CS 621 - Deep Learning for NLP
Assignment 3

Muhammad Adnan Rizqullah (2403851)

Supervised by Dr. Fawaz Al-Salmi

Assignment Report

This project explores three types of recurrent neural network architectures: Standard RNNs, LSTMs, and Bidirectional RNNs/Transformers on two important NLP tasks: machine translation and text generation.

This report consists of 3 sections:

1. Architecture and hyperparameters: describes the RNN architecture and hyperparameters that is used for various conducted experiments
2. Results: the results of the experiments
3. Data preparation: the data preparation steps for the project

Github repository of project: https://github.com/madnanrizqu/cs681-assignment-3

Architecture and hyperparameters

This section discusses the architectures and hyperparameters used across both the text translation and text generation tasks.

Common Components

All model architectures share several common components:

1. **Embedding layer**: Maps token indices to dense vector representations (256 dimensions)
2. **Vocabulary management**: Limited vocabulary size (5,000 for translation, 10,000 for generation)
3. **Loss function**: Custom masked loss for handling padded sequences for translation. Regular cross entropy loss for generation
4. **Optimizer**: Adam optimizer

Text Translation Architectures

All translation models follow an encoder-decoder architecture with attention, but differ in the type of recurrent units used:

Standard RNN

```
# Encoder
self.rnn = tf.keras.layers.SimpleRNN(units,
# Return the sequence and state
return_sequences=True,
recurrent_initializer='glorot_uniform')
```

```
# Decoder
self.rnn = tf.keras.layers.SimpleRNN(units,
return_sequences=True,
return_state=True,
recurrent_initializer='glorot_uniform')
```

LSTM Model

```
# Encoder
self.rnn = tf.keras.layers.LSTM(units,
return_sequences=True,
recurrent_initializer='glorot_uniform')

# Decoder
self.rnn = tf.keras.layers.LSTM(units,
return_sequences=True,
return_state=True,
recurrent_initializer='glorot_uniform')
```

The LSTM model includes additional gating mechanisms to better capture long-range dependencies, with cross-attention using 4 attention heads.

Bidirectional LSTM

```
# Encoder
self.rnn = tf.keras.layers.Bidirectional(
merge_mode='sum',
layer=tf.keras.layers.LSTM(units,
return_sequences=True,
recurrent_initializer='glorot_uniform'))

# Decoder
self.rnn = tf.keras.layers.LSTM(units,
return_sequences=True,
return_state=True,
recurrent_initializer='glorot_uniform')
```

The Bidirectional LSTM processes input in both forward and backward directions, enhancing contextual understanding with 4-headed cross-attention.

Text Generation Architectures

For text generation, we implemented three architectures:

Standard RNN

```
self.rnn = tf.keras.layers.SimpleRNN(rnn_units,
```

```
return_sequences=True,
return_state=True)
```

LSTM Model

```
self.rnn = tf.keras.layers.LSTM(rnn_units,
return_sequences=True,
return_state=True)
```

The LSTM includes additional gating mechanisms (input gate, forget gate, and output gate) to better capture long-range dependencies in text.

Bidirectional LSTM

```
self.rnn = tf.keras.layers.Bidirectional(
tf.keras.layers.LSTM(rnn_units//2, # Half the units for each direction
return_sequences=True,
return_state=True)
)
```

Training Hyperparameters

All models were trained with consistent hyperparameters:

Embedding dimensions	256	256
Hidden units	256	512
Batch size	64	64
Maximum vocabulary size	5,000	10,000
Optimizer	Adam	Adam
Training epochs	20 (with early stopping)	20 (with early stopping)
Steps per epoch	100	Not determined
Validation steps	20	Not determined
Early stopping patience	3	5

Results

This section presents the results from both the text translation and text generation experiments.

Text Translation Results

The following table summarizes the performance metrics across all three translation models:

Model	Test Set Accuracy	Test Set Loss	Test BLEU
Standard RNN	38.40%	3.6717	0.0261
LSTM	64.44%	2.1192	0.5294
LSTM Bidirectional	65.17%	2.1213	0.5547

We runned the notebook here to have advantage of google's more powerfull computing:

1. rnn notebook: https://colab.research.google.com/drive/1B2IBb6WLiyc_8YCBAT-aa-NZF0ww8gSO?usp=sharing
2. lstm notebook: https://colab.research.google.com/drive/1umSJAMZCq_C4YeH0Ra0ICCb_H1SRELT_?usp=sharing
3. lstm bidirectional: https://colab.research.google.com/drive/1_fd6P_QZAsU0zAEvvC-fxeRsDAfad3F6?usp=sharing

Training Dynamics for Translation

1. The Standard RNN failed converged compared with LSTM-based models
2. The LSTM improved massively compared with standard RNN
3. The Bidirectional LSTM showed minor training stability compared to unidirectional models

Text Generation Results

The following table summarizes the performance metrics across all three generation models:

Model Test Perplexity Test Loss Qualitative
Assestment

Model	Test Set Perplexity	Test Loss	Qualitative Assessment (A score till F)
Standard RNN	5.5830	1.7197229862213135	AB

LSTM	5.3288	1.67312049865 72266	AB
LSTM Bidirectional	1.0510	0.04974830895 662308	F

We runned the notebook here to have advantage of google's more powerfull computing:

1. rnn notebook: https://colab.research.google.com/drive/1kxC-RZOFHkfO3Z5Ne4FhuZ5fkPFtIcSi?usp=sharing
2. lstm notebook: https://colab.research.google.com/drive/1PWLNLdJdr1ph35P5YQPCeoHrGB_Mrq6k?usp=sharing
3. lstm bidirectional: https://colab.research.google.com/drive/1NmnK-n-BWA_1NsiH4EhSzAB8rRITemox?usp=sharing

Training Dynamics for Generation

1. The LSTM Bidirectional showed good perplexity and loss scores on test dataset but somehow fails to generate good text
2. The LSTM and RNN performed quite similar

Data preparation

This section details the data preparation process for both tasks.

Text Translation Dataset

For the translation task, we used an English-Indonesian parallel corpus containing approximately 13,500 sentence pairs. The preprocessing pipeline included:

1. **Text normalization**:
    a. Conversion to lowercase
    b. Unicode normalization (NFKD)
    c. Regular expression filtering
    d. Adding spaces around punctuation marks
2. **Tokenization**:
    a. Adding special tokens `[START]` and `[END]`
    b. Converting tokens to integer indices
3. **Dataset splits**:
    a. 75% training / 15% validation / 15% test split
    b. Batch size of 64 examples

Text Generation Dataset

For the text generation task, we used a corpus of Romeo & Juliet from Project Gutenberg. The preprocessing pipeline included:

1. **Text normalization**:
   a. Conversion to lowercase
   b. Unicode normalization (NFKD)
   c. Regular expression filtering
   d. Adding spaces around punctuation marks
2. **Tokenization**:
   a. Adding special tokens `[START]` and `[END]`
   b. Converting tokens to integer indices
3. **Dataset splits**:
   a. 80% training / 10% validation / 10% test split
   b. Batch size of 64 examples
   c. Sequence length of 100 tokens per example

Common Preprocessing Steps

Both tasks employed similar preprocessing strategies to ensure fair model comparison:

1. Text normalization to standardize input
2. Limited vocabulary size with OOV tokens handled via `[UNK]` token
3. Consistent data batching methodology
4. Specialized processing for sequence beginnings and endings

This preprocessing approach ensured that all models were evaluated on a level playing field, with differences in performance attributable to architectural variations rather than data preparation discrepancies.