

# Linear regression

ML Instruction Team, Fall 2022

CE Department  
Sharif University of Technology

# Backpropagation Through Time (BPTT)

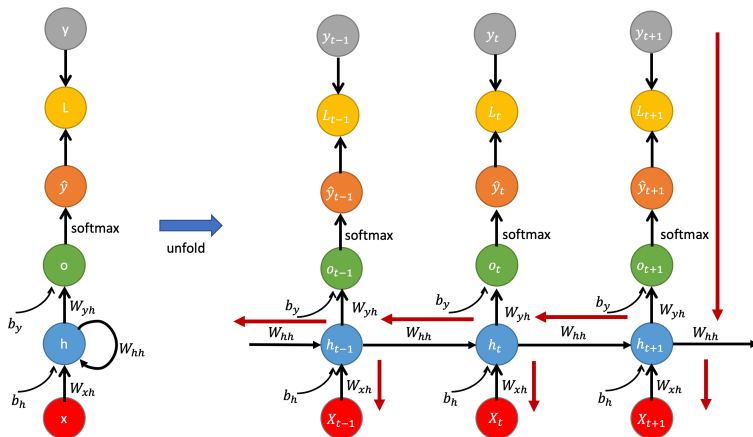


Figure: Simple RNN Computational Graph, [source](#)

# Backpropagation Through Time (BPTT)

- Suppose that in this example, we have

$$h_t = \tanh(X_t.W_{xh} + h_{t-1}.W_{hh} + b_h)$$

$$o_t = h_t.W_{yh} + b_y$$

$$y_t = \text{Softmax}(o_t)$$

- And our loss function is Log Loss. So as you remember, we have

$$L(y, \hat{y}) = \sum_{t=1}^T L_t(y_t, \hat{y}_t) = - \sum_{t=1}^T y_t \log \hat{y}_t = - \sum_{t=1}^T y_t \log [\text{Softmax}(o_t)]$$

# Backpropagation Through Time (BPTT)

- Now, we are going to calculate derivative of  $L$  w.r.t  $W_{yh}, W_{hh}, W_{xh}, b_y, b_h$

► Part I: The Straight Ones

$$\begin{aligned}\frac{\partial L}{\partial W_{yh}} &= \sum_{t=1}^T \frac{\partial L_t}{\partial W_{yh}} \\&= \sum_{t=1}^T \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial o_t} \frac{\partial o_t}{\partial W_{yh}} \\&= \sum_{t=1}^T (\hat{y}_t - y_t) \otimes h_t \\ \frac{\partial L}{\partial b_y} &= \sum_{t=1}^T \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial o_t} \frac{\partial o_t}{\partial b_y} = \sum_{t=1}^T (\hat{y}_t - y_t)\end{aligned}$$

# Backpropagation Through Time (BPTT)

■ Cont.

► Part II: The Tricky Ones

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial W_{hh}}$$

we know that  $h_t$  is a function of  $h_{t-1}$  and  $W_{hh}$ ,  $h_{t-1}$  itself is a function of  $W_{hh}$  and  $h_{t-2}$ , and so on. Thus, we have

$$\frac{\partial h_t}{\partial W_{hh}} = \left( \frac{\partial h_t}{\partial W_{hh}} \right)_{h_{t-1}} + \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W_{hh}}$$

$$\frac{\partial h_{t-1}}{\partial W_{hh}} = \left( \frac{\partial h_{t-1}}{\partial W_{hh}} \right)_{h_{t-2}} + \frac{\partial h_{t-1}}{\partial h_{t-2}} \frac{\partial h_{t-2}}{\partial W_{hh}}$$

In conclusion, the following equation holds (by substitution)

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \left( \sum_{k=1}^t \left( \prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j} \right) \left( \frac{\partial h_k}{\partial W_{hh}} \right)_{h_{k-1}} \right)$$

# Backpropagation Through Time (BPTT)

- Cont. It's also true that

$$\prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j} = \frac{\partial h_t}{\partial h_k}$$

That leads us to

$$\frac{\partial L_t}{\partial W_{hh}} = \sum_{k=1}^t \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \left( \frac{\partial h_k}{\partial W_{hh}} \right)_{h_{k-1}}$$

and

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^T \sum_{k=1}^t \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \left( \frac{\partial h_k}{\partial W_{hh}} \right)_{h_{k-1}}$$

# Backpropagation Through Time (BPTT)

- Cont. Similar to the mentioned way, we could compute derivative of  $L$  w.r.t  $W_{xh}, b_h$

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^T \sum_{k=1}^t \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \left( \frac{\partial h_k}{\partial W_{xh}} \right)_{h_{k-1}}$$

$$\frac{\partial L}{\partial b_h} = \sum_{t=1}^T \sum_{k=1}^t \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \left( \frac{\partial h_k}{\partial b_h} \right)_{h_{k-1}}$$

# Backpropagation Through Time (BPTT)

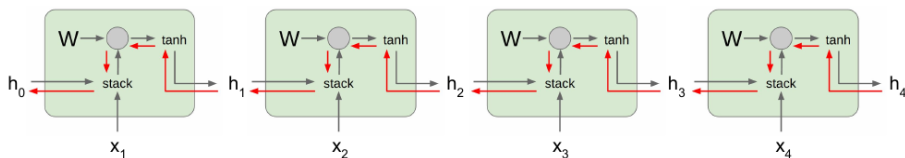


Figure: Vanilla RNN Gradient Flow, [source](#)

## ■ RNN Training Issues

### ► Exploding Gradients

What can we do to solve exploding?

### ► Vanishing Gradients

What can we do to solve vanishing?



# RNN Unit

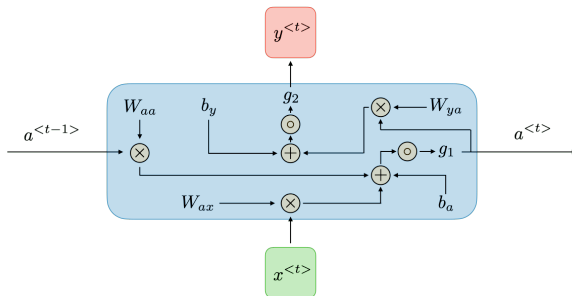


Figure: RNN Unit, [source](#)

- For a simple RNN unit we had:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

# Gated Recurrent Unit (GRU)

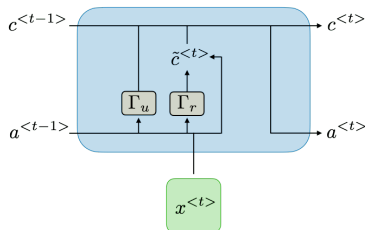


Figure: GRU Unit, [source](#)

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * a^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_r = \sigma(W_r[a^{<t-1>}, x^{<t>}] + b_r)$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

$$c^{<t>} = a^{<t>}$$

# Long Short-Term Memory (LSTM)

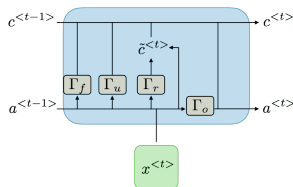


Figure: LSTM Unit, [source](#)

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * a^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_r = \sigma(W_r[a^{<t-1>}, x^{<t>}] + b_r)$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (\Gamma_f * c^{<t-1>})$$

$$a^{<t>} = \Gamma_o * c^{<t>}$$

# Type of Gates

- Update Gate  $\Gamma_u$ 
  - ▶ How much past should matter
- Relevance Gate  $\Gamma_r$ 
  - ▶ Drop previous information
- Forget gate  $\Gamma_f$ 
  - ▶ Erase a cell or not
- Output gate  $\Gamma_o$ 
  - ▶ How much to reveal of a cell

# LSTM Walk Through

- The first step in our LSTM is to decide what information we're going to throw away from the cell state.

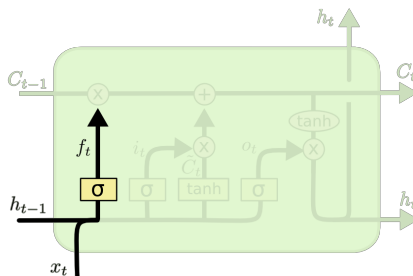


Figure: Forget Gate Layer, [source](#)

# LSTM Walk Through

- The next step is to decide what new information we're going to store in the cell state.

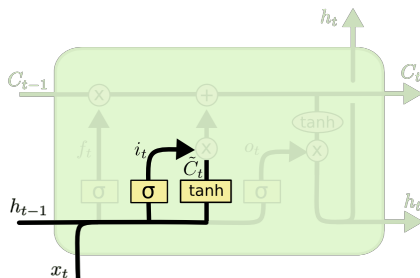


Figure: Input Gate Layer, [source](#)

# LSTM Walk Through

- It's now time to update the old cell state,  $C_{t-1}$ , into the new cell state  $C_t$ .

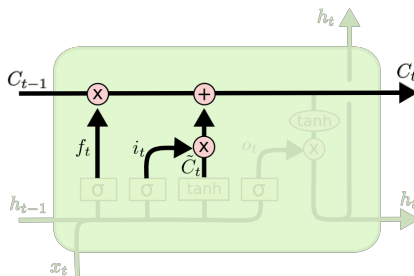


Figure: Update Cell State, [source](#)

# LSTM Walk Through

- Finally, we need to decide what we're going to output. This output will be based on our cell state

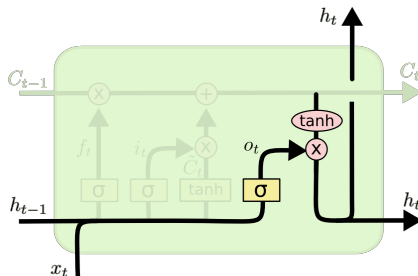


Figure: Output Gate Layer, **source**



# Why LSTMs?

- The LSTM does have the ability to remove and add information to the cell state.
- The gates in the previous slide let the information to pass through the units.
- The gates value are between zero and one and specify **how much information should be let through**.
- They also somehow solve the **vanishing gradient**.
- We can use more blocks of them so there will be more information to remember.

# How LSTMs solve vanishing gradients

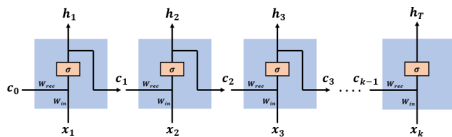


Figure: Simple Recurrent Neural Network, [source](#)

$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_k}{\partial W} = \frac{\partial L_k}{\partial h_k} \frac{\partial h_k}{\partial c_k} \cdots \frac{\partial c_2}{\partial c_1} \frac{\partial c_1}{\partial W} = \frac{\partial L_k}{\partial h_k} \frac{\partial h_k}{\partial c_k} \left( \prod_{t=2}^k \frac{\partial c_t}{\partial c_{t-1}} \right) \frac{\partial c_1}{\partial W}$$

$$\frac{\partial c_t}{\partial c_{t-1}} = \sigma'(W_{rec} \cdot c_{t-1} + W_{in} \cdot x_t) W_{rec}$$

# How LSTMs solve vanishing gradients

$$\frac{\partial L_k}{\partial W} = \frac{\partial L_k}{\partial h_k} \frac{\partial h_k}{\partial c_k} \left( \prod_{t=2}^k \sigma'(W_{rec} \cdot c_{t-1} + W_{in} \cdot x_t) W_{rec} \right) \frac{\partial c_1}{\partial W}$$

- For large  $K$  the gradient tends to vanish or if  $W_{rec}$  is large enough it causes exploding gradient which is solved by *Gradient Clipping*.

# How LSTMs solve vanishing gradients

- In LSTM we also have

$$\frac{\partial L_k}{\partial W} = \frac{\partial L_k}{\partial h_k} \frac{\partial h_k}{\partial c_k} \left( \prod_{t=2}^k \frac{\partial c_t}{\partial c_{t-1}} \right) \frac{\partial c_1}{\partial W}$$

- But

$$c^t = \Gamma_u * \tilde{c}^t + (\Gamma_f) * c^{t-1}$$
$$\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial \Gamma_f}{\partial c_{t-1}} \cdot c_{t-1} + \Gamma_f + \frac{\partial \Gamma_u}{\partial c_{t-1}} \cdot \tilde{c}_t + \frac{\partial \tilde{c}_t}{\partial c_{t-1}} \cdot \Gamma_u$$

- Consider that the *forget gate* is added to other terms and allows better control of gradient values. But it doesn't guarantee that there is no vanishing or exploding in gradient.

# Bidirectional RNN

- How to get information from future?

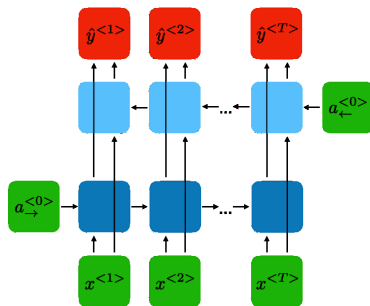


Figure: BRNN, source

$$y^{<t>} = g(W_y[\vec{a}^{<t>}, a^{<T-t>}] + b_y)$$

# Bidirectional RNN

- They are usually used in natural language processing.
- They are powerful for modeling dependencies between words and phrases in both directions of the sequence because every component of an input sequence has information from both the past and present.

# Back-propagation in BRNN

- It is exactly the same as simple RNN

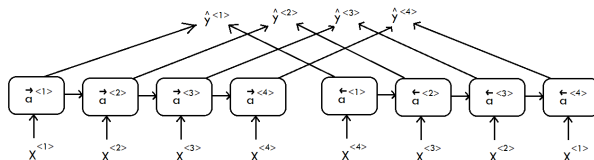


Figure: BRNN, [source](#)

# Encoder-Decoder

- If we want an output with a **different length** from the input (e.g. Machine Translation), what should we do?
- Which **alignment** method should we use?

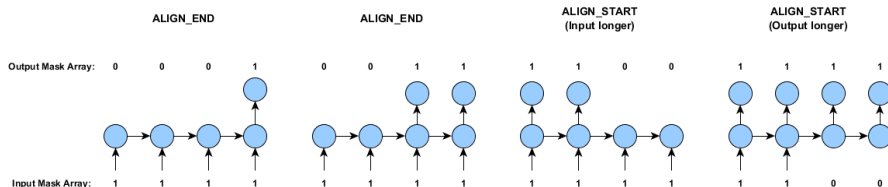


Figure: Problems in Sequence-to-Sequence Models, **source**



# Encoder-Decoder

Encoder-Decoder architecture could be a solution.

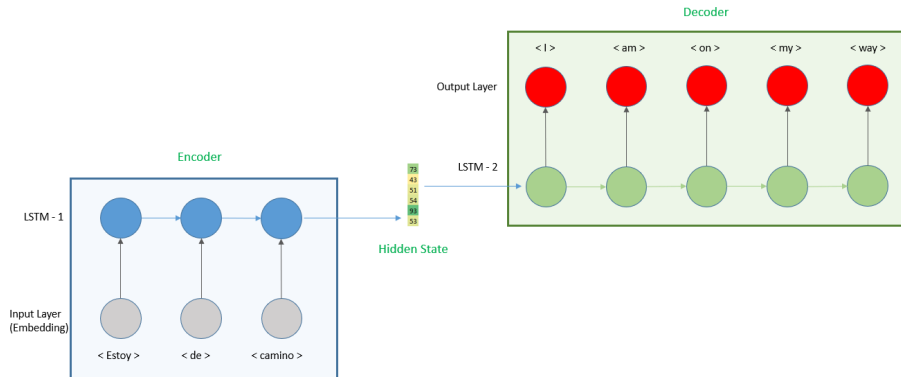


Figure: Encoder-Decoder, [source](#)

# Encoder-Decoder

## Encoder

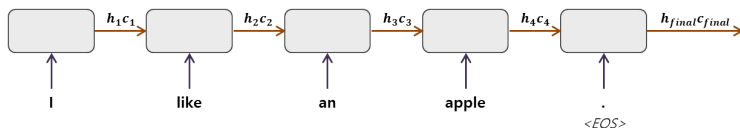


Figure: Encoder, **source**

- **Encoder:** The encoder processes the input sequence and compresses the information into a fixed length vector, known as the context vector.

# Encoder-Decoder

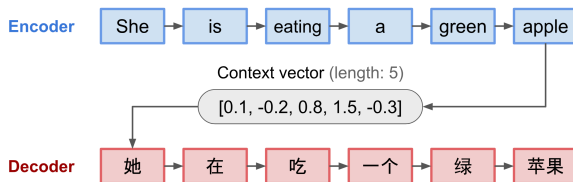


Figure: Context Vector, source

- **Context Vector (Hidden State):** This vector is expected to convey the meaning of the whole source sequence from the encoder to the decoder. (The early work considered the last state of the encoder network as the context vector.)

# Encoder-Decoder

## Decoder

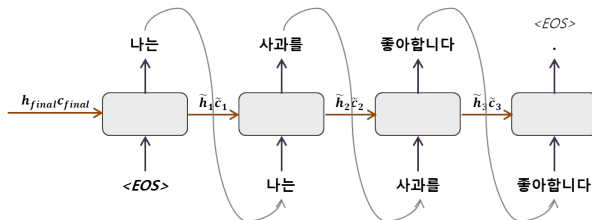


Figure: Decoder, source

- **Decoder:** The decoder uses the context vector to output the desired target sequence.

# Encoder-Decoder Limitations

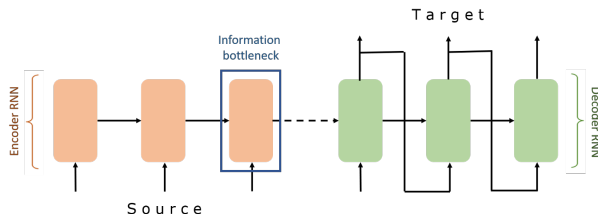


Figure: Bottleneck Phenomenon, source

- The context vector is **bottleneck**. By increasing the length of the input sequence, the model captures the essential information roughly. How to solve it? Attention! Define and use the the context vector better.

# Encoder-Decoder Limitations

- Slow convergence in training. How to solve it? Teacher Forcing.

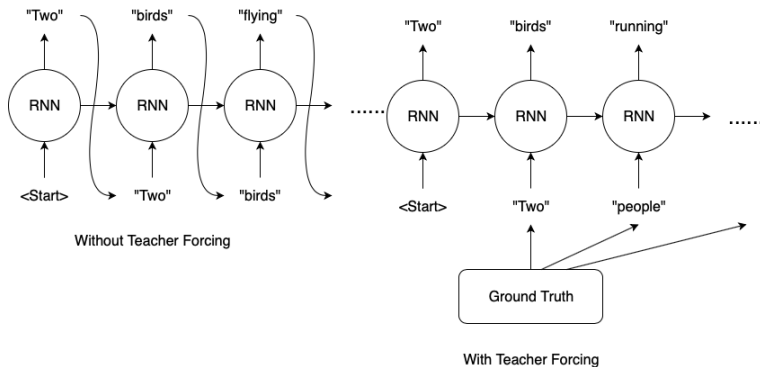


Figure: Teacher Forcing in Decoder Side, **source**

# Teacher Forcing

- Teacher forcing is a method for training recurrent neural networks more efficiently.
- Teacher forcing works by using the actual output at the current time step  $y^{(t-1)}$  as input in the next time step, rather than the  $o^{(t-1)}$  generated by the network.

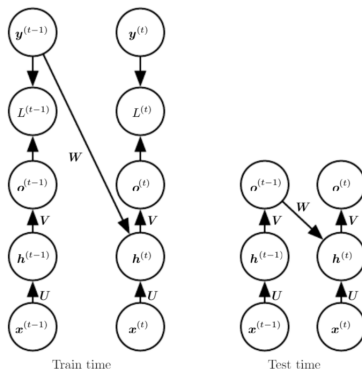


Figure: Teacher Forcing, [source](#)

# Teacher Forcing

- At the early stages of training, the predictions of the model are very bad. If we do not use Teacher Forcing, the hidden states of the model will be updated by a sequence of wrong predictions, **errors will accumulate** , and it is difficult for the model to learn from that.
- The problem with RNNs is that we need previous time step output as input for next time step.
- This technique allows us to prevent backpropagation through time which was complex and time-consuming.
- With teacher forcing the model will be **trained faster** .



# Search Strategy

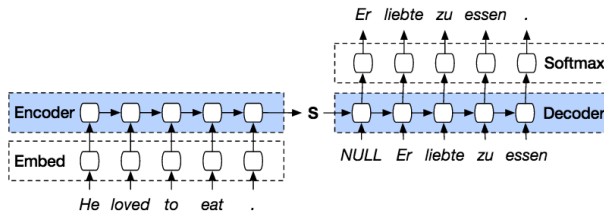


Figure: Simple Machine Translation Model, [source](#)

- By now, we have discussed some important elements of sequence-to-sequence models (e.g. Machine Translation), including word embedding, LSTM, encoder, decoder, context vector, and softmax. But, there is a missing part! The search strategy for inference.

$$\hat{y}_T, \hat{y}_{T-1}, \dots, \hat{y}_1 = \underset{\tilde{y}_T, \tilde{y}_{T-1}, \dots, \tilde{y}_1}{\operatorname{argmax}} P(\tilde{y}_T, \tilde{y}_{T-1}, \dots, \tilde{y}_1 | S)$$

**Notice:**  $\tilde{y}_t$  is the corresponding word to time step  $t$ .

# Search Strategy

## ■ Exhaustive Search (Brute-force Search)

Iterate over all possible combinations of  $\tilde{y}_t, y_{t-1}, \dots, \tilde{y}_1$ .

- ▶ T: Total Time Step (the number of decoder units)
- ▶ V: Vocabulary Size (the number of possible words)
- ▶ Time Complexity:  $O(V^T)$
- ▶ So, it's not feasible.

# Search Strategy

## ■ Greedy Search

$$\max P(y_T, y_{T-1}, \dots, y_1 | S) = \max \prod_{t=1}^T P(y_t | y_{t-1}, y_{t-2}, \dots, y_1, S)$$

$$\max \prod_{t=1}^T P(y_t | y_{t-1}, y_{t-2}, \dots, y_1, S) \approx \prod_{t=1}^T \max P(y_t | y_{t-1}, y_{t-2}, \dots, y_1, S)$$

Time step	1	2	3	4
A	0.5	0.1	0.2	0.0
B	0.2	0.4	0.2	0.2
C	0.2	0.3	0.4	0.2
<eos>	0.1	0.2	0.2	0.6

Figure: Greedy Search Example, [source](#)

$$\begin{aligned}
 P((A, B, C, eos) | S) &= P(A | (), S) P(B | (A), S) P(C | (A, B), S) P(eos | (A, B, C), S) \\
 &= 0.048
 \end{aligned}$$

# Search Strategy

## Greedy Search

### ► Advantages

- Time complexity:  $O(TV)$
- Sometimes it's a good approximation

### ► Disadvantages

- It is somehow too naive.
- It can be very inaccurate.

Time step	1	2	3	4
A	0.5	0.1	0.1	0.1
B	0.2	0.4	0.6	0.2
C	0.2	0.3	0.2	0.1
<eos>	0.1	0.2	0.1	0.6

Figure: Greedy Search Example, [source](#)

$$\begin{aligned}
 P((A, C, B, eos)|S) &= P(A|(), S)P(C|(A), S)P(B|(A, C), S)P(eos|(A, C, B), S) \\
 &= 0.054
 \end{aligned}$$

# Beam Search

## ■ Beam Search

It's something between the two previous methods.

- Keeping a number of candidates (beam width) instead of one in Greedy Search and all of the combinations in Exhaustive Search.

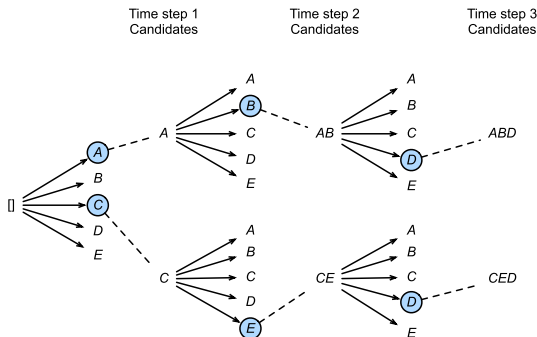


Figure: A Beam Search example in which beam width equals 2, [source](#)

# Beam Search

## ■ Beam Search

It's something between the two previous methods.

- ▶  $k$ : Beam Width
- ▶ Time Complexity:  $O(\log(k)kVT)$  (By using max heap in each time step)

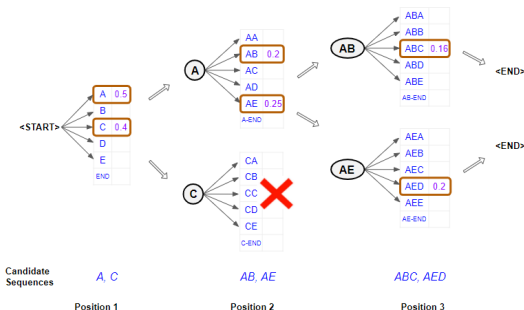


Figure: Another Beam Search example in which beam width equals 2, [source](#)

# References

- <https://lilianweng.github.io/posts/2018-06-24-attention/>
- <https://mmuratarat.github.io/2019-02-07/bptt-of-rnn>
- [https://d2l.ai/chapter\\_recurrent-modern/encoder-decoder.html](https://d2l.ai/chapter_recurrent-modern/encoder-decoder.html)
- [https://d2l.ai/chapter\\_recurrent-modern/seq2seq.html](https://d2l.ai/chapter_recurrent-modern/seq2seq.html)
- [https://d2l.ai/chapter\\_recurrent-modern/beam-search.html](https://d2l.ai/chapter_recurrent-modern/beam-search.html)

**Thank You!**

**Any Question?**