Comparing Seq2Seq Encoder and Decoder Performance for English to Japanese Translation

Penelope King
pking@ucsd.edu

Abstract

This project aims to build and test different types of Sequence-to-Sequence (Seq2Seq) architectures and compare their performance on a translation task (English to Japanese). Using this exploration, I aim to identify the strengths and weaknesses of different Seq2Seq architectures, which can provide better understanding of similar natural language processing tasks. This exploration is motivated by the increasing importance of communication in an ever growing and connected world. The 4 encoder/decoder structures tested are Gated Recurrent Unit (GRU), Long Short Term Memory (LSTM), Transformers, and Convolutional Neural Networks (CNN). And the data used is the Japanese-English Subtitle Corpus (JESC). In this project, I show that CNNs performed the strongest out of the 4 models due to its ability to converge faster and its ability to capture local connections when translating individual sentences.

Code: https://github.com/PenelopeKing/translation-project

1 Introduction

Translation is a powerful tool that can help bridge different spheres of the world together that may be blocked off due to language barriers. With the growing connection of society, machine translation has become critical in natural language processing. Translation specifically comes with unique challenges due to the complexity and nuance of linguistic differences and nuances across languages.

Sequence to Sequence (Seq2Seq) models are an ideal model architecture for translation tasks. A Seq2Seq model uses 2 Recurrent Neural Networks (RNNs) to transform one sequence into another. It is ideal for tasks when input and output sequences can vary in length, which translation tasks often deal with. It's architecture is made up of an encoder, decoder and intermediate step. RNNs assume all inputs are independent and are crafted to capture temporal dependencies by using a hidden state that persists and evolves over time. Parameters are also shared, and having the same weights across all steps makes RNNs efficient for sequential data. Some limitations of RNNs are vanishing gradients (gradients become very small) especially when training long sequences. And a difficulty with long-term dependencies. Advanced RNN architectures like Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU) tackles some of these limitations, improving the handling of long-term dependencies. Other approaches, such as Convolutional Neural Networks (CNNs) and Transformer models, offer further innovations in efficiency and accuracy, making them promising alternatives for translation tasks.

My work centers on translating English to Japanese and determining what architecture will be the best fit. This task comes with challenges; Japanese and English differ in terms of grammar, cultural context, characters used, and more. To address these challenges, I explore and compare various Seq2Seq architectures—CNNs, GRUs, LSTMs, and Transformers—evaluating their performance on translation tasks.

Through this exploration, I aim to identify strengths and limitations of different Seq2Seq architectures in handling English to Japanese translation, providing insights for similar future natural language processing tasks. I find that CNNs significantly enhances the performance of a Seq2Seq model trained on the data.

2 Data

The dataset I used for the project is a cleaned dataset from the Japanese-English Subtitle Corpus (JESC). It is created from crawling the internet for movie and TV subtitles and aligning their captions. It is also one of the largest freely available EN-JA corpus datasets. The dataset is large, with over 2 million data points. This dataset is available at Stanford NLP (2023). Each single training point consists of a sentence in English with its respective Japanese sentence (phrase pairs). In total there are 2,801,388 phrase pairs. The dataset comes with a predefined train-validation-testing split which was: 2,797,388/2000/2000, respectively. Due to the large size of the training data, only a smaller random subset of the training data was used in my project due to limitations of computational power and time.

Data was processed by tokenizing the words in each sentence. Then I built an English and Japanese vocabulary using the training data. These vocabularies also included special tokens such as padding, start, end and unknown tokens. I limited my vocabulary size to the 9000 most frequent tokens.

3 Models

I trained and tested 4 different type of Seq2Seq RNNs using GRU, LSTM, Transformer, and CNN architectures as the encoders and decoders.

RNNs are mathematically defined as the following:

The input sequence is defined as:

$$\mathbf{X} = \{x_1, x_2, \ldots, x_T\}, \quad x_t \in \mathbb{R}^d$$

where T is the length of the input sequence, and x_t is the t-th input vector with d features.

The output sequence is defined as:

$$\mathbf{Y} = \{y_1, y_2, \ldots, y_{T'}\}, \quad y_t \in \mathbb{R}^k$$

where T' is the length of the output sequence, and y_t is the t-th output vector with k features.

The encoder processes the input sequence to produce a hidden state representation. At each time step t, the hidden state is updated as:

$$h_t^{\text{enc}} = f_{\text{enc}}(W_x x_t + W_h h_{t-1}^{\text{enc}} + b_h), \quad h_t^{\text{enc}} \in \mathbb{R}^n$$

where:

- h_0^{enc} is the initial hidden state of the encoder (e.g., $h_0^{\text{enc}} = 0$ or a learnable parameter),
- f_{enc}: Activation function (e.g., tanh, ReLU),
- $W_x \in \mathbb{R}^{n \times d}$: Input-to-hidden weight matrix,
- $W_h \in \mathbb{R}^{n \times n}$: Recurrent weight matrix,
- $b_h \in \mathbb{R}^n$: Bias vector for the hidden state.

The final hidden state of the encoder, h_T^{enc}, is used to initialize the decoder. The decoder generates the output sequence one time step at a time, conditioned on the encoder's final hidden state. At each time step t, the decoder's hidden state is updated as:

$$s_t = f_{\text{dec}}(W_y y_{t-1} + W_s s_{t-1} + W_c h_T^{\text{enc}} + b_s), \quad s_t \in \mathbb{R}^m$$

where:

- $s_0 = W_{\text{init}} h_T^{\text{enc}}$: The decoder's initial hidden state, initialized from the encoder's final hidden state,
- f_{dec}: Activation function (ReLU),

- $W_y \in \mathbb{R}^{m \times k}$: Input-to-hidden weight matrix for the decoder input,
- $W_s \in \mathbb{R}^{m \times m}$: Recurrent weight matrix for the decoder,
- $W_c \in \mathbb{R}^{m \times n}$: Context weight matrix connecting the encoder to the decoder,
- $b_s \in \mathbb{R}^m$: Bias vector for the decoder's hidden state.

The decoder generates an output at each time step as:

$$\hat{y}_t = \text{Softmax}(W_o s_t + b_o), \quad \hat{y}_t \in \mathbb{R}^k$$

where:

- $W_o \in \mathbb{R}^{k \times m}$: Weight matrix for the output layer,
- $b_o \in \mathbb{R}^k$: Bias vector for the output layer,
- Softmax ensures the output is a probability distribution for classification tasks.

The Seq2Seq model is trained to minimize the sequence cross-entropy loss:

$$\mathscr{L} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

For the experiment, I trained four different Seq2Seq models in total, each differing in what architecture is used for their encoders and decoders: CNN, Transformer, GRU, and LSTM.

These encoders and decoder architectures can be mathemtically described as the following:

2.1 CNN Encoder-Decoder

CNNs have convolutional layers that prioritize local connections. It takes a small patch of the input and runs a smaller neural network (kernel). It includes a pooling layer that down samples the input to reduce computation. This makes CNNs efficient to run due to there being less parameters to learn. This often can lead to faster convergence, which is especially important when dealing with large data.

The CNN encoder processes an input sequence $\mathbf{X}$, using 1D convolution layers:

$$\mathbf{H} = \text{Conv1D}(\mathbf{X}, W_{\text{enc}}) + b_{\text{enc}}, \quad \mathbf{H} \in \mathbb{R}^{T \times n}$$

where:

- $W_{\text{enc}} \in \mathbb{R}^{n \times d \times k}$: Convolution filter with k kernels and d input channels,
- $b_{\text{enc}} \in \mathbb{R}^n$: Bias vector,
- $\mathbf{H} = \{h_1, h_2, \ldots, h_T\}$: Encoded sequence representation.

The CNN decoder generates the output sequence $\mathbf{Y} = \{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{T'}\}$ using another convolution:

$$\hat{\mathbf{Y}} = \text{Conv1D}(\mathbf{H}, W_{\text{dec}}) + b_{\text{dec}}$$

where:

- $W_{\text{dec}} \in \mathbb{R}^{k \times n \times d'}$: Decoder convolution filter,
- $b_{\text{dec}} \in \mathbb{R}^{d'}$: Bias vector for the decoder.

2.2 GRU Encoder-Decoder

The GRU architecture uses gate mechanisms to update some hidden states of the network at each step. These gating mechanisms control what comes in and out of the network and there are 2 types: the reset and update gates. The reset gate is in charge of how much the previous hidden state is remembered, and the update gate is in charge of what input should be used to update the current hidden state. This allows for more efficient computation of the model. It handles moderately long-term dependencies and reaches convergence faster than LSTMs may.

The GRU encoder updates its hidden state at each time step t as follows:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

where:

- z_t, r_t: Update and reset gates,
- $W_z, W_r, W_h \in \mathbb{R}^{n \times d}$, $U_z, U_r, U_h \in \mathbb{R}^{n \times n}$: Weight matrices,
- $b_z, b_r, b_h \in \mathbb{R}^n$: Bias vectors,
- $\odot$: Element-wise multiplication.

The GRU decoder uses similar equations. The encoder's final hidden state h_T^{enc} is:

$$s_t = (1 - z_t) \odot s_{t-1} + z_t \odot \tilde{s}_t$$

$$\hat{y}_t = \text{Softmax}(W_o s_t + b_o)$$

where:

- s_t: Decoder hidden state,
- $W_o \in \mathbb{R}^{k \times n}$, $b_o \in \mathbb{R}^k$: Output layer parameters.

2.3 LSTM Encoder-Decoder

LSTMs also have gates like GRU, but has 3 gates instead of 2. These are the input, forget, and output gate. These gates determine the information to add, remove and return from each memory cell, respectively. This architecture allows LSTMs to control what to remember and throw away, making it able to learn long-term dependencies. LSTMs address the issues of vanishing gradients, but process words one by one in a sequence.

The LSTM encoder updates its hidden state and cell state as follows:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$h_t = o_t \odot \tanh(c_t)$$

where:

- f_t, i_t, o_t: Forget, input, and output gates,
- c_t: Cell state,
- $W_f, W_i, W_o, W_c \in \mathbb{R}^{n \times d}$, $U_f, U_i, U_o, U_c \in \mathbb{R}^{n \times n}$: Weight matrices,
- $b_f, b_i, b_o, b_c \in \mathbb{R}^n$: Bias vectors.

The LSTM decoder uses the final encoder hidden state h_T^{enc} and cell state c_T^{enc} to initialize the decoder:

$$s_t, c_t = \text{LSTM}(y_{t-1}, s_{t-1}, c_{t-1})$$
$$\hat{y}_t = \text{Softmax}(W_o s_t + b_o)$$

where s_t is the decoder hidden state and c_t is the decoder cell state.

2.4 Transformer Encoder-Decoder

Transformers use attention mechanisms to capture relationships between distant elements in a sequence. The attention mechanism uses scaled dot-products and calculate the weighted sum of a query, key, and value vector. This computes how much each element should attend to others. This helps to capture more complex relationships within the data. However, transformers are very costly to compute and take quadratic complexity.

Attention is calculated as the following:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Where Q is the query matrix, K is the key matrix, and V is the value matrix (contains information to be passed). And d_k is the dimensionality of the key vectors.

Each transformer layer computes multi-headed attention and is calculated as the following:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O$$

Each head is the following:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The encoder processes the input sequence $\mathbf{X}$ using a series of attention layers and feed forward layers.

Encoder Layer

$$\text{Attention: } \text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

where:

- $Q = W_Q\mathbf{X}$, $K = W_K\mathbf{X}$, $V = W_V\mathbf{X}$,
- $W_Q, W_K, W_V \in \mathbb{R}^{d_k \times d}$.

Feedforward Layer

$$\text{FFN}(x) = \sigma(W_1 x + b_1)W_2 + b_2$$

where $W_1 \in \mathbb{R}^{d_{\text{ff}} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{\text{ff}}}$.

Decoder Layer

The decoder combines encoder outputs $\mathbf{H}$ with the decoder input $\mathbf{Y}$:

$$\text{Attention}(\mathbf{Y}, \mathbf{H}) = \text{Softmax}\left(\frac{Q'K^\top}{\sqrt{d_k}}\right)V$$

where:

- $Q' = W'_Q\mathbf{Y}$, $K = W_K\mathbf{H}$, $V = W_V\mathbf{H}$.

2.5 Models

The optimizer I used was: Adam(lr = 0.001, wd = 1e-5). The architecture and parameters of the models are described in the following tables (Table 1):

Table 1: Model Architecture Parameters

	Transformer	LSTM	GRU	CNN
Embedding Dimension	15	15	15	15
Layers	3	3	3	-
Heads	3	-	-	-
Dropout	0.2	0.2	0.2	-
Kernel Size	-	-	-	3
Channels	-	-	-	5

After training, validation, and testing these models on the dataset's predefined training and testing split, I compared all the models to compare their performance across datasets.

The performance metric I used was average accuracy of predicted versus true tokens (words) across a sentence, excluding padding and special tokens (e.g. <START>, <UNKNOWN>, etc...).

4 Results

All models were trained on 50 epochs each and parameters were set to attempt to give "equal attention" to all models to ensure balanced comparison. Using these models, I calculated their final training and testing accuracy (Table 2):

Table 2: Model Train and Test Accurarcies

Model	Train Accuracy	Test Accuracy
Transformer	0.283	0.234
LSTM	0.239	0.187
GRU	0.191	0.133
CNN	0.946	0.939

5 Conclusion

The CNN model performed the best by far, likely because this model was able to converge faster than the rest of the models due to CNN focusing more on local connections. And since many of the sentence sequences were short (the maximum length of a sentence was around 15 words per sentence), there may have not been a need to incorporate long range dependencies in the model architecture. Furthermore, as a language, sentences tend to be shorter in Japanese compared to some other language. This may also be the reason why the CNN model performed better. Japanese as a language relies heavily on context, which also contributes to shorter sentences. Character density is also shorter as well. The Transformer model performed second best, followed by LSTM, and GRU. The other 3 models may have had more difficulty converging to find an optimal solution for learned parameters, which may be why these models did not perform as strongly. There are a lot of data points in the dataset, and limited time and computational power to be able to fully optimize these models.

Transformers rely on self-attention mechanisms, which are ideal for capturing long range dependencies, and may not have been able to fully use its strength in capturing global context with short sentence lengths, resulting in the subpar performance. The quadratic time complexity of transformers also makes it difficult to tune and train, which may also have contributed to its performance. LSTMs and GRUs both are sequential and process tokens one by one, which may have been less efficient compared to the CNN and Transformer models, resulting in both model architectures performing the worst. And since GRUs are a more simplified version of LSTMs, it have have struggled to capture the moderately long

range dependencies and lack of focus on local dependencies, which most likely led to it performing the worst of out all the models.

In a future project, it would be insightful to further explore into these architectures and maybe also different ones as well with stronger computational power and extended time. Furthermore, it would be interesting to work on entire documents rather than individual sentences. Individual sentences tend to be shorter, which LSTMs, GRU, and Transformers cannot fully take advantage of, as they deal with longer range dependencies. This is likely why CNNs performed the best out of the 4 models. In word documents (multiple sentences), it would be interesting to see if CNN performance increases or decreases as well as the other models. Another interesting direction would be to try different language combinations and see if CNNs still perform well on those translations tasks. It would be insightful to see if maybe CNNs performed well due to the nature of Japanese's shorter sentence structures, and if a language with longer sentences would have performed worse.