Coursera Deep Learning Specialization Notes: Sequence Models

Amir Masoud Sefidian

Version 1.0, February 2023

Contents

Preface

A couple of years ago I completed Deep Learning Specialization taught by AI pioneer Andrew Ng. I found this series of courses immensely helpful in my learning journey of deep learning. After years, I decided to prepare this document to share some of the notes which highlight key concepts I learned in the fifth course of this specialization, Sequence Models. In this course, we will become familiar with Sequence Models and their exciting applications such as speech recognition, music synthesis, chatbots, machine translation, natural language processing (NLP), and more. You will be able to build and train Recurrent Neural Networks (RNNs) and commonly-used variants such as GRUs and LSTMs. Moreover, you will apply RNNs to Character-level Language Modeling, gain experience with Natural Language Processing and Word Embeddings, and use HuggingFace tokenizers and Transformer Models to solve different NLP tasks such as Named Entity Recognition and Question Answering.

The content of this document is mainly adapted from this GitHub repository. I have added some explanations, illustrations, and visualization to make some complex concepts easier to grasp for readers. This document could be a good reference for Machine Learning Engineers, Deep Learning Engineers, and Data Scientists to refresh their minds on the fundamentals of deep learning. Please don't hesitate to contact me via my website (Sefidian Academy) if you have any questions.

Happy Learning!
Amir Masoud Sefidian

1 Sequence models

1.1 Applications

- Speech recognition
- Music generation
- Sentiment classification
- DNA sequence analysis
- Machine translation
- Video activity recognition
- Name entity recognition

1.2 Notation

Motivating example (Name entity recognition): Which words in the sentence are names?

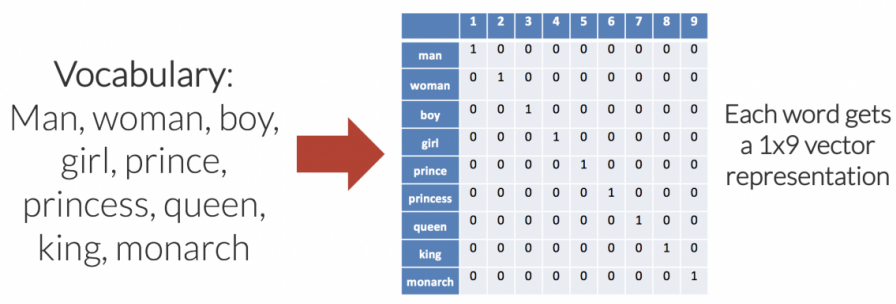
```
x: Harry Potter and Hermione Granger invented a new spell.
y: 1     1      0   1        1       0        0 0    0
```

Elements in each position (9 positions in this case). t stands for timestamp:

$$x^{<1>}, x^{<2>}, ..., x^{<t>}, ..., x^{<9>}$$
$$y^{<1>}, y^{<2>}, ..., y^{<t>}, ..., y^{<9>}$$

- $x^{(i)<t>}$ is the t^{th} sequence element of example i.
- $y^{(i)<t>}$ is the t^{th} sequence element of output i.
- $T_x^{(i)}$ is the total number of sequence elements of training example i (9 in this case).
- $T_y^{(i)}$ is the total number of sequence elements of output example i (9 in this case).

Dictionary or **Vocabulary**: list of all words that we use in the representations.
One possible word representation: **One-Hot** vectors, where each word is a binary vector of the size of the whole vocabulary.



	1	2	3	4	5	6	7	8	9
man	1	0	0	0	0	0	0	0	0
woman	0	1	0	0	0	0	0	0	0
boy	0	0	1	0	0	0	0	0	0
girl	0	0	0	1	0	0	0	0	0
prince	0	0	0	0	1	0	0	0	0
princess	0	0	0	0	0	1	0	0	0
queen	0	0	0	0	0	0	1	0	0
king	0	0	0	0	0	0	0	1	0
monarch	0	0	0	0	0	0	0	0	1

Figure 1: One Hot Representation

1.3 Recurrent neural networks (RNN)

Why not use a standard network?

1) Inputs and outputs can be different lengths in different samples. 2) Typical networks do not share features learned across different positions of text.

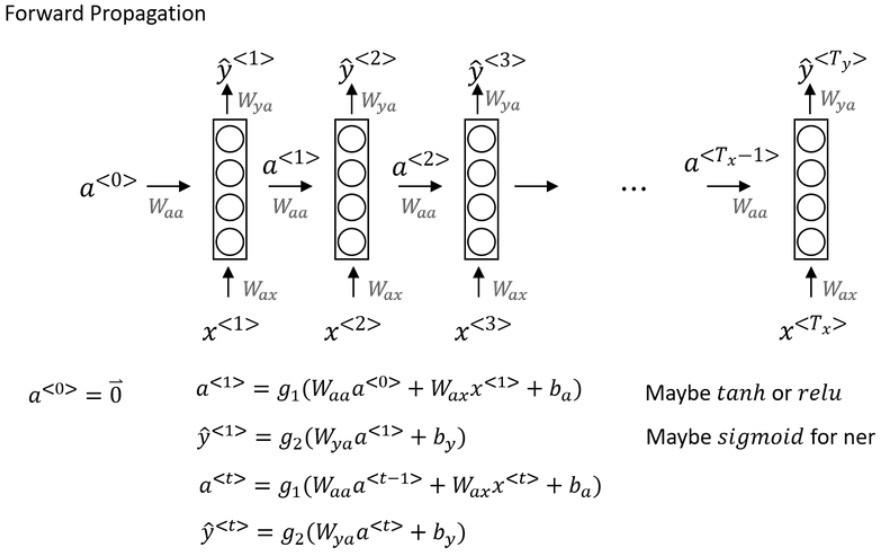


Figure 2: RNN forward propagation

RNN Forward Propagation The idea is to use the activation at $t-1$ as well as the input at time t:

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

That can be simplified by horizontally stacking W_{aa} and W_{ax} into $W_a = [W_{aa}; W_{ax}]$, and vertically stacking $a^{<t-1>}$ and $x^{<t>}$ as $[a^{<t-1>}, x^{<t>}]$:

$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$

And we also have:

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

Which is usually simplified by renaming W_{ya} to W_y:

$$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$$

Note that all weights (W_{aa}, W_{ax}, etc.) are shared among all time steps.

Dimensions of matrices assuming that the vocabulary is of the size 10000 and there are 100 hidden units:

$x^{<t>} : (10000, 1)$, $a^{<t>} : (100, 1)$

$W_{aa} : (100, 100)$, $W_{ax} : (100, 10000)$, $W_a : (100, 10100)$

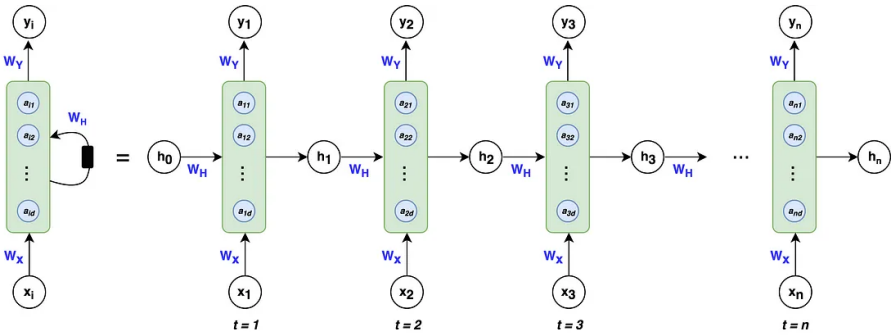


Figure 3: RNN: Folded and Unfolded

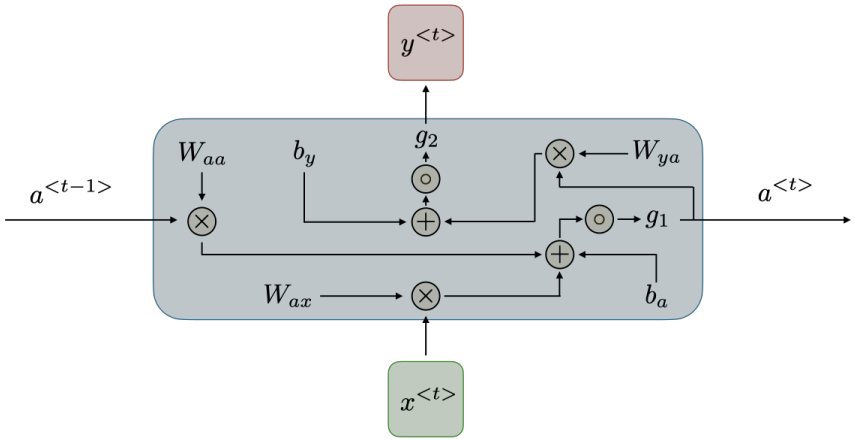


Figure 4: RNN unit

RNN backpropagation Loss is again the cross entropy loss (standard logistic regression loss). Loss at time step t:

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -[y^{<t>} \log(\hat{y}^{<t>}) + (1 - y^{<t>}) \log(1 - \hat{y}^{<t>})]$$

Total loss on the entire sequence:

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

- **Backpropagation through time**: Gradient flows backward to the matrix multiplication node where we compute the gradients w.r.t. both the weight matrix and the hidden state. The gradient w.r.t. the hidden state and the gradient from the previous time step meet at the copy node where they are summed up.

Different types of RNNs Paper: "The unreasonable effectiveness of recurrent neural networks."

- Many-to-many: many inputs and many outputs, when $T_x = T_y$. Name Entity Recognition.
- Many-to-many: many inputs and many outputs, when $T_x != T_y$, e.g. machine translation. Two parts, one encoder (e.g. from the source language) of size T_x, one decoder (e.g. to translate to the target language) of size T_y.

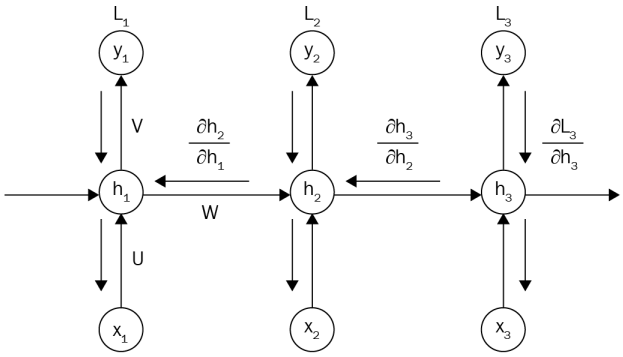


Figure 5: Backpropagation through time

- Many-to-one: many inputs and only one output (e.g. for sentiment analysis)
- One-to-one: just for the sake of completeness - really just a standard NN.
- One-to-many: one input, sequence of outputs (e.g. music generation). Note when generating sequences we usually take each output of $t-1$ and feed it as the input of t (sometimes the single input can be 0).

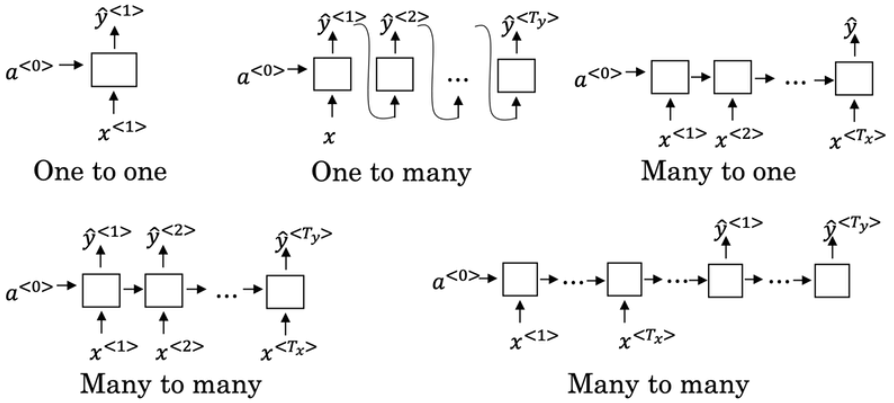


Figure 6: RNN Types

1.4 Language Model and Sequence generation

The goal of Language Model (LM):
1) Calculating the probability of a given sentence,i.e. What is P(sentence) =?
For example:
P('The apple and pair salad') = 10^{-15}
P('The apple and pear salad') = 10^{-10}
2) Given a sequence, what is the probability of the next element? Compute P('word'|'The apple and pear ...') for each 'word'.

How to learn an LM with RNNs:

- Training set: a large corpus of text from which a vocabulary is derived.

- Text is first tokenized (e.g. separated in words). It is generally useful to have an <EOS> (end of sentence) token. Words not in the vocabulary are replaced with the <UNK> token.

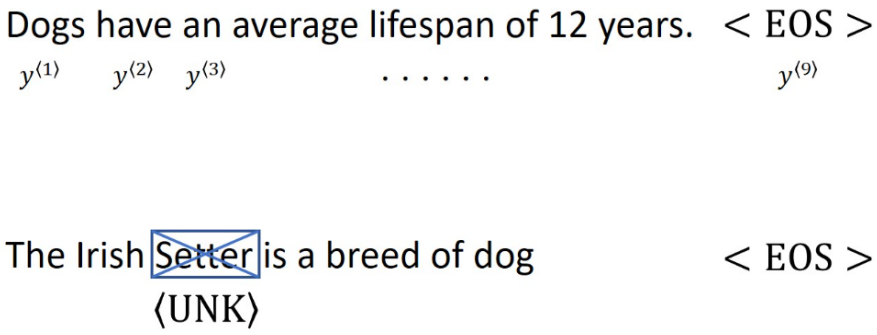


Figure 7: Tokenization [1]

- In a simplistic model, the size of the input vectors for each word is the size of the vocabulary (One-hot encoded).
- Set $x^{<t>} = y^{<t-1>}$.
- The output of each step of the RNN is a vector of the size of the vocabulary, and for each word in that vocabulary, it contains the conditional probability of that word, *given the previous words* already fed to the RNN.

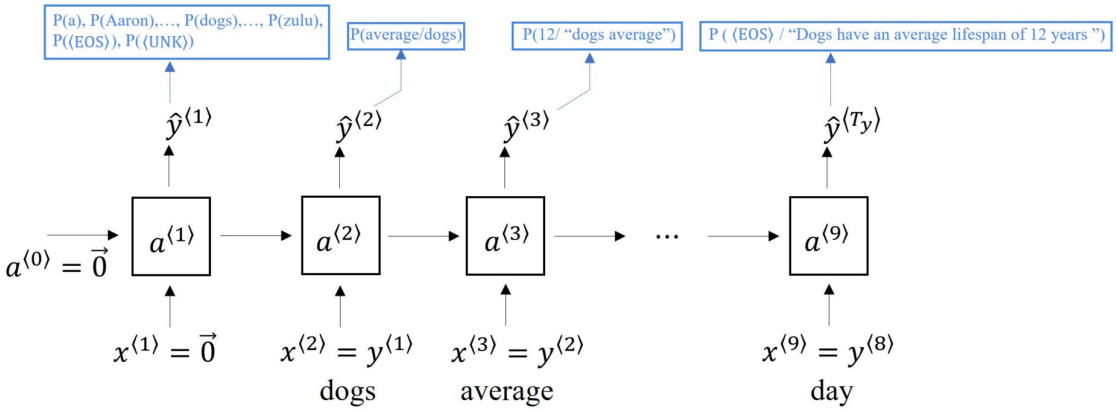


Figure 8: Train an LM using RNN [1]

- **Forward Propagation**: The forward propagation of our RNN model begins at time 0, wherein some activation $a^{\langle 1 \rangle}$ is computed as a function of some input $x^{\langle 1 \rangle}$, which in turn is set to all zeros or a 0-vector. The initial activation $a^{\langle 0 \rangle}$ by convention is also an event of zeros. Now, the main role that $a^{\langle 1 \rangle}$ plays is that it makes a Softmax prediction to try to predict the probability of the first word of our training example – "Dogs". This is denoted by $\hat{y}^{\langle 1 \rangle}$.

  A Softmax prediction picks each word from the dictionary we have created and tries to predict the probability of that word occurring in a particular sentence. For example, computing the chance that the first word was "Zulu" or the chance that the first word is an Unknown Word, and so on. In short, Softmax helps us get to the output at each stage, that is, $\hat{y}^{\langle 1 \rangle}$.

Continuing with the forward propagation, the RNN model uses the activation $a^{\langle 1 \rangle}$ to predict the next word. At this time step, the network will also be given the correct value of the first word. In our case, we will tell the RNN that the first word was "Dogs". This way the output of the previous time step also becomes the input for the next time step, that is, $y^{\langle 1 \rangle} = x^{\langle 2 \rangle}$. We move to the next time step and the whole process with Softmax is repeated as it predicts the second word. In the third time step, we tell the network the correct second word which is "have" and the next activation value $a^{\langle 3 \rangle}$ is computed. In this step, the input is the first two words "Dogs have" and hence, the output of the previous step is again equal to the input of the next step, that is, $x^{\langle 3 \rangle} = y^{\langle 2 \rangle}$.

The process moves from left to right till it reaches the EOS token. In each step of the RNN, previous words act as input for the next word, and this way, the RNN learns to predict one word at a time propagating in the forward direction.

- **Back Propagation**: Now that we have performed forward propagation for our Language model using RNN, we shall look at how we are going to define the cost function and compute the loss using backpropagation. We are going to train our network by calculating loss at a certain time t. So, if at this time the correct word was .. and the neural network Softmax predicted the output .., we can easily compute the Softmax loss at this particular time.

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log(\hat{y}_i^{<t>})$$

The total loss, as we understood previously, is nothing but the sum of losses computed for each time step and individual predictions.

$$\mathcal{L}(\hat{y}, y) = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

If we train this neural network for a larger training set, given an initial set of words such as "Dogs have an average" or "Dogs have an average lifespan", our neural network will be able to predict the probabilities of the subsequent words using Softmax, such as $P(y^{\langle 2 \rangle}|y^{\langle 1 \rangle})$ and so on. Hence, for a three-word sentence or a four-word sentence, the total probability will be the product of all the previous probabilities of those three or four words.

- Calculate the probability of a new sentence:
$P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>}).P(y^{<2>}|y^{<1>}).P(y^{<3>}|y^{<2>}y^{<1>})$

Sampling novel sequences

- Using Softmax, our trained model predicts the chance or probability of a particular sequence of words and this can be represented as follows.

$$P\left(y^{\langle 1 \rangle}, \ldots, y^{\langle T_x \rangle}\right)$$

- An important aspect to be explored, once a Language Model has been trained, is how well it can generate new or novel sequences.

- The sampling begins after we have initiated the first input $x^{\langle 1 \rangle} = 0$ and set the initial activation value $a^{\langle o \rangle} = 0$. That produces a vector $\hat{y}^{<1>}$ with the probability of each word in the vocabulary (Softmax), from which we can sample a few with `np.random.choice`. Then we pick a word from

our sample (e.g. one which has a high probability of being the first word in a sentence such as "The"), and we feed it as the input $x^{<2>}$.

Then we pass on the output we sampled above, $\hat{y}^{\langle 1 \rangle}$, and pass as input to the second time step. Softmax distribution will, then, make a prediction for the second output $\hat{y}^{\langle 2 \rangle}$ based on this input, which is nothing but the sampled output of the previous time step.

- We obtain a vector $y^{<2>}$ corresponding to the conditional probability of each word in the vocabulary given the first chosen words. For example $P(y^{<2>}|"The")$ if the first chosen word was "The".
- We repeat this until we have a sequence of the intended size or an <EOS> token is generated.
- Sometimes a <UNK> token can be generated, though it can be explicitly ignored.

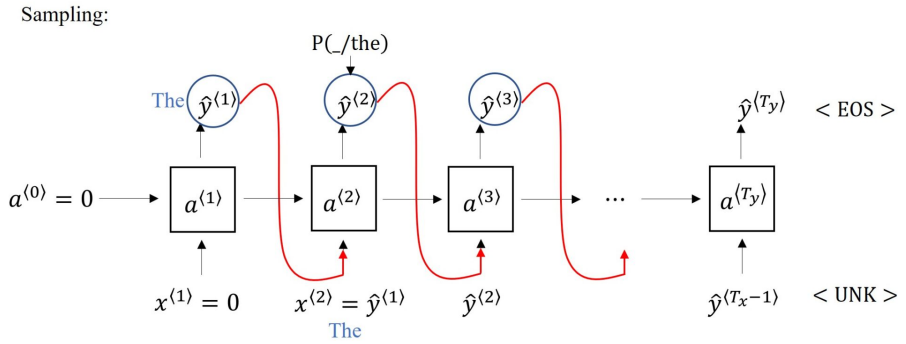


Figure 9: Sampling a sequence from an LM [1]

Note that the vocabulary can also be at the character level, which means that the sequence becomes the character sequence:

- Ends up with much longer sequences, computationally expensive, that don't capture relationships between words.
- They have the advantage of not having to deal with <UNK> tokens.

Vanishing gradients with RNNs

- RNNs are not good at capturing long-range dependencies, for example: "The cat, which already ate ...., was full." The subject "cat" can be distant from the predicate "was".
- This is due to vanishing gradients: the most common problem when training RNNs. Exploding gradients are catastrophic and usually cause NANs in the output, so at least they are easy to spot (gradient clipping can be used but that is not ideal).

Gate Recurrent Unit (GRU)

- Paper: "On the properties of neural machine translation: Encoder-decoder approaches"
- Paper: "Empirical evaluation of Gated Recurrent Neural Networks on Sequence Modeling"

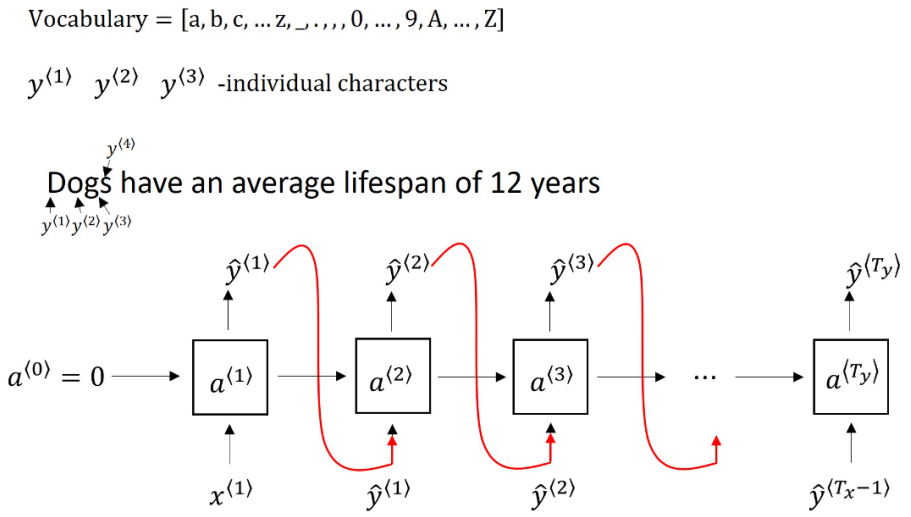


Figure 10: Character level LM [1]

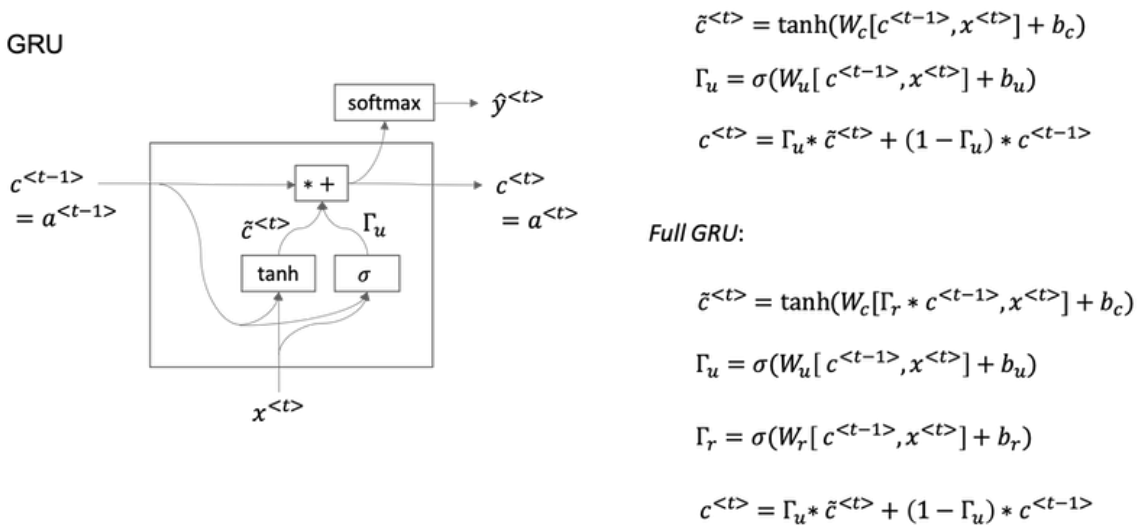


Figure 11: GRU

- You can find a detailed tutorial about GRU on my website:
  Understanding Gated Recurrent Unit (GRU) with PyTorch code

Notation:

$$c = \text{memory cell}$$
$$c^{<t>} = a^{<t>}$$

c and a are the same in this case, but they will be different in LSMTs, hence the distinction.
Example: c remembers if "cat" was singular or plural.
At each time instant we compute a *candidate replacement* for $c^{<t>}$ referred to as $\tilde{c}^{<t>}$.

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

The relevance gate term Γ_r ponders the relevance of $c^{<t-1>}$ for the new candidate replacement:

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

Then we also have an update "Gate" (hence the use of the letter gamma Γ), that decides if the new candidate should replace the value of the memory gate or not.

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

This function is always very close to 0 or very close to 1, so it can be considered close to the binary output. Then the decision on whether to retain previous $c^{<t-1>}$ or update it with new candidate $\tilde{c}$ is made with:

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

- The $*$ operation is element-wise multiplication.
- Γ_u, $c^{<t>}$, and $\tilde{c}^{<t>}$ all have the same dimensions (equal to the number of hidden units).
- The Γ_u gate allows memorizing items like the "cat" for as long as needed, e.g. until the "was" token is found, in which case c can be replaced with $\tilde{c}$.
- It also solves the vanishing gradient problem for the most part.

1.5 LSTM

- Paper: "Long short-term memory" (hard to read!).
- Compared to the GRU:
  - LSTMs are much older than GRUs!
  - LSTMs are computationally more expensive, due to more gates.
  - GRU is a simplification of the more complex LSTM model.
  - No consensus over the better algorithm.
  - It does not use Γ_r.
  - $c^{<t>}$ (long-term memory) is no longer equivalent to $a^{<t>}$ (short-term memory).
  - Γ_u is replaced with two gates: Γ_u (How much to add information from the new candidate?) and Γ_f (How much to forget/remove information from the previous memory?)
  - It has an explicit output gate Γ_o
- Very good at memorizing values within long-range dependencies, such as the GRU.

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

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

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

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

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

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

In some variants there is a "peephole connection", adding $c^{<t-1>}$ to ALL gates, where the i^{th} element of $c^{<t-1>}$ affects the i^{th} element of Γ_u,Γ_f, and Γ_o.

You can find detailed tutorials about LSTM on my website:

- A complete guide to understanding Long Short Term Memory (LSTM) Networks
- Implementing LSTM Networks in Python with Keras
- Understanding Long Short-Term Memory Networks (LSTM) with PyTorch codes

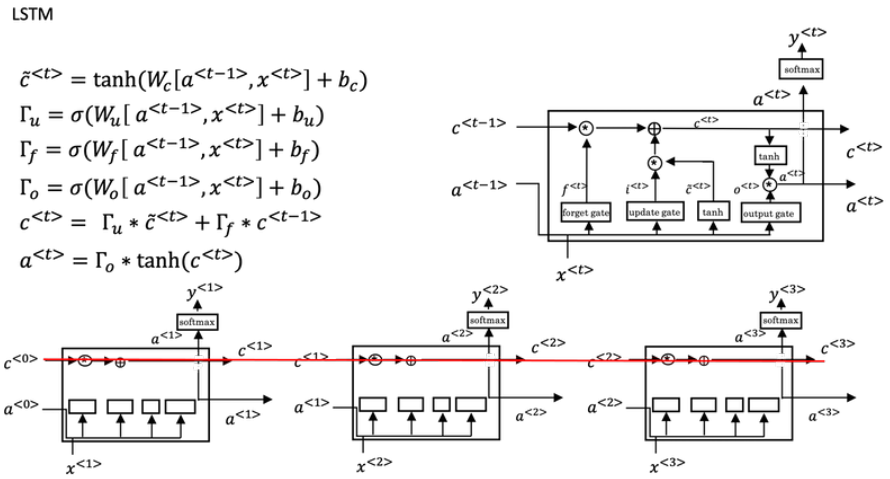


Figure 12: LSTM architecture

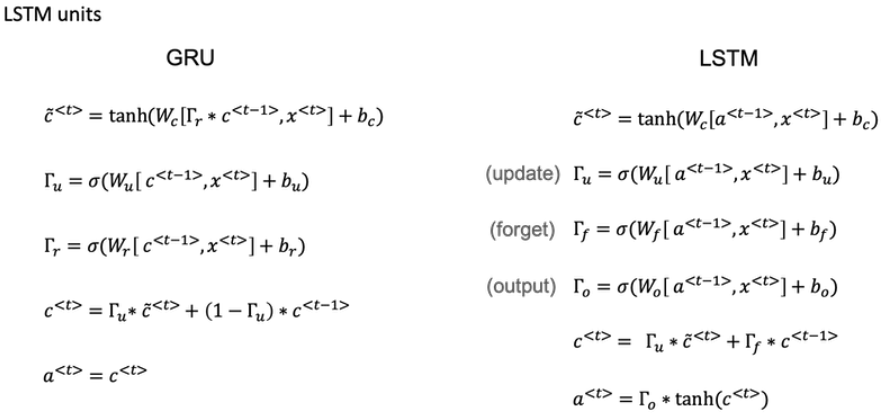


Figure 13: LSTM vs. GRU

1.6 Bidirectional RNNs (BRNN)

- Forward propagation goes both forward and back in time, and there are separate activations for each direction for each instant t.
- BRNNs work with GRU and LSTM (LSTM seems to be more common).
- Able to make predictions in the middle of the sequence from both the future and past elements in the sequence.
- Does require the entire sequence of data before making predictions (does not work well for speech recognition in real-time, for example).

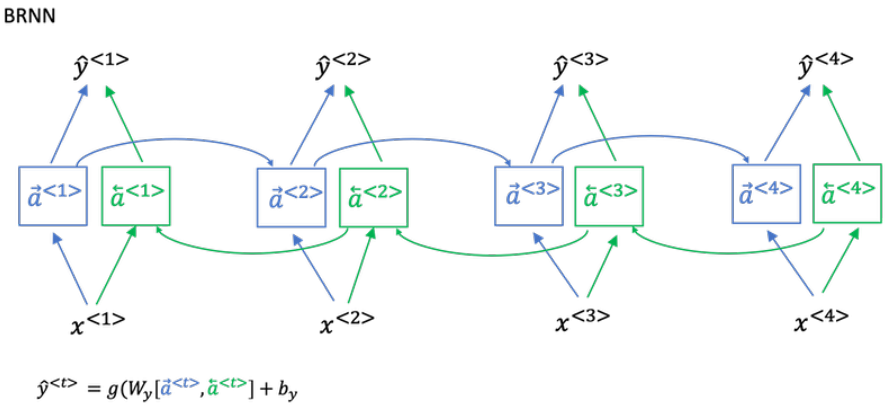


Figure 14: Bidirectional RNN

1.7 Deep RNN

- RNNs can be stacked "vertically", where each layer l has the same number of time units t, and the output $y^{<t>}$ is connected as the next layer's $(l+1)$ input.
- $a^{[l]<t>}$ represents the activation of layer l at time t.
- Usually, there aren't many layers; 3 is already quite a lot.
- In some architectures, each output $y^{<t>}$ can be connected as the input to another neural network (e.g. densely connected).
- LSTMs and GRUs can work in this architecture as well as BRNNs.

Updated RNNs layer indexing notation

- Superscript $[l]$ denotes an object associated with the l^{th} layer.
  - Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.
- Superscript (i) denotes an object associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example input.

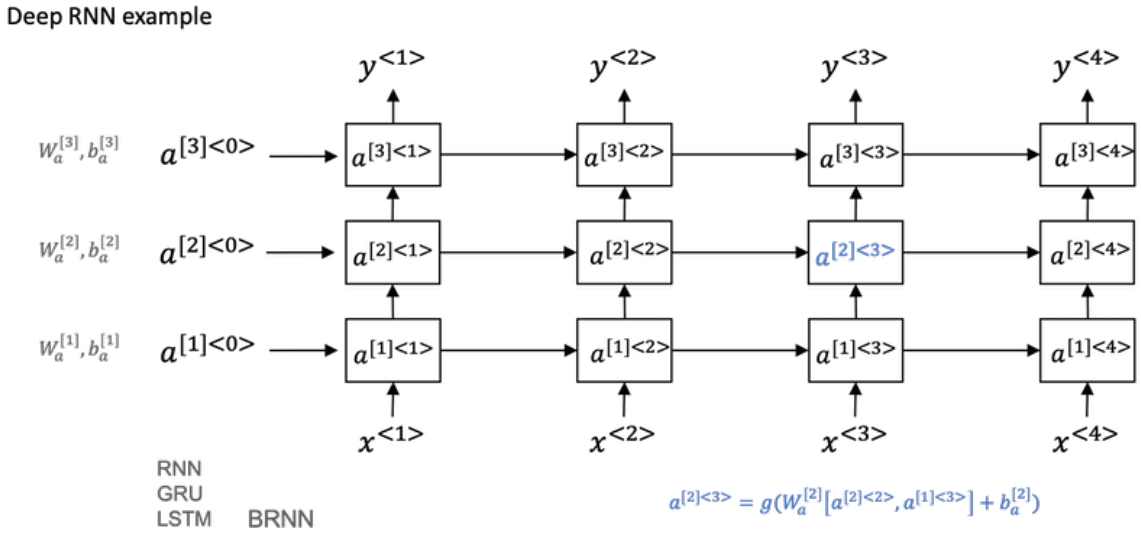


Figure 15: Deep RNNs

- Superscript $\langle t \rangle$ denotes an object at the t^{th} time-step.
  - Example: $x^{\langle t \rangle}$ is the input x at the t^{th} time-step. $x^{(i)\langle t \rangle}$ is the input at the t^{th} timestep of example i.
- Subscript i denotes the i^{th} entry of a vector.
  - Example: $a_i^{[l]}$ denotes the i^{th} entry of the activations in layer l.

1.8 NLP and Word embeddings

One-hot representation: Each word is represented by a binary vector for the size of the vocabulary with a 1 at the position that corresponds to a specific word in that vocabulary. The vectors generated *do not encode semantic proximity.* For example, the meaning of apple and orange in the following sentences when trying to find the next word:

`I want a glass of orange _____` `I want a glass of apple _____`

Word embedding (featurized representation): Each word is represented by high-dimensional feature vectors that include a measure of distance between the word represented by the vector and a series of features or other terms. Each word embedding can be named by the reference where the number is the index of the word in the vocabulary; e_{5391} or e_{9853} for example for the first and second column, respectively.

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1.00	1.00	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
...	...	...	...	...	...	...

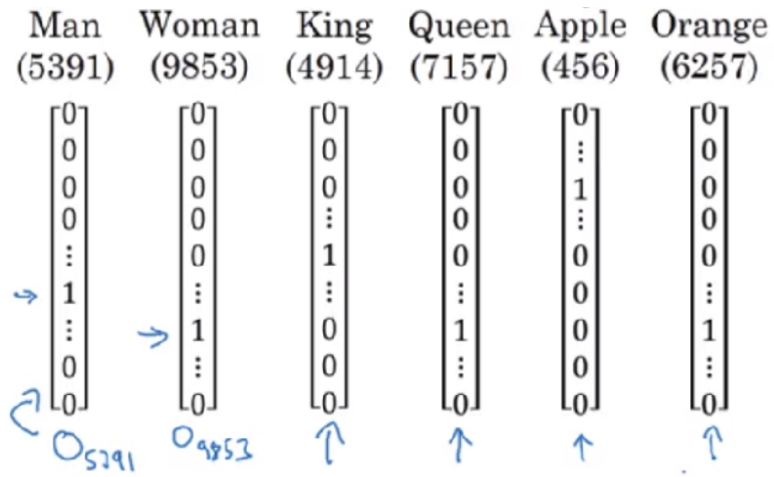


Figure 16: One-Hot representation of words

- Word vectors can be compared to find close relationships between terms and this helps generalize. In the example above, if we know that the missing word is "juice", the vectors for "apple" and "orange" should be close enough that a learning algorithm can infer the next word "juice" in both cases.
- Words can then be visualized together with t-SNE (plot high dimensional spaces into 2d or 3d spaces), where the word proximity is visible.

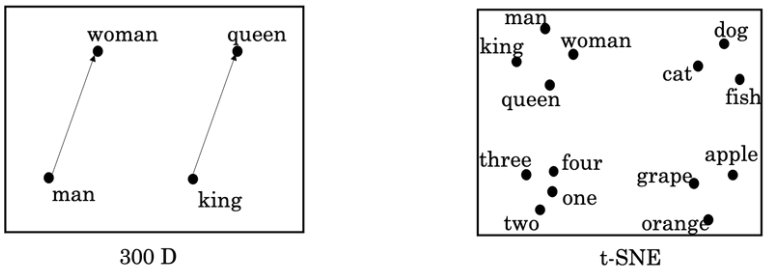


Figure 17: Word embedding

Using word embeddings

- It is possible to use "transfer learning" to learn word embeddings from a large body of text, for example, 1 billing word from the internet, and then use those embeddings with a smaller 100k word training set for the new task (e.g. Name Entity Recognition).
- This also allows representing words with smaller (compared to 1-hot) dense vectors of embeddings (e.g. size 300 instead of 10k). Word representation sizes are no longer restricted to the size of the vocabulary.

Properties of word embeddings

- Paper: "Linguise regularities in continuous space word representations"

- Allow finding analogies: e.g. by subtracting the word embeddings of "Man" and "Woman" or "King" and "Queen" (from the table above) we find that the main difference between them is the "Gender".
- With word embeddings we can use equations for finding the most appropriate words (**analogy tasks**):

$$e_{man} - e_{woman} \approx e_{king} - e_w$$

$$\text{Find word } w\text{: } argmax_w sim(e_w, e_{king} - e_{man} + e_{woman})$$

- The most common similarity (sim) used is the cosine similarity:

$$sim(u, v) = cos(\theta) = \frac{u^\intercal v}{\|u\|_2 \|v\|_2}$$

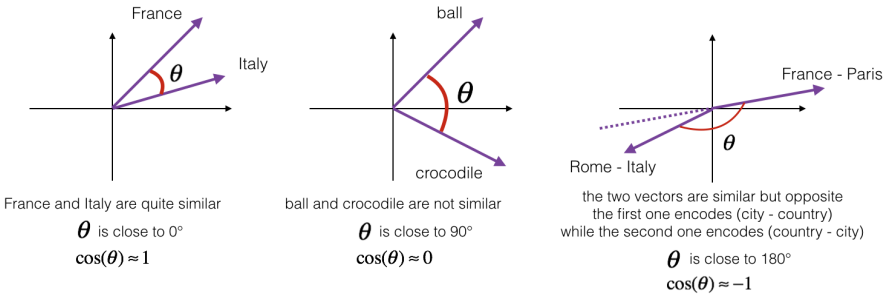


Figure 18: Cosine Similarity

- Another possible measure is the squared distance (euclidean), though it is a distance measure, not a similarity measure, so we would need to take its negative:

$$\|u - v\|^2$$

- The difference between the cosine similarity and using the euclidean distance is how they normalize for lengths of the vectors.

Embedding matrix

- An embedding matrix E, has dimensions $n\_embeddings \times V$, where V is the size of the vocabulary.
- The i^{th} column in E is e_i, e.g. e_{123}.
- If we have a One-Hot vector o_i of size $V \times 1$, then:

$$e_i = E \cdot o_i$$

- In practice, use a specialized function to look up an embedding instead of multiplication.
- In Keras, there is an `Embedding` layer that allows retrieving the right embedding for the word more efficiently.

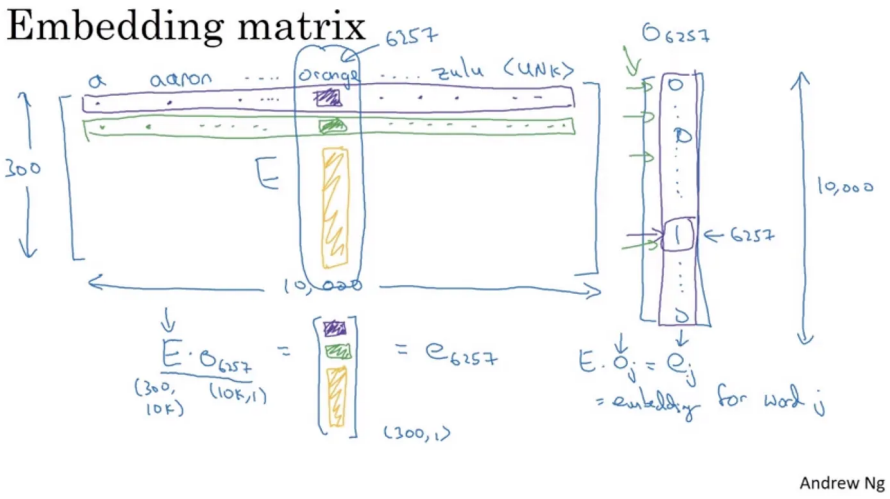


Figure 19: Embedding Matrix

How to learn word embeddings (Embedding Matrix)?

- Paper: "A neural probabilistic language model"
  Neural Language Model
- The idea is to use E as a parameter matrix in a neural network.
  - The embedding from a window/**context** (a fixed number) of words is concatenated (e.g. 4 embeddings of size 300 create a vector of size 1200) and is fed to a dense layer that is subsequently fed to a Softmax layer (with the size of the vocabulary). The word embeddings are treated as parameters in that they are part of the optimization process (e.g. Gradient Descent). The output (target value) of the neural network is the *next word in a sequence*. The inputs and the output are fed as one-hot vectors.
  - Works because the algorithm tends to make words that are used together or interchangeably (in the same context) be represented closer together in the embeddings matrix since these words tend to share the same *neighborhood*.

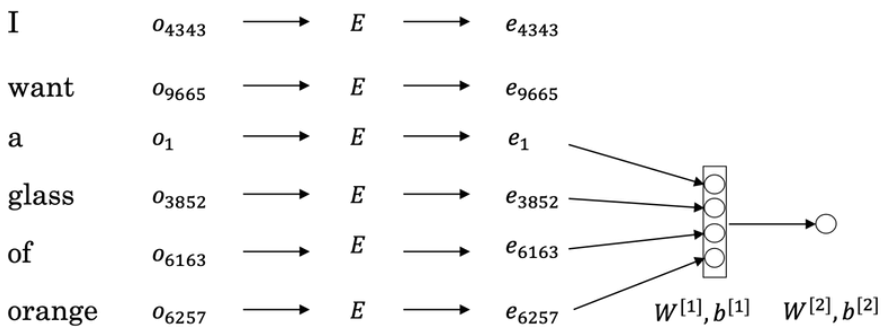


Figure 20: Neural Language Model for learning Word Embeddings

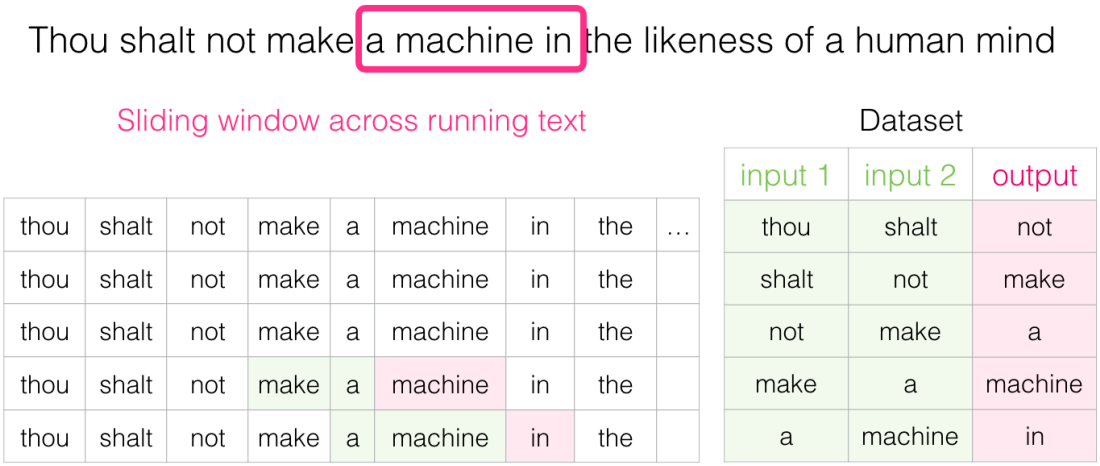


Figure 21: Sliding window technique

Different approaches for selecting the Context

- 4 words on the right
- 4 words on the right in the left (and predict the word in the middle) e.g. **CBOW** algorithm.
- The last 1 word
- Better yet, the "nearby" 1 word: **skip-grams** also works well!

1.9 Word2Vec (CBOW and Skip-Gram)

- Paper: "Efficient estimation of word representations"
- **Skip-grams**: Given a **context** word, try to estimate **target** word(s) within the neighborhood (a window) of the context word.
- **Continuous Bag-of-Words Model (CBOW)**: Predicts the middle/target word based on surrounding context words.
- Example sentence (Skip-Gram, randomly select target(s) within the neighborhood of the context (*orange*)):

  `I want a glass of orange juice to go along with my cereal.`

Context	Target
orange	juice
orange	glass
orange	my

- Imagine a neural network to guess the target word given the context word. This is a hard problem. But the objective is not to do well on the classification problem, it is to obtain good word embeddings!
- Now we take the **context** word c (e.g. "orange") and a **target** word (e.g. "juice") t. (O is a one-hot representation):

$$O_c \rightarrow E \rightarrow e_c \rightarrow softmax \rightarrow \hat{y}$$

$$\text{Softmax: } p(t|c) = \frac{e^{\theta_t^\intercal e_c}}{\sum_{j=1}^{V} e^{\theta_j^\intercal e_c}}$$

Where θ_t is the parameters of Softmax associated with the output t, that is the chance that output t is the label. V is the vocab size.

The loss function will be:

$$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{V} y_i \log(\hat{y}_i)$$

Even though there are parameters θ_t, the resulting embeddings E are still pretty good.
Both context and target are fed as One-Hot encoded vectors.

- The problem is the computational cost of summation in Softmax, even for a V=10k word vocabulary.
- A possible solution (in the literature) is to have a **hierarchical** Softmax classifier, that splits the words in the vocabulary into a binary tree and at each step tries to decide the branch to which the word should be long. This is actual $O(log(V))$.
- In practice the hierarchical Softmax classifier doesn't use a balanced tree, it instead is developed so that the most common words are on top, whereas the least frequent words are deeper.

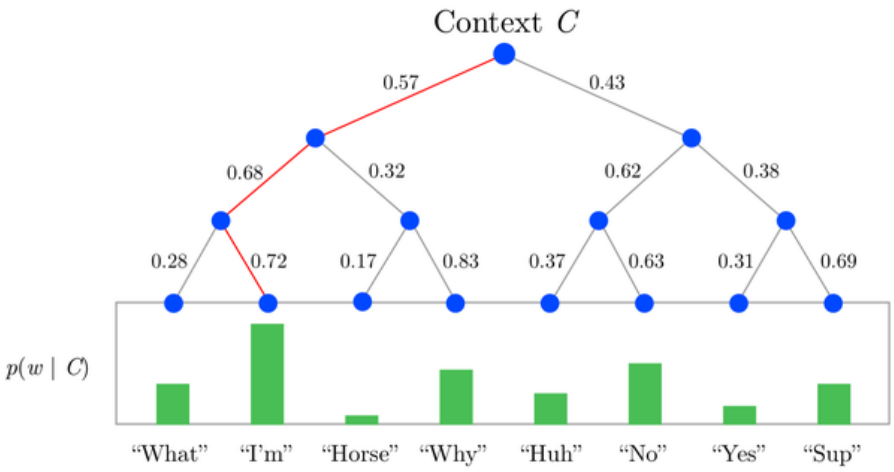


Figure 22: Hierarchical Softmax

Notes on sampling context c

- Some words are extremely frequent (the, of , a, and, to, ...) while others are less frequent (orange, apple, durian, ...)
- In practice the distribution P(c) is not random, it instead must balance out between more frequent and less frequent words.

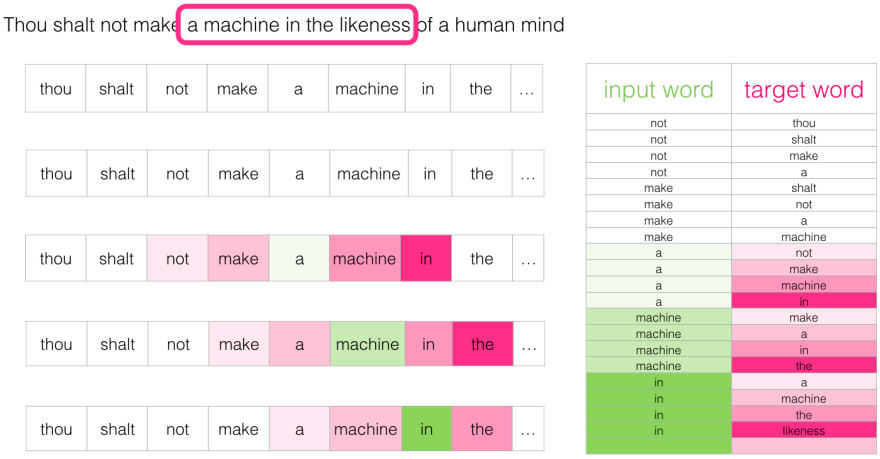


Figure 23: Sliding Window for Skip-Gram

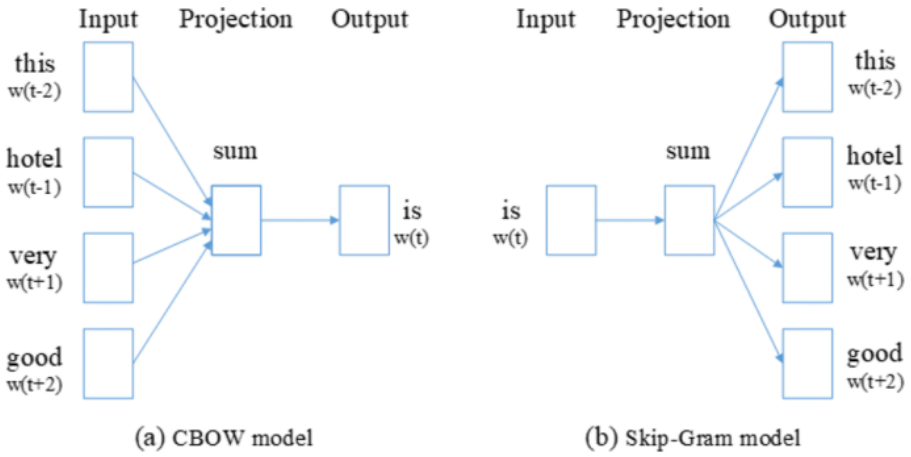


Figure 24: CBOW vs. Skip-Gram

1.10 Negative Sampling (Skip-Gram revised)

- Paper: "Distributed representation of words and phrases and their compositionality"
- Similar to skip-gram in performance but much faster.

  `I want a glass of orange juice to go along with my cereal.`

- Switch the model's task from predicting a neighboring word to a model that takes the input and output word and outputs a score indicating if they're neighbors or not (0 for "not neighbors", 1 for "neighbors"). A supervised binary classification problem.
- Similarly to skip-gram, we first choose a context word c, then choose a random target word T and set a boolean "target?" flag to one (**positive example**).
- We then choose k random words from the vocabulary and set the "target?" flag to 0 (**negative examples**). It is not a problem if by chance the random word is in the neighborhood of the context word (such as the word "of" in the table below).
  - $k \in [5, 20]$ for smaller datasets
  - $k \in [2, 5]$ for larger datasets

The result is a table as follows:

Context	T Word	Target?
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0
...	...	...

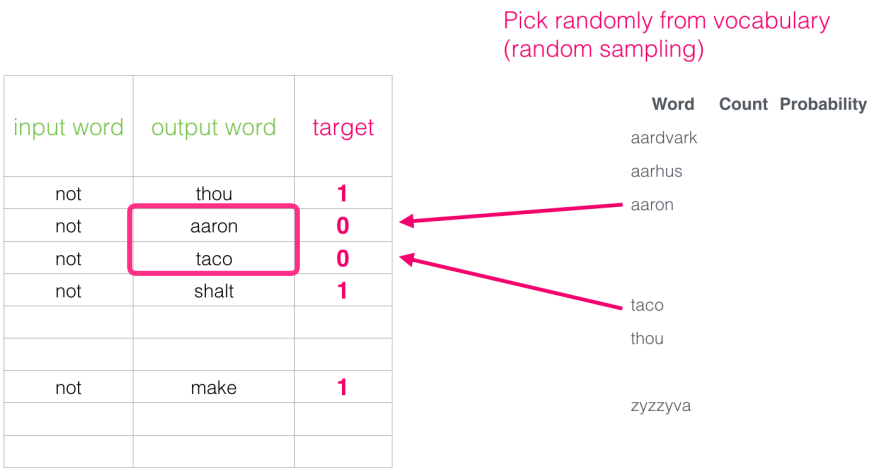


Figure 25: Negative Sampling with $k = 2$

The model uses the logistic function instead of Softmax and becomes:

$$P(y = 1|c, t) = \sigma(\theta_t^\intercal e_c)$$

$$O_c \rightarrow E \rightarrow e_c \rightarrow logistic \rightarrow \hat{y}$$

- This model must be thought of as training many individual logistic regression models, as many as the size of the vocabulary. V logistic regression to determine each word of the vocabulary is the target of our context or not.
- We update k (negative) + 1 (positive) logistic models in one iteration (one context word).
- Random sampling would pick up the more frequent words (of, the, and, ...) most of the time, thus under-representing less frequent words.
- Uniform sampling of each word in the vocabulary would give an even chance to each word $\frac{1}{|V|}$ (where V is the size of the vocabulary). But that would under-represent the most frequent words. To sample words as a function of their frequency ($f(w_i)$) using the special ratio parameter 3/4:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{V} f(w_j)^{3/4}}$$

1.11 GloVe (global vectors for word representation)

- Paper: "GloVe: Global vectors for word representation"
- Not as used as the Word2Vec model, but gaining momentum due to its simplicity.
- works based on the co-occurrence of words.

$$X_{ij} = \text{number of times } i \text{ appears in the context of } j$$

- Depending on how context and target word are defined it could happen that $X_{ij} = X_{ji}$

Model

$$\text{minimize} \sum_{i=1}^{V}\sum_{j=1}^{V} f(X_{ij})(\theta_i^\intercal e_j + b_i + b_j - \log(X_{ij}))^2$$

- Where $\theta_i^\intercal e_j$ can be seen like the $\theta_t^\intercal e_c$ of the skip-gram model, though they are symmetrical and should be randomly initialized.
- b_i and b_j are bias terms.
- V is the size of the vocabulary.
- $f(X_{ij})$ is a weighting term, that:
  - is 0 if $X_{ij} = 0$, and we don't need to calculate $log(X_{ij})$
  - Compensates the imbalance due to words that are more frequent (e.g. the, is, of , a,..) than others (e.g. durian).
- In this model θ_i and θ_j are symmetric, and therefore to calculate the final embedding for each word we can take the average:

$$e_w^{(final)} = \frac{e_w + \theta_w}{2}$$

1.12 Interpretability of word embeddings

- Methods used to create word an embedding matrix like Skip-Gram or GloVe do not guarantee that the embeddings are humanly interpretable. In other words, the rows of the embedding matrix will not have a humanly understandable meaning such as the rows of the motivational table that was previously used:

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1.00	1.00	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
...	...	...	...	...	...	...

- In algebraic terms, it is not even possible to guarantee that the rows of the embedding matrix generated by these methods are orthogonal spaces.

- For example, in the case of GloVe, it is not possible to guarantee that there is a matrix A, given GloVe's parameters θ_i and e_j, such that:

$$(A\theta_i)^\intercal(A^{-\intercal}e_j) = \theta_i^\intercal A^\intercal A^{-\intercal}e_j = \theta_i^\intercal e_j$$

Further reading on Word Embeddings:
You can find detailed tutorials about Word Embeddings on my website:

- What are Word Embeddings and how do they work? An introduction to Word2Vec (CBOW and Skip Gram)
- What is Word2vec word embedding?
- Understanding Word2vec embedding with Tensorflow implementation
- Understanding GloVe embedding with Tensorflow implementation

1.13 Sentiment classification

Simple sentiment classification model

$$O_c \rightarrow E \rightarrow e_c \rightarrow Average \rightarrow softmax \rightarrow \hat{y}$$

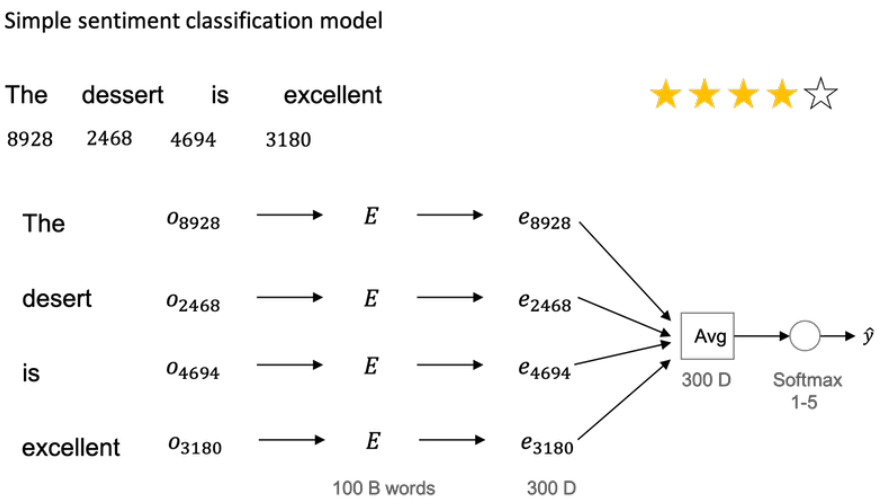


Figure 26: Simple Sentiment Classification Model

- In this model, the average is calculated for all the dimensions of the embedding vectors corresponding to a text's words, producing a single vector with the same dimensions as a single embedding vector.
- This embedding average vector is then fed to Softmax output layer with as many units as the classes that we are trying to identify (e.g. 5 for 5 star review systems).
- It completely ignores word order, thus having bad results when order matters.

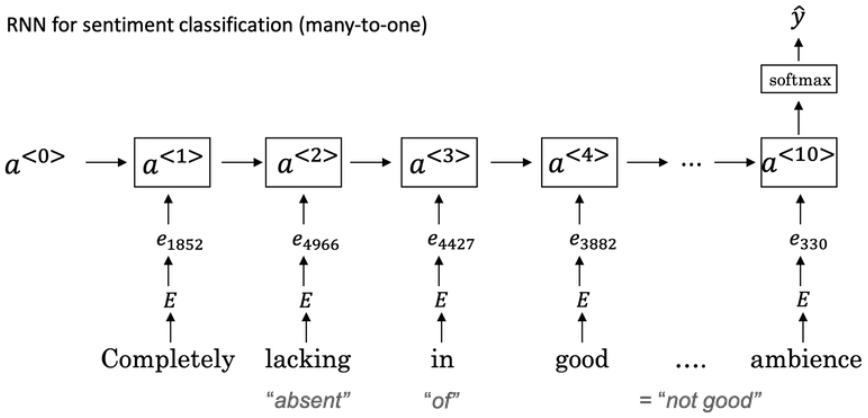


Figure 27: Sentiment Classification with RNNs

RNN for sentiment classification For each word at position t:

$$O_c^{<t>}(\text{One-Hot}) \rightarrow E^{<t>}(\text{Embedding}) \rightarrow e_c^{<t>} \rightarrow \text{RNN } a^{<t>} \rightarrow softmax \rightarrow \hat{y}$$

- We take the softmax value for the last word (the hidden state of the latest time step) as the final output.

1.14 Debiasing word embeddings

- Paper: "Man is to computer programmer as woman is to homemaker? Debiasing word embeddings"

Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model. e.g.:
`Man:Computer_Programmer as Woman:Homemaker`
`Father:Doctor as Mother:Nurse`
Even if the same embeddings work in the case of:
`Man:Woman as King:Queen`

There are some words that intrinsically capture gender: (grandmother, grandfather), (girl, boy), and (she, he) are gender intrinsic in the definition. There are other words like doctor and babysitter that we want to be gender-neutral.

Steps to address the bias problem in word embeddings

1. Identify the bias direction

   - For example for gender, take a few gender embedding pairs for *definitional* words as:
     $e_{he} - e_{she}$
     $e_{male} - e_{female}$
     $e_{grandfather} - e_{grandmother}$
     $e_{...} - e_{...}$

   - Take the average of such embeddings, to identify the bias direction (in practice this is done with **SVD - Single Value Decomposition**, since the bias direction can be multidimensional)

- The Non-bias direction is orthogonal to the bias direction.

2. Neutralize: For every word that is not definitional (e.g. not like grandmother and grandfather), project (in the non-bias axis) to get rid of bias. See Figure (28).

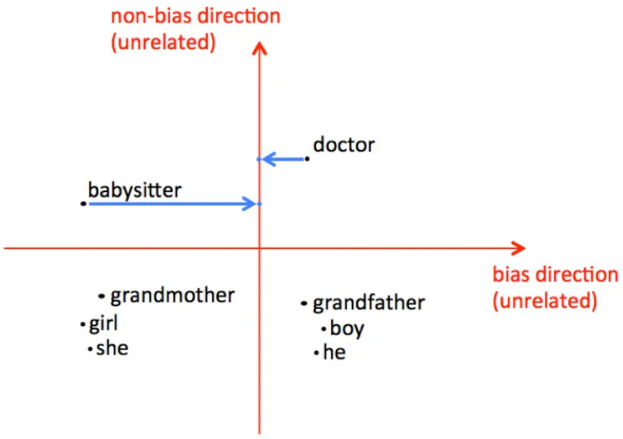


Figure 28: Neutralizing step - Addressing Bias

3. Equalize pairs - For example, after projecting the (non-definitional word) "babysitter" in the non-bias axis, move "grandmother" and "grandmother" so that they have the same distance from the non-bias and bias axis, hence having the same distance from the word "babysitter". See Figure (29).

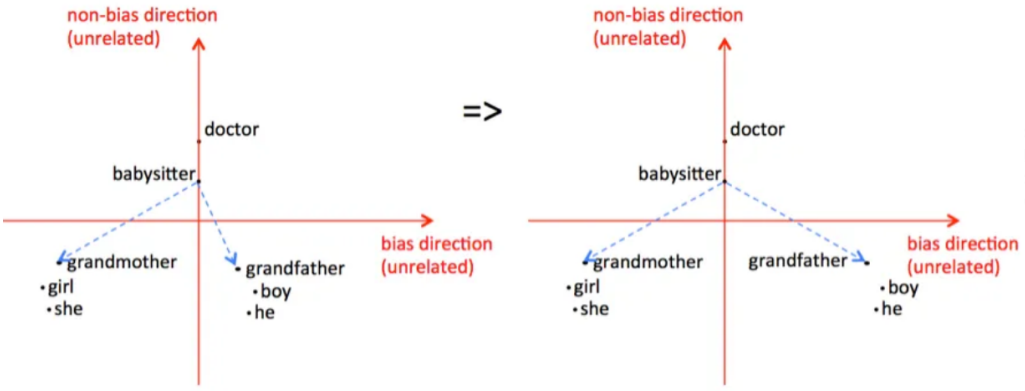


Figure 29: Equalizing pairs - Addressing Bias

Challenge: What words are definitional? Most words are not definitional for a specific bias. It is possible to train a classifier to identify words that are definitional for a specific bias (e.g. grandmother/grandfather for gender).

1.15 Basic Sequence Models

Sequence to sequence models

- Paper: "Sequence to sequence learning with neural networks"

- Paper: "Learning phrase representations using RNN encoder-decoder for statistical machine translation"

Example: Machine translation

encoder RNN (in: sentence in language A) → decoder RNN (out: sentence in language B)

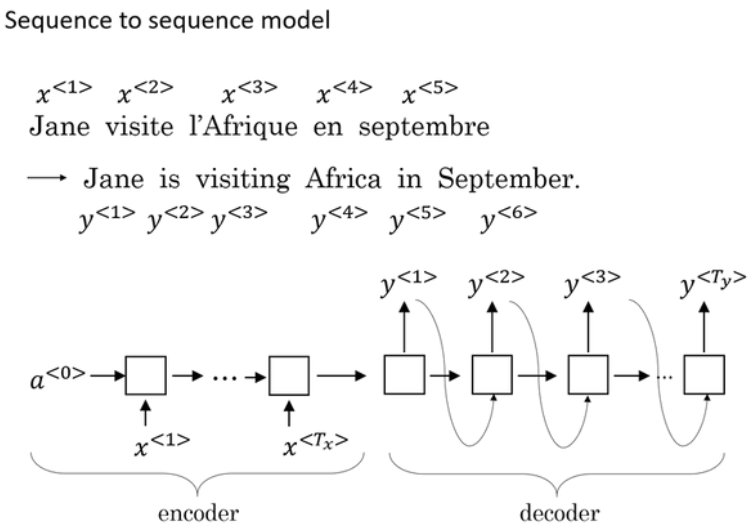


Figure 30: Machine Translation

Example: Image captioning

Conv Net (in: cat picture) → decoder RNN (out: image description)

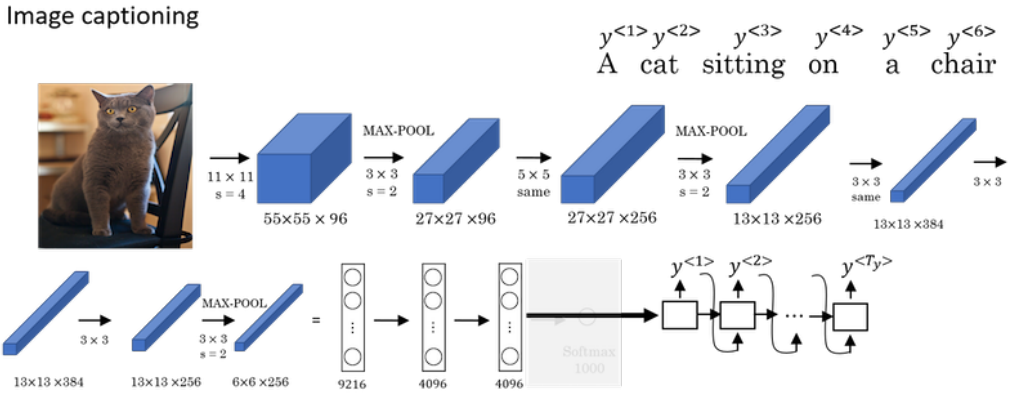


Figure 31: Image Captioning

Picking the most likely sentence

- The goals of a Language Model were: 1) To calculate the probability of a sequence $P(\hat{y}^1, \cdots, \hat{y}^{<T_y>})$ and 2) Generate a novel sequence by sampling.
- **Machine Translation example**: We are trying to translate from French (A) to English (B) with the model:

$$\text{encoder RNN (in language A)} \rightarrow \text{ decoder RNN (out language B)}$$

- The output of the encoder module is a vector which is the input of the decoder ($a^{<0>}$) of the English language model.
- The resulting model is called a **conditional language model**, which **maximizes** the conditional probability of an English sentence given a French sentence:

$$\underset{y^{<1>},\ldots,y^{<T_y>}}{\text{argmax}} \; P(y^{<1>}, ..., y^{<T_y>} | x^{<1>}, ..., x^{<T_x>})$$

- The goal of Machine Translation is to find a sequence (a sentence in English) that maximizes the above conditional probability.

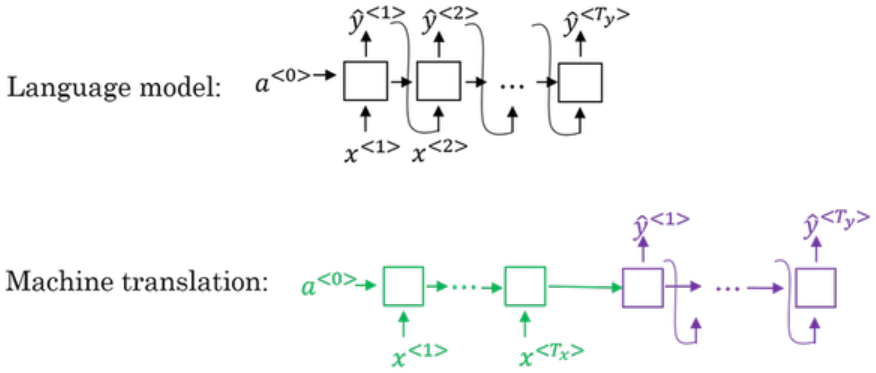


Figure 32: Language Model vs. Machine Translation

- Why not just use "greedy search", i.e. always chose the highest probability word on the decoder stage?
  - Because the sequence of all highest probability words is not necessarily the optimal sentence to maximize $P(y^{<1>}, \cdots, y^{<T_y>} | x^{<1>}, ..., x^{<T_x>})$.
- Moreover, the total number of English words in a sentence is exponentially large. It is a huge space to check all combinations of words, which is why **approximate search** algorithms are used to try to find the sentence that maximizes P.

1.16 Beam search algorithm

1. Feed the sentence in the source language (x) to the encoder, then, on the decoder module, take the most likely first B words $P(\hat{y}^{<1>}|x)$.
2. For each of the first B words selected, feed $\hat{y}^{<1>}$ as the input for the next stage $t = 2$ in order to get $\hat{y}^{<2>}$, and calculate the top B combinations of $\hat{y}^{<1>}$ and $\hat{y}^{<2>}$ that maximize:

$$P(\hat{y}^{<1>}, \hat{y}^{<2>}|x) = P(\hat{y}^{<1>}|x)(\hat{y}^{<2>}|x, \hat{y}^{<1>})$$

3. So now we get 3 combinations of $\hat{y}^{<1>}$ and $\hat{y}^{<2>}$, and for each of these combinations we want to find the top B combinations of $\hat{y}^{<1>}$, $\hat{y}^{<2>}$ and $\hat{y}^{<3>}$ that maximize $P(\hat{y}^{<1>}, \hat{y}^{<2>}, \hat{y}^{<3>}|x)$.

4. Repeat the step above until we get the top B combinations that maximize $P(\hat{y}^{<1>}, ..., \hat{y}^{<t>}|x)$, and finally, select only the most likely (since we now have the entire sequence).

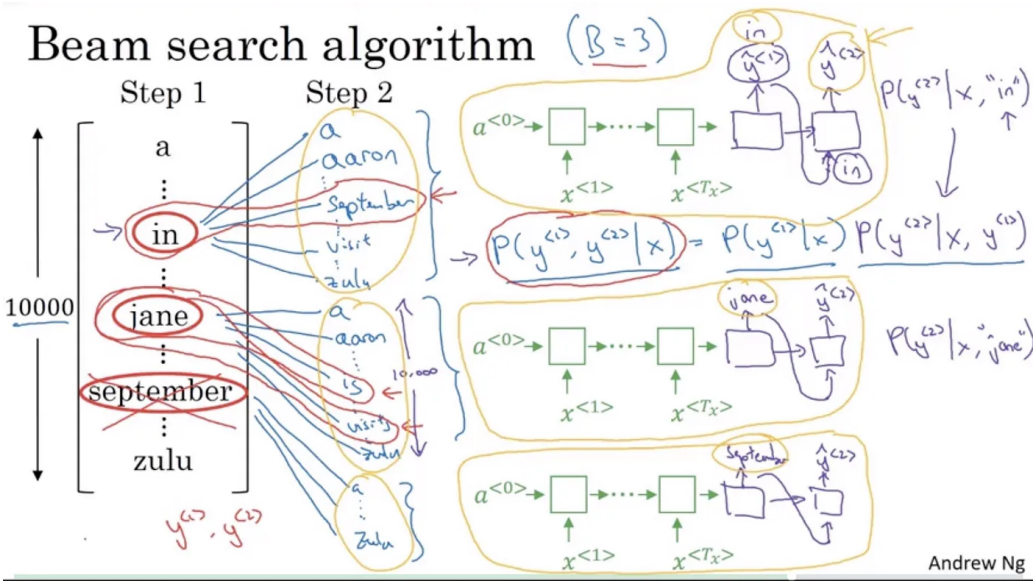


Figure 33: Beam Search

- This algorithm implies creating B copies of the network at each stage.
- If $B = 1$ then Beam search becomes a greedy algorithm, but with $B > 1$ the algorithm finds better results.
- You can find a detailed example of Beam Search at the following link:

  Example of Beam search in Sequence to Sequence models

Refinements to Beam search Generalizing the conditional probability used by the Beam search algorithm:

$$\arg\max_{y} \prod_{t=1}^{T_y} P(y^{<t>}|x, y^{<1>}, \cdots, y^{<t-1>})$$

In practice, using the sum of log we get the same result, with more numerical stability (less chance of overflow):

$$\arg\max_{y} \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \cdots, y^{<t-1>})$$

Length normalization: Since $P(.) < 1$, the $\log P(.) < 0$, then the target function has the problem that long sentences have low probabilities, so unnaturally short sentences are preferred. To solve this one thing to do is to normalize by the number of words:

$$\arg\max_y \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \cdots, y^{<t-1>})$$

where α is usually set to 0.7 based on the heuristic.

Choosing Beam width B

- Large B: better results, slower (high memory usage)
- Small B: worse results, faster (less memory usage)
- Typical values of B are 10 for production. 100 would be very large for production, but in research settings, usually, values of 1000 to 3000 are used.
- Increasing B pretty much never hurts the performance.

Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find the exact maximum for $\arg\max_y P(y|x)$.

Error analysis on Beam search Given an input sequence x ('Jane visite l'Afrique en septembre'), we feed both the output of translated sentence produced by the model $\hat{y}$ ('Jane visits Africa in September') and an optimal human-translated version y^* ('Jane visited Africa last September') to the RNN and calculate $P(\hat{y}|x)$ and $P(y^*|x)$. If it is found that the machine translation is wrong (in a meaningful way), we can try to find if the cause is either the RNN or Beam Search algorithm.

- Case 1: $P(y^*|x) > P(\hat{y}|x)$
  - Conclusion: Beam search is at fault because it could not find the best sequence of words (y) that maximize $P(y|x)$. It chose $\hat{y}$ while the y^* actually attains a much bigger value. We could try to increase B in this case.
- Case 2: $P(y^*|x) \leq P(\hat{y|x})$
  - Conclusion: RNN model is at fault because $y*$ is a better translation than $\hat{y}$, however, according to the RNN, $P(y^*|x)$ is less than $P(\hat{y}|x)$.

We could try to perform an error analysis (looking at a batch of examples with bad translation) to try to find which of these problems is more prevalent, in order to decide what we should spend our time troubleshooting.

1.17 Bleu Score

- A metric to measure the accuracy of a translated sentence according to different ground truth translations of the same sentence.
- Paper: "Bleu: A method for automatic evaluation of machine translation"

- Bleu stands for "Bilingual Evaluation Understudy". In the theater world "understudy" is someone who learns and can take the role of a more senior actor if necessary. In machine translation, it means that we find a machine-generated score that can replace human-generated translation references.

Given reference human translations $R_1, \cdots, R_n$, and the machine translation MT for a given text/sentence, the Bleu score can be generally formalized as P^n, or "Precision" for "n-grams" (unigrams, bigrams, etc.) as:

$$P_n = \frac{\sum_{\text{n-grams} \in \hat{y}} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-grams} \in \hat{y}} \text{Count}(\text{n-gram})}$$

Where:

- $\text{Count}_{\text{clip}}$: The maximum number of times that an n-gram is presented in the machine translation MT appears in any of the reference translations $R_1, \cdots, R_n$.
- Count: The number of times a specific n-gram is present in the machine translation MT.

Example:
French sentence: 'Le chat est sur le tapis.'
R_1: 'The cat is on the mat.'
R_2: 'There is a cat on the mat.'
MT: 'The cat the cat on the mat.'

Bigram	$Count$	$Count_{\text{clip}}$
the cat	2	1
cat the	1	0
cat on	1	1
on the	1	1
the mat	1	1

$P_2 = \frac{4}{6}$

Bleu details:
P_n = Bleu score on n-grams only
Combined Bleu score (using up to n-grams with $n = 4$):

$$\text{BP} \exp(\frac{1}{4} \sum_{n=1}^{4} P_n)$$

Where BP is the **brevity penalty**, which penalizes translations that are shorter than the original:

$$\text{BP} = \begin{cases} 1 \text{ if MT\_output\_length} < \text{ reference\_output\_length} \\ \exp(1 - \frac{\text{MT\_output\_length}}{\text{reference\_output\_length}}) \text{ otherwise} \end{cases}$$

1.18 Attention Model

- Paper: "Neural machine translation by jointly learning to align and translate."

- With traditional models, the Bleu score tends to fall within the length of sequences.

  Consider the French sentence:
  " Jane s'est rendue en Afrique en septembre dernier, a apprécié la culture et a rencontré beaucoup de gens merveilleux; elle est revenue en parlant comment son voyage était merveilleux, et elle me tente d'y aller aussi."

  and its English translation:
  " Jane went to Africa last September, and enjoyed the culture and met many wonderful people; she came back raving about how wonderful her trip was, and is tempting me to go too."

- The way a human translator would translate this sentence is not to first read the whole French sentence and then memorize the whole thing and then regurgitate an English sentence from scratch (this is what happens in an encoder-decoder model). Instead, what the human translator would do is read the first part of it, maybe generate part of the translation, look at the second part, generate a few more words, look at a few more words, generate a few more words, and so on.

- The Encoder-Decoder architecture works quite well for short sentences, so we might achieve a relatively high Bleu score, but for very long sentences, maybe longer than 30 or 40 words, the performance comes down. By translating long sentences in small chunks (like humans tend to do) it is possible to retain a high Bleu score independently of the size of the sentence.

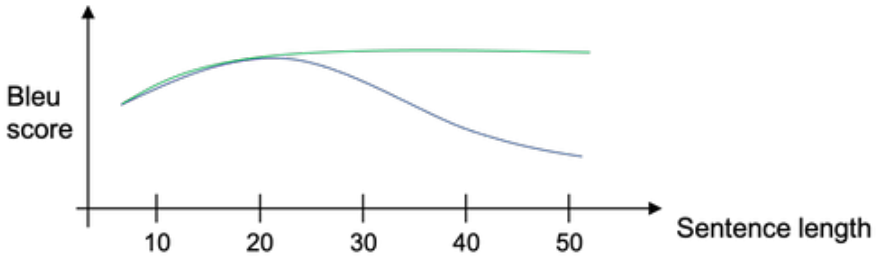


Figure 34: Bleu score drops for long sentences. Blue line: MT, Green line: Attention model

BRNNs (Bidirectional RNN) are normally used for machine translation where **Attention Models** are usually employed. With that in mind:

- Timesteps in the encoder section are referred to by t'.
- Timesteps in the decoder section are referred to by t.
- The two sets of activations of the encoder section are $a^{<t'>} = (\overrightarrow{a}^{<t'>}, \overleftarrow{a}^{<t'>})$.
- Attention weights: $\alpha^{<t,t'>}$ = the amount of "attention" to $t'^{\,th}$ input of encoder when generating the t^{th} output of decoder $y^{<t>}$. For example, $\alpha^{<1,1>}$ denotes when you're generating the first words, how much should you be paying attention to the first piece of information in the input. $\alpha^{<1,2>}$ which tells us when we are trying to compute the first word of Jane, how much attention we are paying to the second word from the input, and so on. See Figure (35).
- The input $C^{<t>}$ (or context) to each decoder unit/timestep is a weighted average of the "attention" and the outputs of the encoder units, thus allowing each decoder output to take into consideration of the closest set of words to the translated word being generated:

$$C^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>} \text{ where } \sum_{t'} \alpha^{<t,t'>} = 1$$

The $\alpha^{<t,t'>}$ parameter is in turn the application of a Softmax function (add up to one) to the exponentiation of another parameter $e^{<t,t'>}$:

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$

$\alpha^{<t,t'>}$ and $e^{<t,t'>}$ depend on $s^{<t-1>}$ (hidden state activation from the previous time step in the decoder) and $a^{<t'>}$ (features from time step t'). But we don't know what the function is. So one thing you could do is just train a very small neural network to learn whatever this function should be. And use backpropagation and gradient descent to learn the right function. Therefore, $e^{<t,t'>}$ is a parameter learned by a simple neural network (e.g. 1 dense layer) with inputs $S^{<t-1>}$ (the previous decoder state) and $a^{<t'>}$. See Figure (37).

- The downside of this algorithm is that it runs in quadratic time, that is $T_x \times T_y$, which is asymptotically $O(n^2)$. This might be an acceptable cost, assuming that sentences are not that long.
- This technique has also been applied to other problems such as **image captioning**: Paper: "Show, attend, and tell: Neural image caption generation with visual attention"
- You can find more detailed tutorials about Attention Mechanism on my website:
  - An illustrated guide to Attention Mechanism in Sequence Models with PyTorch code
  - Understanding Attention Mechanism in Sequence 2 Sequence Machine Translation
  - Understanding Attention Mechanism with example
  - Implementing Attention Mechanism in Python
  - Bahdanau and Luong Attention Mechanisms explained

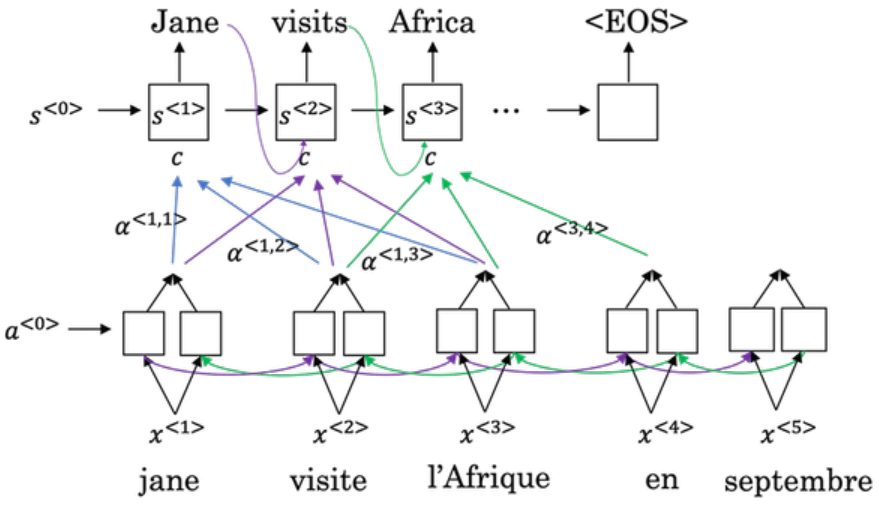


Figure 35: Attention Mechanism

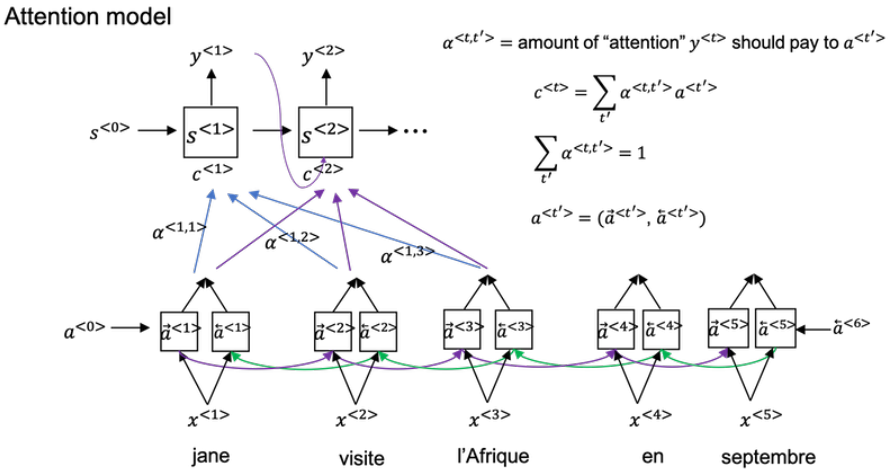


Figure 36: Attention Model

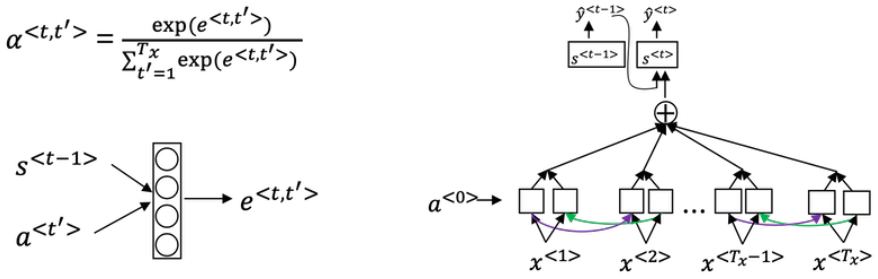


Figure 37: Attention Weights

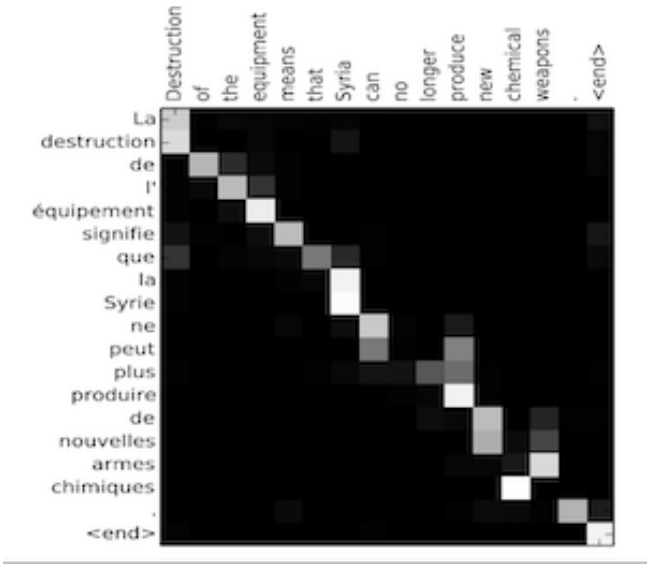


Figure 38: Visualization of attention weights

1.19 Speech recognition

- Problem: input(x): Audio clip, output(y): Transcript of the audio
- Time domain and spectral features.

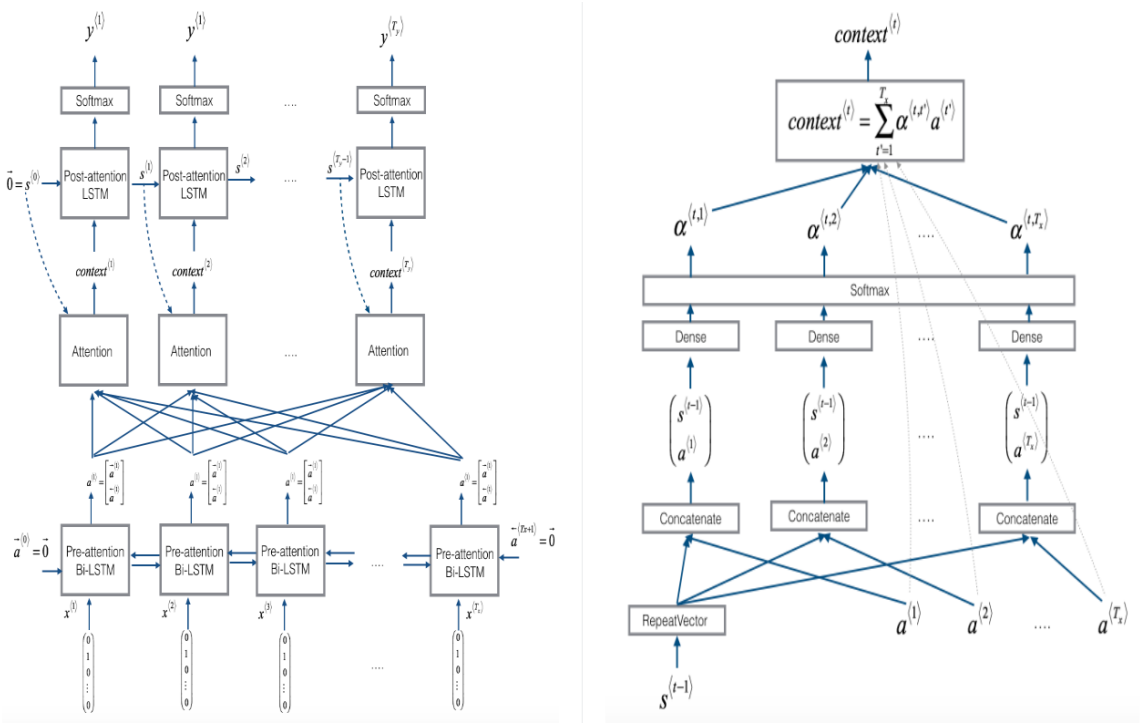


Figure 39: Example of an attention model (left) and calculation of the attention weights in a single time step (right)

Symbol	description	calculation
$a^{<t'>}$	feature for the decoder network at chunk-timestep t'	$a^{<t'>} = (\overrightarrow{a}^{<t'>}, \overleftarrow{a}^{<t'>})$
$\alpha^{<t,t'>}$	amount of attention $\hat{y}^{<t>}$ should pay to chunk-feature $a^{<t'>}$ at time step t ($\sum_{t'} \alpha^{<t,t'>} = 1$)	$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$
$c^{<t>}$	context for the decoder network at time step t	$c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>}$
$\hat{y}^{<t>}$	prediction of decoder network at time step t	

Figure 40: Attention Summary

- With deep learning phonemes are no longer needed!
- Commercial systems are now trained from 10k to 100k hours of audio!
- The attention model is applicable.

CTC cost for speech recognition

- Paper: "Connectionist Temporal Classification: Labeling unsegmented sequence data with recurrent neural networks"
- The first part of an RNN for speech recognition (normally a BRNN), uses a large number of input features and also possibly has a relatively large resolution in time, meaning that many data points are generated per time unit. Notice that the number of time steps here is very

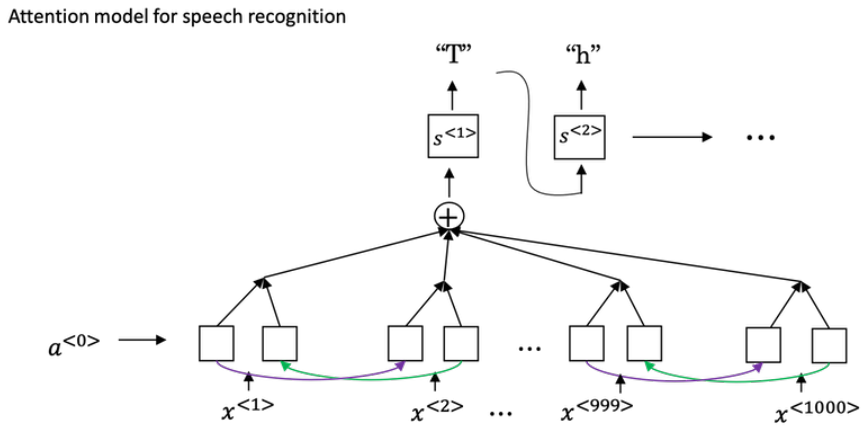


Figure 41: Attention for speech recognition

large and in speech recognition, usually, the number of input time steps is much bigger than the number of output time steps. For example, if you have 10 seconds of audio and your features come at 100 hertz so 100 samples per second, then a 10-second audio clip would end up with a thousand inputs. But your output might not have a thousand alphabets, might not have a thousand characters. The CTC cost function allows the RNN to generate a valid output like:

`ttt_h_eee___ ___qqq___ii...`

- Underscores are "blank" characters, not "spaces".
- The basic rule for the CTC cost function is to collapse repeated characters not separated by "blank".

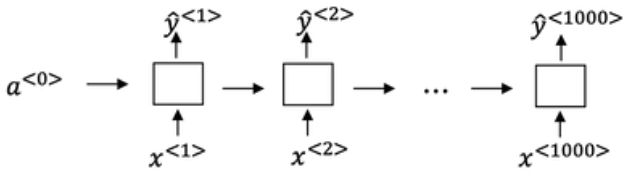


Figure 42: Speech recognition

Trigger Word Detection

- Examples: "Okay Google", "Alexa", "xaudunihao" (Baidu DuerOS), "Hey Siri"

  One possible solution:

- Create an RNN that inputs sound spectral features, and always outputs 0 except when it has detected the end of a trigger word in which case it outputs 1.
  - Problem: it creates an imbalanced training set, with a lot more 0s than 1s. Thus, "accuracy" is not a good performance metric in this case because a classifier that always returns 0 can have more than 90% accuracy. F-score, precision, and recall are more appropriate.
  - One possible hack is to make it output a series of 1s after the trigger word, not just one 1, though that doesn't solve the problem definitively.

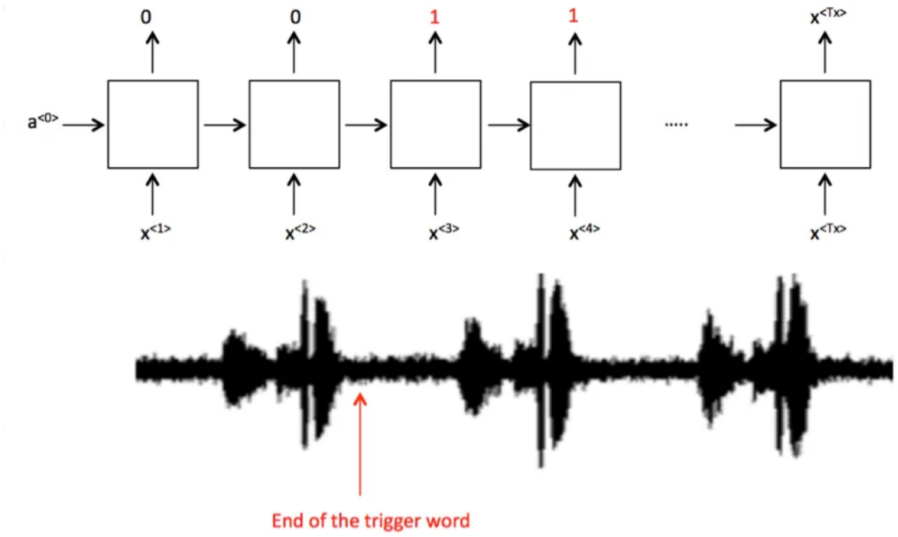


Figure 43: Trigger word detection labeling

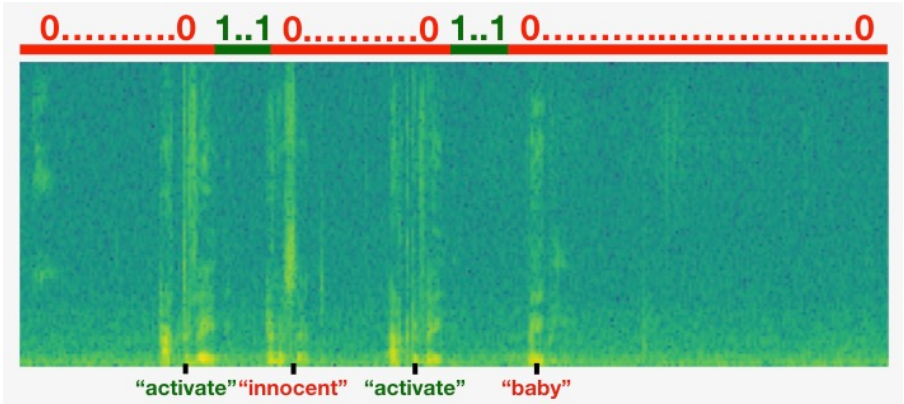


Figure 44: Trigger word detection spectrogram

References

[1] S. Zivkovic. Language Modelling and Sampling Novel Sequences. https://datahacker.rs/004-rnn-language-modelling-and-sampling-novel-sequences/.