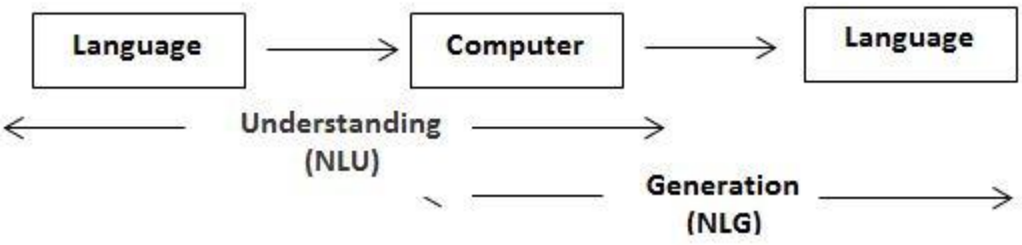
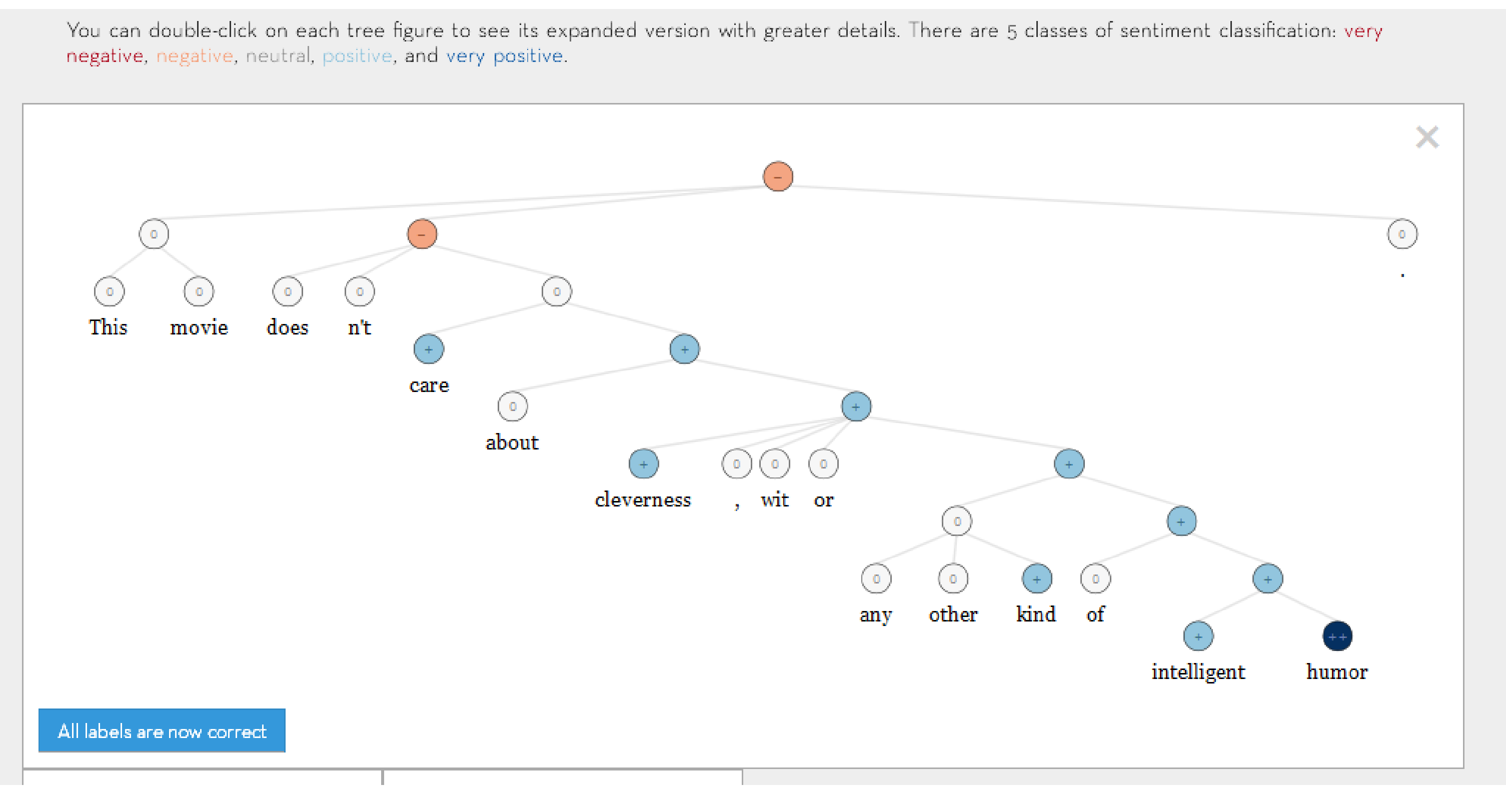
Overview



- NLP usually includes 2 parts: NLU (Understanding) and NLG (Generation)
- Application
    - Spell Checking
    - Spam Detection
    - Part-of-speech tagging: Colorless(Adj) green(Adj) ideas(Noun) sleep(Verb) furiously(Adv)
    - Name Entity Recognition: Einstein(Person) met with UN(Organization) officials in Princeton(Location)
    - Sentiment Analysis
    - Coreference Resolution: Carter told Peter he shouldn't run again → he here is Carter or Peter
    - Word Sense Disambiguation: I need new batteries for my mouse
    - Parsing
    - Machine Translation: Need to process both NLU and NLG
    - Information Extraction: You're invited to our dinner party, Friday May 27 at 5:30 → Party May 27 add
    - Paraphrase, Summarization
    - Question and Answering
- Sentiment analysis



- Use RNN to encode input sentence and decode vector to output sentence ?

Frequency based Embedding

CountVectorizer

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int64)
```

- We have N unique tokens in D documents (D1, D2, …, Dn). We have D x N matrix, each row in the matrix contains the frequency of tokens in document Di. For example,

D1: He is a lazy boy. She is also lazy

D2: Dick is a lazy person

The 2 x 6 matrix:

	He	She	lazy	boy	Dick	person
D1	1	1	2	1	0	0
D2	0	0	1	0	1	1

- In practice, the matrix can be very sparse and inefficient for computation

→ Vector for 'he': (1, 0)

TFIDF for 3 documents

- Common words like 'is', 'the', 'a' etc. tend to appear quite frequently in comparison to the words which are important to a document
- For example, the document A about Messi must have lots of word Messi, but the common words like 'the' is also going to present in higher frequency
- TFIDF will penalize the common words by assigning them lower weights while giving more importance to words like Messi in a particular document

Document 1: The game of life is a game of everlasting learning

Document 2: The unexamined life is not worth living

Document 3: Never stop the learning

- Step 1: Term Frequency: measures the number of times word occurs in a document
  - **TF for Document 1**

Document1	**the**	**game**	**of**	**life**	**is**	**a**	**everlasting**	**learning**
Term Frequency	1	2	2	1	1	1	1	1

  - **TF for Document 2**

Document2	**the**	**unexamined**	**life**	**is**	**not**	**worth**	**living**
Term Frequency	1	1	1	1	1	1	1

  - **TF for Document 3**

Document3	**never**	**stop**	**learning**	**the**
Term Frequency	1	1	1	1

- **Normalized TF for Document 1**

Document1	**the**	**game**	**of**	**life**	**is**	**a**	**everlasting**	**learning**
Normalized TF	$\frac{1}{10}$	$\frac{2}{10}$	0.2	0.1	0.1	0.1	0.1	0.1

- **Normalized TF for Document 2**

Document2	**the**	**unexamined**	**life**	**is**	**not**	**worth**	**living**
Normalized TF	$\frac{1}{7}$	0.142857	0.142857	0.142857	0.142857	0.142857	0.142857

- **Normalized TF for Document 3**

Document3	**never**	**stop**	**learning**	**the**
Normalized TF	$\frac{1}{4}$	0.25	0.25	0.25

- Step 2: Inverse Document Frequency:
    - **IDF(the) = $\log_e$(Total Number Of Documents / Number Of Documents with term game in it)= $\log_e$(3 / 3)=0**
    - If a word has appeared in all documents, then probably that word is not relevant to a particular document, but if it has appeared in a subset of documents then probably the word may relevant to the documents it is present in
- Step 3: TF * IDF

→ vector for ‘the’ = (0, 0, 0)

Co-occurrence Matrix

- Idea: Similar words tend to occur together and will have similar context. E.g. Apple is a fruit. Mango is a fruit → Apple and mango tend to have a similar context
- Here we use Context Windows of 2 and the corpus: He is not lazy. He is intelligent. He is smart

	He	is	not	lazy	intelligent	smart
He	0	4	2	1	2	1
is	4	0	1	2	2	1
not	2	1	0	1	0	0
lazy	1	2	1	0	0	0
intelligent	2	2	0	0	0	0
smart	1	1	0	0	0	0

He	is	not	lazy	He	is	intelligent	He	is	smart
He	is	not	lazy	He	is	intelligent	He	is	smart
He	is	not	lazy	He	is	intelligent	He	is	smart
He	is	not	lazy	He	is	intelligent	He	is	smart

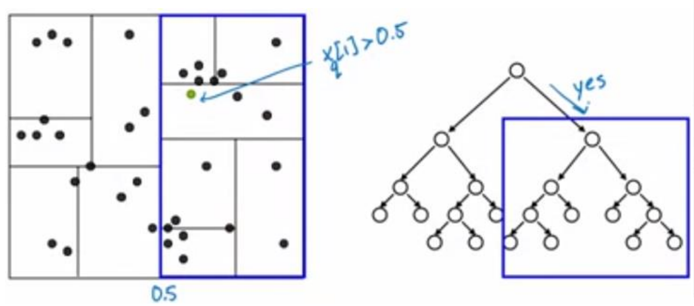
- While the word ‘lazy’ has never appeared with ‘intelligent’ in the context window, therefore assigned 0 in the blue box

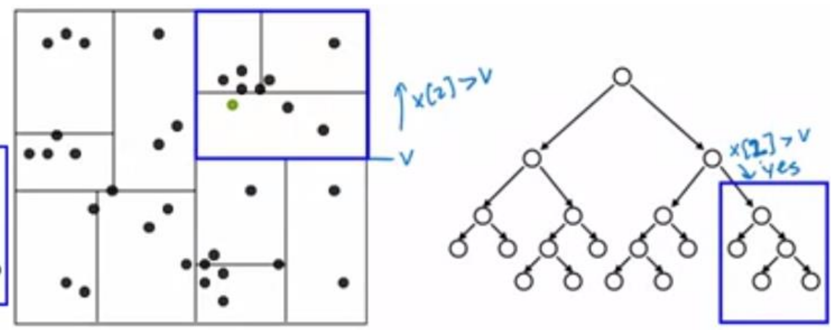
- If we have N unique tokens, vocabulary size = N, the co-occurrence matrix is N x N, so it is very large. To partially solve it, we can construct the M x N co-occurrence matrix with M is the subset of N by removing irrelevant words like stop words, but it is still very large and difficult to compute
- However, **this co-occurrence matrix is not the word vector representation**
- But, if it performs PCA on N x N matrix, you can choose k components out of these N components so the matrix will be N x k matrix
- PCA will decompose co-occurrence matrix into 3 matrices = $U \bullet S \bullet V^T$, $U \bullet S$ **is the word vector representation**

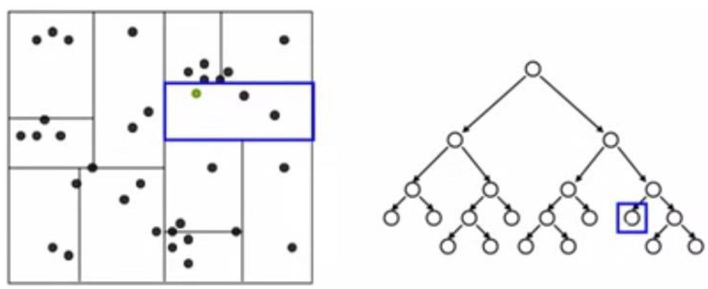
- Advantage
    - It preserves the semantic relationship between words. i.e man and woman tend to be closer than man and apple
    - It uses SVD at its core, which produces more accurate word vector representations than existing methods.

KD Tree for Nearest Neighbor Search

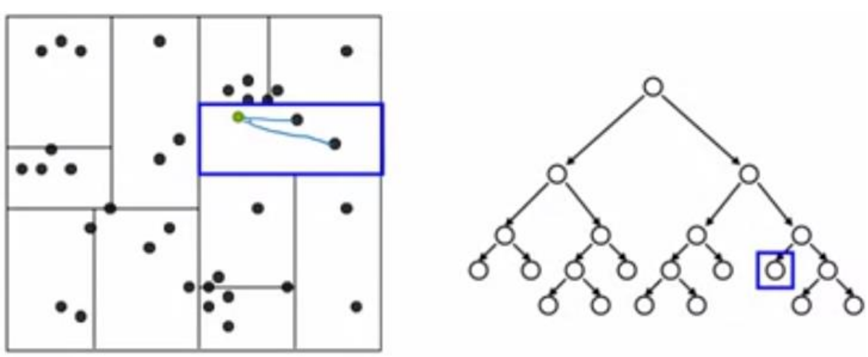
- For the dataset, you partition based on the x and y axis. Based on you partitioning, you draw the binary tree to keep track
- Step 1: Start by exploring leaf node containing query point



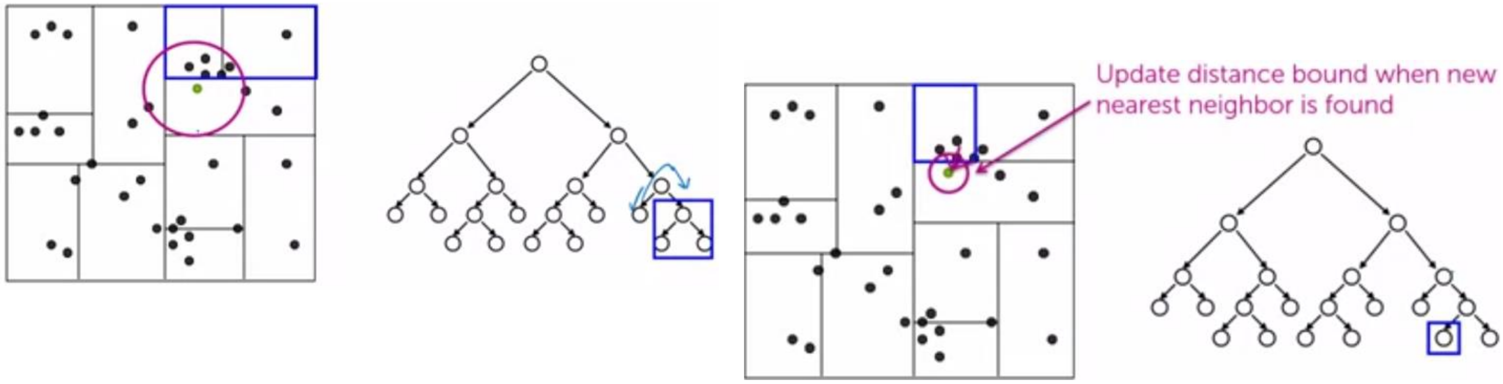


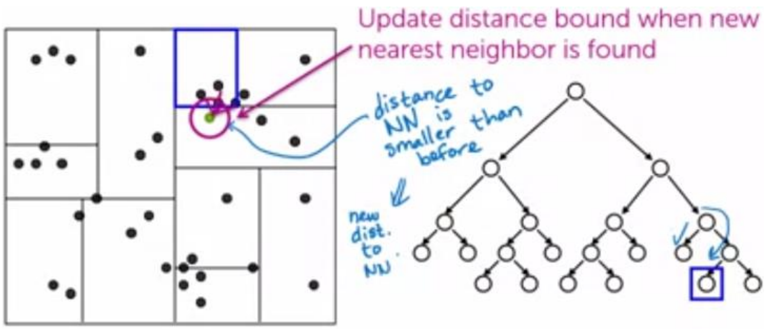


- Step 2; Compute distance to each other point at leaf node

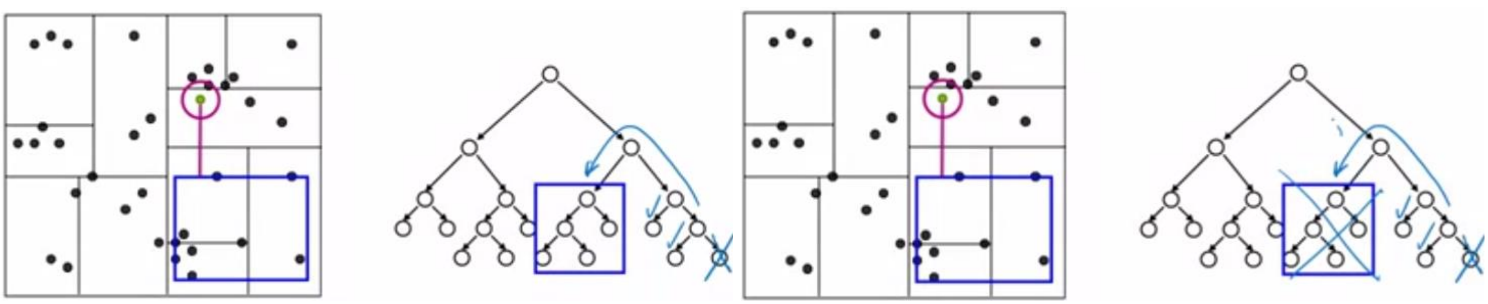


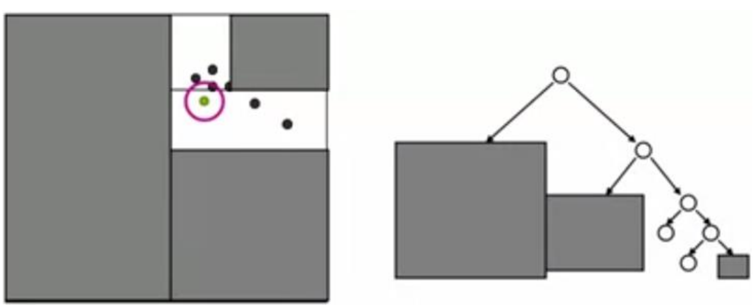
    - But in this case, update threshold the distance between query point and points in your considering leaf nodes. Find another leaf node
- Step 3: Backtrack and try other branch at each node visited





- Step 4: Use your updated threshold to prune the tree





Word2Vec/ Prediction based Vector

- **Transform single word to vector** is very important in NLP
- Word2vec helps you to do it, it includes 2 models:
  - Skip-gram
  - Continuous Bag-of-words
- Both of these are shallow neural network which map words to the input. They learn weights which act as **word vector representation**

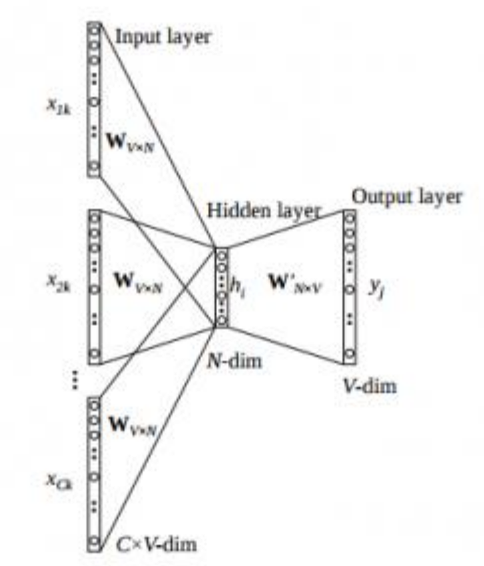
Continuous Bag of words (CBOW)

- The way CBOW work is that it tends to predict the probability of a word given a context (n words before and after). Consider example with context size of 1, we have corpus C = ‘Hey, this is sample corpus using only one context word’
- The corpus may be converted into

Input	Output			Hey	This	is	sample	corpus	using	only	one	context	word
Hey	this		Datapoint 1	1	0	0	0	0	0	0	0	0	0
this	hey		Datapoint 2	0	1	0	0	0	0	0	0	0	0
is	this		Datapoint 3	0	0	1	0	0	0	0	0	0	0
is	sample		Datapoint 4	0	0	1	0	0	0	0	0	0	0
sample	is		Datapoint 5	0	0	0	1	0	0	0	0	0	0
sample	corpus		Datapoint 6	0	0	0	1	0	0	0	0	0	0
corpus	sample		Datapoint 7	0	0	0	0	1	0	0	0	0	0
corpus	using		Datapoint 8	0	0	0	0	1	0	0	0	0	0
using	corpus		Datapoint 9	0	0	0	0	0	1	0	0	0	0
using	only		Datapoint 10	0	0	0	0	0	1	0	0	0	0
only	using		Datapoint 11	0	0	0	0	0	0	1	0	0	0
only	one		Datapoint 12	0	0	0	0	0	0	1	0	0	0
one	only		Datapoint 13	0	0	0	0	0	0	0	1	0	0
one	context		Datapoint 14	0	0	0	0	0	0	0	1	0	0
context	one		Datapoint 15	0	0	0	0	0	0	0	0	1	0
context	word		Datapoint 16	0	0	0	0	0	0	0	0	1	0
word	context		Datapoint 17	0	0	0	0	0	0	0	0	0	1

The target for a single datapoint say Datapoint 4 is shown as below?

Hey	this	is	sample	corpus	using	only	one	context	word
0	0	0	1	0	0	0	0	0	0



- The input and output layer are one-hot vector with size of 1 x 10
- 2 weight matrix: one between the input and hidden layer V x N = 10 x 4, second between hidden and output N x V = 4 x 10
- No activation at any layers
- The hidden input gets multiplied by hidden- output weights and output is calculated.
- Error between output and target is calculated and propagated back to re-adjust the weights.
- The weight between the hidden layer and the output layer is **taken as the word vector representation of the word.**

- If input layer is 'this' and target is 'sample'

Context

C1	this	0	1	0	0	0	0	0	0	0	0

Input-Hidden Weight

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32
33	34	35	36
37	38	39	40

Hidden Activation

5	6	7	8

- If input layer is 'this', 'corpus', 'context' and the target is 'sample', it's different with above because we take average in hidden layer

Context

C1	this	0	1	0	0	0	0	0	0	0	0
C2	corpus	0	0	0	0	1	0	0	0	0	0
C3	context	0	0	0	0	0	0	0	0	1	0

Input-Hidden Weight

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32
33	34	35	36
37	38	39	40

Hidden Activation

5	6	7	8
17	18	19	20
33	34	35	36

Average hidden Activation

18.33333333	19.33333333	20.33333333	21.33333333

- Disadvantage
  - CBOW takes the average of the context of a word (as seen above in calculation of hidden activation)

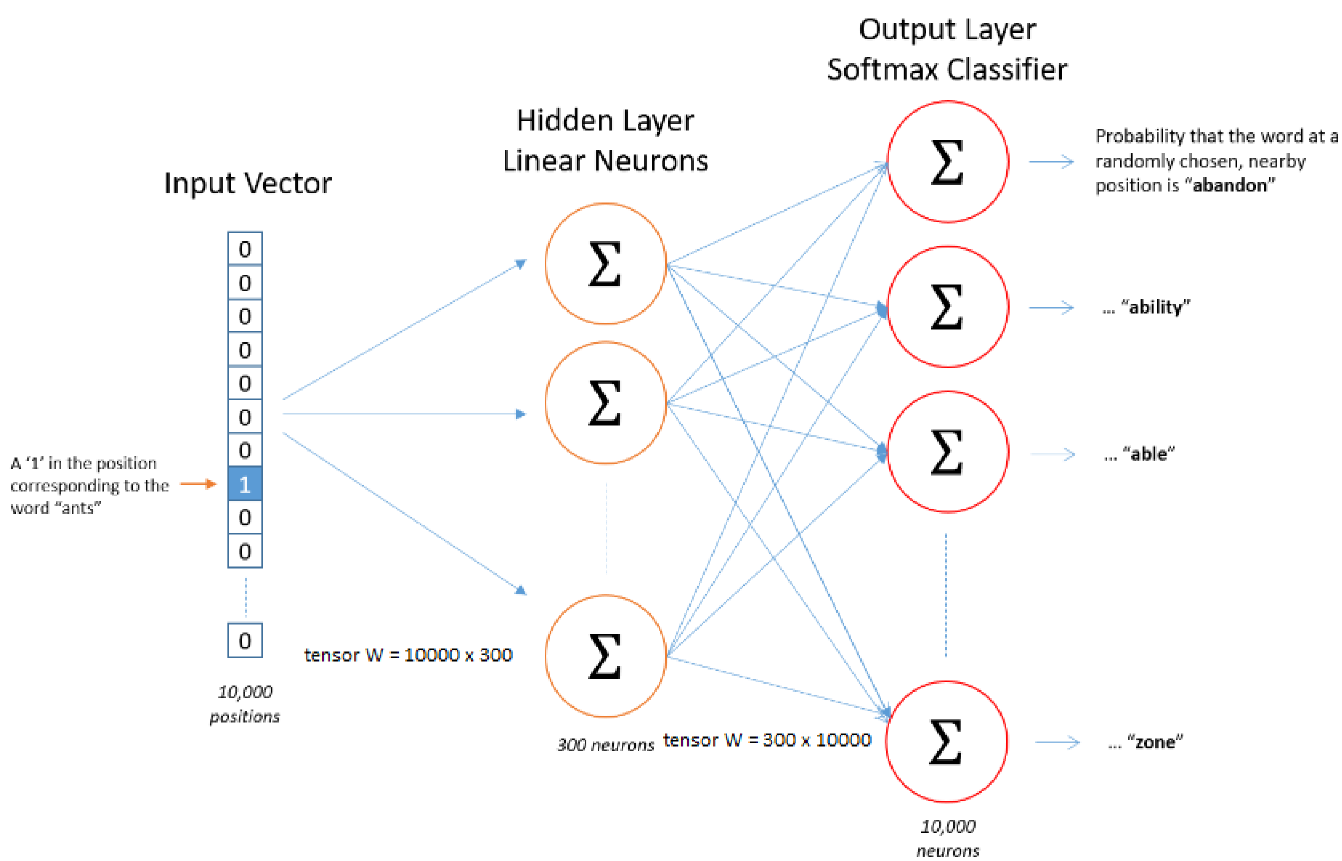
- For example, Apple can be both a fruit and a company but CBOW takes an average of both the contexts and places it in between a cluster for fruits and companies

Skip-gram

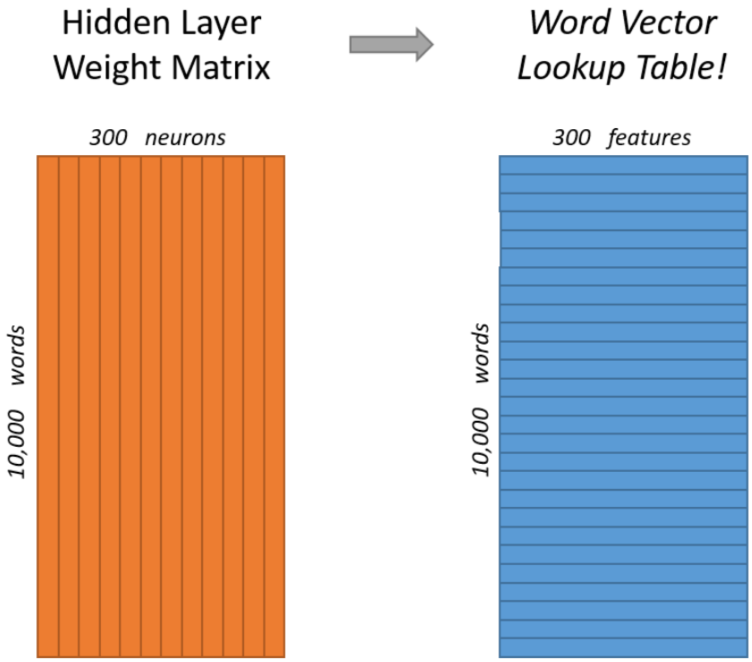
- Given the word, predict the n words before and n words after based on the probability
    - For example, if you gave the trained network the input word "Soviet", the output probabilities are going to be much higher for words like "Union" and "Russia" than for unrelated words like "watermelon" and "kangaroo".
    - The weight between the input and hidden layer is **taken as the word vector representation of the word**
- First of all, you cannot feed the raw text string to neural network. For example, you have corpus (list of sentence):
    - The ants eat my cake
    - You are my boy friend

	The	ants	eat	my	cake	You	are	boy	friend
The	1	0	0	0	0	0	0	0	0
ants	0	1	0	0	0	0	0	0	0

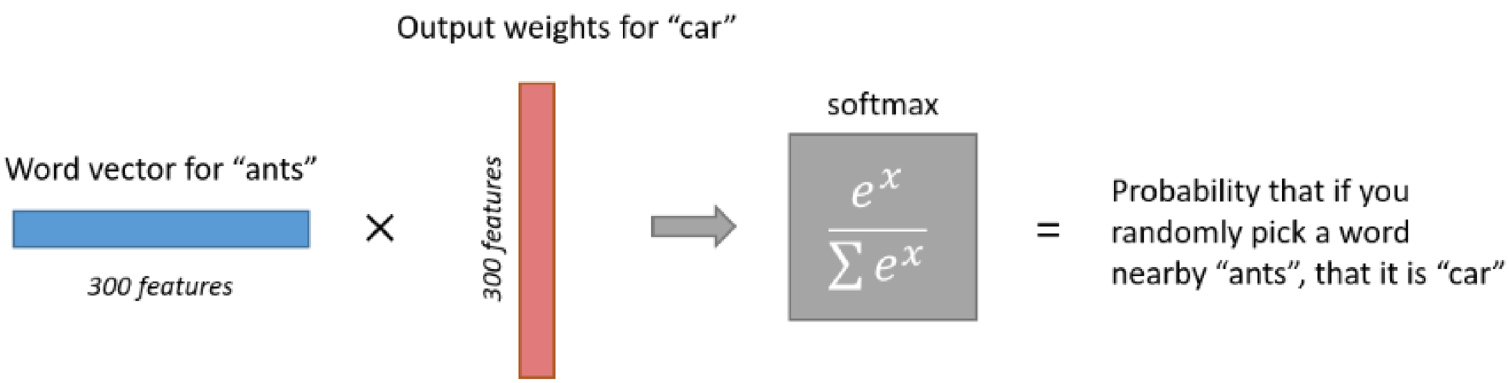
    - Create one-hot for ants: $\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
- In our example, we have 1 rows x 10000 components/ columns (one for every word in our vocabulary) for ants in input of NN, and output is single vector with 10000 components, the probability that a randomly selected nearby word is that vocabulary word

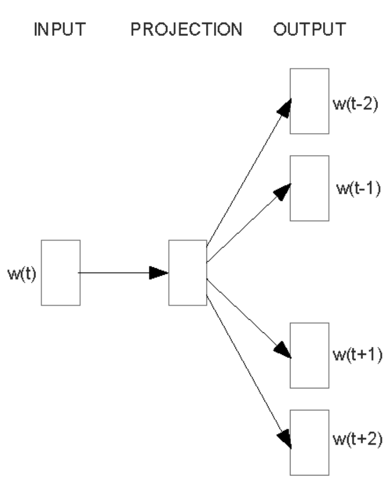


- There is **no activation function** on the hidden layer neurons, but the output neurons use softmax
- Every word is represented for class and 300 neurons means 300 features that a word include, 300 features is just hyperparameter, we need to tune model to find out



- So the end goal of all of this is really just to **learn this hidden layer weight matrix** – the output layer we'll just toss when we're done
- Because our input is one-hot vector which is almost all zeros, if you multiply 1 x 10000 one hot to hidden layer matrix 10000 x 300, the output is just the word vector for the input word
- For the output layer, here's an illustration of calculating the output of the output neuron for the word "car"





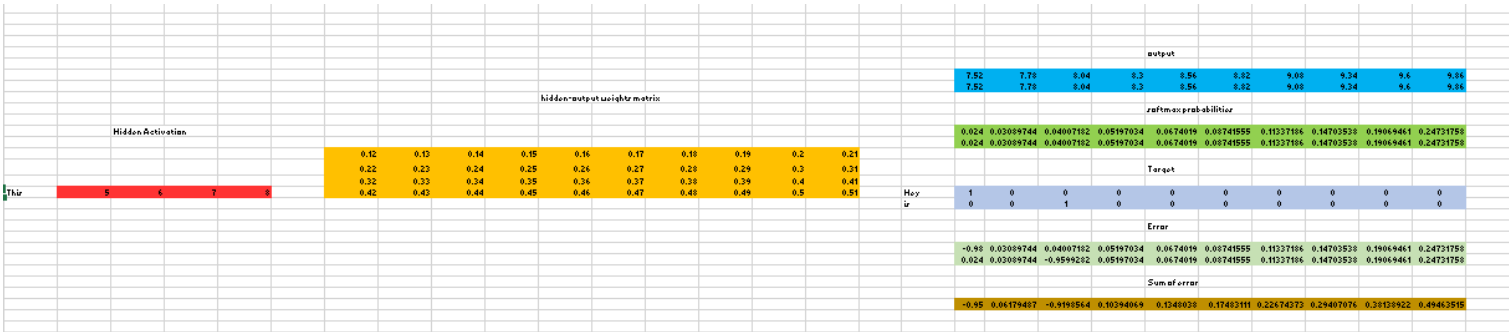
Skip-gram

- So, we use 1 hidden layer and 1 softmax as output layer:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c,\, j\ne 0} \log p\left(\omega_{t+j} \mid \omega_t\right)$$

- T: number of subset 2c+1 in the corpus

- Another example, continue example in CBOW, take 'this' as input and 'Hey', 'is' as target



- The red row in second picture is hidden layer. You multiply it to hidden-output weight matrix, you will get 1 row, but because you have 2 targets, so you must duplicate this row (blue matrix)
- Error is calculated by substracting the grey matrix(target) to green matrix(output) element-wise
- Advantage:
  - Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit

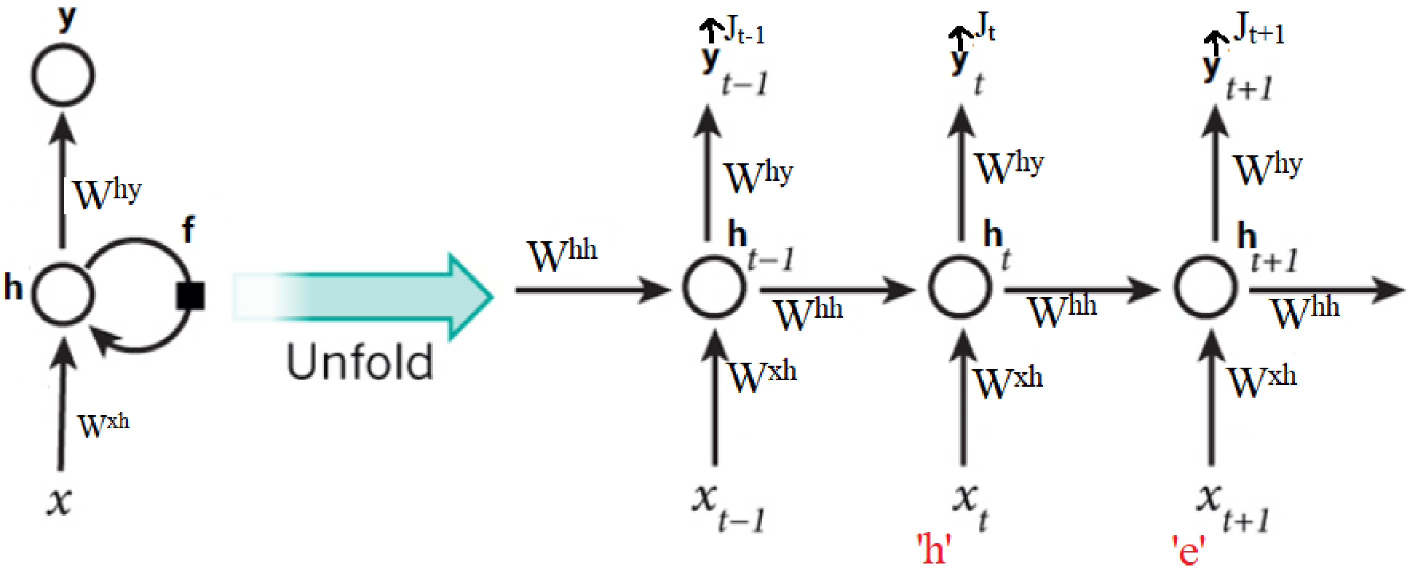
Phrase Representation

$$score(\omega_i,\omega_j)=\frac{count(\omega_i\omega_j)-\delta}{count(\omega_i)count(\omega_j)}$$

Recurrent Neural Network

Language Model

- Language Model calculates the probability of the string in the corpus: $p(\omega_1,...,\omega_T)$
- Given i-1 words, predict the word #i, conditional probabilities: $p(\omega_1,...,\omega_T)=\prod_{i=1}^{m}p(\omega_i\,|\,\omega_1,...,\omega_{i-1})$
- Markov Assumption helps to calculate $p(\omega_2\,|\,\omega_1)=\frac{count(\omega_1,\omega_2)}{count(\omega_1)}$ or $p(\omega_3\,|\,\omega_1,\omega_2)=\frac{count(\omega_1,\omega_2,\omega_3)}{count(\omega_1,\omega_2)}$
- Because Language Model need to calculate $count(\omega_1,\omega_2)$, $count(\omega_1,\omega_2,\omega_3)$ so the complex will be explode exponentially
- Markov Order: the number of word we consider to predict the target word
  - Markov-order = 1, $p(\omega_2\,|\,\omega_1)$
  - Markov-order = 2, $p(\omega_3\,|\,\omega_1,\omega_2)$



- Given a list of word vectors: $x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T$
- At a single step
    - $h_t = \sigma\left(W^{hh} h_{t-1} + W^{xh} x_t\right)$
    - $\hat{y}_t = soft\max\left(W^{hy} h_t\right)$
    - $\hat{P}\left(x_{t+1} = v_j \mid x_t, \ldots, x_1\right) = \hat{y}_{t,j}$
- Feed Forward in RNN: Instead of words, we just consider characters, given {h, e, l, l}, predict the next character and the target is {o}
- ddd h'_{t+1}

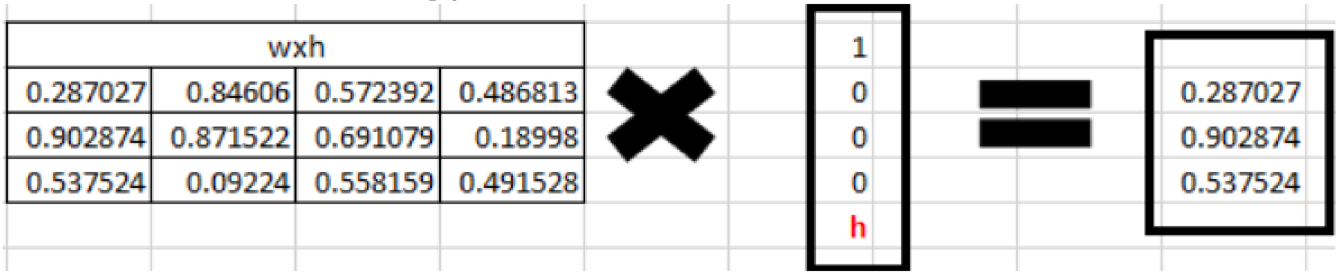
1	0	0	0
0	1	0	0
0	0	1	1
0	0	0	0
h	e	l	l

- First of all, we need to determine the dimension of h_{t-1}, h_t, h_{t+1} , here we choose $\mathbb{R}^{3x1}$, so $h_{t-1} = \begin{bmatrix}0 & 0 & 0\end{bmatrix}^T$. From the input, we can see the dimension of $x_t \in \mathbb{R}^{4x1}$
- To fulfil the first formula $h_t = \sigma\left(W^{hh} h_{t-1} + W^{xh} x_t\right) = \sigma\left(W\left[h_{t-1}, x_t\right] + b\right)$,(σ: tanh function) W^{hh} could be followed 2 approaches:
    - W^{hh} includes 3x3 matrix: $\begin{bmatrix}0.427 & 0 & 0 \\ 0 & 0.427 & 0 \\ 0 & 0 & 0.427\end{bmatrix}$ and 3x1 matrix bias: $\begin{bmatrix}0.567 \\ 0.567 \\ 0.567\end{bmatrix}$, so

      $$W_{hh} h_{t-1} = \begin{bmatrix}0.427 & 0 & 0 \\ 0 & 0.427 & 0 \\ 0 & 0 & 0.427\end{bmatrix}\begin{bmatrix}0 \\ 0 \\ 0\end{bmatrix} + \begin{bmatrix}0.567 \\ 0.567 \\ 0.567\end{bmatrix}$$

    - $W^{hh} \in \mathbb{R}^{3x4} = \begin{bmatrix}0.427 & 0 & 0 & 0.567 \\ 0 & 0.427 & 0 & 0.567 \\ 0 & 0 & 0.427 & 0.567\end{bmatrix}$ and add one to $h_{t-1} = \begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^T$
- In this case, to make it simpler, we will choose first approach

- Randomize the initial W^{xh} and multiply to x_t



wxh			
0.287027	0.84606	0.572392	0.486813
0.902874	0.871522	0.691079	0.18998
0.537524	0.09224	0.558159	0.491528

- $W \in \mathbb{R}^{3x7} = \begin{bmatrix} 0.427 & 0 & 0 & 0.287 & 0.84 & 0.572 & 0.486 \\ 0 & 0.427 & 0 & 0.902 & 0.871 & 0.691 & 0.189 \\ 0 & 0 & 0.427 & 0.537 & 0.092 & 0.558 & 0.491 \end{bmatrix}$

- $W^{hh}h_{t-1} + W^{xh}x_t = \left(\begin{bmatrix} 0.427 & 0 & 0 \\ 0 & 0.427 & 0 \\ 0 & 0 & 0.427 \end{bmatrix} \mathbf{x} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.567 \\ 0.567 \\ 0.567 \end{bmatrix}\right) + \begin{bmatrix} 0.287 \\ 0.902 \\ 0.537 \end{bmatrix} = \begin{bmatrix} 0.85 \\ 1.46 \\ 1.10 \end{bmatrix}$

- $h_t = \sigma\left(W^{hh}h_{t-1} + W^{xh}x_t\right) = \begin{bmatrix} 0.7 & 0.81 & 0.75 \end{bmatrix}^T$

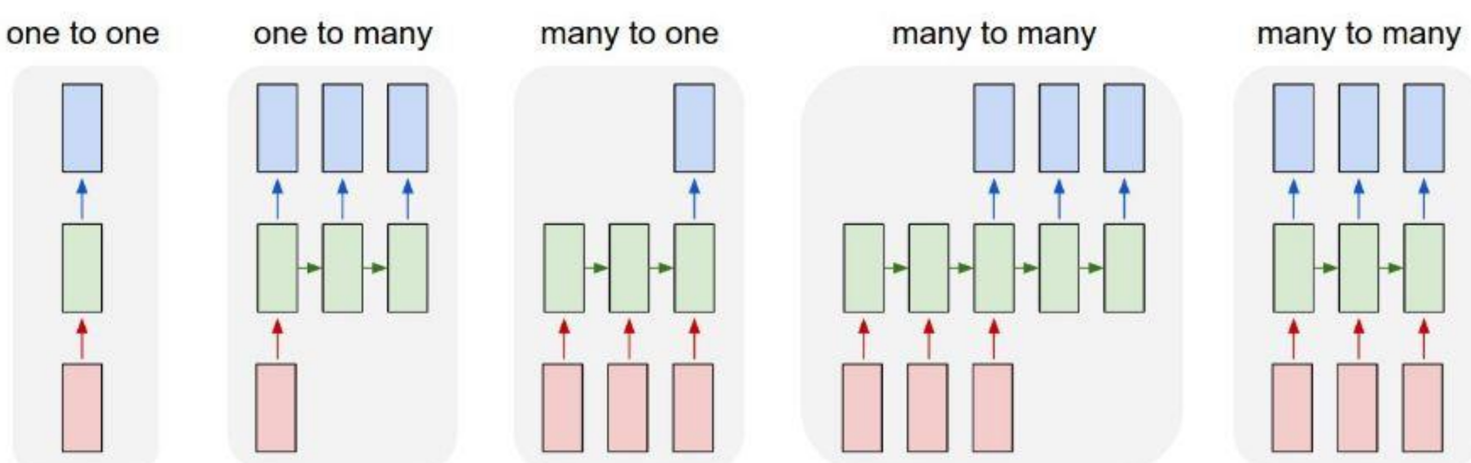
- Randomize initial weight $W^{hy} = \begin{bmatrix} 0.37 & 0.97 & 0.83 \\ 0.39 & 0.28 & 0.65 \\ 0.64 & 0.09 & 0.33 \\ 0.91 & 0.32 & 0.14 \end{bmatrix}$,

$$\hat{y}_t = soft\max\left(W^{hy}\ \mathbf{x}\ h_t\right) = soft\max\left(\begin{bmatrix} 0.37 & 0.97 & 0.83 \\ 0.39 & 0.28 & 0.65 \\ 0.64 & 0.09 & 0.33 \\ 0.91 & 0.32 & 0.14 \end{bmatrix} x \begin{bmatrix} 0.7 \\ 0.81 \\ 0.75 \end{bmatrix}\right) = soft\max\left(\begin{bmatrix} 1.69 \\ 0.98 \\ 0.76 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 0.41 \\ 0.2 \\ 0.16 \\ 0.21 \end{bmatrix}$$

- At **t**, $\begin{cases} y_t = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T \rightarrow y_{t,0} = y_{t,'h'} = 1 \\ \hat{y}_t = \begin{bmatrix} 0.41 & 0.2 & 0.16 & 0.21 \end{bmatrix} \rightarrow \hat{y}_{t,0} = \hat{y}_{t,'h'} = 0.41 \end{cases}$,

$$J^{(t)} = -\sum_{j=1}^{|V|} y_{t,j} \log\left(\hat{y}_{t,j}\right) \rightarrow J = -\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \log\left(\hat{y}_{t,j}\right)$$

- Architecture

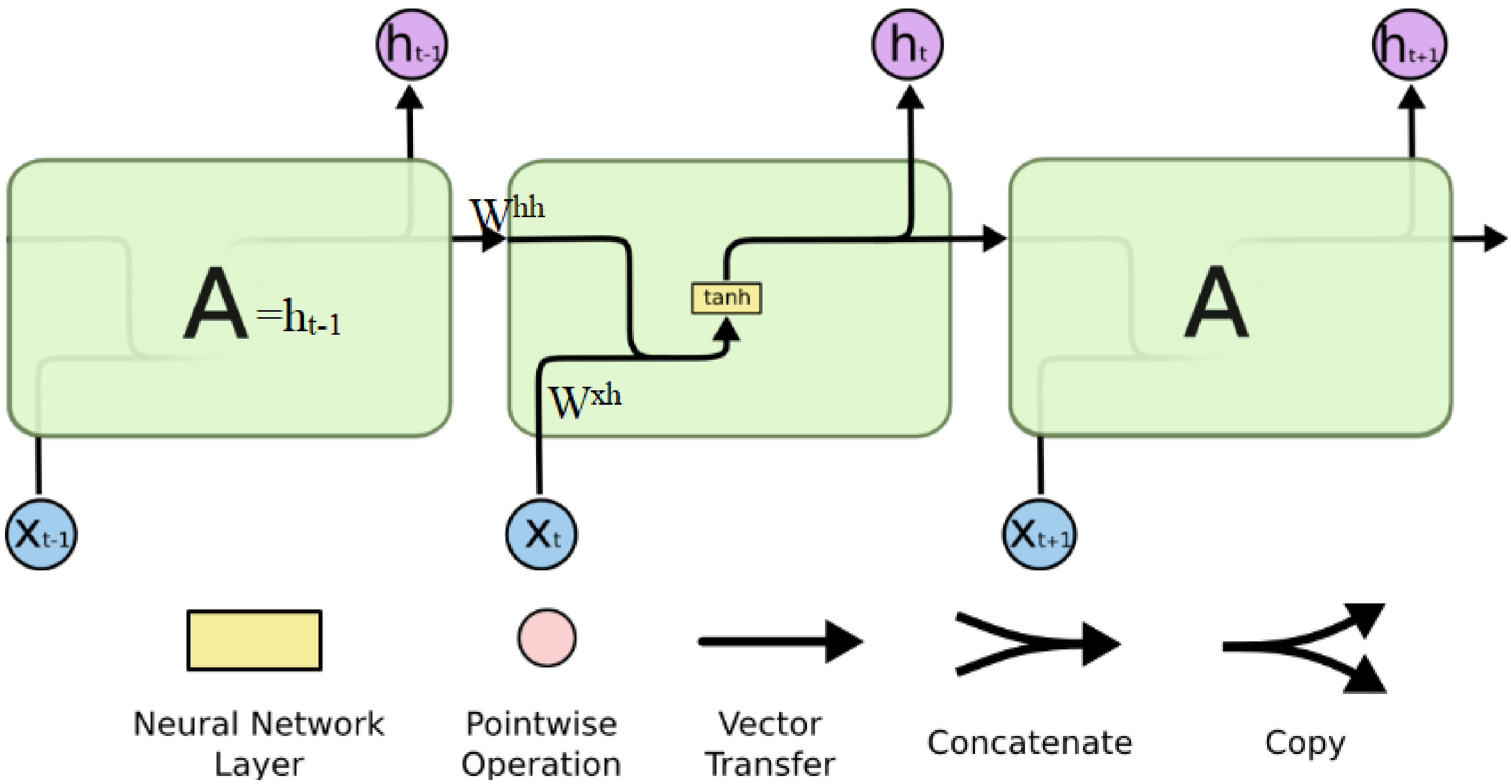


  - The example above is many to many

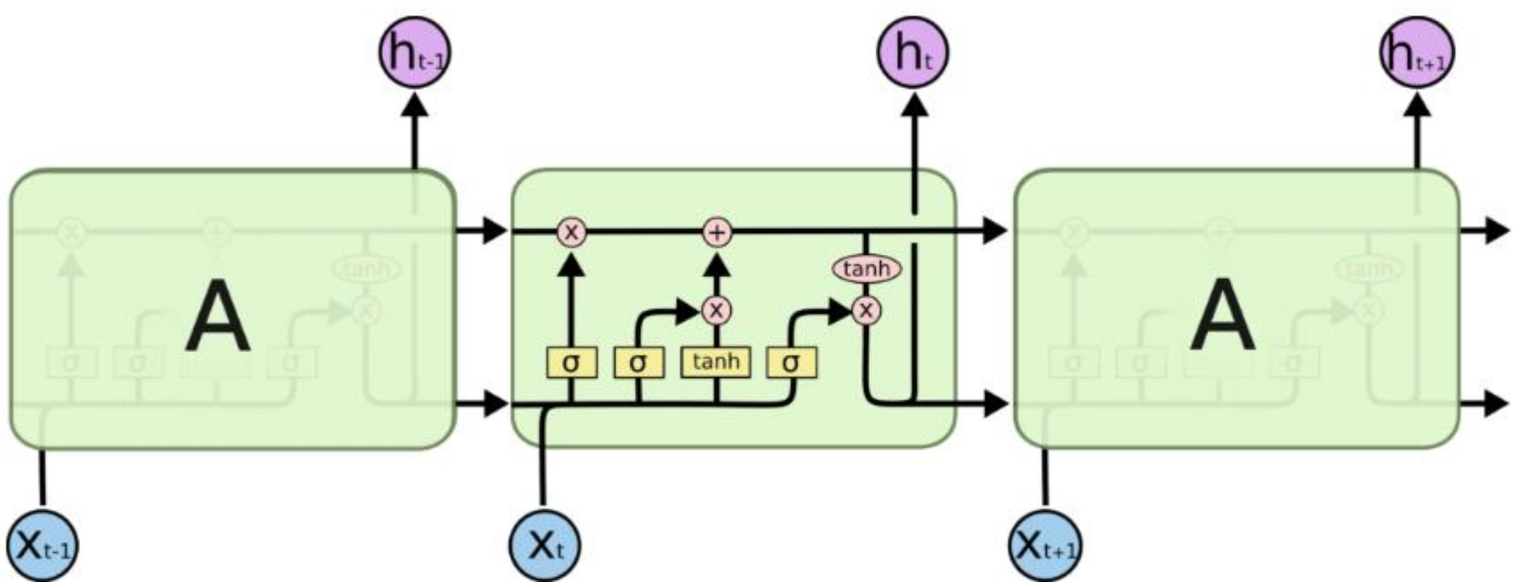
Problem in RNN

Long Term dependencies and Short Term attention

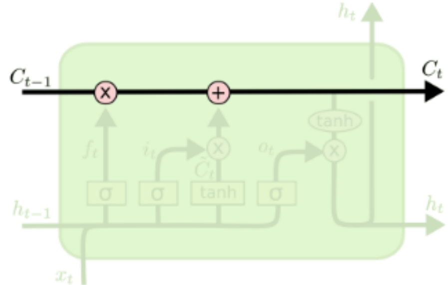
- In France, I had a great time and I learnt some of the …
- We can fill in this blank: 'French', but our model above is not trained to remember the key words here is 'France'. Generally, it is **not trained to capture the long-term dependencies**, so the word we predict will mostly depend in the previous few words, not much earlier ones
- **LSTM/ GRU helps RNN to calculate h_t**, capable of learning long-term dependencies
- Main idea in LSTM:
    - Create the memory to record the words before
    - Record the words before strongly or softly based on the word we are considering through gate or σ (here σ is sigmoid not tanh)



- Each lines carries an entire vector
- This figure is for standard RNN, below is LSTM

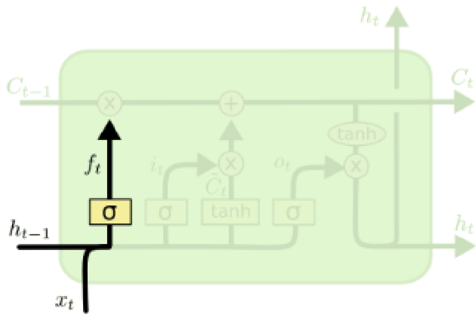


- LSTM also have the chain structure like standard RNN, but instead of single neural layer, they have four, interacting each other
- Another main difference is the **horizontal line through the top of the diagram (HL)** or the **cell state** in original diagram, so the LSTM can remove or add information from other parts of corpus
- Sigmoid layer describes how much the words should be added or removed
    - 0 means "completely get rid of this."
    - 1 means "completely keep this"



Step-by-Step LSTM

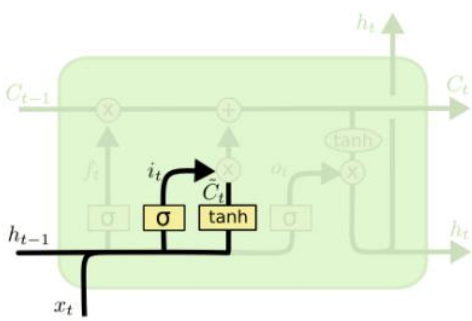
Forget gate



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

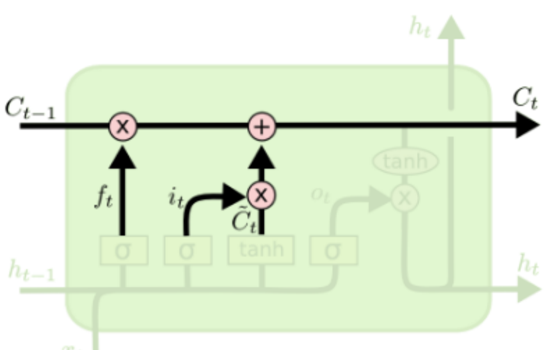
- In the first step, determine how much information in C_{t-1} should be forgot
- Example of a language model trying to predict the next word based on all the previous ones
    - When we see the new subject, we want to remove the gender of the old subject

Input gate



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

- Next step, determine the new information should be saved in cell state, E,g. add the gender of the new subject to the hidden state, to replace the old one we're forgetting

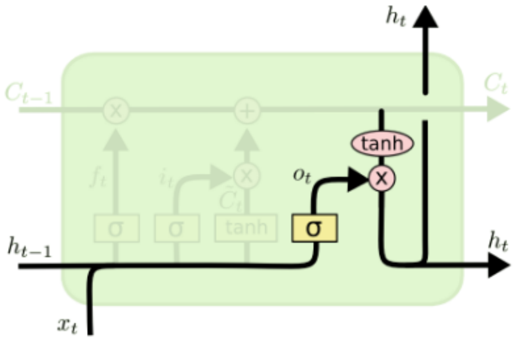


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Update the old hidden state C_{t-1}, into the new cell state C_t

Output gate

- Determine h_t based on cell state C

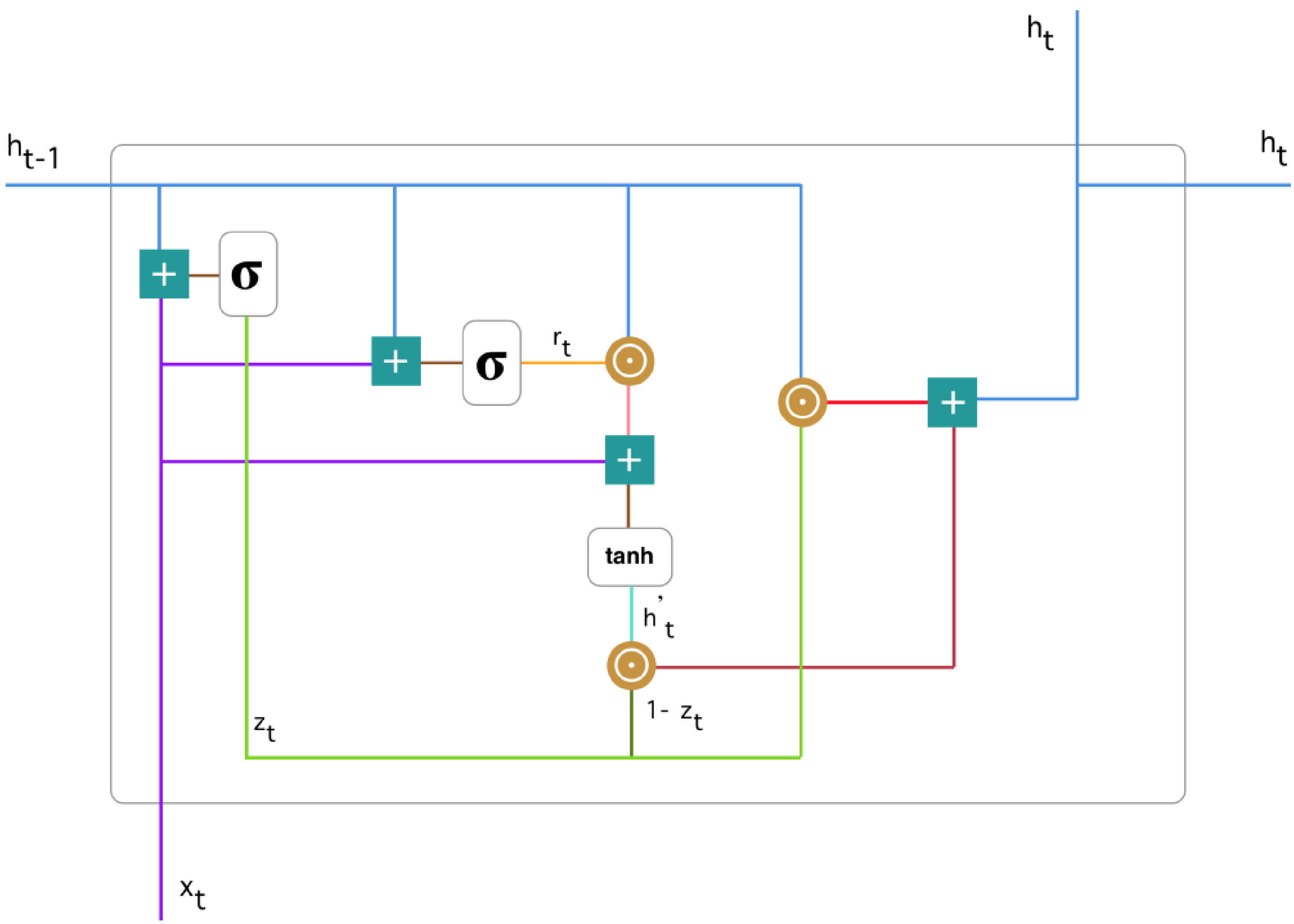


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

- Run a sigmoid layer which decides what parts of the cell state we're going to output
- Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.
- For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next
- For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

GRU

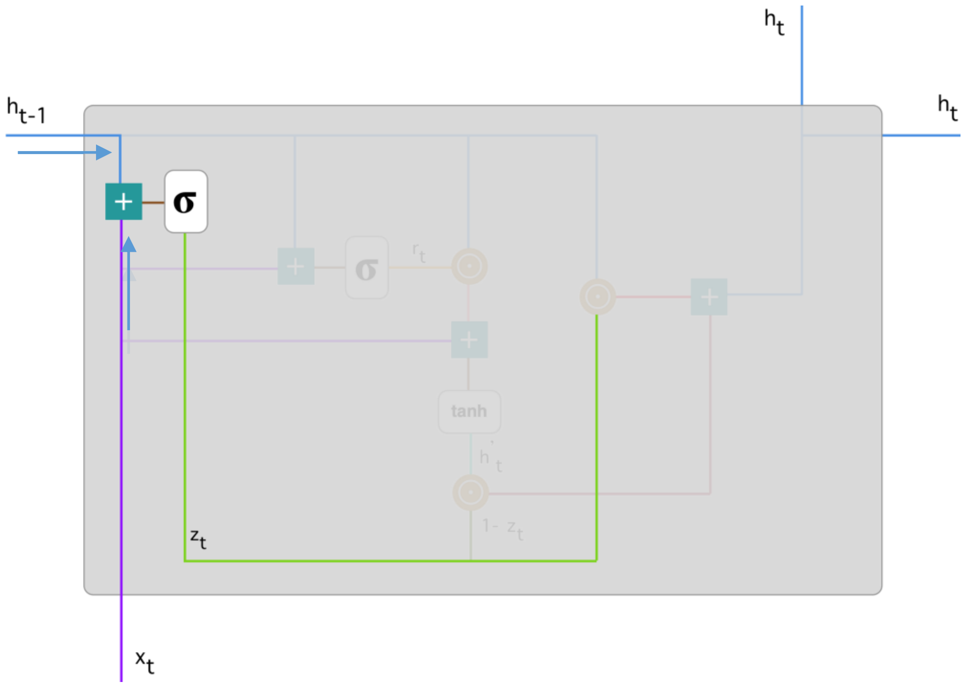
- Instead of using Forget, Input and output gate, GRU uses Update and Reset gate. Basically, there are 2 gates which decide what information should be passed to the output
- They can be trained to keep relevant information from long ago without vanishing or remove though time



"plus" operation "sigmoid" function "Hadamard product" operation "tanh" function

How the GRU works

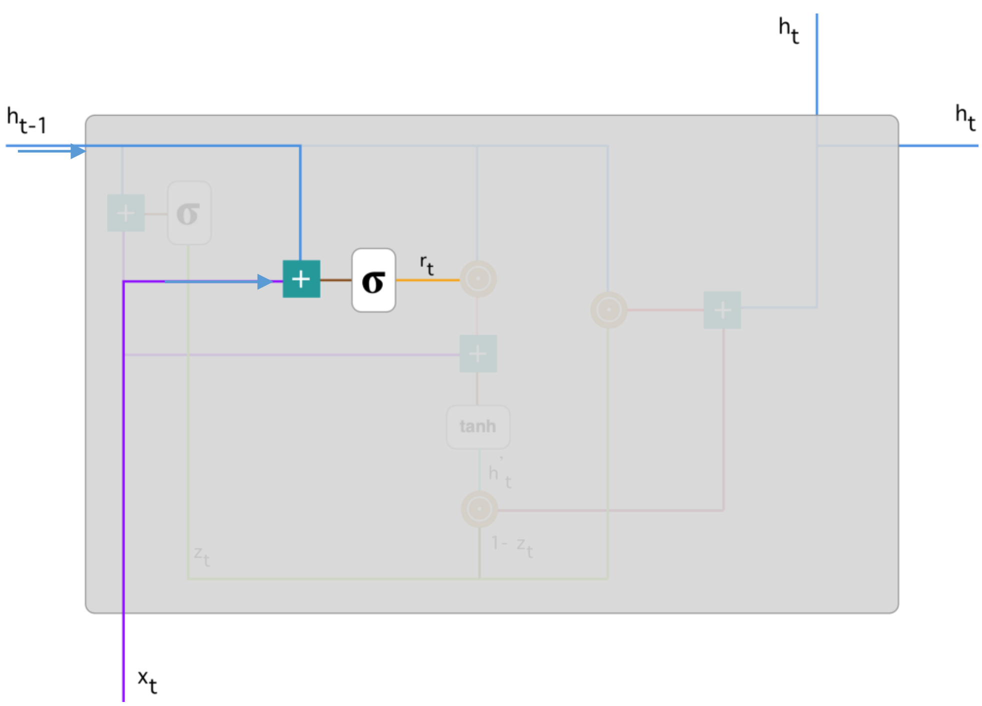
- Update gate



$$z_t = \sigma\left(W^z x_t + U^z h_{t-1}\right)$$

  - Update gate helps the model to determine how much of the past information needs to be passed along the future, so the model can decide to copy all information and reduce the vanishing problem

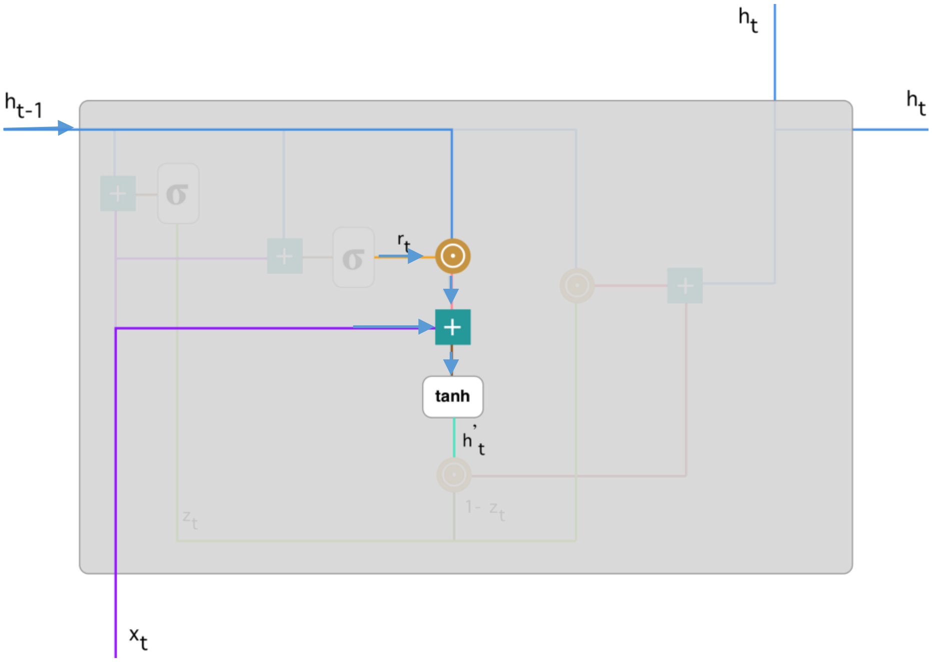
- Reset gate



$$r_t = \sigma\left(W^r x_t + U^r h_{t-1}\right)$$

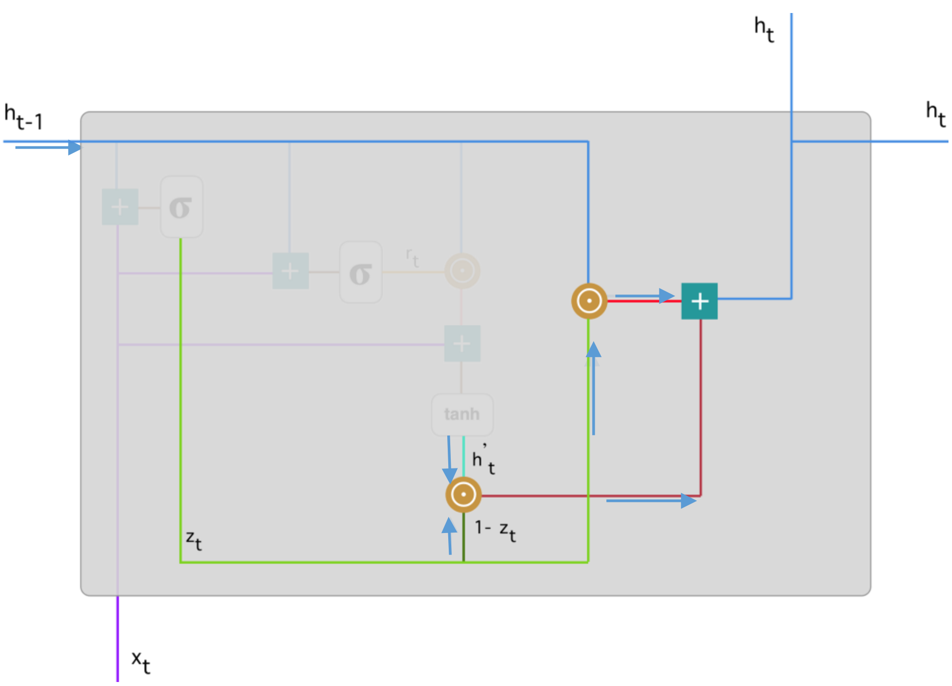
  - This gate is used to decide how much of the past information to forget
  - The formula is almost same as the update gate, but the weights

- Current Memory Content



$$h'_t = \tanh\left(Wx_t + r_t \odot Uh_{t-1}\right)$$

  - $r_t \odot Uh_{t-1}$ helps to remove the should-forget information, which determined in Reset gate. E.g. The text starts with "This is a fantasy book which illustrates…" and after a couple paragraphs ends with "I didn't quite enjoy the book because I think it captures details which seem irrelevant to me.". To determine the level of satisfaction, we only need last part of this review, so it will learn to $r_t \approx 0$ to wash out the past and focus on the last sentence
  - If $h'_t = 0$ means we don't need to care about all words before, just care about the current word
- Final Memory



$$h_t = z_t \odot h_{t-1} + \left(1 - z_t\right) \odot h'_t$$

  - GRU doesn't use Cell State, but it uses h'_t

Vanishing Gradient

- Derivative of loss function J of variable W^{hh}: $\frac{\partial J}{\partial W^{hh}} = \sum_t \frac{\partial J^{(t)}}{\partial W^{hh}}$
- Derivative of loss function $J^{(t)}$ of variable W^{hh}: $\frac{\partial J^{(t+1)}}{\partial W^{hh}} = \frac{\partial J^{(t+1)}}{\partial y_{t+1}} \frac{\partial y_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial W^{hh}}$

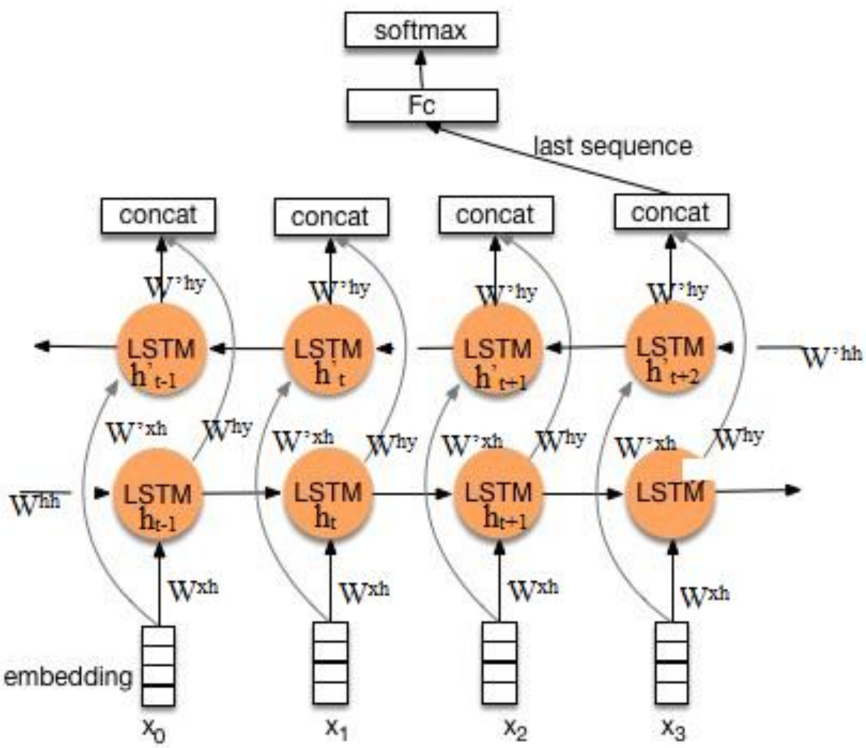
- However, $\frac{\partial h_{t+1}}{\partial W^{hh}} = \frac{\partial h_{t+1}}{\partial h_t}\frac{\partial h_t}{\partial W^{hh}} + \frac{\partial h_{t+1}}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial W^{hh}}$

$\rightarrow \frac{\partial J_n}{\partial W^{hh}} = \sum_{k=0}^{n} \frac{\partial J_n}{\partial y_{t+1}}\frac{\partial y_{t+1}}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial h_k}\frac{\partial h_k}{\partial W^{hh}}$, at k=0, $\frac{\partial h_{t+1}}{\partial h_0} = \frac{\partial h_{t+1}}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial h_0}$

Fasttext

- An extension to Word2Vec proposed by Facebook
- Instead of feeding individual words into RNN, we break words into n-grams
- For instance, the tri-grams for the word apple is app, ppl, and ple

sBidirectional RNN



  - While doing the first row h_{t-1}, h_t,…, you also do the second row like the first row but in the reverse way

Deep/ Stacked RNN

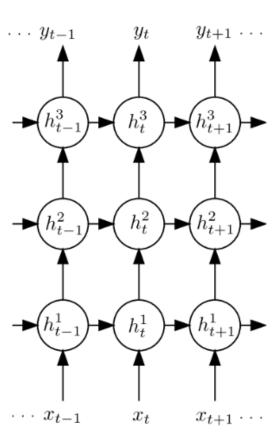
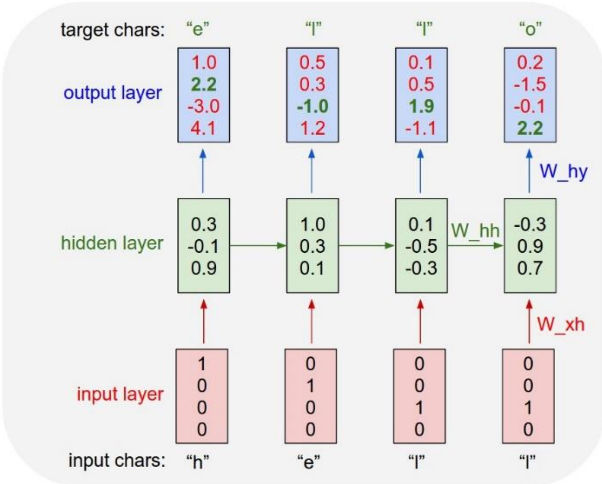


Fig. 3. Deep Recurrent Neural Network

Character-level RNN

- Some words are Name, weird words, it's called OOV(Out-of-vocabulary). Character-level RNN are applied



- Character-level has covered all words but it cannot apply the pre-trained weights
- Moreover, there are some ways of word embedding, train weights
  - from scratch
  - apply pre-trained weight with
    - trainable = T: continue to train the weight with pre-trained
    - trainable = F: apply weights without any train

Seq2seq

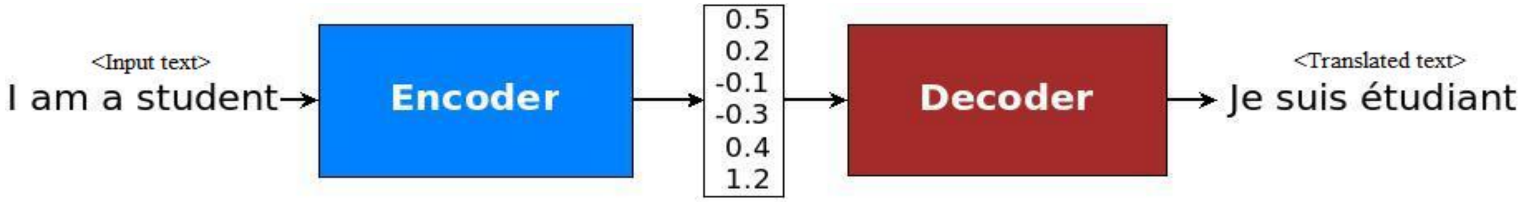
- I/O: Input is a sentence(corpus) and output is also a sentence(corpus)
- The problem Seq2seq can apply:
  - Machine translation
  - Image Captioning: input is image instead of corpus and output is still output, so in the encode session, we replace RNN by CNN
  - Summarization
  - Headline Generate
  - Add accent in Vietnamese
  - Punctuation

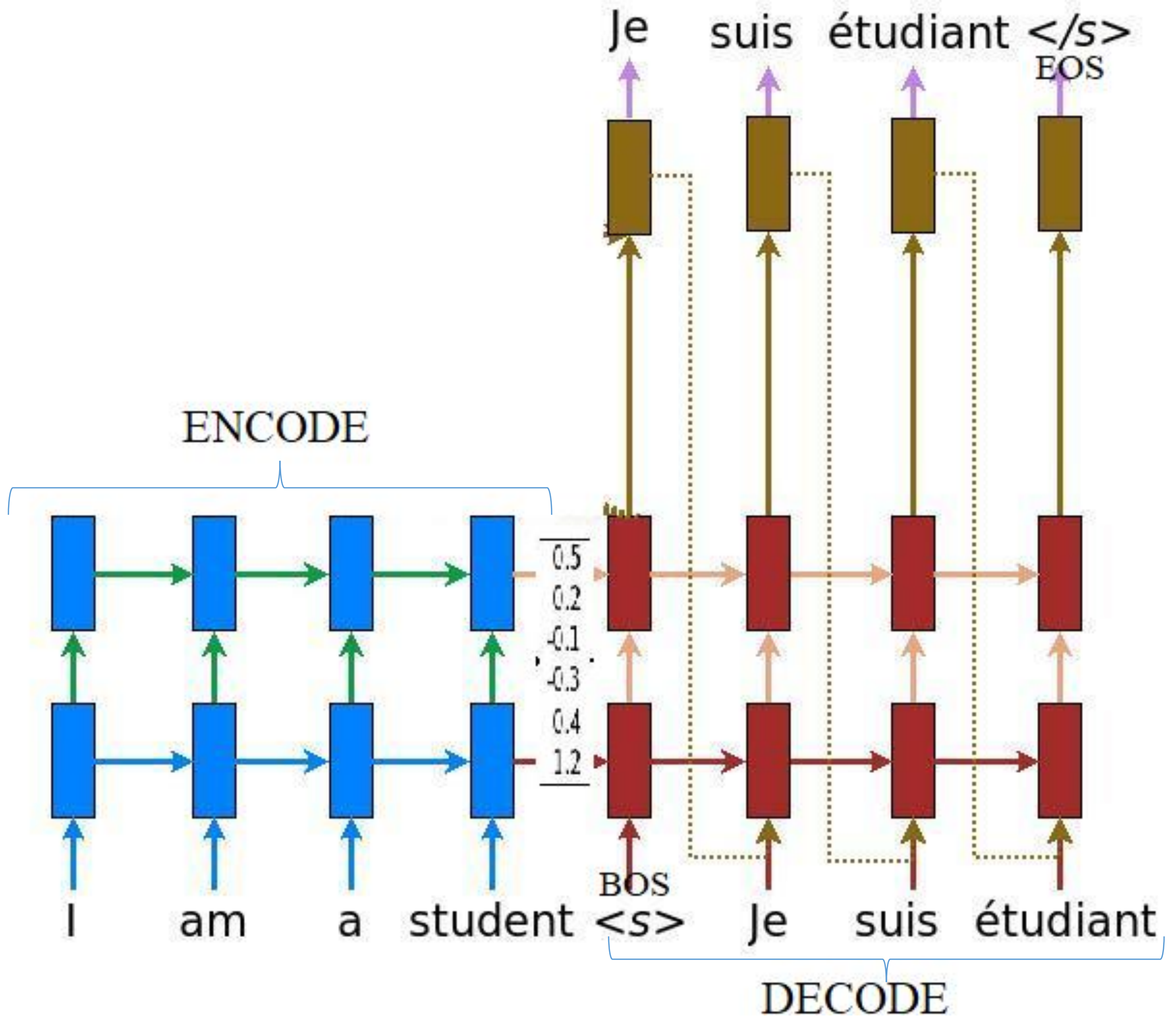
Conditional Language Model

- Suppose you need to translate 'I am a student' to the French, you need to calculate:

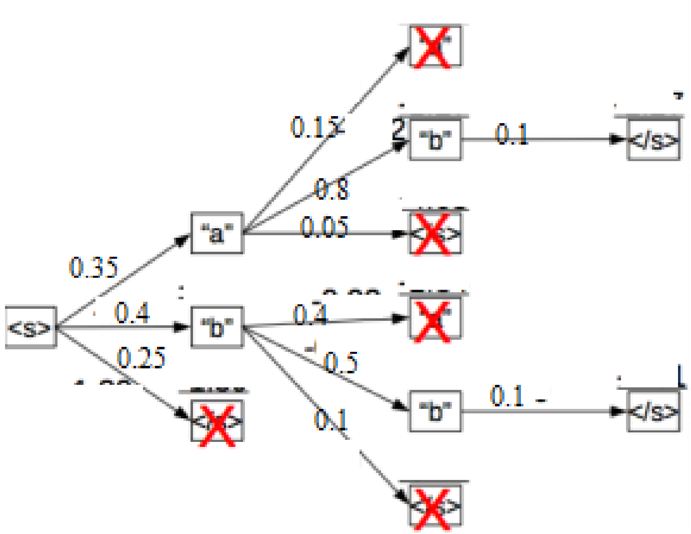
  $P(y_1 y_2 \dots y_N \mid x) = \prod_{i=1}^{N} P(y_i \mid y_1 \dots y_{i-1}, x)$ in which N: number of word in translated sentence, $y_1, y_2, \dots, y_N$ is words in translated sentence
- Cost function: $J(\theta) = -\sum_{j=1}^{m} \sum_{i=1}^{N} \log P\left(y_i^{(j)} \mid y_1^{(j)} \dots y_{i-1}^{(j)}, x^{(j)}\right)$ in which m is number of sentence
- Inference/ Feed Forward Process: $\arg\max_y P(y_1 y_2 \dots y_N \mid x) = \arg\max_y \prod_{i=1}^{N} P(y_i \mid y_1 \dots y_{i-1}, x)$
- However, $\prod_{i=1}^{N} P(y_i \mid y_1 \dots y_{i-1}, x)\ easily \rightarrow 0$, We just log them,

  $$\arg\max_y \log P(y_1 y_2 \dots y_N \mid x) = \arg\max_y \prod_{i=1}^{N} \log P(y_i \mid y_1 \dots y_{i-1}, x)$$
- Architecture



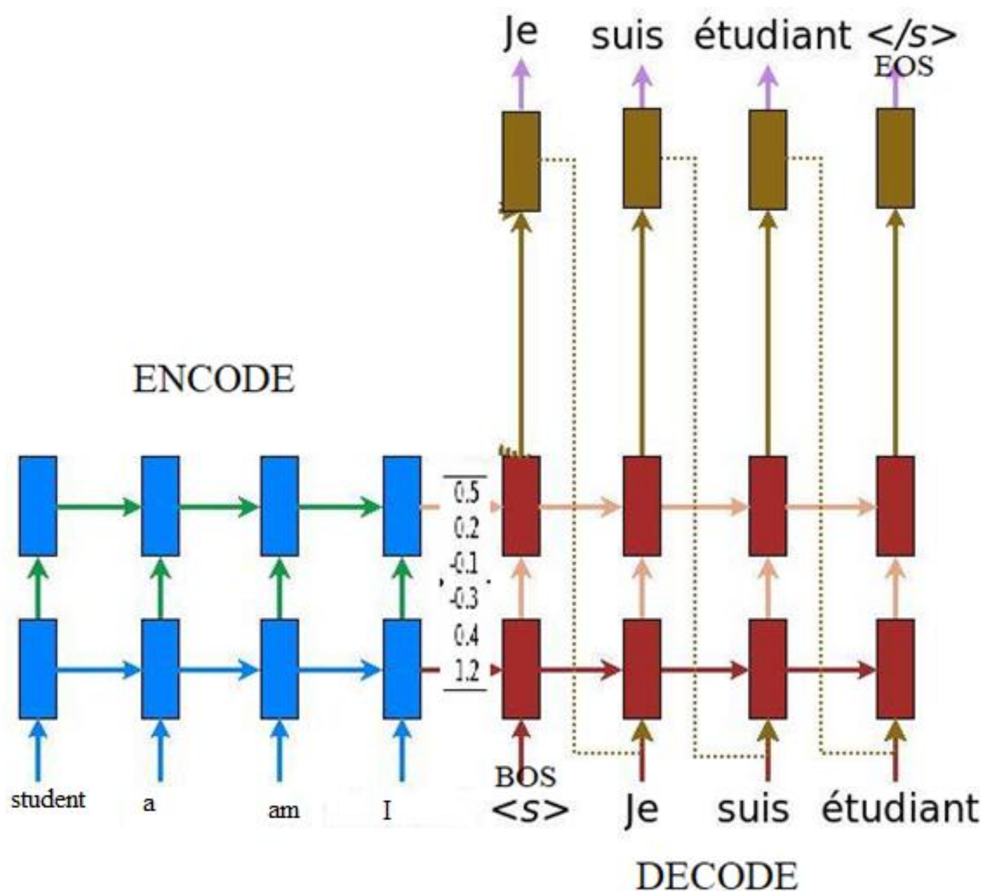


- The input is word vector of ‘I’, ‘am’, ‘a’ and ‘student’, the word vector is built on English words only
- The output for ‘Je’ is the list of all probability of all French words which have probability of ‘Je’ is the largest
- This architecture has modelled the Inference/ Feed forward process. E.g.
  $P(Je \mid BOS, I, am, a, student) \times P(suis \mid Je, BOS, I, am, a, student) \times \ldots \times P(EOS \mid \text{etudiant}, \text{suis}, \text{Je}, \text{BOS}, \text{I}, \text{am}, \text{a}, \text{student})$
- The problem occurs when you just choose the largest probability to be your output because the model mention $\arg\max_y \prod_{i=1}^{N} P(y_i \mid y_1 \ldots y_{i-1}, x)$, we can solve by some options
  - Random Sampling (Ancestral Sampling): in the step i in N, you choose the word based on the distribution which follows distribution of decode output at step i. Application in ChatBot
  - Greedy Search: mentioned above which choose the largest probability in the output at step i
  - Beam Search: at step i, we just keep k/ Beam Size/ Beam Width string which has length of i. E.g. Beam Size = 2



- The string here: <s> a b </s>

- Another problem: for instance, you translate "I am a student" to French, so "I" should be translated to "Je", but in the architecture we consider, "Je" has to go though "student", "a", "am", so the weight of "I" impacts on prediction of "Je" is now a little. To solve it, we reverse the sentence

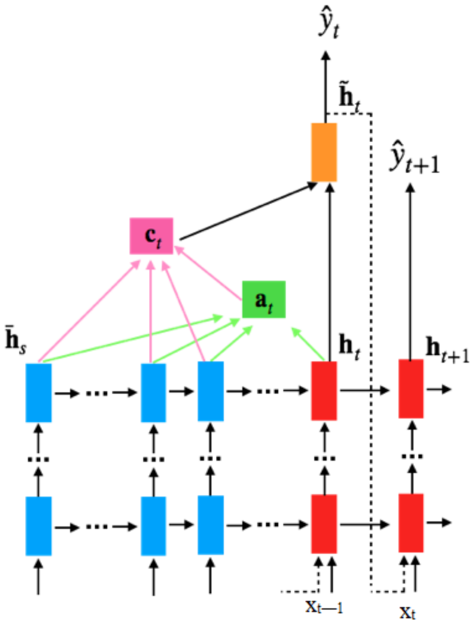


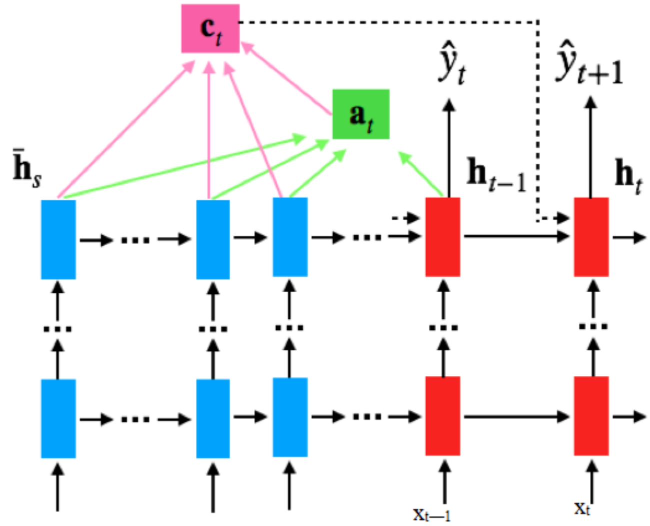
- However, we also have another method to solve completely this problem, called Attention, or Bahdanau Attention

Attention

- Calculate the score to measure the alignment of target to the all words of input sentence
- Now, we want to find out h_t, which is x_t is 'Je'

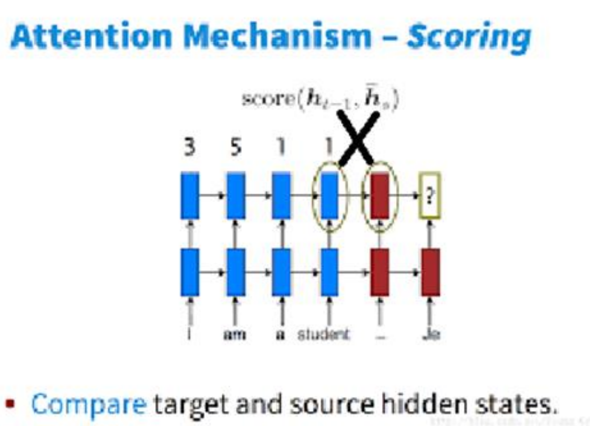
Luong Attention Mechanism	Bahdanau Attention Mechanism
$\mathbf{h}_t \rightarrow \mathbf{a}_t \rightarrow \mathbf{c}_t \rightarrow \tilde{\mathbf{h}}_t$	$\mathbf{h}_{t-1} \rightarrow \mathbf{a}_t \rightarrow \mathbf{c}_t \rightarrow \mathbf{h}_t$

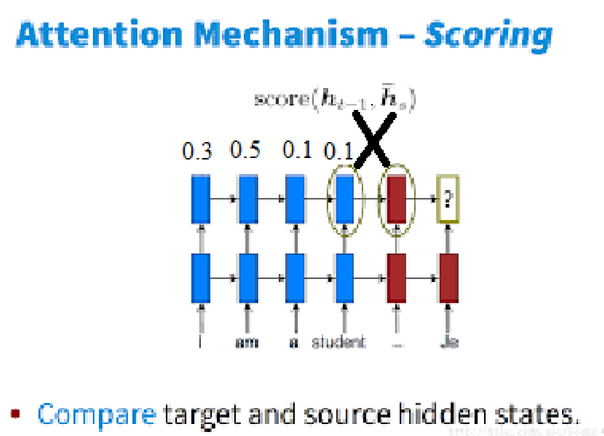




Luong	Bahdanau
$\mathbf{a}_t(s) = \text{align}(\mathbf{h}_t, \bar{\mathbf{h}}_s)$	$\mathbf{a}_t(s) = \text{align}(\mathbf{h}_{t-1}, \bar{\mathbf{h}}_s)$
$\mathbf{c}_t = \sum a_t \mathbf{h}_s$	$\mathbf{c}_t = \sum a_t \mathbf{h}_s$
$\tilde{\mathbf{h}}_t = \tanh(W_c[\mathbf{c}_t; \mathbf{h}_t])$	$\mathbf{h}_t = \text{RNN}(\mathbf{h}_{t-1}^{l-1}, [\mathbf{c}_t; \mathbf{h}_{t-1}])$

- Bahdanau Attention
  - $align\left(h_{t-1}, \bar{h}_s\right) = score\left(h_{t-1}, \bar{h}_s\right) = v_\alpha^T \tanh\left(W_1 h_{t-1} + W_2 \bar{h}_s\right)$
  - v_α, W_1, W_2 is variable like w_{hh}, w_{xh} and initialize these variable in the beginning





  - Then, put the score vector of h_{t-1} to the softmax: $$\alpha_{ts} = \frac{\exp\left(score\left(h_{t-1}, \bar{h}_s\right)\right)}{\sum_{s'} \exp\left(score\left(h_{t-1}, \bar{h}_{s'}\right)\right)}$$
  - $$c_t = \sum_s \alpha_{ts} \bar{h}_s$$

- $h_t = f\left(h_{t-1}, x_t, c_t\right) = \sigma\left(W^{hh} h_{t-1} + W^{xh} x_t + Cc_t\right)$
- $f\left(h_{t-1}, x_t, c_t\right)$,we can use LSTM, GRU or Vanilla as long as we add Cc_t when calculating h_t

- Luong Attention
  - Assume we have $\tilde{h}_{t-1}$ and x_{t-1} and we use LSSTM/GRU or Vanilla to calculate h_t
  - $$align\left(h_t, \overline{h}_s\right) = score\left(h_t, \overline{h}_s\right) = \left[\begin{array}{c} h_t^T \overline{h}_s \\ h_t^T W \overline{h}_s \\ v_\alpha^T \tanh\left(W_1 h_t + W_2 \overline{h}_s\right) \end{array}\right.$$
  - Then, put the score vector of h_t to the softmax: $\alpha_{ts} = \dfrac{\exp\left(score\left(h_t, \overline{h}_s\right)\right)}{\sum_{s'} \exp\left(score\left(h_t, \overline{h}_{s'}\right)\right)}$
  - $$c_t = \sum_s \alpha_{ts} \overline{h}_s$$
  - $$\tilde{h}_t = \tanh\left(W_C \left[c_t, h_t\right]\right)$$
  - Now, we have $\tilde{h}_t$ and x_t to calculate h_{t+1}