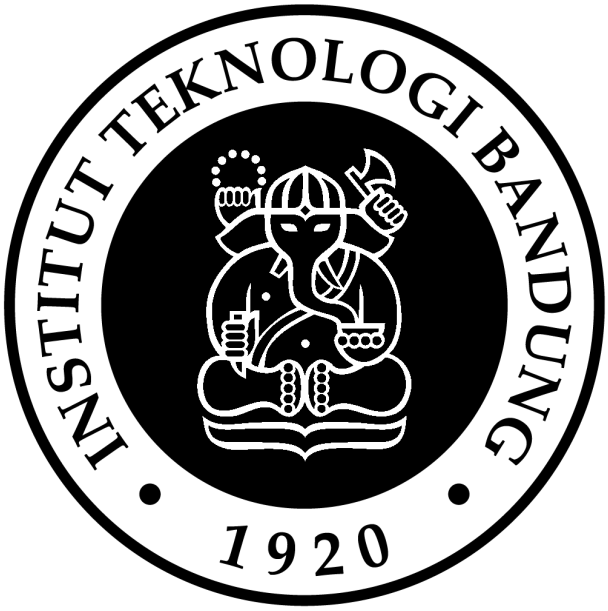
Laporan Tugas Besar 2 IF3270 Pembelajaran Mesin
Feedforward Neural Network

Semester II Tahun 2024/2025



Disusun Oleh :

Maulana Muhamad Susetyo	13522127
Andi Marihot Sitorus	13522138
Muhammad Rasheed Qais Tandjung	13522158

INSTITUT TEKNOLOGI BANDUNG

2024

Bab I
Deksripsi Persoalan

Persoalan ini adalah untuk membangun implementasi forward propagation dari nol untuk model Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), dan Long Short-Term Memory (LSTM). Implementasi from-scratch ini harus mampu memuat dan menggunakan bobot (nilai weights dan biases) yang telah dihasilkan dari proses pelatihan model Keras sebelumnya. Tujuannya adalah untuk mendemonstrasikan pemahaman mendalam tentang operasi internal setiap jenis *layer* dan memverifikasi konsistensi hasil prediksi antara model Keras yang dilatih dan implementasi manual menggunakan bobot yang sama.

Bab II
Penjelasan Implementasi

A. CNN

Berikut adalah model *forward propagation* CNN dari *library* keras dan yang akan digunakan sebagai dasar implementasi model *forward propagation from scratch*. Model ini akan dilatih dengan database CIFAR-10 dan *weight* hasil dari training akan disimpan ke dalam file .h5. Weight yang didapatkan akan digunakan sebagai parameter dari implementasi *forward propagation from scratch*.

```
modelKeras = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu',
padding='same', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),

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

    layers.Flatten(),

    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

a. Fungsi load_weights_from_keras

```
def load_weights_from_keras(file_path):
    weights = {}
    with h5py.File(file_path, 'r') as f:
        layers_group = f['layers']
        for layer_name in layers_group:
            layer_group = layers_group[layer_name]
            if 'vars' in layer_group:
                vars_group = layer_group['vars']
                if '0' in vars_group and '1' in
vars_group:
                    weights[f"{layer_name}/kernel:0"]
= vars_group['0'][()]
                    weights[f"{layer_name}/bias:0"] =
vars_group['1'][()]
    return weights
```

Fungsi ini digunakan untuk mengekstrak weight dari file .h5 hasil penyimpanan prosedur save_weight dari library keras.

b. Fungsi conv2d

```
def conv2d(x, weight, bias, stride=1,
padding='same'):
    kh, kw, in_channels, out_channels = weight.shape
    h, w, _ = x.shape

    if padding == 'same':
        pad_h = max((np.ceil(h / stride) - 1) *
stride + kh - h, 0)
        pad_w = max((np.ceil(w / stride) - 1) *
stride + kw - w, 0)
        pad_top = int(pad_h // 2)
        pad_bottom = int(pad_h - pad_top)
        pad_left = int(pad_w // 2)
        pad_right = int(pad_w - pad_left)
        x = np.pad(x, ((pad_top, pad_bottom),
(pad_left, pad_right), (0, 0)), mode='constant')

        h, w, _ = x.shape

    out_h = (h - kh) // stride + 1
    out_w = (w - kw) // stride + 1

    if out_h <= 0 or out_w <= 0:
        raise ValueError(f"Output shape invalid:
({out_h}, {out_w}). Input: {x.shape}, kernel: ({kh},
{kw})")

    out = np.zeros((out_h, out_w, out_channels))

    for oc in range(out_channels):
        for i in range(out_h):
            for j in range(out_w):
                patch = x[i*stride:i*stride+kh,
j*stride:j*stride+kw, :]
                out[i, j, oc] = np.sum(patch *
weight[:, :, :, oc]) + bias[oc]

    return np.maximum(0, out)
```

Fungsi ini melakukan konvolusi 2D pada input citra x menggunakan bobot weight dan bias.

Langkah-langkah:

i. Padding (jika 'same'): menghitung dan menambahkan padding di sekitar input agar ukuran output tetap proporsional.
ii. Output shape: menghitung ukuran output (out_h, out_w) berdasarkan padding dan stride.
iii. Looping per patch: untuk setiap posisi (i,j) dan channel output oc, potong patch dari input dan lakukan: patch * filter + bias
iv. hasil akhir aktivasi menggunakan fungsi aktivasi ReLU

c. Fungsi max_pool2d

```
def max_pool2d(x, size=2, stride=2):
    h, w, c = x.shape
    out_h = (h - size) // stride + 1
    out_w = (w - size) // stride + 1
    out = np.zeros((out_h, out_w, c))

    for ch in range(c):
        for i in range(out_h):
            for j in range(out_w):
                patch = x[i*stride:i*stride+size,
j*stride:j*stride+size, ch]
                out[i, j, ch] = np.max(patch)
    return out
```

Fungsi ini melakukan max pooling 2D.
Langkah-langkah:

i. Mengiterasi patch per window (size x size) per channel.
ii. Menyimpan nilai maksimum dari setiap patch ke output.
iii. Hasil akhir adalah matriks yang lebih kecil dengan fitur dominan per wilayah.

d. Fungsi flatten

```
def flatten(x):
    return x.flatten()
```

Mengubah x menjadi vektor 1 dimensi.

e. Fungsi dense

```
def dense(x, weight, bias, activation='relu'):
    z = np.dot(x, weight) + bias
    if activation == 'relu':
```

```
        return np.maximum(0, z)
    elif activation == 'softmax':
        exp_z = np.exp(z - np.max(z))
        return exp_z / np.sum(exp_z)
    return z
```

Fungsi ini mengalikan fitur x dengan weight lalu mengaktivasi nilai x dengan fungsi aktivasi yang dipilih.

f. Fungsi forward_from_scratch

```
def forward_from_scratch(x, weights_dict):
    x = conv2d(x, weights_dict['conv2d/kernel:0'],
weights_dict['conv2d/bias:0'], padding='same')

    x = max_pool2d(x)

    x = conv2d(x, weights_dict['conv2d_1/kernel:0'],
weights_dict['conv2d_1/bias:0'], padding='same')

    x = max_pool2d(x)

    x = flatten(x)

    x = dense(x, weights_dict['dense/kernel:0'],
weights_dict['dense/bias:0'], activation='relu')

    x = dense(x, weights_dict['dense_1/kernel:0'],
weights_dict['dense_1/bias:0'], activation='softmax')

    return x
```

Fungsi ini adalah implementasi dari model library keras yang ditunjukkan di awal.

g. Perbandingan Hasil Implementasi From Scratch Dengan Model Library Keras

```
y_true = y_test.flatten()
y_pred_scratch = []

for i in range(len(x_test)):
    prediction = forward_from_scratch(x_test[i], weights)
    y_pred_scratch.append(np.argmax(prediction))

f1 = f1_score(y_true, y_pred_scratch, average='macro')
print(f"Macro F1-score (from scratch): {f1:.4f}")

keras_preds = modelKeras.predict(x_test, verbose=0)
keras_labels = np.argmax(keras_preds, axis=1)
f1_keras = f1_score(y_true, keras_labels, average='macro')
print(f"Macro F1-score (Keras): {f1_keras:.4f}")
```

```
Macro F1-score (from scratch): 0.6995
Macro F1-score (Keras): 0.6995
```

Hasil dari implementasi forward propagation CNN from scratch memberikan hasil prediksi yang sama dengan forward propagation dari library keras. Hal ini bisa terjadi karena weight yang digunakan untuk implementasi forward propagation berasal dari hasil latih model library keras.

B. RNN

a. Fungs Membuat Model RNN Keras

```
def create_rnn_model(rnn_layers, rnn_cells,
rnn_type):
     model = models.Sequential()

model.add(layers.Input(shape=(sequence_length,)))
     model.add(layers.Embedding(vocab_size, 128))

     for i in range(rnn_layers):
     if rnn_type == 'unidirectional':
          return_sequences = True if i < rnn_layers -
1 else False
          model.add(layers.SimpleRNN(rnn_cells[i],
return_sequences=return_sequences))
     elif rnn_type == 'bidirectional':
          return_sequences = True if i < rnn_layers -
1 else False
```

```
model.add(layers.Bidirectional(layers.SimpleRNN(rnn_c
ells[i], return_sequences=return_sequences)))

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

     model.compile(optimizer='adam',

loss='sparse_categorical_crossentropy',
                   metrics=['accuracy'])
     return model
```

Fungsi untuk membuat model RNN keras dengan input berupa jumlah RNN layer, banyak cell per layer, dan tipe RNN (unidirectional/bidirectional)

b. Fungsi Training dan Evaluating Model Keras

```
def train_and_evaluate(model_name, rnn_layers,
rnn_cells, rnn_type):
     print(f"\n--- Training Model: {model_name} ---")
     model = create_rnn_model(rnn_layers, rnn_cells,
rnn_type)
     history = model.fit(X_train_vec, y_train,
                         epochs=15,

validation_data=(X_valid_vec, y_valid),
                         verbose=1)

     model_filename = f'models/model_{model_name}.h5'
     model.save(model_filename)
     print(f"Model saved to {model_filename}")

     y_pred_keras =
np.argmax(model.predict(X_test_vec), axis=1)
     f1_keras = f1_score(y_test, y_pred_keras,
average='macro')
     print(f"Keras Model Test Macro F1-score:
{f1_keras:.4f}")

     plt.figure(figsize=(10, 4))
     plt.subplot(1, 2, 1)
     plt.plot(history.history['loss'],
label='Training Loss')
```

```
    plt.plot(history.history['val_loss'],
label='Validation Loss')
    plt.title(f'{model_name} - Loss per Epoch')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history['accuracy'],
label='Training Accuracy')
    plt.plot(history.history['val_accuracy'],
label='Validation Accuracy')
    plt.title(f'{model_name} - Accuracy per Epoch')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.tight_layout()
    plt.show()

    return model, f1_keras, history, model_filename
```

Fungsi untuk melakukan *training* dan evaluasi *loss* dari model Keras.

c. Implementasi Embedding Layer

```
class EmbeddingLayerScratch:
    def __init__(self, weights):
    self.weights = weights

    def forward(self, inputs):
    return self.weights[inputs]
```

Kelas untuk melakukan *embedding* pada RNN. Kelas ini cukup mengambil *weights* yang sudah didapatkan dari hasil *training*.

d. Implementasi SimpleRNN

```
class SimpleRNNLayerScratch:
    def __init__(self, weights,
return_sequences=False):
    self.W = weights[0]
    self.U = weights[1]
    self.b = weights[2]
    self.return_sequences = return_sequences

    def forward(self, inputs):
```

```
     if inputs.ndim != 3:
          raise ValueError(f"SimpleRNNLayerScratch
expected 3D input, got {inputs.ndim}D with shape
{inputs.shape}")

     batch_size, sequence_length, input_dim =
inputs.shape
     hidden_dim = self.W.shape[1]

     h_t = np.zeros((batch_size, hidden_dim))

     all_hidden_states = []

     for t in range(sequence_length):
          x_t = inputs[:, t, :]
          h_t = np.tanh(np.dot(x_t, self.W) +
np.dot(h_t, self.U) + self.b)
          if self.return_sequences:
               all_hidden_states.append(h_t)

     if self.return_sequences:
          return
np.array(all_hidden_states).transpose(1, 0, 2)
     else:
          return h_t
```

Kelas untuk melakukan iterasi RNN layer. Akan dilakukan iterasi sebanyak *timestep*, dan hasil masing-masing akan dimasukkan ke fungsi tanh.

e. Implementasi Bidirectional RNN Layer

```
class BidirectionalSimpleRNNLayerScratch:
     def __init__(self, forward_weights,
backward_weights, return_sequences=False):
     self.forward_rnn =
SimpleRNNLayerScratch(forward_weights,
return_sequences=return_sequences)
     self.backward_rnn =
SimpleRNNLayerScratch(backward_weights,
return_sequences=return_sequences)
     self.return_sequences = return_sequences

     def forward(self, inputs):
     forward_output =
self.forward_rnn.forward(inputs)
```

```
     reversed_inputs = inputs[:, ::-1, :]
     backward_output =
self.backward_rnn.forward(reversed_inputs)

     if self.return_sequences:
          return np.concatenate([forward_output,
backward_output], axis=-1)
     else:
          return np.concatenate([forward_output,
backward_output], axis=-1)
```

Akan membuat dua buah RNN layer, salah satu adalah versi terbalik dari yang lainnya.

f. Implementasi Dropout Layer

```
class DropoutLayerScratch:
     def __init__(self, rate):
     self.rate = rate

     def forward(self, inputs):
     return inputs
```

Implementasi dari dropout layer. Tidak melakukan apa-apa karena dropout hanya dilakukan saat training.

g. Implementasi Dense Layer

```
class DenseLayerScratch:
     def __init__(self, weights):
     self.W = weights[0]
     self.b = weights[1]

     def forward(self, inputs):
     return np.dot(inputs, self.W) + self.b

class SoftmaxActivationScratch:
     def forward(self, inputs):
     exp_inputs = np.exp(inputs - np.max(inputs,
axis=-1, keepdims=True))
     return exp_inputs / np.sum(exp_inputs, axis=-1,
keepdims=True)
```

Dense layer untuk melakukan propagation seperti FFNN biasa. Untuk layer terakhir akan menggunakan layer dense bertipe softmax.

h. Implementasi RNN From Scratch

```
class FromScratchRNNModel:
     def __init__(self, keras_model):
     self.layers = []

     for i, layer in enumerate(keras_model.layers):
          layer_keras_weights = layer.get_weights()

          if isinstance(layer, layers.Embedding):

self.layers.append(EmbeddingLayerScratch(layer_keras_
weights[0]))
          elif isinstance(layer, layers.SimpleRNN):
               return_sequences_keras =
layer.return_sequences

self.layers.append(SimpleRNNLayerScratch(layer_keras_
weights, return_sequences=return_sequences_keras))
          elif isinstance(layer,
layers.Bidirectional):
               forward_weights =
layer_keras_weights[:3]
               backward_weights =
layer_keras_weights[3:]
               return_sequences_keras =
layer.forward_layer.return_sequences

self.layers.append(BidirectionalSimpleRNNLayerScratch
(forward_weights, backward_weights,
return_sequences=return_sequences_keras))
          elif isinstance(layer, layers.Dropout):

self.layers.append(DropoutLayerScratch(layer.rate))
          elif isinstance(layer, layers.Dense):

self.layers.append(DenseLayerScratch(layer_keras_weig
hts))

     self.softmax = SoftmaxActivationScratch()

     def predict(self, inputs):
     x = inputs
     for i, layer in enumerate(self.layers):
          x = layer.forward(x)
     return self.softmax.forward(x)
```

Kelas yang akan menggabungkan seluruh layer sesuai arsitektur.

C. LSTM

a. Text Vectorization

```
vocab_size = 10000
sequence_length = 100

vectorizer = TextVectorization(
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length
)
vectorizer.adapt(X_train)

X_train_vec = vectorizer(X_train)
X_valid_vec = vectorizer(X_valid)
X_test_vec = vectorizer(X_test)
print(f'train {X_train_vec.shape}, valid {X_valid_vec.shape},
test {X_test_vec.shape}')
```

Di atas adalah tahap *text vectorization*, mengubah teks menjadi *sequence of integer* berukuran 100 untuk tahap selanjutnya sesuai dengan *vocabulary* yang dibangkitkan.

b. Fungsi build_lstm_model

```
def build_lstm_model(num_lstm_layers=1, lstm_units=64,
bidirectional=False, dropout_rate=0.5, vocab_size=10000,
embedding_dim=128, sequence_length=100, num_classes=3):
    model = Sequential()
    model.add(Embedding(input_dim=vocab_size,
output_dim=embedding_dim, input_length=sequence_length,
mask_zero=True))
    for i in range(num_lstm_layers):
        return_sequences = (i < num_lstm_layers - 1)
        lstm_layer_instance = LSTM(lstm_units,
return_sequences=return_sequences)
        if bidirectional:
            model.add(Bidirectional(lstm_layer_instance))
        else:
```

```
            model.add(lstm_layer_instance)
    model.add(Dropout(dropout_rate))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=Adam(),
        metrics=['accuracy']
    )
    return model
```

Di atas adalah fungsi untuk membangun model menggunakan *library* keras. Terdiri dari layer Embedding, LSTM (Unidirectional/Bidirectional), Dropout, dan Dense.

c. Fungsi plot_multi_loss

```
def plot_multi_loss(histories, labels, title='Loss Curve'):
    plt.figure(figsize=(10, 6))
    prop_cycler = plt.rcParams['axes.prop_cycle']
    colors = prop_cycler.by_key()['color']

    for i, (h, label) in enumerate(zip(histories, labels)):
        color = colors[i % len(colors)]
        train_line, = plt.plot(h.history['loss'],
label=f'{label} train', color=color)
        plt.plot(h.history['val_loss'], linestyle='--',
label=f'{label} val', color=train_line.get_color(),
alpha=0.7)

    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Di atas adalah fungsi untuk menampilkan grafik *train loss* dan *validation loss* yang didapat dari training.

d. Fungsi train_lstm_model

```
def train_lstm_model(model, X_train_vec, y_train,
X_valid_vec, y_valid, X_test_vec, y_test, model_label):
    print(f'\nTraining {model_label}...')
    history = model.fit(
        Xtrain_vec, y_train,
        validation_data=(X_valid_vec, y_valid),
        epochs=10,
        batch_size=32,
        verbose=0
    )
    y_pred_proba = model.predict(X_test_vec)
    y_pred = np.argmax(y_pred_proba, axis=1)
    f1 = macro_f1(y_test, y_pred)
    print(f'Macro F1-score (test) for {model_label}:
{f1:.4f}')
    return history, f1
```

Di atas adalah fungsi untuk *training* model, dengan parameter data yang digunakan untuk melatih model, training dilakukan selama 10 epoch.

e. Fungsi-fungsi *Forward Propagation LSTM from Scratch*

```
def embedding_forward(x, embedding_matrix):
    return embedding_matrix[x]

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

def dropout_forward(x, rate):
    # Pass-Through
    return x

def dense_forward(x, W, b):
    return np.dot(x, W) + b

def softmax(x):
    e_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e_x / np.sum(e_x, axis=1, keepdims=True)
```

```
def lstm_forward(x, W, U, b, mask):
    batch_size, seq_len = x.shape
    units = U.shape[0]

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

    for t in range(seq_len):
        xt = x[:, t, :]
        current_token_mask = mask[:, t].reshape(-1, 1)

        ht_minus_1 = np.copy(h)
        ct_minus_1 = np.copy(c)

        z = np.dot(xt, W) + np.dot(ht_minus_1, U) + b

        i = sigmoid(z[:, :units])
        f = sigmoid(z[:, units:2*units])
        a = np.tanh(z[:, 2*units:3*units])
        o = sigmoid(z[:, 3*units:])

        state = f * ct_minus_1 + i * a
        out = o * np.tanh(state)

        c = np.where(current_token_mask, state, ct_minus_1)
        h = np.where(current_token_mask, out, ht_minus_1)

    return h
```

Berikut adalah penjelasan fungsi-fungsi LSTM *from scratch*:

- embedding_forward: mengambil vektor untuk indeks x
- dropout_forward: tidak melakukan apa-apa karena hanya forward
- dense_forward: melakukan forward prop layaknya FFNN
- lstm_forward: fungsi perhitungan lstm, menggunakan rumus untuk perhitungan tiap gate, mask digunakan untuk mengabaikan *padding*

D. Penjelasan *Forward Propagation*

a. CNN

Algoritma forward propagation pada CNN bekerja dengan memproses data input berupa gambar di beberapa layer. Layer-layer CNN terdiri atas layer konvolusi, layer pooling, layer flatten, dan layer dense. Layer konvolusi akan mengekstrak feature map dari input menggunakan filter-filter yang digerakkan dengan stride tertentu. Layer pooling akan memperkecil ukuran feature map sesuai ukuran kernel pooling. Layer flatten akan mengubah feature map menjadi matriks 1 dimensi. Layer dense akan digunakan untuk menjalankan fungsi aktivasi agar feature map dapat digunakan untuk prediksi.

b. RNN

Algoritma forward propagation pada RNN bekerja dengan memproses data sekuensial langkah demi langkah. Untuk setiap langkah waktu, RNN menerima input saat ini dan juga hidden state (keadaan tersembunyi) dari langkah waktu sebelumnya, yang berfungsi sebagai "memori" dari informasi sebelumnya. Kedua input ini kemudian digabungkan (biasanya melalui penjumlahan terbobot) dan dilewatkan melalui fungsi aktivasi tanh untuk menghasilkan hidden state baru untuk langkah waktu saat ini dan juga output untuk langkah waktu tersebut. Proses ini berulang untuk setiap elemen dalam sekuens, memungkinkan RNN menangkap dependensi temporal dan menghasilkan prediksi berdasarkan seluruh konteks urutan input.

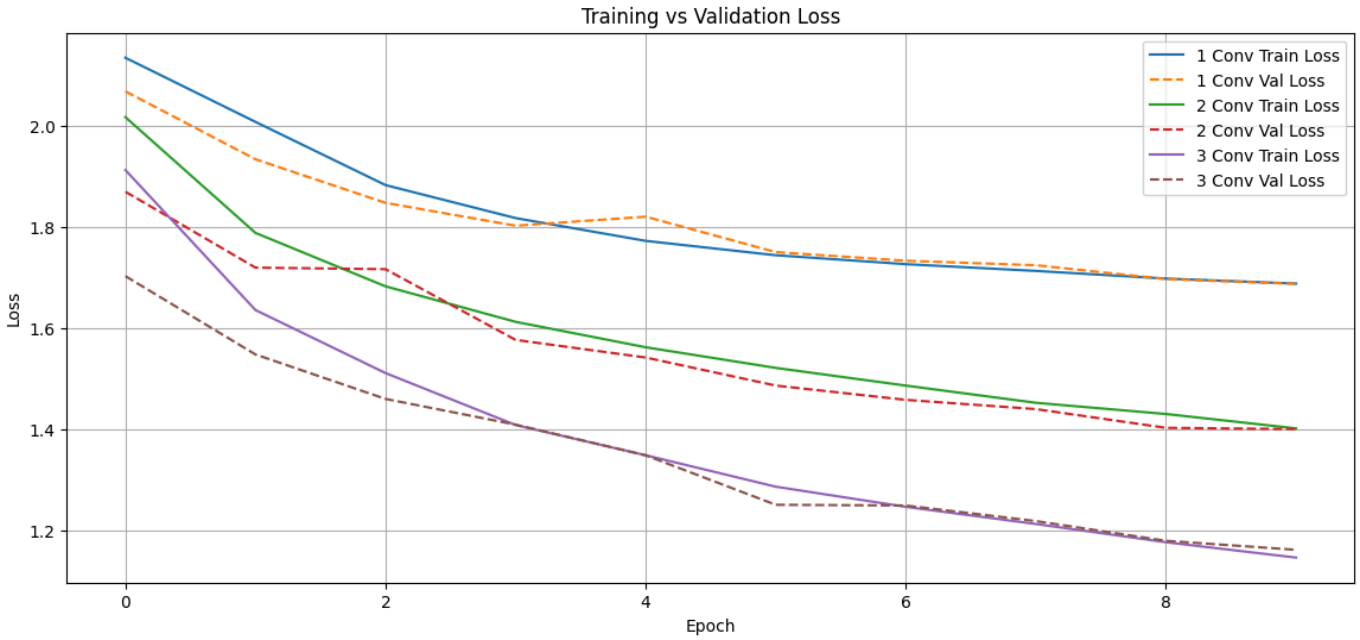
c. LSTM

Algoritma forward propagation mirip seperti RNN. Di setiap langkah, LSTM menerima input saat ini dan hidden state serta cell state dari sebelumnya. Cell state adalah memori utama, alirannya dikontrol oleh tiga gerbang: forget gate (menentukan apakah informasi disimpan atau dilupakan), input gate (menentukan informasi baru yang disimpan ke cell state), dan output gate (menentukan informasi dari cell state yang dikeluarkan sebagai hidden state baru). Gerbang-gerbang memperbarui cell state dan menghasilkan hidden state baru.

Bab III
Hasil Pengujian

A. *CNN*

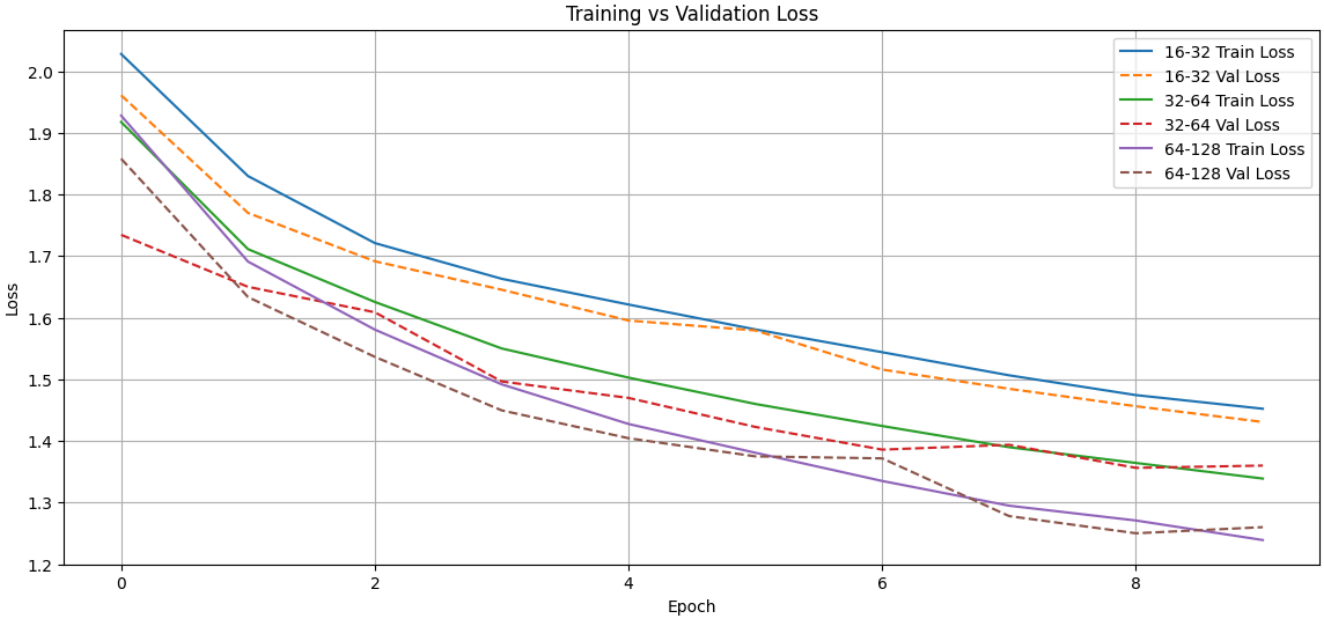
a. Pengaruh jumlah layer konvolusi



1 Conv Layer - Macro F1-score: 0.3587
2 Conv Layers - Macro F1-score: 0.4789
3 Conv Layers - Macro F1-score: 0.5821

Variasi jumlah layer konvolusi pada CNN menunjukkan bahwa 3 layer konvolusi memberikan hasil terbaik dari segi F1-score dan juga grafik train loss dan val loss. Dari pola yang terlihat, semakin banyak layer konvolusi, hasil prediksi dari model akan semakin baik. Namun nilai f1-score yang terlalu tinggi juga tidak baik karena akan terjadi overfit sehingga perlu dibuat batasan untuk banyaknya layer konvolusi yang digunakan.

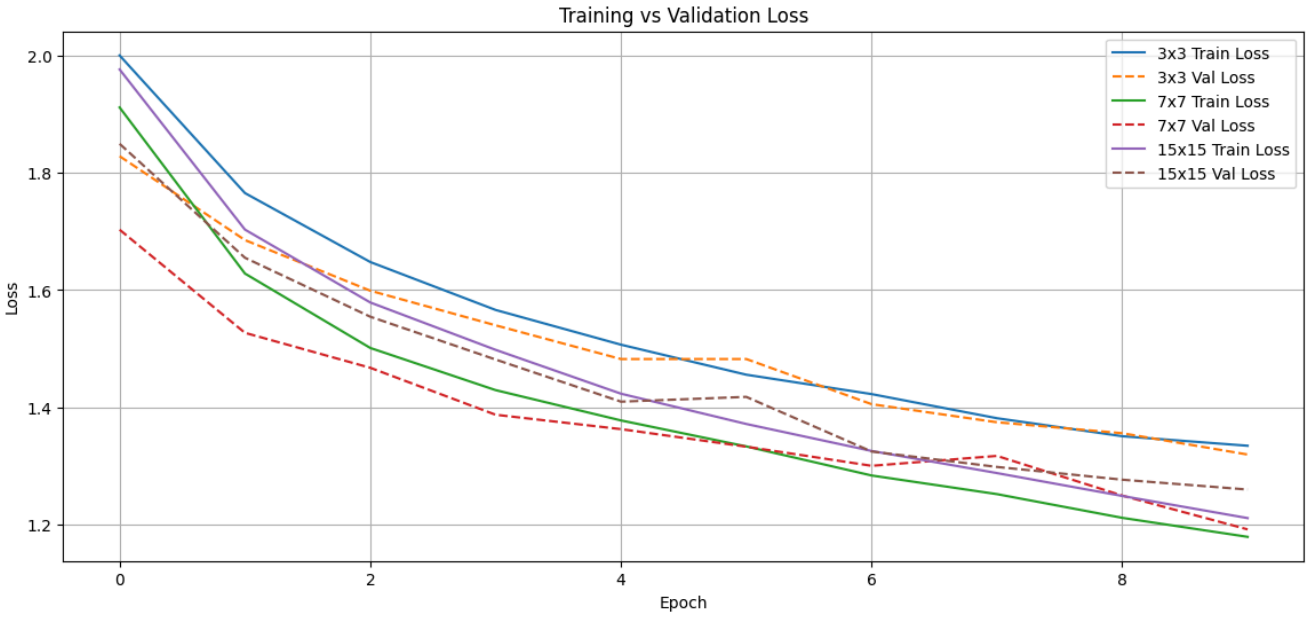
b. Pengaruh banyak filter per layer konvolusi



16-32 Filter - Macro F1-score: 0.4640
32-64 Filter - Macro F1-score: 0.4919
64-128 Filter - Macro F1-score: 0.5316

Variasi jumlah layer konvolusi pada CNN menunjukkan bahwa 64-128 filter per layer konvolusi memberikan hasil terbaik dari segi F1-score dan juga grafik train loss dan val loss.. Dari pola yang terlihat, semakin banyak layer konvolusi, hasil prediksi dari model akan semakin baik.

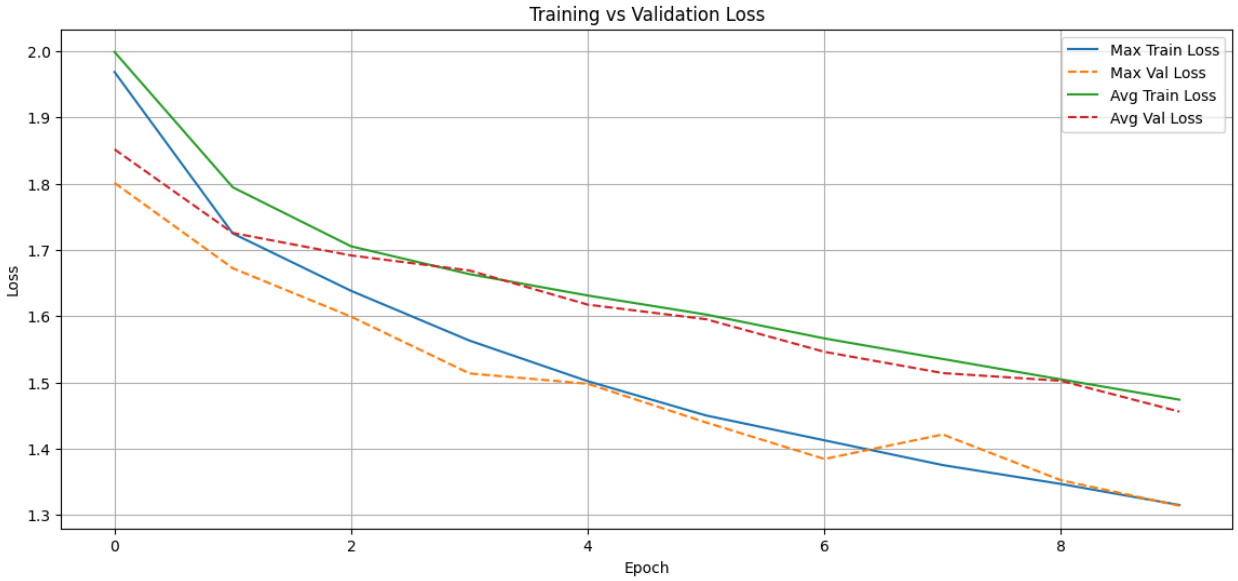
c. Pengaruh ukuran filter per layer konvolusi



3x3 Kernel - Macro F1-score: 0.4993
7x7 Kernel - Macro F1-score: 0.5677
15x15 Kernel - Macro F1-score: 0.5475

Variasi ukuran filter pada layer konvolusi pada CNN menunjukkan bahwa filter 7x7 memberikan hasil terbaik dari segi F1-score dan juga grafik train loss dan val loss. Dari pola yang terlihat, kualitas prediksi model akan meningkat seiring bertambahnya ukuran filter hingga mencapai titik terbaik lalu menurun. Kualitas model terhadap ukuran akan membentuk grafik berupa parabola.

d. Pengaruh jenis pooling layer



Max Pooling - Macro F1-score: 0.5098
Avg Pooling - Macro F1-score: 0.4377

Variasi jenis pooling di layer pooling pada CNN menunjukkan bahwa max pooling memberikan hasil terbaik dari segi F1-score dan juga grafik train loss dan val loss.

e. Perbandingan implementasi from-scratch dengan model Keras

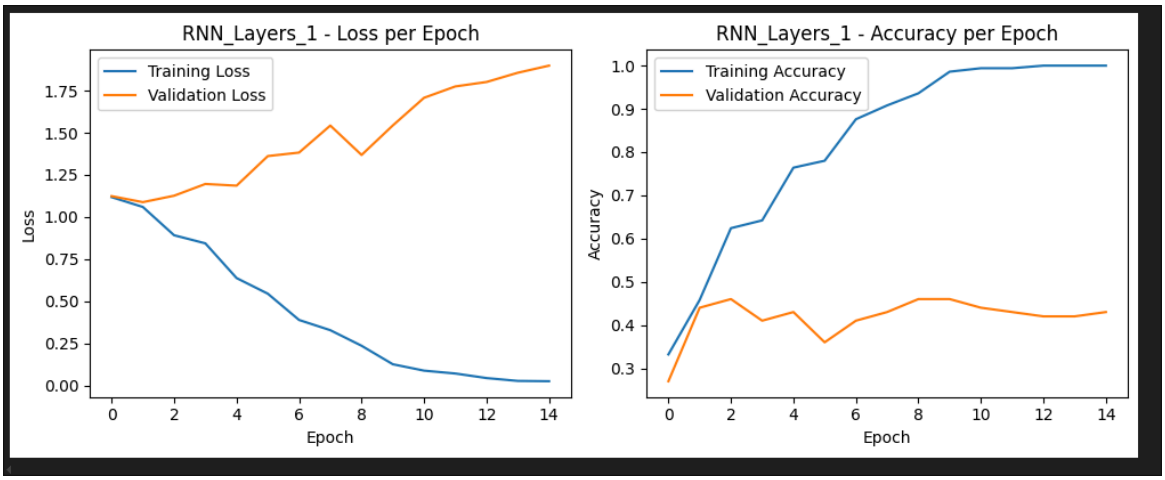
```
y_true = y_test.flatten()
y_pred_scratch = []

for i in range(len(x_test)):
    prediction = forward_from_scratch(x_test[i], weights)
    y_pred_scratch.append(np.argmax(prediction))

f1 = f1_score(y_true, y_pred_scratch, average='macro')
print(f"Macro F1-score (from scratch): {f1:.4f}")

keras_preds = modelKeras.predict(x_test, verbose=0)
keras_labels = np.argmax(keras_preds, axis=1)
f1_keras = f1_score(y_true, keras_labels, average='macro')
print(f"Macro F1-score (Keras): {f1_keras:.4f}")
```

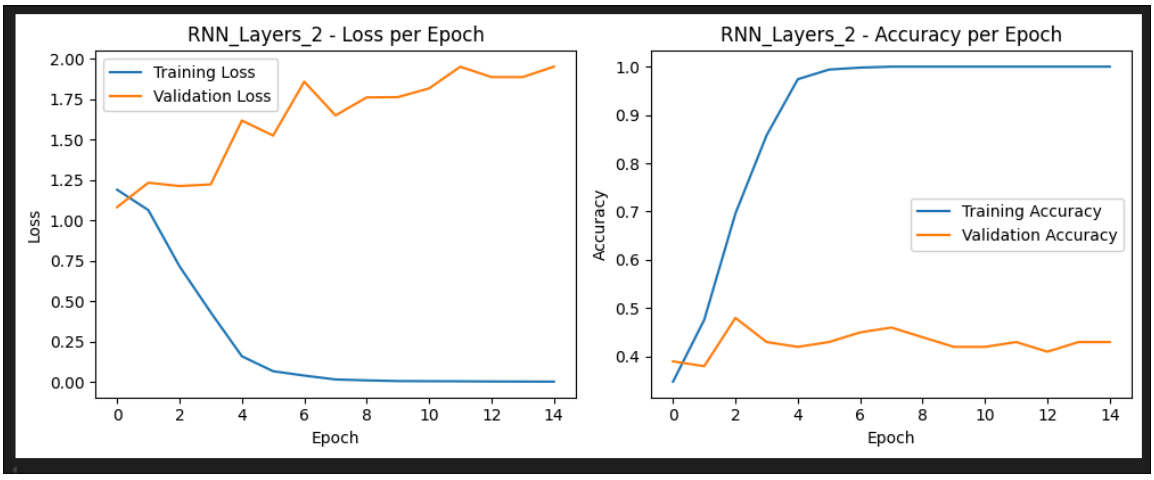
```
Macro F1-score (from scratch): 0.6995
Macro F1-score (Keras): 0.6995
```

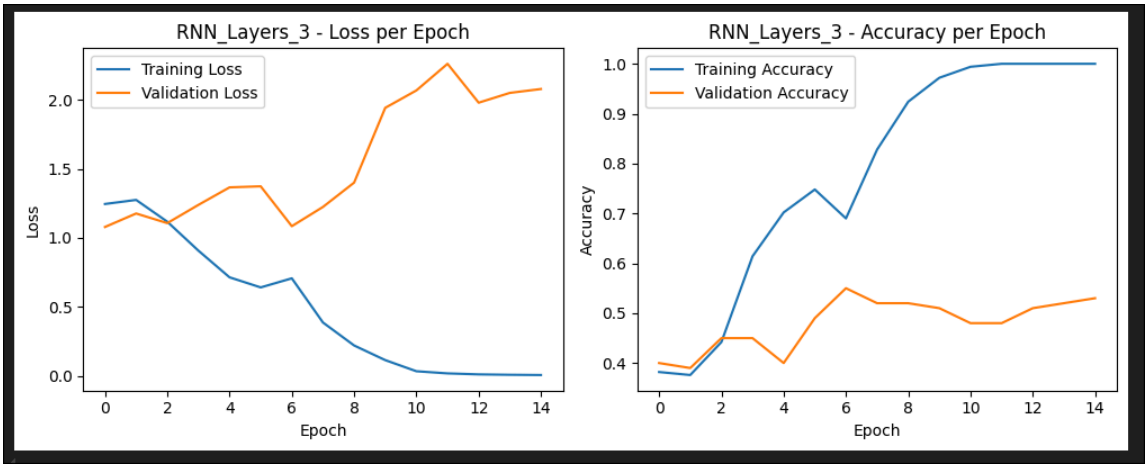
Hasil dari implementasi forward propagation CNN from scratch memberikan hasil prediksi yang sama dengan forward propagation dari library keras. Hal ini bisa terjadi karena weight yang digunakan untuk implementasi forward propagation berasal dari hasil latih model library keras.

B. RNN

a. Pengaruh jumlah layer



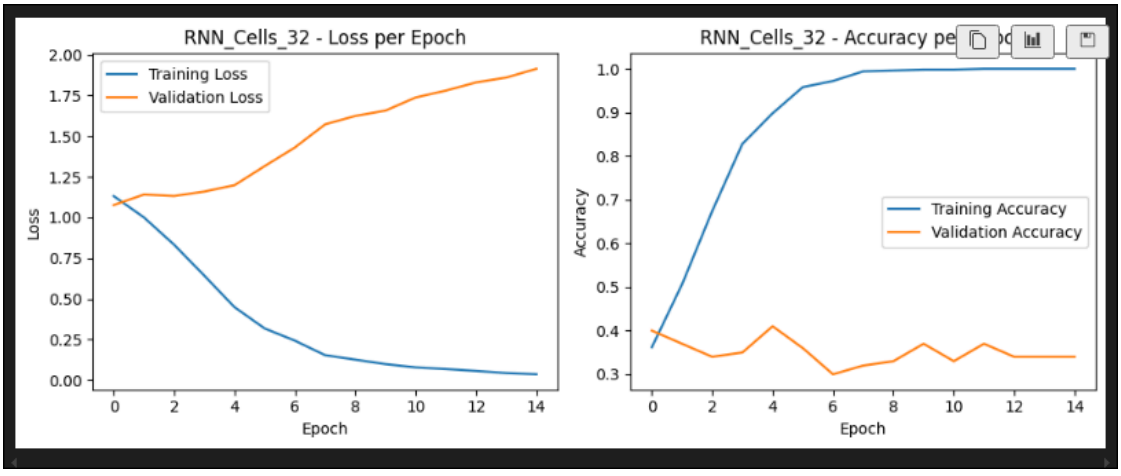


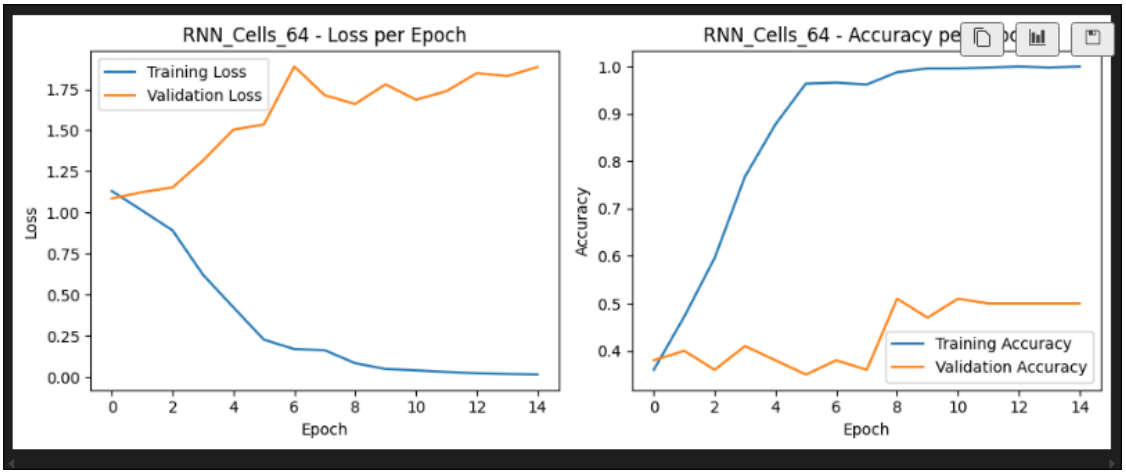


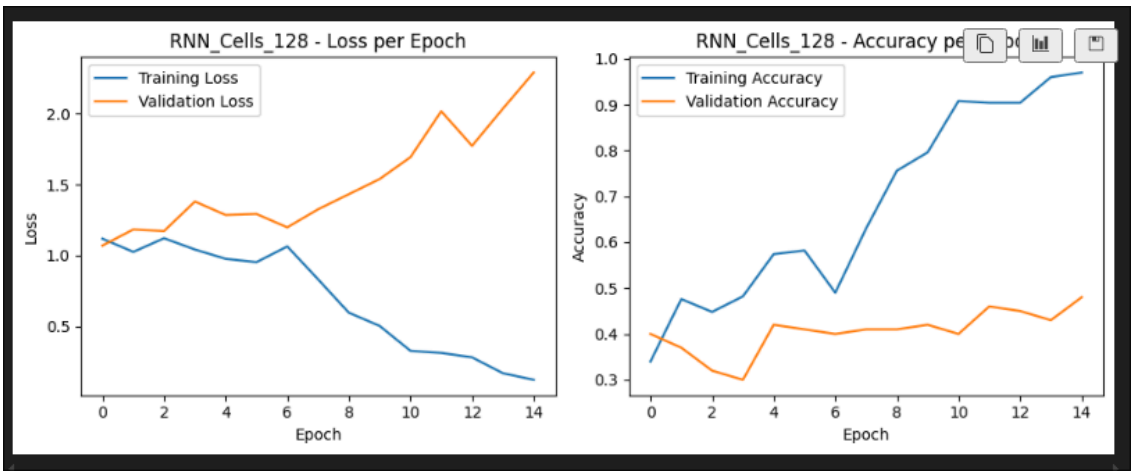
Comparison of F1-scores for different RNN layer counts:
1 RNN Layers: Macro F1-score = 0.3749
2 RNN Layers: Macro F1-score = 0.3942
3 RNN Layers: Macro F1-score = 0.5022

Dengan variasi jumlah layer, dapat dilihat bahwa semakin banyak layer yang digunakan, semakin besar loss yang didapatkan untuk model. Dapat diasumsikan bahwa menambah layer akan meningkatkan overfitting dari model.

b. Pengaruh banyak cell per layer





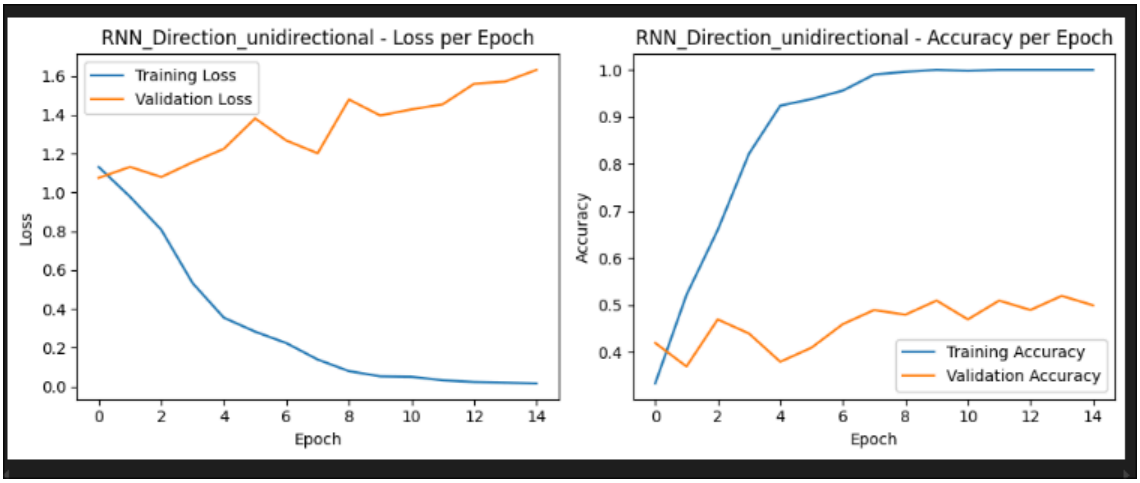


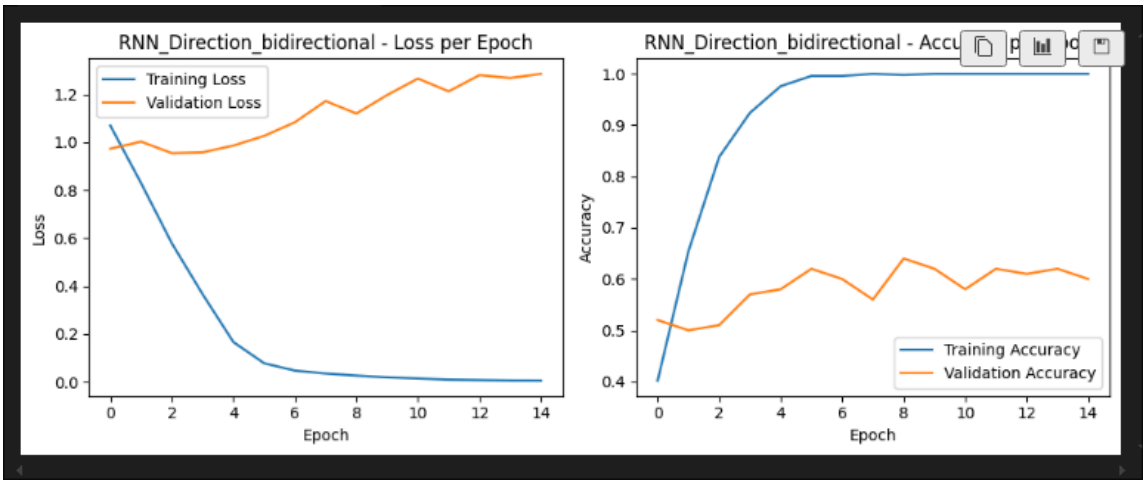
Comparison of F1-scores for different RNN cell counts per layer:
32 RNN Cells: Macro F1-score = 0.4094
64 RNN Cells: Macro F1-score = 0.4192
128 RNN Cells: Macro F1-score = 0.4014

Menggunakan variasi jumlah sel, terlihat training loss memiliki pola yang mirip, kecuali untuk 128 cell per layer yang memiliki loss lebih banyak di awal, mungkin karena banyaknya parameter membuat model cukup kebingungan terkait inferensi.

Untuk validation loss, terlihat pola yang sama untuk seluruh model. Dapat diasumsikan bahwa overfitting tidak terlalu dihilangkan dengan mengubah jumlah cell per layer.

c. Pengaruh jenis layer berdasarkan arah

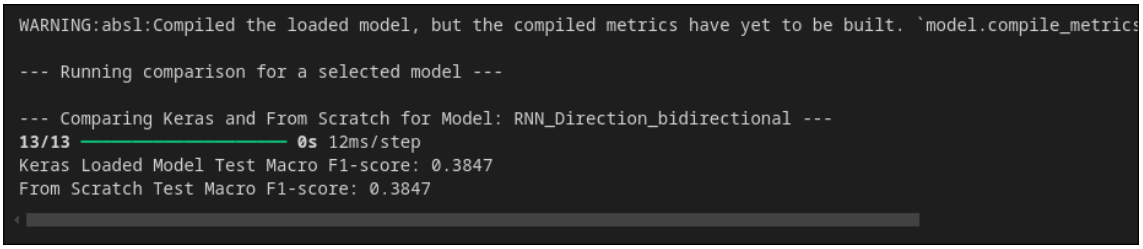




Keras Model Test Macro F1-score (unidirectional): 0.3847
Keras Model Test Macro F1-score (bidirectional) : 0.5015

Menggunakan variasi arah (unidirectional/bidirectional), terlihat pola training loss cukup sama, namun validation loss untuk bidirectional jauh lebih bagus, walaupun masih overfitting dan nilainya naik.

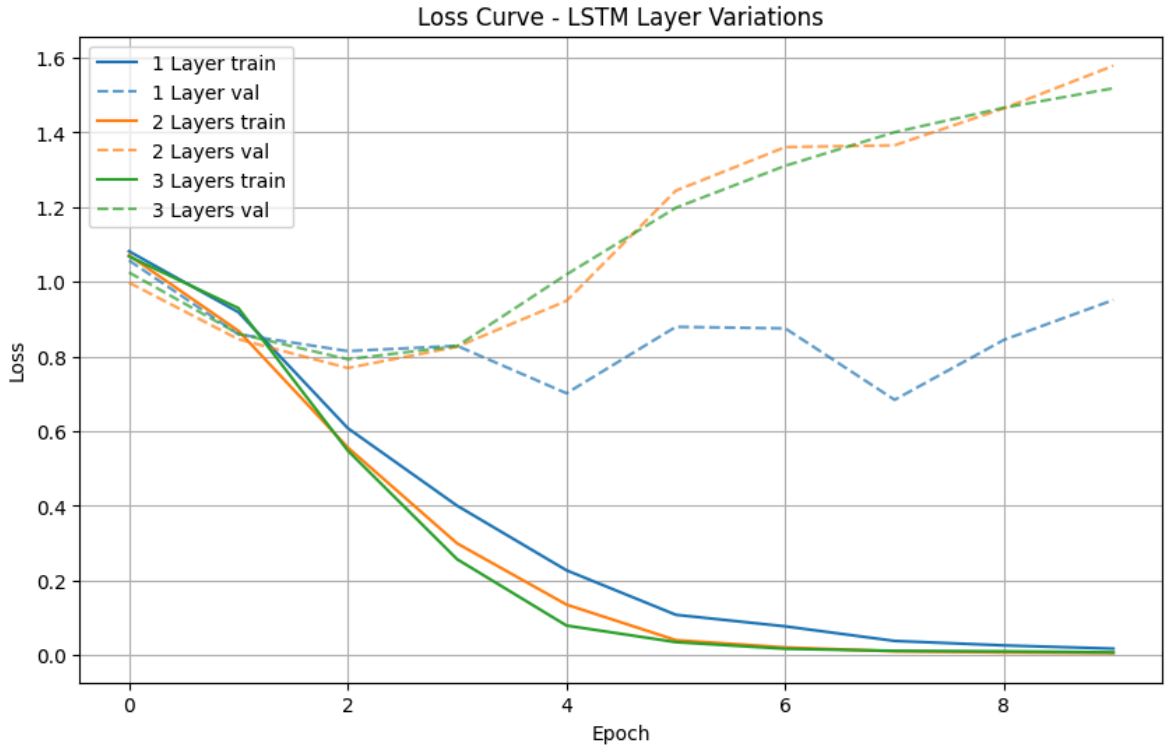
d. Perbandingan implementasi from-scratch dengan model Keras



Menggunakan bobot yang didapatkan dari hasil training model Keras, dan menggunakannya sebagai input pada model from scratch, dapat dilihat bahwa hasil inferensi menunjukkan hasil yang sama dengan model Keras.

C. LSTM

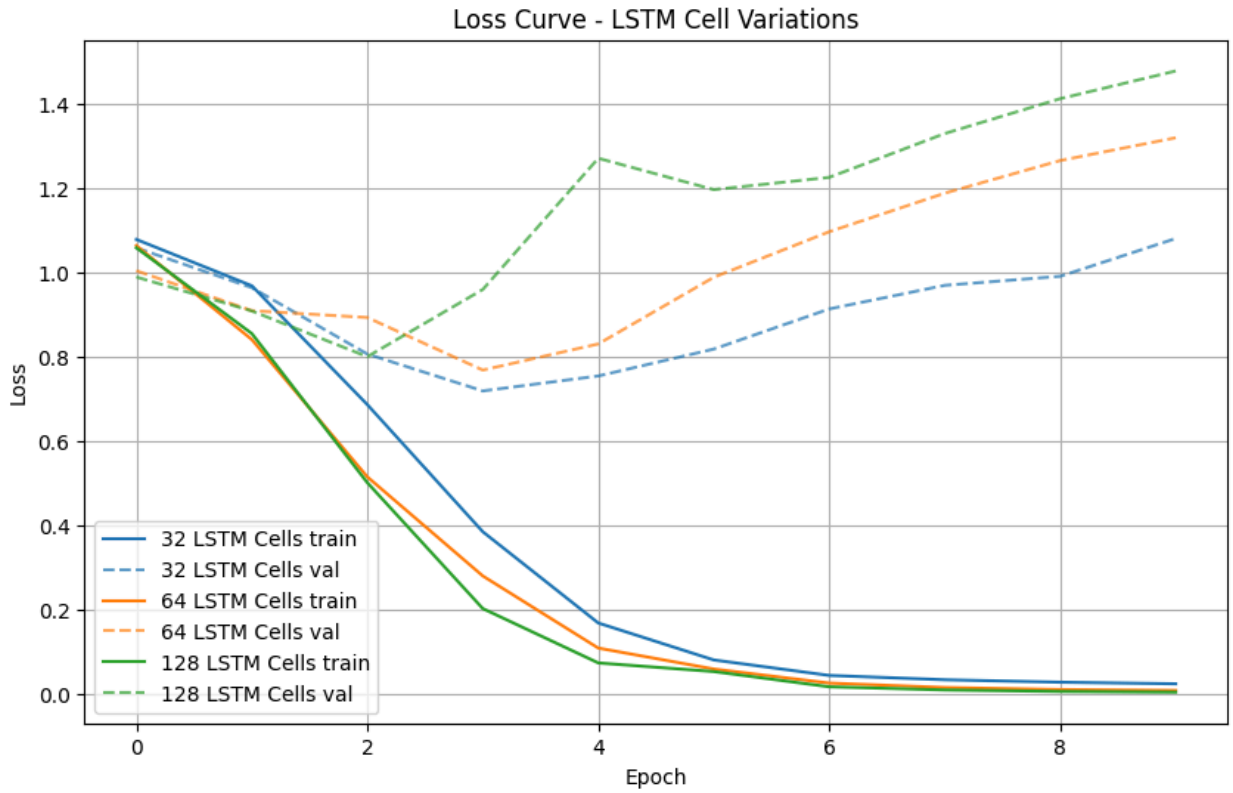
a. Hasil Variasi Jumlah Layer



LSTM with 1 layer(s): Macro F1 = 0.7031
LSTM with 2 layer(s): Macro F1 = 0.7108
LSTM with 3 layer(s): Macro F1 = 0.7318

Penambahan jumlah layer LSTM cenderung meningkatkan nilai Macro F1-Score pada hasil akhir prediksi. Akan tetapi, dari grafik loss, model dengan jumlah layer yang lebih banyak menunjukkan kecenderungan overfitting yang lebih cepat dan lebih besar, terlihat dengan meningkatnya validation loss setelah beberapa epoch awal meskipun training loss terus menurun.

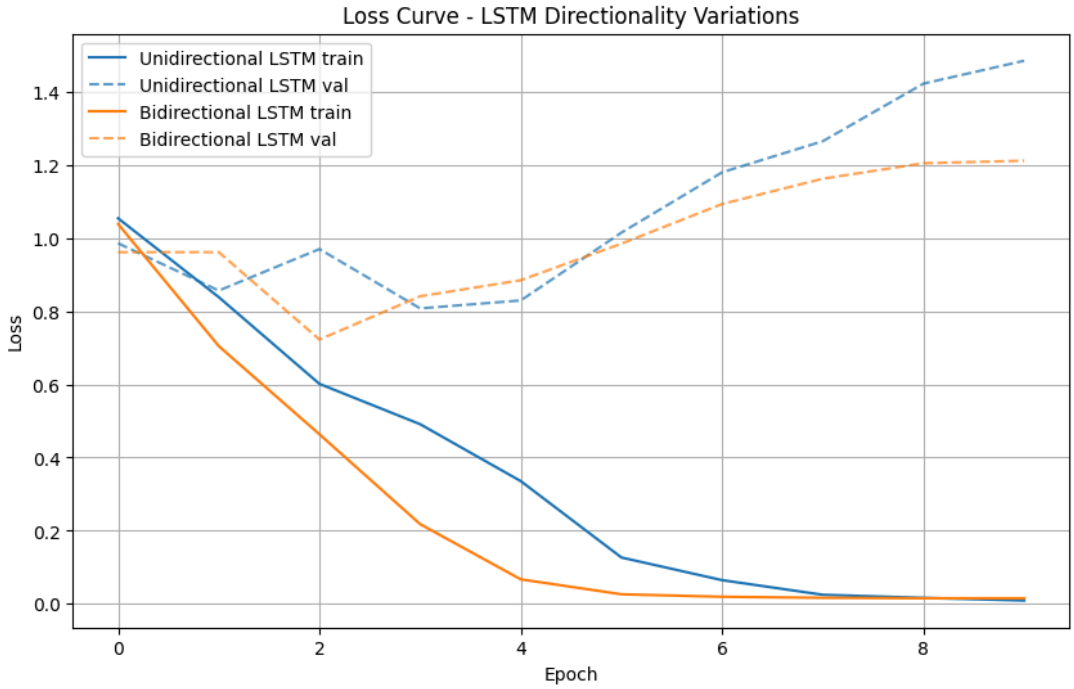
b. Hasil Variasi Jumlah Cell per Layer



32 LSTM cell(s) per layer: Macro F1 = 0.7390
64 LSTM cell(s) per layer: Macro F1 = 0.7185
128 LSTM cell(s) per layer: Macro F1 = 0.7379

Variasi jumlah sel LSTM per layer menunjukkan bahwa 32 sel menghasilkan Macro F1-Score tertinggi. Terlihat juga di grafik loss dimana model dengan 32 sel menunjukkan *validation loss* yang paling stabil dan rendah. Model dengan jumlah sel yang lebih banyak (64 dan 128 sel) mengalami *overfit* meskipun kapasitas modelnya lebih besar untuk mempelajari data training.

c. Hasil Variasi Arah



Unidirectional: Macro F1 = 0.6860
Bidirectional: Macro F1 = 0.7254

Variasi arah menunjukkan bahwa layer Bidirectional LSTM menghasilkan Macro F1-Score yang lebih tinggi dibandingkan Unidirectional LSTM. Grafik loss juga menunjukkan bahwa Bidirectional LSTM memiliki *validation loss* yang lebih rendah dan lebih stabil dibandingkan Unidirectional LSTM. Bidirectional LSTM juga mencapai *plateau train loss* lebih cepat dibandingkan Unidirectional LSTM

d. Perbandingan implementasi from-scratch dengan model Keras

```
embedding_matrix = result_weights[0]
lstm_W = result_weights[1]
lstm_U = result_weights[2]
lstm_b = result_weights[3]
dense_W = result_weights[4]
dense_b = result_weights[5]

X_test_vec_np = X_test_vec.numpy() if hasattr(X_test_vec,
'numpy') else np.array(X_test_vec)
```

```
emb_out = embedding_forward(X_test_vec_np, embedding_matrix)
input_mask = (X_test_vec_np != 0)
lstm_out = lstm_forward(emb_out, lstm_W, lstm_U, lstm_b,
input_mask)
drop_out = dropout_forward(lstm_out, rate=0.5)
dense_out = dense_forward(drop_out, dense_W, dense_b)

probs_scratch = softmax(dense_out)
y_pred_scratch = np.argmax(probs_scratch, axis=1)

# Compare with Library
probs_keras = keras_model_scratch.predict(X_test_vec)
y_pred_keras = np.argmax(probs_keras, axis=1)

f1_scratch = macro_f1(y_test, y_pred_scratch)
f1_keras = macro_f1(y_test, y_pred_keras)

print(f'From-scratch LSTM Macro F1-score: {f1_scratch:.4f}')
print(f'Keras LSTM Macro F1-score: {f1_keras:.4f}')
print('Prediction equal?', np.all(y_pred_scratch ==
y_pred_keras))
if (np.all(y_pred_scratch != y_pred_keras)):
    print('Are probabilities close (within 1e-5)?',
np.allclose(probs_scratch, probs_keras, atol=1e-5))
```

From-scratch LSTM Macro F1-score: 0.7585
Keras LSTM Macro F1-score: 0.7585
Prediction equal? True

Di atas adalah kode untuk membandingkan implementasi *from scratch* dengan implementasi melalui keras. Pertama melakukan training dengan model keras, kemudian mengekstrak bobot dari training tersebut untuk digunakan pada forward propagation *from scratch*. Kemudian hasil keduanya dibandingkan. Dan berdasarkan hasil, dapat dilihat implementasi *from scratch* menghasilkan nilai macro F1 yang sama dengan implementasi keras

Bab IV
Kesimpulan dan Saran

A. Kesimpulan

CNN: Forward propagation pada CNN dapat diimplementasikan dengan cukup sederhana, namun model tersebut masih perlu mendapatkan hasil bobot pelatihan dari model Keras karena implementasi *backwards propagation* CNN yang masih cukup sulit.

RNN: Forward propagation pada RNN dapat diimplementasikan dengan cukup sederhana, namun model tersebut masih perlu mendapatkan hasil bobot pelatihan dari model Keras karena implementasi *backwards propagation* RNN yang masih cukup sulit. Diperlukan dataset yang lebih banyak untuk mendapatkan model RNN yang bagus untuk melakukan prediksi, karena data terkait teks yang memiliki kemungkinan input yang sangat banyak sangat sensitif untuk melakukan *overfitting.*

LSTM: Forward propagation pada LSTM dapat diimplementasikan dengan cukup sederhana, namun karena datasetnya yang kecil, model yang complex (jumlah layer lebih banyak, jumlah cell lebih banyak) cenderung overfit (terlihat dari val loss yang naik terus). Tetapi dibandingkan RNN, masih lebih baik hasilnya jika dilihat dari Macro F1 Score

B. Saran

CNN: Untuk implementasi selanjutnya dapat dicoba untuk memberikan struktur pasti model Keras untuk diimplementasikan agar pengerjaannya lebih terarah.

RNN: Untuk implementasi selanjutnya dapat mencoba menggunakan dataset yang lebih besar untuk mendapatkan hasil RNN yang lebih bagus untuk melakukan prediksi. Selain itu, proses data cleaning dan preprocessing data sebaiknya dilakukan untuk lebih meningkatkan hasil model.

LSTM: Untuk implementasi selanjutnya menggunakan dataset yang lebih besar untuk memberikan model waktu untuk berlatih.

Bab V
Pembagian Tugas

NIM	Nama	Tugas
13522127	Maulana Muhamad Susetyo	Implementasi dan Laporan Bagian LSTM
13522138	Andi Marihot Sitorus	Implementasi dan laporan bagian CNN
13522158	Muhammad Rasheed Qais Tandjung	Implementasi dan laporan bagian RNN

Bab VI
Referensi

- Pranala Github: github.com/trimonuter/ML_IF3270_TugasBesar2
- https://d2l.ai/chapter_recurrent-modern/lstm.html
- https://d2l.ai/chapter_recurrent-modern/deep-rnn.html
- https://d2l.ai/chapter_recurrent-modern/bi-rnn.html
- https://d2l.ai/chapter_recurrent-neural-networks/index.html
- https://d2l.ai/chapter_convolutional-neural-networks/index.html
- https://numpy.org/doc/2.1/reference/generated/numpy.einsum.html