Sequence Data and Recurrent Neural Networks (RNNs)

Introduction and Architecture:

Sequence data are type of data which has some positional information or there exist dependencies between data points. Sequence data are generally ordered data. Some examples of sequence data are: time-series data, text data, gene data, etc.

While working with sequence data, we might encounter with three different cases:

1. Input is a sequence; output is not a sequence (sequence to one | Many to One)
2. Input is not a sequence; output is a sequence (one to sequence | One to Many)
3. Input is a sequence, and output is also a sequence (sequence to sequence | Many to Many)

Recurrent Neural Networks are class of Artificial Neural Networks that can process data that has sequence structure such as time-series data, natural language, music, gene sequence data and many more. One of the important information in sequence data unlike in tabular data, is positional information. RNNs attempt to integrate positional information in the model.

For example, in text completion task: "Jack and Jill went up the *hill* ", to predict "hill" at the end of the sentence, the RNN would use some information from all words preceding it. An RNN architecture consists of recurrent units, typically represented as *cells*, which are connected to each other in a loop. This recurrent structure is what allows information to flow through the network over time.

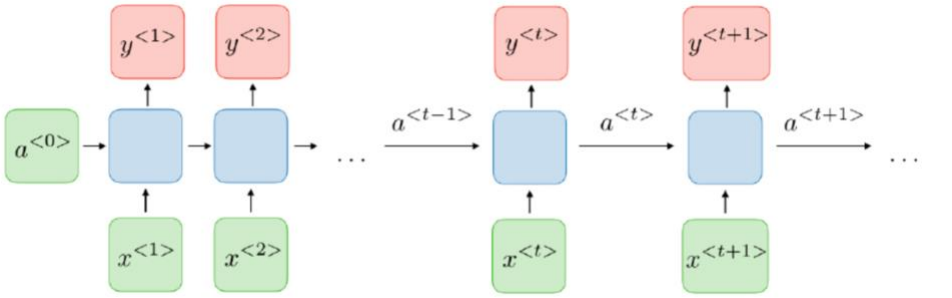


Figure: RNN Architecture. Each blue box is a RNN Cell.

At each time step t, the activation $a^{<t>}$ *and output* $y^{<t>}$ are obtained as:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad \text{and} \quad y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

Here, W and b represent weight and bias parameters to be learned and x represents the input sequence and $g(\cdot)$ is an activation. $g_1(\cdot)$ is used to create next activation and $g_2(\cdot)$ is used to create the output y . The subscripts 'ax' in W_{ax} represents that this weight operates on input 'x' to produce new activation a. A schematic of a RNN cell is shown below:

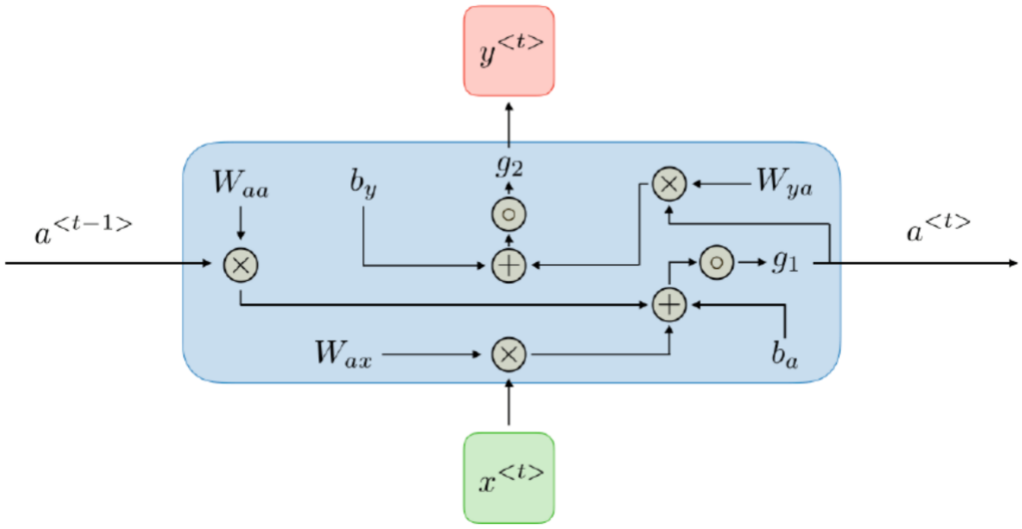


Figure: Schematic of a cell in an RNN

Parameter Sharing in RNN:

Note, in the Architecture figure above, Weights (W) and bias (b), the neural network parameters are shared among all cells across different *t*. Parameter sharing in Recurrent Neural Networks (RNNs) is a fundamental concept that sets them apart from traditional feedforward neural networks. RNN uses same set of model parameters across different time steps in the sequence. This characteristic of parameter sharing across the sequence is how RNN maintains memory and can capture sequential dependencies in the sequential data. i.e., the same set of weights and biases are used for each time step, and these parameters are shared across the entire sequence. This shared parameterization allows the network to maintain a hidden state that carries information from previous time steps and influences predictions at the current time step.

Advanced variants of RNNs, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks also employ shared parameters but incorporate gating mechanisms to better control the flow of information through the network, mitigating some of the issues associated with vanishing and exploding gradients.

Vanishing and Exploding Gradients in RNN:

In our discussion of backpropagation, we saw that gradients are key to finding the optimal weights (training process). However, we also mentioned that the gradients in neural net are locally influenced and propagated (Recall: Using chain rule for Backpropagation). Because RNN share same weights and biases (parameters) across entire sequences RNNs are susceptible to vanishing gradients which occur when gradients become very small. Vanishing gradients makes capturing long term dependencies in sequence data challenging. Exploding gradients can also occur in RNN. Exploding gradients occur when gradients grow rapidly, leading to unstable training. Exploding gradient can occur because of poor initialization, large values of the weights, divide by zero cases and many others. To mitigate exploding gradient, gradient clipping is used. In Gradient clipping, a threshold for gradient is set and gradient is stabilized if at any time gradient is larger than threshold. Gradient exploding due to divide by zero during batch normalization can be avoided by using a small tolerance value in the denominator to avoid infinite value or NaNs.

Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM)

GRUs and LSTMs helps to mitigate two major challenges in RNNs:

1. Vanishing Gradient
2. Inability to capture long term dependencies.

For example: In text data "The **cat** which already ate **was** full**.**" The model needs to remember "cat" is singular to be able to use singular verb "was". However, once such sequence is generated the model might need to forget that dependencies. To do so, GRU and LSTM introduces a new learnable parameter often called memory cell and represented by $c^{<t>}$. Several types of gates (Typically Sigmoid Function and denoted by Γ) are used to make decision on updating $c^{<t>}$. GRU and LSTM differs from each other on how these memory cell parameter $c^{<t>}$ and Γ are used. $\tilde{c}^{<t>}$ is a candidate to update $c^{<t>}$ and the gates Γ determines whether to update or not.

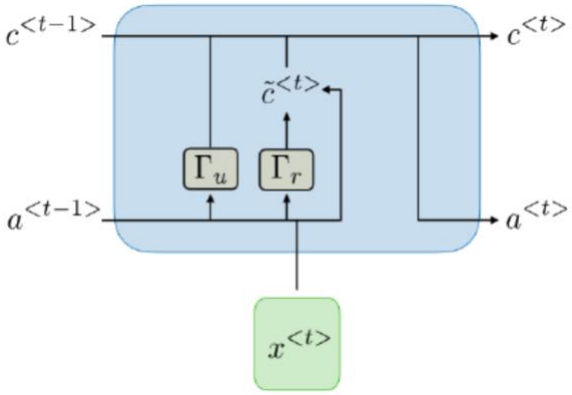


Figure: Schematic of a Gated Recurrent Unit (GRU cell).

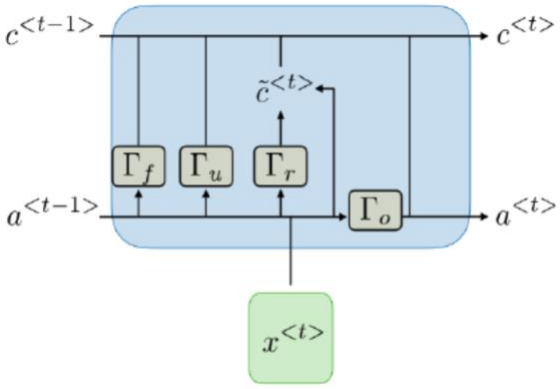


Figure: Schematic of a LSTM cell

The major difference between GRU and LSTMs are:

1. In GRUs, $a^{<t>} = c^{<t>}$, refer to the schematic above.
2. GRUs uses 2 gates, whereas LSTM uses 4 gates.

Cell and Gates expressions in GRU and LSTM:

GRU:

1. $\tilde{c}^{<t>} = tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$
2. $\Gamma_u = \sigma(W_c[c^{<t-1>}, x^{<t>}] + b_u)$ **(Update Gate)**
3. $\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$ **(Relevance Gate)**
4. $c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$

(Γ_u: Determines whether $c^{<t>}$ is updated by $\tilde{c}^{<t>}$ or not)

LSTM:

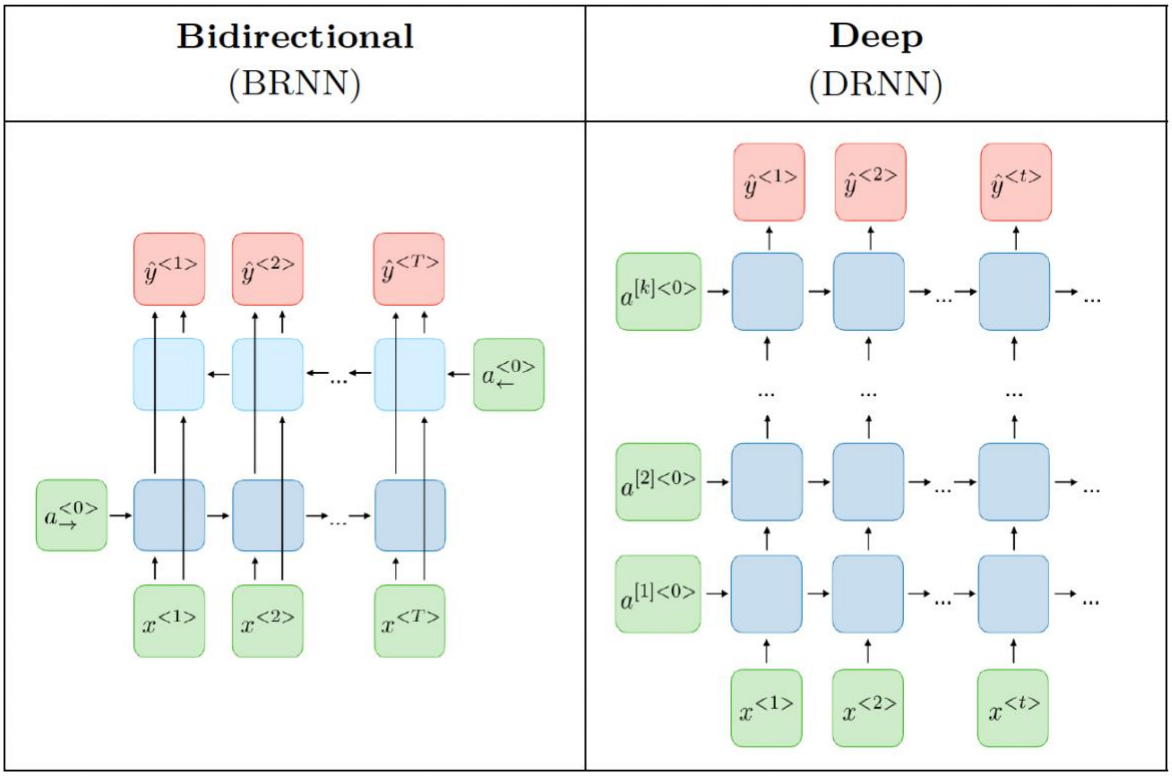
1. $\tilde{c}^{<t>} = tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$
2. $\Gamma_u = \sigma(W_c[a^{<t-1>}, x^{<t>}] + b_u)$

   Unlike in GRU, LSTM uses two gates Γ_f (Forget) and Γ_o (Output) to make update decision.

3. $\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$
4. $\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$
5. $c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (\Gamma_f) * c^{<t-1>}$ (Compare this with update rule for GRU).
6. $a^{<t>} = \Gamma_u * \tanh(c^{<t>})$

Bidirectional RNN and Deep RNN

Bidirectional RNNs and Deep RNNs are two different variants of RNNs. Bidirectional RNN has backward flow of information as well which is captured by another learnable parameter **a**, as shown in the figure below. On the other hand, Deep RNN are simply stacking many RNN layers on top of each other before collecting the output layer.



Text Data:

Text data or natural language data is another type of sequential data that can employ Recurrent Neural Network and Sequential Models. However, text data are different than other numeric sequential data such as stock-price time series data because i) text data are mostly non-numeric, qualitative data and have semantic information unlike numeric sequential data such as stock-price history data.

Preparing text data for Natural Language Processing (NLP) involves several essential steps to transform unstructured text into a format that is suitable for analysis and machine learning models.

Some Steps in Preparing Text Data for Processing

Text Cleaning:

Remove any irrelevant characters, symbols, or formatting issues from the text, such as special characters, HTML tags.

Lowercasing:

Convert all text to lowercase. This helps maintain consistency by treating words with different cases as the same, preventing the duplication.

Stopword Removal:

Is the process of removing common words (stopwords) that do not contribute much to the overall meaning of the text. Examples include "the," "and," and "is." It helps reduce the dimensionality of the data and improve the efficiency of analysis.

Tokenization:

Tokenization is a process of breaking text into smaller units, such as words or sub-words. Tokenization is a crucial step that converts the text into a format that can be processed by computers or machines.

Text Vectorization:

Text Vectorization is the process of converting text into numerical vectors. One reason for vectorization is the hope that representation of words in the vector space could capture some-sort of relation between words or text which can be helpful to understand the text semantics. For example, if vectorization of two words *"Machine"* and *"Learning"* are [0.5, 0.7, -0.3] and [0.7, 1.2, -0.3] respectively then some sort of mathematical computation such as cosine distance can be used to understand how related or similar are these words.

Common Text Vectorization techniques include:

i) Count Based Methods

Bag of Words (BoW): Represents each document as a vector of word frequencies.

Term Frequency-Inverse Document Frequency (TF-IDF): Reflects the importance of words in a document relative to their frequency across all documents.

ii) Word Embeddings

Dense vector representations of words that capture semantic relationships. Word embeddings are dense vector representations of words in a continuous vector space. Unlike the sparse and high-dimensional vectors in the Bag of Words model, word embeddings are lower-dimensional and capture semantic relationships between words. Some word embedding techniques include Word2Vec, GloVe (Global Vectors for Word Representation), BERT, etc. These models learn to

represent words in a way that similar words have similar vector representations, and they can capture semantic relationships and context.

In our Application Example code, Sentiment Analysis using LSTM, we will see following steps carried for preparing text data to bed processed using LSTM or any other ML model.

For illustrative purpose, let's take following List with 3 elements as our text data. You can think each element in this list as a stamen about the Machine Learning subject from a student.

Sample text =

['I love machine learning',

'It is an interesting subject',

'Machine Learning is a very boring subject

]

1. **Tokenization:**
   Creates a dictionary of key:value pair where each word in our sample text will be a key and a numeric value will be assigned to it. Apart from words in our vocab (sample_text), a special token <00V> would be also created to represent any words that are not present in the vocab.
   Example:

```
# Create and instance of a Tokenizer
tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)

# creates a dictionary of word and values using train data.
# i.e for each words in train review map it to some number
tokenizer.fit_on_texts(sample_text)
word_index = tokenizer.word_index
word_index
```

```
{'<OOV>': 1,
 'machine': 2,
 'learning': 3,
 'is': 4,
 'subject': 5,
 'i': 6,
 'love': 7,
 'it': 8,
 'an': 9,
 'interesting': 10,
 'a': 11,
 'boring': 12}
```

   Output:

2. **Convert the Text Data to Sequences:**
   This process takes our sample_text and convert each item in the list as sequence using the numeric 'value' created for each word in vocab created during the tokenization.
   Example:

```
# For sample_text
# using tokens above convert each review into a sequence.
sample_sequences = tokenizer.texts_to_sequences(sample_text)
sample_sequences
```

   Output:

```
[[6, 7, 2, 3], [8, 4, 9, 10, 5], [2, 3, 4, 11, 12, 5]]
```

   Note: Our first text data in the sample_text is "I love Machine Learning" and corresponding value pairs from tokenization are: 6, 7, 2, and 3. Hence the sequence for the text data : "I love Machine Learning" ~ [6, 7, 2, 3], and similarly for other text data in the sample_text.
   Note: Our sequences have different lengths. For processing sequence data using Machine Learning Algorithms such as LSTM and other models, we often require every sequence data to be of same length. This is achieved through padding and truncating.

3. **Padding and Truncation:** Padding and Truncation is used to make sure all sequence data are of some length.

Example:

```
# pad/truncate sequence to make each review of length max_length
sample_padded = pad_sequences(sample_sequences, maxlen=5, padding='post', truncating=trunc_type)
sample_padded
```

Output:

```
[[ 6,  7,  2,  3,  0],
 [ 8,  4,  9, 10,  5],
 [ 2,  3,  4, 11, 12]]
```

Compare this new padded and truncated sequences to the one above before padding and truncation.

Now, our text data is ready to be fed to the neural network. We will first use embedding layer in our model to get embedding for each sequence data. Then build the preferred neural network.

Example of Embedding Layer:

```
from tensorflow.keras.layers import Embedding
# Create an embedding layer
embedding_layer = Embedding(1000, 10)

# Pass some text to the embedding layer
embedded_text = embedding_layer(sample_padded)
# Print the shape of the output
print('The embedded text data shape is',embedded_text.shape)
print('An example of embedding of our first text sequence data is:\n')
embedded_text[1,:,:]
```

Output:

```
The embedded text data shape is (3, 5, 10)
An example of embedding: First embedded text sequence data is:

<tf.Tensor: shape=(5, 10), dtype=float32, numpy=
array([[-0.04125395,  0.04855437,  0.02515086, -0.01305473,  0.01466948,
         0.03884846, -0.00540029,  0.04975407, -0.0115853 ,  0.02742013],
       [-0.02488901,  0.0254776 , -0.0177028 ,  0.04949569,  0.03940037,
        -0.03260408,  0.04433412, -0.01016321, -0.02398716, -0.02276855],
       [ 0.01211165, -0.04683459,  0.01782714, -0.03366586, -0.04917466,
         0.02816759, -0.02666187, -0.03555292, -0.01121656, -0.02065084],
       [-0.00860788,  0.00218011, -0.03720338, -0.01439076,  0.03707841,
         0.00723063, -0.00357102, -0.01245975, -0.02958282, -0.00718694],
       [ 0.00868963, -0.04361991, -0.04320996, -0.03954545, -0.03536152,
        -0.02187313,  0.0062754 ,  0.03871896,  0.00609164, -0.00854009]],
      dtype=float32)>
```

Note: The embedded data set is of size (3,5,10) where first dimension (3) is equal to number of data samples in our sample_text, second dimension (5) is the sequence length we declared and the third dimension (10) is the embedding dimension we declared. So, each word is our data is represented as a 10-dimension numeric vector.

For code and details, Please refer to:

a) Text_Processing_Example.ipynb and
b) 06_SentimentAnalysis_IMDB.ipynb, in the code folder in Course One Drive.