerate(self.model.layers):
            layer_info.append({
                'index': i,
                'name': f"layer_{i}_{layer.__class__.__name__}",
                'type': layer.__class__.__name__,
                'layer': layer
            })

        return layer_info

    def sigmoid(self, x):
        """Sigmoid activation function"""
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))

    def tanh(self, x):
        """Tanh activation function"""
        return np.tanh(x)

    def softmax(self, x):
        """Softmax activation function dengan numerical
```

```
stability"""
        if x.ndim == 1:
            x = x.reshape(1, -1)

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

    def embedding_forward(self, input_ids, embedding_weights):
        """Forward pass untuk embedding layer"""
        if input_ids.ndim == 1:
            input_ids = input_ids.reshape(1, -1)

        return embedding_weights[input_ids]

    def lstm_cell_forward(self, x, h_prev, c_prev, weights):
        """Forward pass untuk single LSTM cell"""
        # LSTM gates computation
        # Keras menggunakan format: [W_xi, W_xf, W_xc, W_xo]
        # dan [W_hi, W_hf, W_hc, W_ho] untuk recurrent weights

        W_input, W_recurrent, b = weights[0], weights[1],
weights[2]

        # Ensure proper shapes
        if x.ndim == 1:
            x = x.reshape(1, -1)
        if h_prev.ndim == 1:
            h_prev = h_prev.reshape(1, -1)
        if c_prev.ndim == 1:
            c_prev = c_prev.reshape(1, -1)

        # Split weights untuk 4 gates (input, forget, cell, output)
        units = W_recurrent.shape[0]

        W_xi = W_input[:, :units]
        W_hi = W_recurrent[:, :units]
        b_i = b[:units]

        W_xf = W_input[:, units:2*units]
        W_hf = W_recurrent[:, units:2*units]
        b_f = b[units:2*units]

        W_xc = W_input[:, 2*units:3*units]
        W_hc = W_recurrent[:, 2*units:3*units]
        b_c = b[2*units:3*units]

        W_xo = W_input[:, 3*units:]
        W_ho = W_recurrent[:, 3*units:]
        b_o = b[3*units:]
```

```
        i_t = self.sigmoid(np.dot(x, W_xi) + np.dot(h_prev, W_hi) +
b_i)
        f_t = self.sigmoid(np.dot(x, W_xf) + np.dot(h_prev, W_hf) +
b_f)
        c_tilde = self.tanh(np.dot(x, W_xc) + np.dot(h_prev, W_hc)
+ b_c)
        o_t = self.sigmoid(np.dot(x, W_xo) + np.dot(h_prev, W_ho) +
b_o)

        c_t = f_t * c_prev + i_t * c_tilde
        h_t = o_t * self.tanh(c_t)

        return h_t, c_t

    def lstm_forward(self, x, weights, return_sequences=False):
        if x.ndim == 2:  # Sesuaikan batch dimension supaya sesuai
format
            x = x.reshape(1, x.shape[0], x.shape[1])

        batch_size, seq_len, input_dim = x.shape

        hidden_dim = weights[1].shape[0]

        h = np.zeros((batch_size, hidden_dim))
        c = np.zeros((batch_size, hidden_dim))

        if return_sequences:
            outputs = np.zeros((batch_size, seq_len, hidden_dim))
            for t in range(seq_len):
                h, c = self.lstm_cell_forward(x[:, t, :], h, c,
weights)
                outputs[:, t, :] = h
            return outputs
        else:
            for t in range(seq_len):
                h, c = self.lstm_cell_forward(x[:, t, :], h, c,
weights)
            return h

    def bidirectional_lstm_forward(self, x, weights):

        if x.ndim == 2:
            x = x.reshape(1, x.shape[0], x.shape[1])

        num_weights_per_direction = len(weights) // 2
        forward_weights = weights[:num_weights_per_direction]
        backward_weights = weights[num_weights_per_direction:]

        forward_out = self.lstm_forward(x, forward_weights,
return_sequences=False)
```

```
        x_reversed = x[:, ::-1, :]
        backward_out = self.lstm_forward(x_reversed,
backward_weights, return_sequences=False)

        if forward_out.ndim == 1:
            forward_out = forward_out.reshape(1, -1)
        if backward_out.ndim == 1:
            backward_out = backward_out.reshape(1, -1)

        combined_out = np.concatenate([forward_out, backward_out],
axis=1)
        return combined_out

    def dense_forward(self, x, weights):
        W, b = weights[0], weights[1]

        if x.ndim == 1:
            x = x.reshape(1, -1)

        return np.dot(x, W) + b

    def predict(self, X_test):
        print(f"\nPredicting for {X_test.shape[0]} samples...")

        predictions = []

        for i in range(X_test.shape[0]):
            # Get single sample
            input_ids = X_test[i:i+1]  # Keep batch dimension,
jangan X_test[i] pasti error karena dimensinya gak sesuai format
            current_output = input_ids

            # Forward pass semua layer
            for layer_info in self.layer_info:
                layer_name = layer_info['name']
                layer_type = layer_info['type']

                if layer_type == 'Embedding':
                    if layer_name in self.weights:
                        embedding_weights =
self.weights[layer_name][0]
                        current_output =
self.embedding_forward(current_output, embedding_weights)

                elif layer_type == 'LSTM':
                    if layer_name in self.weights:
                        is_last_lstm = True
                        for j in range(layer_info['index']+1,
len(self.layer_info)):
                            if self.layer_info[j]['type'] in
['LSTM', 'Bidirectional']:
```

```
                                is_last_lstm = False
                                break

                        return_sequences = not is_last_lstm
                        current_output = self.lstm_forward(
                            current_output,
                            self.weights[layer_name],
                            return_sequences=return_sequences
                        )

                elif layer_type == 'Bidirectional':
                    if layer_name in self.weights:
                        current_output =
self.bidirectional_lstm_forward(
                            current_output,
                            self.weights[layer_name]
                        )

                elif layer_type == 'Dropout':
                    # Skip dropout in inference
                    continue

                elif layer_type == 'Dense':
                    if layer_name in self.weights:
                        current_output =
self.dense_forward(current_output, self.weights[layer_name])

                        # Apply softmax if this is the last layer
(output layer)
                        if layer_info['index'] ==
len(self.layer_info) - 1:
                            current_output =
self.softmax(current_output)

            predictions.append(current_output[0])  # Remove batch
dimension

            if (i + 1) % 100 == 0:
                print(f"  Processed {i + 1}/{X_test.shape[0]}
samples")

        return np.array(predictions)
```

2.2. Hasil Pengujian

2.2.1. Convolutional Neural Network

Basis pengujian: (3 filter, 3×3 kernel-size, 2 conv layer, maxPooling) w/ model

```
# Library
model = Sequential([
    Input(shape=(32, 32, 3)),
    Conv2D(3, (3, 3), padding='same', name='conv1'),
    ReLU(),
    Conv2D(3, (3, 3), padding='same', name='conv2'),
    ReLU(),
    MaxPooling2D(pool_size=2, strides=2, name='pool'),
    Flatten(),
    Dense(10, activation='softmax', name='fc')
])

# Scratch
myModel = MySequential()
myModel.add(Conv2DScratch(data["conv1_w"], data["conv1_b"],
padding=1))
myModel.add(ReLUScratch())
myModel.add(Conv2DScratch(data["conv2_w"], data["conv2_b"],
padding=1))
myModel.add(ReLUScratch())
myModel.add(MaxPool2DScratch())
myModel.add(FlattenScratch())
myModel.add(DenseScratch(data["fc_w"], data["fc_b"]))
myModel.add(SoftmaxScratch())
```

Library prediction score:

```
o_macro_f1 = f1_score(y_test.flatten(), o_y_pred, average='macro')
print("Macro F1-score:", o_macro_f1)
```

```
Macro F1-score: 0.46777719439512644
```

Scratch prediction score:

```
macro_f1 = f1_score(y_test, y_pred, average='macro')
print("Macro F1-score:", macro_f1)
```

```
Macro F1-score: 0.08048267442564698
```

(3 filter, 3×3 kernel-size, 2 conv layer, averagePooling)
Library prediction score:

```
o_macro_f1 = f1_score(y_test.flatten(), o_y_pred, average='macro')
print("Macro F1-score:", o_macro_f1)
```

```
Macro F1-score: 0.47572486486052645
```

Scratch prediction score:

```
macro_f1 = f1_score(y_test, y_pred, average='macro')
print("Macro F1-score:", macro_f1)
```

```
Macro F1-score: 0.042656360762554545
```

(3 filter, 3×3 kernel-size, 3 conv layer, maxPooling)

Library prediction score:

```
o_macro_f1 = f1_score(y_test.flatten(), o_y_pred, average='macro')
print("Macro F1-score:", o_macro_f1)
```

```
Macro F1-score: 0.48579990542399243
```

Scratch prediction score:

```
macro_f1 = f1_score(y_test, y_pred, average='macro')
print("Macro F1-score:", macro_f1)
```

```
Macro F1-score: 0.07775405312850872
```

(3 filter, 3×3 kernel-size, 4 conv layer, maxPooling)

Library prediction score:

```
o_macro_f1 = f1_score(y_test.flatten(), o_y_pred, average='macro')
print("Macro F1-score:", o_macro_f1)
```

```
Macro F1-score: 0.47540873316105065
```

Scratch prediction score:

```
macro_f1 = f1_score(y_test, y_pred, average='macro')
print("Macro F1-score:", macro_f1)
```

```
Macro F1-score: 0.044742280611098556
```

(2 filter, 3×3 kernel-size, 2 conv layer, maxPooling)

Library prediction score:

```
o_macro_f1 = f1_score(y_test.flatten(), o_y_pred, average='macro')
print("Macro F1-score:", o_macro_f1)
```

```
Macro F1-score: 0.3895190368551136
```

Scratch prediction score:

```
macro_f1 = f1_score(y_test, y_pred, average='macro')
print("Macro F1-score:", macro_f1)
```

```
Macro F1-score: 0.08370834384183609
```

(4 filter, 3×3 kernel-size, 2 conv layer, maxPooling)

Library prediction score:

```
o_macro_f1 = f1_score(y_test.flatten(), o_y_pred, average='macro')
print("Macro F1-score:", o_macro_f1)
```

```
Macro F1-score: 0.5132189270901975
```

Scratch prediction score:

```
macro_f1 = f1_score(y_test, y_pred, average='macro')
print("Macro F1-score:", macro_f1)
```

```
Macro F1-score: 0.066070913505994 84
```

(3 filter, 5×5 kernel-size, 2 conv layer, maxPooling)

Library prediction score:

```
o_macro_f1 = f1_score(y_test.flatten(), o_y_pred, average='macro')
print("Macro F1-score:", o_macro_f1)
```

```
Macro F1-score: 0.47825036080672445
```

Scratch prediction score:

```
macro_f1 = f1_score(y_test, y_pred, average='macro')
print("Macro F1-score:", macro_f1)
```

```
Macro F1-score: 0.047114289145776186
```

(3 filter, 7×7 kernel-size, 2 conv layer, maxPooling)

Library prediction score:

```
o_macro_f1 = f1_score(y_test.flatten(), o_y_pred, average='macro')
print("Macro F1-score:", o_macro_f1)
```

```
Macro F1-score: 0.433556296723384
```

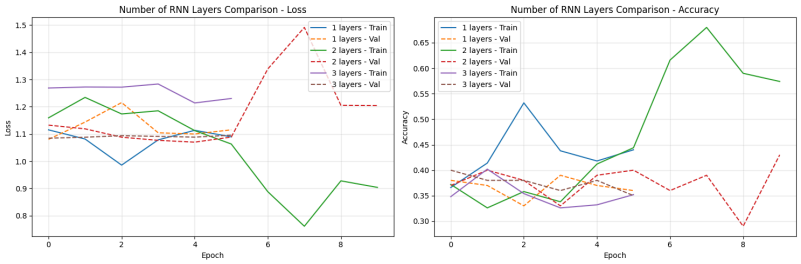
Scratch prediction score:

```
macro_f1 = f1_score(y_test, y_pred, average='macro')
print("Macro F1-score:", macro_f1)
```

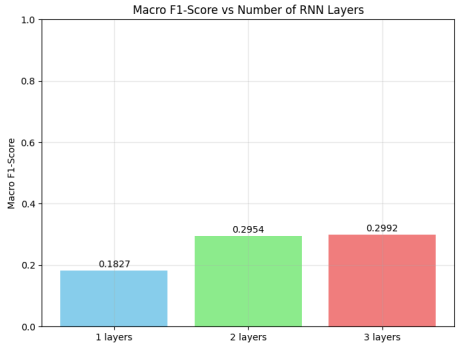
```
Macro F1-score: 0.07573150294509395
```

2.2.2. Recurrent Neural Network

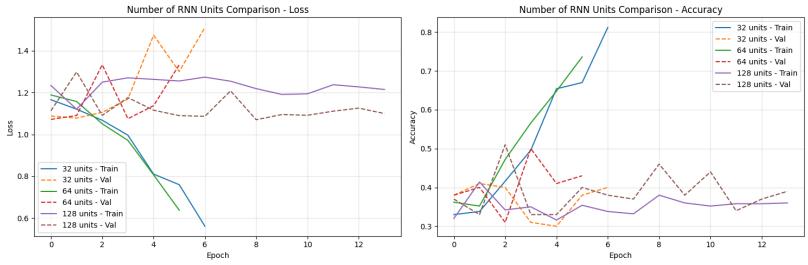
Grafik untuk akurasi dan loss dari perbedaan **jumlah layer RNN**



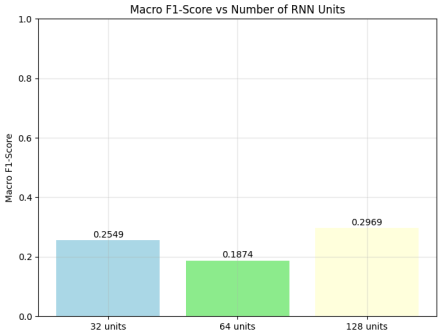
Grafik nilai macro f1 dari perbedaan **jumlah layer RNN**



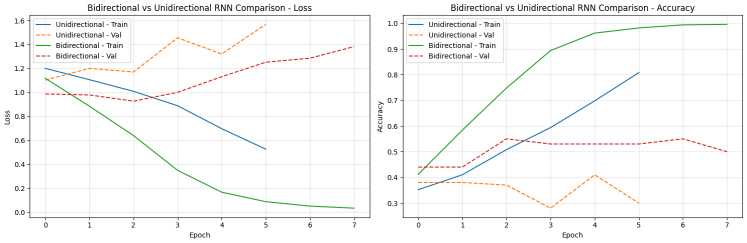
Grafik untuk akurasi dan loss dari perbedaan **jumlah sel dalam RNN**



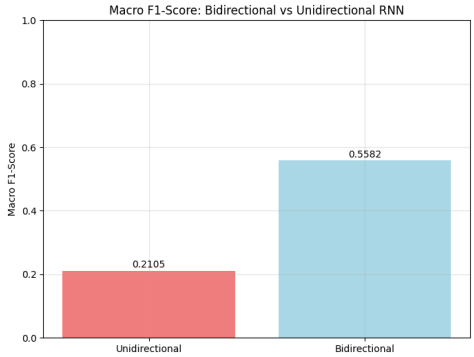
Grafik nilai macro f1 dari perbedaan **jumlah sel dalam RNN**



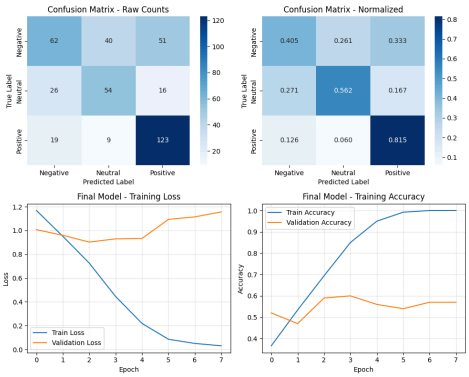
Grafik untuk akurasi dan loss dari perbedaan **jenis arah RNN**



Grafik nilai macro f1 dari perbedaan **jenis arah RNN**



Data model final

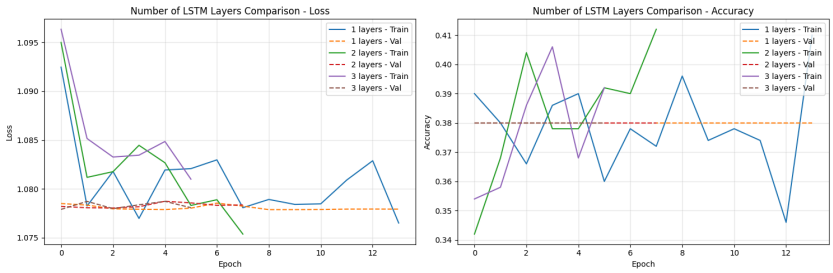



```
First 10 predictions comparison:
```

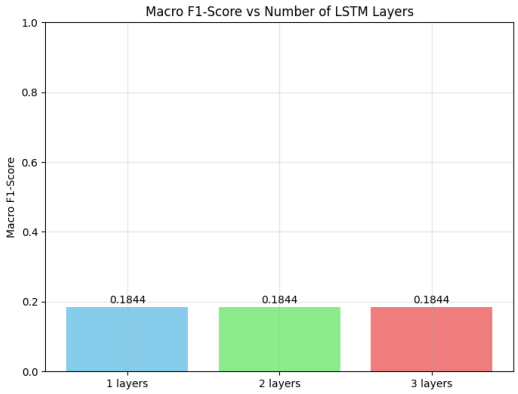
```
Sample | Keras Class | Scratch Class | Match | True Label
-----------------------------------------------------------
     1 |           2 |             0 | ✗     |          2
     2 |           1 |             1 | ✓     |          1
     3 |           1 |             0 | ✗     |          0
     4 |           2 |             0 | ✗     |          2
     5 |           1 |             0 | ✗     |          1
     6 |           0 |             0 | ✓     |          0
     7 |           1 |             0 | ✗     |          1
     8 |           1 |             0 | ✗     |          0
     9 |           2 |             2 | ✓     |          2
    10 |           2 |             2 | ✓     |          2

Implementation accuracy: 0.3325 (133/400 matches)

=== PREDICTION PROBABILITIES COMPARISON (First 5 samples) ===
Sample 1:
  Keras:        [0.2958, 0.0519, 0.6523]
  From Scratch: [0.4124, 0.3307, 0.2568]
  Max diff:     0.395507
Sample 2:
  Keras:        [0.3008, 0.6346, 0.0646]
  From Scratch: [0.3783, 0.3826, 0.2391]
  Max diff:     0.252042
Sample 3:
  Keras:        [0.3602, 0.3799, 0.2599]
  From Scratch: [0.4788, 0.2559, 0.2652]
  Max diff:     0.123950
Sample 4:
  Keras:        [0.1373, 0.0343, 0.8284]
  From Scratch: [0.3546, 0.3368, 0.3086]
  Max diff:     0.519822
Sample 5:
  Keras:        [0.3668, 0.5466, 0.0866]
  From Scratch: [0.4493, 0.3036, 0.2472]
  Max diff:     0.243040
```

2.2.3. Long Short Term Memory

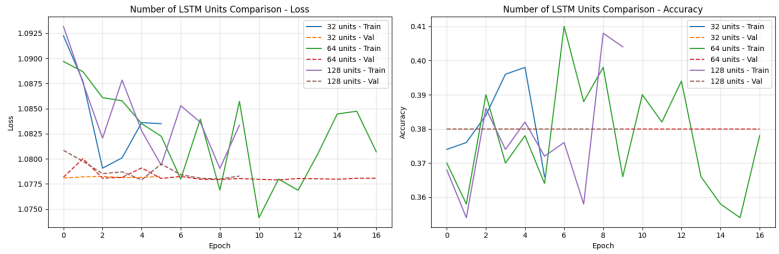
Grafik untuk akurasi dan loss dari perbedaan **jumlah layer LSTM**



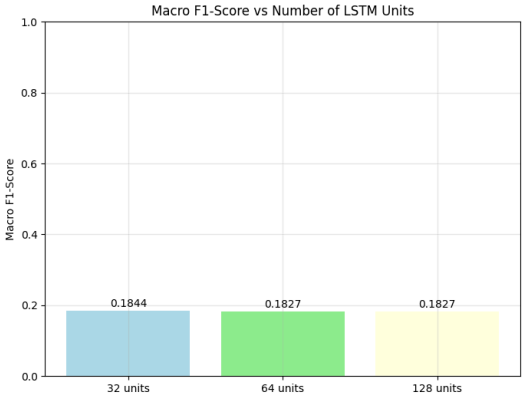
Grafik nilai macro f1 dari perbedaan **jumlah layer LSTM**



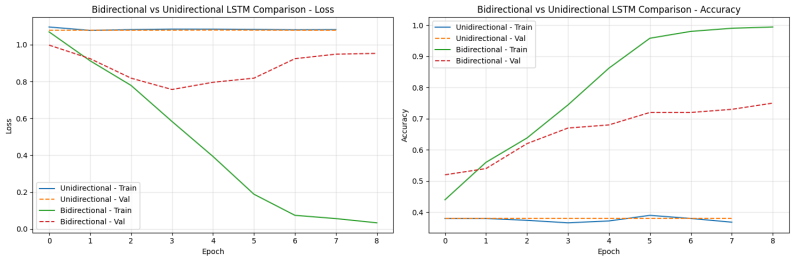
Grafik untuk akurasi dan loss dari perbedaan **banyak sel Layer LSTM**



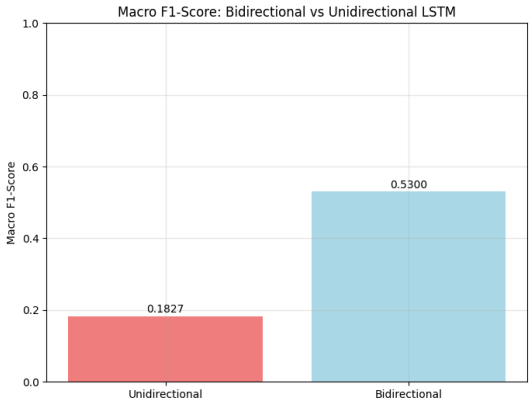
Grafik nilai macro f1 dari perbedaan **banyak sel Layer LSTM**



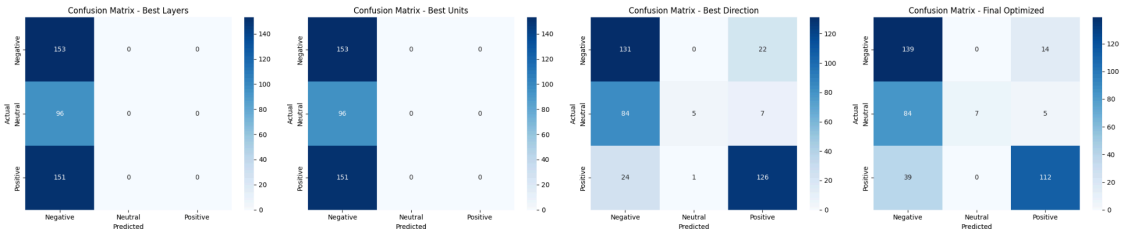
Grafik untuk akurasi dan loss dari perbedaan **jenis arah LSTM**



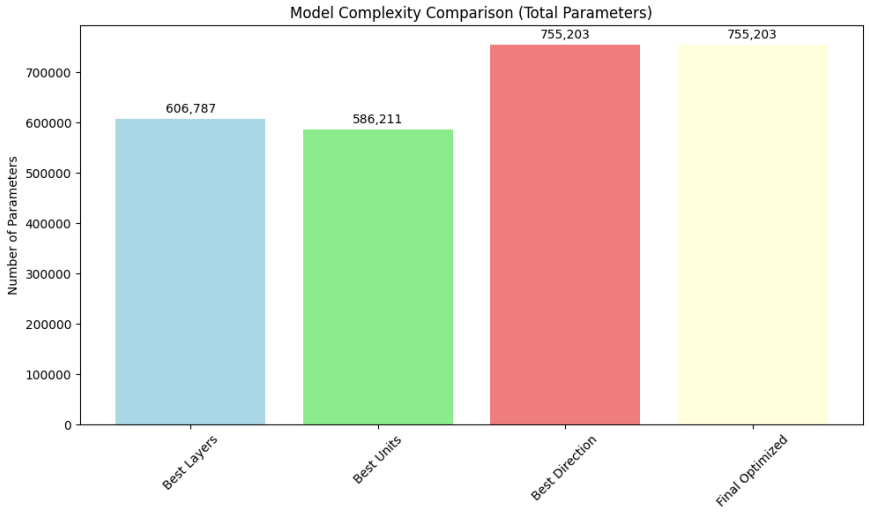
Grafik nilai macro f1 dari perbedaan **jenis arah LSTM**



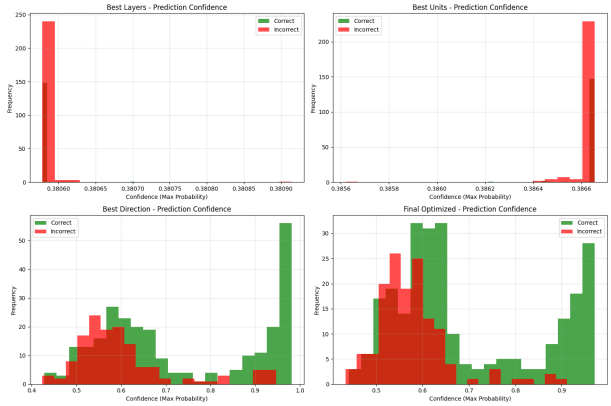
Confusion matrix model final



Jumlah parameter model terbaik setiap variasi fitur



Presisi dari setiap model terbaik


Best Layers - Prediction Confidence
Correct
Incorrect
Frequency
Confidence (Max Probability)
Best Units - Prediction Confidence
Correct
Incorrect
Frequency
Confidence (Max Probability)
Best Direction - Prediction Confidence
Correct
Incorrect
Frequency
Confidence (Max Probability)
Final Optimized - Prediction Confidence
Correct
Incorrect
Frequency
Confidence (Max Probability)

Bab 3 Kesimpulan dan Saran

3.1. Kesimpulan

Untuk CNN, didapat beberapa hal yang bisa ditarik. MaxPooling dan AveragePooling sulit untuk dibandingkan karena dalam beberapa kasus tertentu memiliki keunggulannya masing-masing. Kemudian semakin banyak jumlah filter, bisa mengubah akurasi dari hasil. Perubahan terhadap banyaknya layer konvolusi bisa menghasilkan akurasi yang berbeda pula. Untuk ukuran dari kernel, seperti beberapa penelitian lain, ditemukan kernel berukuran 3x3 menjadi yang terbaik. Maka didapat agar semakin akurat, maka diperlukan penyesuaian jumlah filter, jumlah layer konvolusi, dan ukuran kernel 3x3. Serta karena efek yang tidak terlalu signifikan, gunakan MaxPooling karena algoritma tersebut akan lebih mudah untuk dijalankan.

Untuk RNN, model dengan jumlah layer RNN yang lebih banyak menghasilkan model dengan akurasi yang lebih tinggi. Jumlah sel mungkin menghasilkan model yang lebih akurat tetapi tidak selalu karena akurasi 64 sel kurang dibanding 32 sel. Bidirectional RNN memberikan performa yang lebih baik berkat kemampuan melihat konteks masa depan. Informasi dari kedua arah memberikan representasi yang lebih banyak

Mengenai pengaruh jumlah sel atau hidden units, ditemukan bahwa hubungan antara jumlah sel dan akurasi tidak selalu linear. Ditemukan bahwa akurasi dengan 64 sel justru lebih rendah dibandingkan dengan 32 sel. Hal ini menunjukkan bahwa peningkatan kompleksitas model tidak selalu menghasilkan performa yang lebih baik, dan diperlukan penyesuaian yang tepat untuk menghindari overfitting atau underfitting. Fenomena ini mengindikasikan bahwa terdapat titik optimal dalam pemilihan ukuran hidden state yang perlu disesuaikan dengan kompleksitas dataset.

Keunggulan bidirectional RNN terlihat jelas dalam hasil pengujian. Bidirectional RNN memberikan performa yang lebih baik dibandingkan unidirectional RNN karena kemampuannya untuk melihat konteks masa depan dan masa lalu secara bersamaan memberikan keuntungan signifikan. Informasi dari kedua arah (forward dan backward) menghasilkan representasi yang lebih kaya dan komprehensif, memungkinkan model untuk membuat prediksi yang lebih akurat dengan mempertimbangkan konteks lengkap dari urutan data.

Namun, terdapat tantangan signifikan dalam implementasi from scratch versus library. Terdapat perbedaan akurasi yang signifikan antara implementasi library Keras dan implementasi from scratch, dengan akurasi implementasi from scratch hanya mencapai 33.25% (133/400 matches). Perbedaan probabilitas prediksi cukup besar, dengan max difference mencapai 0.52 pada beberapa sampel. Hal ini mengindikasikan adanya perbedaan dalam pengolahan data dan komputasi antara kedua implementasi yang memerlukan penelusuran lebih lanjut atau perbaikan.

Sekali lagi Keunggulan bidirectional terlihat jelas dalam hasil pengujian model LSTM. Bidirectional memberikan performa yang lebih baik dibandingkan unidirectional. Dalam LSTM khususnya diperlukan model yang dapat menimbang masa depan dan tidak menimbulkan vanishing/exploding problem pada simple RNN. Jumlah layer dan jumlah unit dalam layer pun tidak terlalu berpengaruh. Sehingga model yang terbaik secara signifikan bisa menggunakan bidirectional.

3.2. Saran

Untuk CNN, terdapat perbedaan yang jauh antara kalkulasi library dengan yang dibuat sendiri. Ini mungkin karena perbedaan weight ketika melakukan transpose. Weight bisa jadi tidak terinisialisasi sesuai dengan yang dihasilkan library sehingga akurasi sangatlah buruk dibandingkan hasil library.

Untuk RNN, model yang dibentuk perlu memiliki jumlah layer yang cukup dengan arah bidirectional untuk menghasilkan nilai akurasi yang terbaik dalam dataset yang digunakan. Akurasi yang lebih buruk mungkin karena cara pengolahan data oleh tensor terhadap bobot berbeda dengan pembuatan scratch.

Untuk LSTM, pengaruh arah LSTM menjadi bidirectional sangat berpengaruh ke akurasi. Tetapi tidak seperti RNN, jumlah layer LSTM tidak terlalu mempengaruhi kemampuan prediksi dari model yang dibuat. Begitu juga dengan jumlah sel dalam layer LSTM.

Bab 4 Pembagian Tugas

NIM	Nama	Tugas
13522123	Jimly Nur Arif	RNN, LSTM
13522133	Yosef Rafael Joshua	-
13522160	Rayhan Ridhar Rahman	CNN

Referensi

- https://d2l.ai/chapter_convolutional-neural-networks/index.html
- https://d2l.ai/chapter_recurrent-neural-networks/index.html
- https://d2l.ai/chapter_recurrent-modern/lstm.html
- https://www.tensorflow.org/datasets/catalog/cifar10