Sequence Models

Introduction

Sequence models have transformed the quality of many ML based applications.

- Speech Recognition : Speech → Transcript
- Music : Numerical values → Notes
- Sentiment classification : Sentence → Rating
- DNA Analysis : DNA → Protein identification
- Machine Translation : English → Spanish
- Named Entity Recognition : Passage → Trigger words

Notation

Consider the following sentence. The aim of the task is to perform `Named entity recognition`, i.e. the model would sift through passages of fiction to find the locations where names/terms are mentioned. This is fairly common in search engines as well, in addition to sequence models.

X	"Harry	Potter	and	Hermione	Granger	invented	a	new	spell"
	$x^{<1>}$	$x^{<2>}$	$x^{<3>}$	$x^{<4>}$	$x^{<5>}$	$x^{<6>}$	$x^{<7>}$	$x^{<8>}$	$x^{<9>}$
Y	1	1	0	1	1	0	0	0	0
	$y^{<1>}$	$y^{<2>}$	$y^{<3>}$	$y^{<4>}$	$y^{<5>}$	$y^{<6>}$	$y^{<7>}$	$y^{<8>}$	$y^{<9>}$

The following convention will be regularly followed:

- t^{th} element of training input i will be designated as $X^{(i)<t>}$
- t^{th} element of training output i will be designated as $Y^{(i)<t>}$
- Input sequence length for input i, will be designated as $T_x^{(i)}$
- Output sequence length for output i, will be designated as $T_y^{(i)}$

In the example shown, $T_x = T_y$. To represent words in NLP models, we use a `Vocabulary` or `Dictionary`. Simply put, these are list of words which the model can use for its tasks. Typical vocabulary lists have between 10E4 to 10E6 words. Each word has a numerical location in these dictionaries based on the alphabetical order. We can use `one-hot coding` to represent words belonging to a sequence, basing on this dictionary.

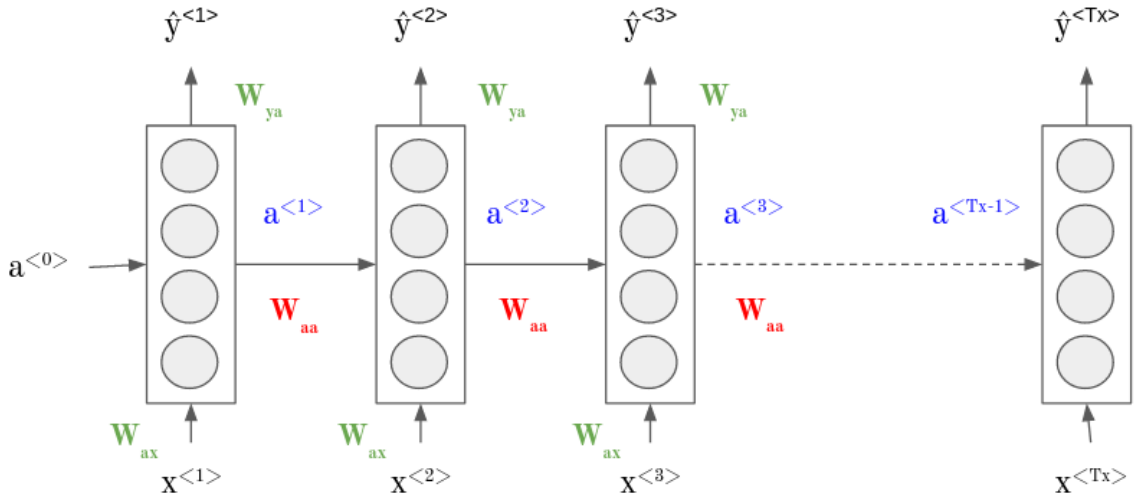
The above example shows each token of the sample sentence being one-hot coded on a dictionary of approximately 10,000 words. (The number next to the one-hot coding represents the position in dictionary according to the alphabetical order). If we encounter a new word in the sequence not belonging to the dictionary, we can set it as `<UNK>` or unknown token.

RNN Model

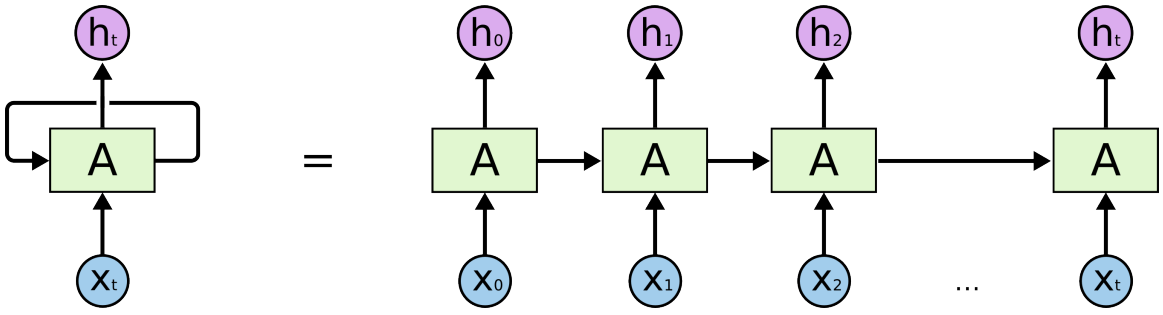
There are few reasons why standard neural network architectures cannot process sequence data.

- The length of input and output sequences (i.e. T_x and T_y) can be different in many circumstances. Standard architecture assumes we have standardized our input and output vector sizes.
- Conventional neural networks do not share features across positions. Hence, a token recurring in another part of the input passage will be treated as an entirely new entity instead of grounding this information as a common feature.
- One-hot coding approach to introduce words into machine learning models makes the size of the models exponentially huge.

To overcome these deficiencies, Recurrent Neural networks have evolved. A diagram of the same is shown below.



Say we input $x^{<1>}$ to a hidden layer and get $y^{<1>}$. When we compute $x^{<2>}$, we use the information from the activation of $x^{<1>}$ (i.e. $a^{<1>}$), to predict $y^{<2>}$ and so on. At each time step, the RNN passes on the activation to next time step. The 0^{th} term $a^{<0>}$ is usually set at random or zeros. In some research papers, this diagram is condensed to another form shown as follows.



RNN reads from left to right & the parameters it uses for each time step, are shared. These shared parameters are denoted as follows: The parameters primarily used to compute activations from inputs are W_{ax}. The horizontal connections are governed by W_{aa} and the output values are dictated by W_{ya}.

A weakness of this RNN design is that it uses inputs from earlier tokens in the sequence. Many times two sentences could be exactly same up to a particular token. In that case, the RNN cannot discriminate given the prior information. The equation to compute forward propagation are given as,

$$
\begin{aligned}
a^{<0>} &= 0 \\
a^{<t>} &= g_1\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right) \\
y^{<t>} &= g_2\left(W_{ya}a^{<t>} + b_y\right)
\end{aligned}
$$

Simplified Notation

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

We write W_a stacked side by side as,

$$W_a = [W_{aa} \mid W_{ax}]$$

Similarly,

$$\left[a^{<t-1>}, x^{<t>}\right] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$$

It is easy to visualize that the product of the above two matrices will result in the unabridged version of the equation.

Backpropagation in Time

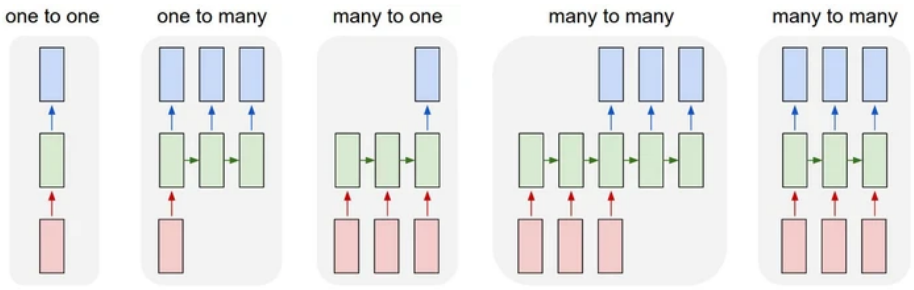
In performing back-propagation, the the parameters have to be updated in such a way that overall loss taken over all time steps is minimized. Backpropagation done in sequence models is commonly referred as `Backpropagation in time`.The loss expression for a single term can be written as,

$$L^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right) = -y^{<t>} \cdot log(\hat{y}^{<t>}) - \left(1 - y^{<t>}\right) \cdot log\left(1 - \hat{y}^{<t>}\right)$$

The overall loss is sum of all individual losses taken over all time steps. It can be expressed as,

$$\mathbb{L} = \sum_{T_x} L^{<t>}\left(\hat{y}^{<t>}, y^{<t>}\right)$$

Types of RNN

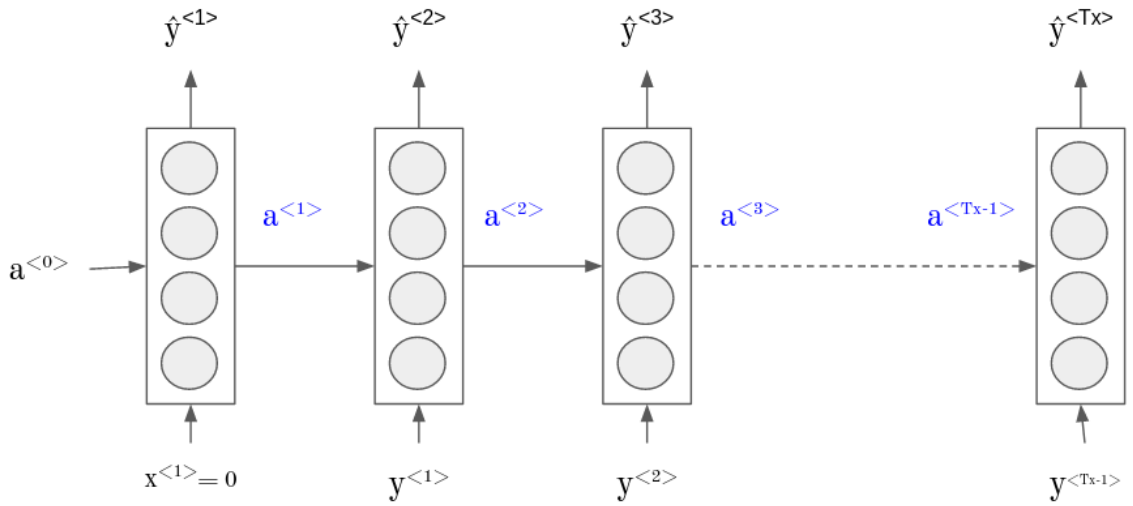


Language Models

RNN is particularly good at learning language models. Consider the following sequence in speech recognition:

The apple and Pear salad was delicious

Making good choices on the language model can help us in learning fair probabilities towards the meaning or completion of sentences. The above sentence if played from audio can be picked up as The apple and pear salad was delicious or The apple and pair salad was delicious. Obviously only one of them makes any semantic sense. We make language models by training on large corpus of texts. In doing so we usually follow these broad practices: (1) Tokenize the sentence and append `<EOS>` if necessary (2) Assign `<UNK>` to unknown words which were not present in the vocabulary.



The RNN's $a^{<1>}$ will perform a `softmax` prediction $\hat{y}^{<1>}$ on what would be the first word. (This `softmax` could be a 10,000-way, if the vocabulary size was 10,000). This $\hat{y}^{<1>}$ will be the next input $x^{<2>}$. In similar fashion, RNN could predict the next word based on the occurrence/prediction of the previous word(s). Note that $y^{<t>}$ is a 10,000-dimension one-hot vector and $\hat{y}^{<1>}$ is a 10000-dimension `softmax` output. Given an example $(y^{<1>}, ..., y^{<T_y>})$, the sample has T_y loss components given by $L(\hat{y}^{<t>}, y^{<t>})$, where $t = 1, ..., T_y$. Because each of $\hat{y}^{<t>}$ and $y^{<t>}$ is say, a 10000-dimension vector the loss associated with this example at the time step t will be,

$$L(\hat{y}^{<t>}, y^{<t>}) = -\sum_{i=1}^{10000} \left(y_i^{<t>} log(\hat{y}_i^{<t>})\right)$$

The index t runs over the length of vocabulary. This is loss value for each token. The overall prediction is given by summing over all the tokens.

$$\mathbb{L} = \sum_{t=1}^{T_x} \left(\hat{y}^{<t>}, y^{<t>}\right)$$

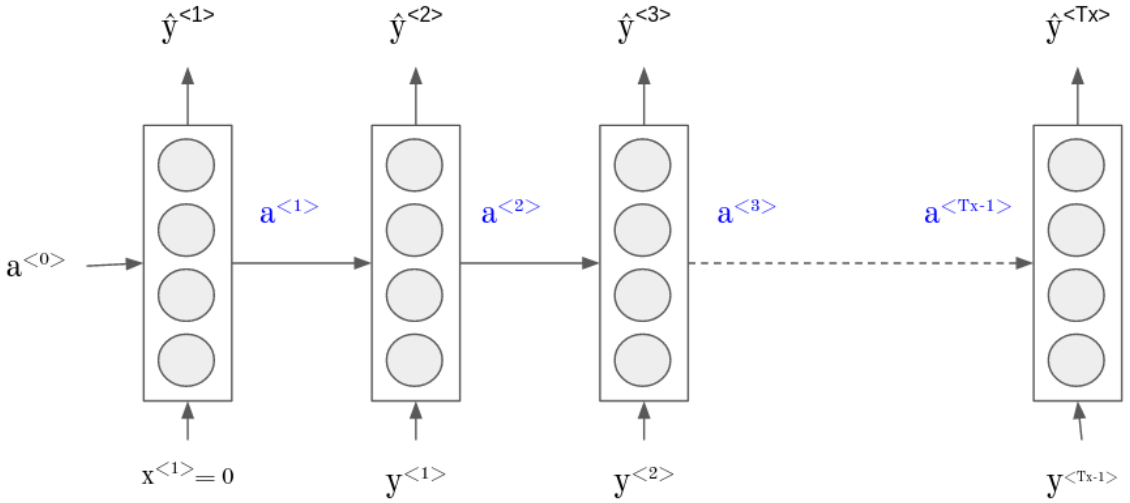
A Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks". Version available at http://karpathy.github.io/2015/05/21/rnn-effectiveness/

The sequence generation is a Markov process which can be written as,

$$P\left(y^{<1>}, y^{<2>}, y^{<3>}\right) = P\left(y^{<1>}\right) \cdot P\left(y^{<2>}|y^{<1>}\right) \cdot P\left(y^{<3>}|y^{<2>}y^{<1>}\right)$$

Sampling Novel Sequences

Sequence models learn any particular $P\left(y^{<1>}, y^{<2>} \dots y^{<N>}\right)$. Such models have to be trained with a large corpus of text as supervised learning. For sampling, if we follow the aforementioned



design, then $\hat{y}^{<1>}$ will give the probabilities of the first word. We then pass the softmax outputs as input to next step (instead of a usual one-hot coded input) and keep doing this until we reach the desired number of time steps. We can reject `<UNK>` token whenever that arises. This method can be called *Word-level RNN*. Alternately, we can even have *Character-level RNN*, where vocabulary is composed of [`a, b, c ... z, 0, 1 ... 9`]. We do not have to worry about `<UNK>` in such cases, but it can create a lot of noise in word generation.

Vanishing Gradients with RNN

Basic RNNs run into vanishing gradients problem. Consider the following,

The soldier who fought valiantly in battle, was very exhausted

The soldiers who fought valiantly in battle, were very exhausted

There is long term dependency in such sentences. Suppose we have a very deep network to process such sentences for MT, the gradients diminish sufficiently by the time they reach earlier token. A token early in the sequence will consequently have very little effect on the forward and back propagation. In absence of long term signal propagation, the dependency may not be very effective. If we do not address it, then RNN will be ineffective in processing meaningful sentences.
Exploding gradients also happen at times, but they can be spotted by the occurrences of `NaNs` in the matrices. We can use Gradient clipping to prevent them. Vanishing gradients are much harder to spot.

Gated Recurrent Unit(GRU)

Gated Recurrent Units (GRU) are a modification of the basic RNN. It introduces a new variable c, which is a `memory cell`, i.e. a tiny memory unit to remember token properties. Let, $c(t) = a(t)$. At every time step t,

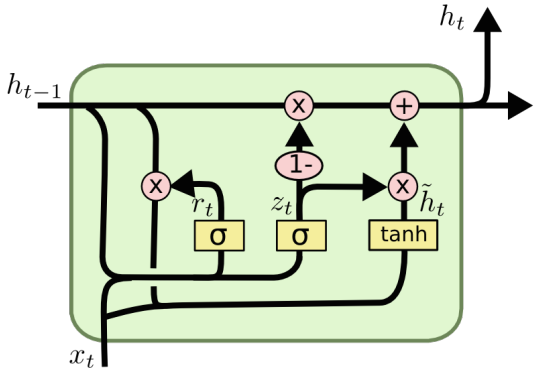
$$\tilde{c}^{<t>} = tanh\left(W_c\left[c^{<t-1>}, x^{<t>}\right] + b_c\right)$$
$$\Gamma_u = \sigma\left(W_u\left[c^{<t-1>}, x^{<t>}\right] + b_u\right)$$

Γ_u has binary values of 0 or 1, and it decides whether to update $c^{<t>}$ with $\tilde{c}^{<t>}$.

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

$\tilde{c}^{<t>}$ updates $c^{<t>}$ only when the update gate variable is 1, else the previous value of $c^{<t>}$ continues. When the GRU is set at a particular token, it will memorize through the variable c to be used as context for later token. The gate is quite easy to be set at 0. For most of the time, the value of $c^{<t>}$ will be maintained. Only when the matrix multiplication produces a very large value, does the gate switches context.

$c^{<t>}$ can be a vector. $c^{<t>}, c^{<t-1>}$ and Γ_u have similar dimensions. The multiplication of $c^{<t>}$ and Γ_u is a element-wise multiplication.



GRU schema

Simple model

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

Advanced model

$$\Gamma_r = \sigma\left(W_r\left[c^{<t-1>}, x^{<t>}\right] + b_r\right)$$
$$\tilde{c}^{<t>} = tanh\left(W_c\left[\Gamma_r * c^{<t-1>}, x^{<t>}\right] + b_c\right)$$
$$\Gamma_u = \sigma\left(W_u\left[c^{<t-1>}, x^{<t>}\right] + b_u\right)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * tanh(c^{<t>})$$

We add a relevancy gate additionally in the full model.

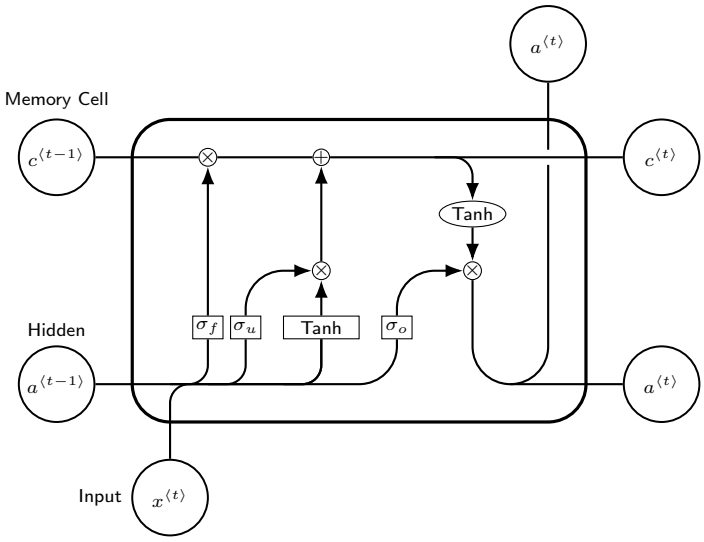
Kyunghyun Cho et al. (2014), "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches", https://arxiv.org/abs/1409.1259

Chung et al. (2014), "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling", https://arxiv.org/abs/1412.3555

Long short-term memory (LSTM)

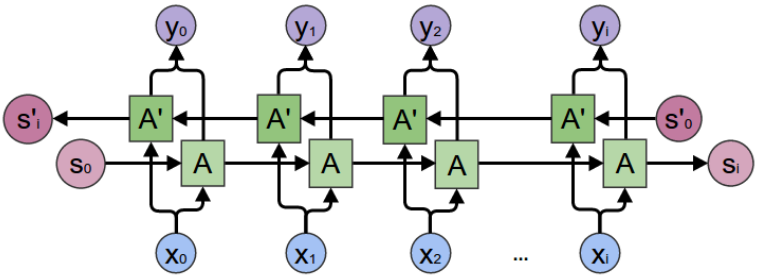
LSTM has 3 gates: `Update, Forget and Output` gates.

$$
\begin{aligned}
\tilde{c}^{<t>} &= tanh\left(W_c\left[a^{<t-1>}, x^{<t>}\right] + b_c\right) \\
\Gamma_u &= \sigma\left(W_u\left[a^{<t-1>}, x^{<t>}\right] + b_u\right) \\
\Gamma_f &= \sigma\left(W_f\left[a^{<t-1>}, x^{<t>}\right] + b_f\right) \\
\Gamma_o &= \sigma\left(W_o\left[a^{<t-1>}, x^{<t>}\right] + b_o\right) \\
c^{<t>} &= \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} \\
a^{<t>} &= c^{<t>}
\end{aligned}
$$



By having some LSTM cells in sequence, we can connect them temporally for more complex networks. It is relatively easy for LSTMs to propagate $c^{<t>}$ a long way, memorizing the long term dependency. This is the basis of the term *peephole* connection when referring to these LSTM chains.

Bi-directional RNN



Bi-directional RNN allows users to pick up information from both future and previous time steps. Often, to infer about a token we require the information coming after the token, and not just the

"Understanding LSTMs", Chris Olah. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
"Long Short-Term Memory" (1997), Schmidhuber et al., https://doi.org/10.1162/neco.1997.9.8.1735

previous ones. Such networks have a forward recurrent component as well as backward recurrent component [`Refer: Acyclic graphs`]. The forward component goes from $a^{<1>} \rightarrow a^{<N>}$, and the backward component $a^{<1>} \leftarrow a^{<N>}$. Prediction is made by both forward and backward activation components.

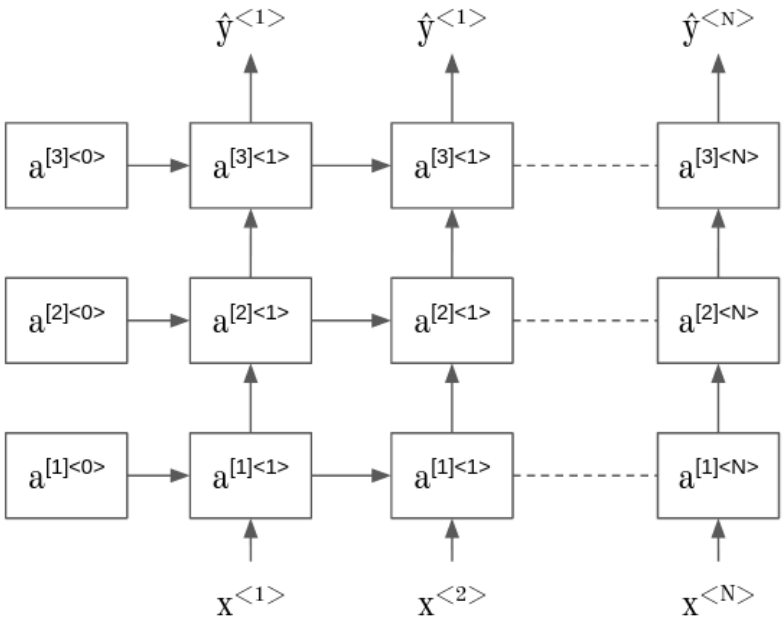
$$\hat{y}^{<t>} = g\left(W_y\left[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}\right] + b_y\right)$$

These blocks can be GRUs or LSTMs or even combination of both. There is only one common disadvantage: The entire sequence is required for the bi-directional model to work. On the fly inference is not available by design. These models have a lot of applications in text processing.

Deep RNNs

It is often valuable to stack RNNs together for learning better models. Any random unit is fed by two inputs - temporal and lateral.

$$\boxed{a^{[N]<l>} = g\left(W_a^{[N]}\left[a^{[N]<l-1>}, a^{[N-1]<l>}\right] + b_a^{[l]}\right)}$$



Word Embedding

Word embedding allow algorithms to understand meaningful relationship between words, so that similar word, substitute etc. can be deduced. So far, in a given vocabulary $\mathbb{V}$, each word is a one-hot coded distinct object. In a pair of sentences as the one shown,the model will not find any pattern or similarity, and may even end up choosing different words.

I want a glass of orange ☐

I want a glass of apple ☐

One reason for the same is that inner-product between any two one-hot coded vectors from a dictionary is zero. *Apple* and *Orange* are no similar than *Apple* and *King*.

Feature enhanced Representation

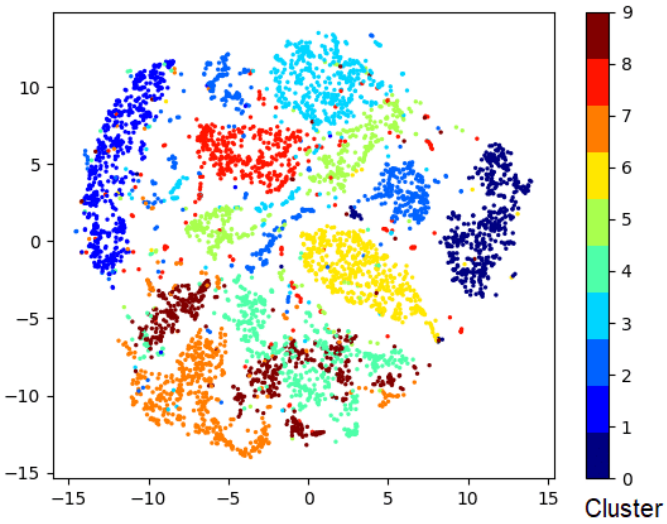
Suppose we add further information about each word in the vocabulary.

	0_{5391} Man	0_{9853} Woman	0_{4914} King	0_{7157} Queen	0_{456} Apple	0_{6257} Orange	0_{930} Car
Gender	-1	1	-1	1	0.98	0.99	0.98
Rank	0.30	0.31	0.95	0.96	-0.01	0.00	0.17
Age	0.03	0.03	0.69	0.70	0.04	0.06	0.20
Food	0.0	0.0	0.0	0.0	0.96	0.97	-0.05
Transport	0.0	0.0	0.0	0.0	0.0	0.0	0.99

We can come up with many different attributes for each word. The different attributes can be used as a *vector of description*. These feature-rich representations are called *embedding*, because they are embedded at a point in a N-dimensional space. By such representations, we can find similarity or relationships between words.

$$e_{5391} = \begin{bmatrix} -1 \\ 0.30 \\ 0.03 \\ 0.0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{Man} \qquad e_{9853} = \begin{bmatrix} 1 \\ 0.31 \\ 0.03 \\ 0.0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{Woman}$$

This feature space may be difficult to interpret, but we have tools such as `t-SNE` which reduces a large dimensional embedding to 2-D for analysis.



Word embedding can establish similarity between concepts by learning on large corpus of data. In the following example of named entity recognition, the model can deduce that *farmer* and *cultivator* are semantically similar.

"Visualizing Data using t-SNE", Hinton et al. (2008), JMLR

Sally Jones is a paddy farmer

Nancy Smith is a orange cultivator

For named entity recognition, we can learn our embedding on a large word corpus (100 Billion) to provide reasonable output on our target task. We can use transfer learning for the new task, which is assumed to have smaller training set (100,000 words). Optimally, we can keep fine-tuning our embedding in the target task after transfer learning.
Note: Word embedding in sequence models are similar to encoding in vision tasks. In embedding, we learn a fixed vector for each word in the vocabulary, whereas encoding is performed for every image input.

Properties of Word Embedding

Embedding can help in analogy and reasoning. Say we want to deduce analogy of the form,

Man→Woman, then King → ?

We can match a suitable word, by finding the difference between the given two words first. Thereafter we can look for words with similar difference with our target word from the embedding.

$$e_{man} - e_{woman} = \begin{bmatrix} -2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad Man \rightarrow Woman \approx King \rightarrow Queen$$

More formally, we can write the relationship of finding the candidate word, given two words as

$$\underset{w}{arg\,max}\ sim\left[e_w, e_{target} - e_1 + e_2\right]$$

Similarity metric

Two simple metrics to perform similarity are,

Cosine similarity

$$sim(u, v) = \frac{u^T v}{||u||_2\,||v||_2}$$

Euclidean similarity

$$sim(u, v) = ||u - v||^2$$

Embedding Matrix

For NLP, we learn the embedding matrix for a vocabulary. Suppose we have a vocabulary of N words given as V,

$$\mathbb{V} = [a, as, \ldots zulu, \langle UNK \rangle]$$

If we learn M attributes for each word, we could create a $M \times N$ matrix, which is called the *Embedding Matrix*.

$$E = \begin{bmatrix} a & \ldots & zulu \\ \vdots & \ddots & \vdots \\ 0.97 & \ldots & -0.01 \end{bmatrix}$$

Mikolov (2013), "Linguistic Regularities in Continuous Space Word Representations", NAACL

An interesting property of this matrix is that if we multiply the one-hot coded position of a word with the matrix, we get the word embedding for the word.

$$E_{M\times N} \cdot O_{N\times 1} = e_{M\times 1}$$

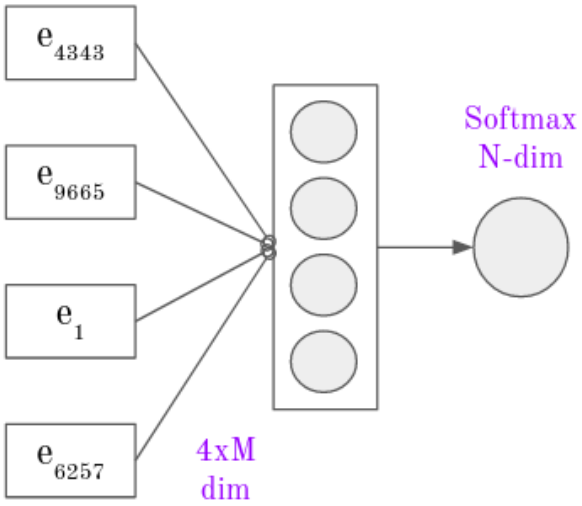
Our goal is to learn this Embedding Matrix $\mathbb{E}$.

Learning the Embedding Matrix $\mathbb{E}$

Making a language model is the logical way to create this word embedding. . Suppose we have a coarse or arbitrary matrix to represent $\mathbb{E}$, where each embedding is of M dimensions. We rely on a large corpus of text to learn this matrix, by asking the a language model to predict correct words, and use a supervised learning approach comparing against ground truth of text. To make the approach standardized we choose fixed number of trailing words to that position, to the feed-forward (FF) network. The output of the network is a `softmax` (SM) unit which produces probabilities of each word in the vocabulary.Tuned parameters are $\mathbb{E}$, $W^{[FF]}, b^{[FF]}, W^{[SM]}, b^{[SM]}$. Over several clips of the text corpus the language model can be tuned, generating a word embedding matrix $\mathbb{E}$.

I want a glass of orange ?

I	0_{4343}	$\xrightarrow{E}$	0_{4343}
want	0_{9665}	$\xrightarrow{E}$	0_{9665}
a	0_{1}	$\xrightarrow{E}$	0_{1}
glass	0_{3990}	$\xrightarrow{E}$	0_{3990}
of	0_{7970}	$\xrightarrow{E}$	0_{7970}
apple	0_{191}	$\xrightarrow{E}$	0_{191}
$\langle word \rangle$	0_{XXYY}	$\xrightarrow{E}$	0_{XXYY}



A second approach is to choose a fixed number of words previous and ahead to a word that we want to predict from the language model. This approach captures the context of usage better.

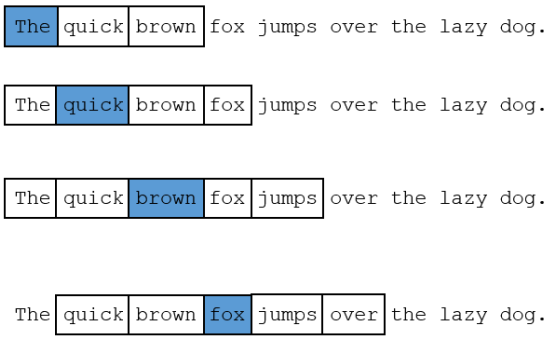
Skip-gram (Word2Vec)

In skip-gram model, rather than having last N-words as context, we randomly choose a word in a

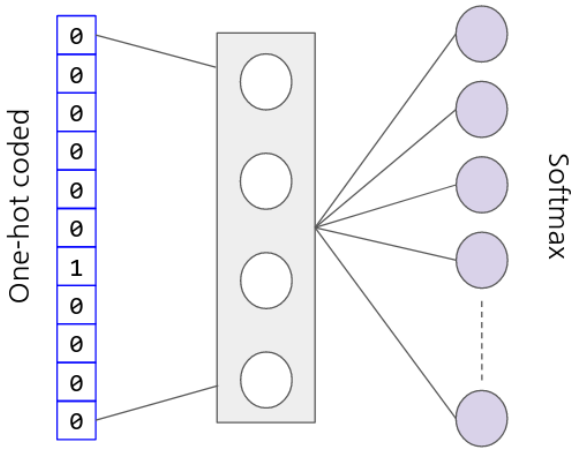
Bengio et al. (2003), "A Neural Probabilistic Language Model"
Mikolov (2013), "Linguistic Regularities in Continuous Space Word Representations", NAACL

sentence, and another as target in vicinity. We create several such context-to-target pairing. Given a context, we try to learn what is a good target by supervised learning. We're going to train a simple neural network with a single hidden layer. The goal is actually just to learn the weights of the hidden layer, which are effectively the word vectors.

`The quick brown fox jumped over the lazy dog`



Let us say we use a vocabulary of 10,000 words. We want to learn the mapping `Context C` $\rightarrow$ `Target t`. Given a specific context word in the sentence, a shallow neural network will learn the probability for every word in our vocabulary of being the nearby word that we chose. When training this network on word pairs, the input is a one-hot vector representing the input word and the training output is also a one-hot vector representing the output word.
If we are learning on 10,000 words, each with 300 features, we could have a hidden layer matrix of $300 \times 10,000$ without an activation (The one-hot coded word multiplied with this matrix then sifts out the particular word embedding). The output from operating on this hidden layer (the word embedding) is fed to a `softmax` units. Each output neuron (one per word) will produce an output between 0 and 1, and the sum of all these output values should add to 1. Let e_c denote the sifted word embedding for context word and θ_t denote the parameter for the target. The `softmax` outputs will show likely it is find each candidate word nearby our context word. Using y and $\hat{y}$, when we train the model, we can tune the embedding matrix (hidden layer weights) and the softmax units.



$$p\left(t|c\right) = \frac{e^{{\theta_t}^T \cdot e_c}}{\sum_{j=1}^{10000} e^{{\theta_j}^T \cdot e_c}}$$

$$L(y, \hat{y}) = -\sum_{i=1}^{10000} y_i \cdot log(\hat{y_i})$$

Once context word is sampled, we can choose targets in a defined window. The choice of context word is done heuristically.

Intuition

If two different words have very similar contexts (i.e., words that are likely to appear around them), then our model needs to output very similar results for these two words. And one way for the network to output similar context predictions for these two words is if the word vectors are similar. So, if two words have similar contexts, then our network is motivated to learn similar word vectors.

Sub-sampling and Negative sampling

In traditional `Word2vec`, there is one disadvantage : the structure of the network is huge and could be slow to train. There are two approaches to mitigate:

1. Sub-sampling frequent words to decrease the number of training examples.
2. Negative sampling to update only a small portion of model weights.

Sub-sampling

`Word2Vec` implements a *sub-sampling* scheme to address occurrence of common words such as *a, an, the* etc. For each word we encounter in our training text, there is a chance that we will effectively delete it from the text. The probability that we retain the word is related to the word's frequency.

$$P(w_i) = \left[\sqrt{\frac{z(w_i)}{0.001}} + 1\right] \cdot \frac{0.001}{z(w_i)}$$

Here, w_i is the word, $z(w_i)$ is the fraction of the total words in the corpus which happen to be that word. For example, if the word *peanut* occurs 1,000 times in a 1 billion word corpus, then `z(peanut) = 1E-6`.

Negative Sampling

Training a neural network means taking a training example and adjusting all of the neuron weights slightly so that it predicts that training sample more accurately. In other words, each training sample will tweak all of the weights in the neural network. The size of word vocabulary is proportional to skip-gram neural network's number of weights, all of which would be updated slightly by every one of the billions of training samples. In Negative sampling, this is addressed by having each training sample only modify a small percentage of the weights, rather than all of them. Given a pair of words, we will predict if it belongs to a legitimate context-target pair. For every positive pair, k-negative word pairs will be introduced.

Context	Target	Label
Orange	Juice	1
Orange	King	0
Orange	Car	0
Orange	Queen	0

Word2vec Tutorial, http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Mikolov et al.(2013), "Distributed Representations of Words and Phrases and their Compositionality", arxiv:1310.4546

We design a supervised learning problem, over a large text corpus. Statistically, it is then possible to learn which words appear together. We could use Logistic Regression to update only (K+1) relevant output units (1 positive-match neuron, and K negative-match neurons) instead of using all of them. This is a fraction of the the total number of parameters to update. [Only one pair corresponds to positive example and the rest are negative examples. In each iteration, (K+1) samples are trained and we leave the rest of the binary classification units intact.]

$$p\left(y=1|t,c\right)=\sigma\left({\theta_t}^T\cdot e_c\right)$$

The way of selecting a word for negative sampling is based on a heuristic technique given by the authors. If the operator `P` refers to probability of word selection and `f` is the frequency of word in the corpus,

$$P(W_i)=\frac{f(W_i)^{\frac{3}{4}}}{\sum_{j=1}^{10,000}f(W_j)^{\frac{3}{4}}}$$

Global vectors for word representations (GloVE)

GloVE is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. Previously, we sampled context and target words. Let X_{ij} be the number of times word j appears in the context of i. Therefore, $< i, j >$ play the role of $< c, t >$ respectively. It should be noted that $X_{ij} \neq X_{ji}$. GloVE algorithms optimizes the following for a vocabulary of 10000 words.

$$min\sum_{i=1}^{10000}\sum_{i=1}^{10000}f(x_i)\left[\theta_i^Te_j+b_i+b_j'-log(X_{ij})\right]^2$$

$f(x_i) = 0$ if $X_{ij} = 0$ to avoid log function encountering 0. This weighing factor is also used to appropriately scale frequent stock words such as *a, an, the* etc. The GLoVE algorithm is nice because θ_i and e_j are symmetric. They have the same optimization objective and hence can be switched without any concerns. The intuition is that we are trying to predict a word based on frequency and yet retaining the context.
Previously, we examined finding word analogy by subtracting embedding. GLoVE does not guarantee that feature dimensions will be orthogonal (i.e. the dimensions of GLoVE model could be a linear transformation of existing feature axes - making interpretation harder).

Sequence to Sequence Models

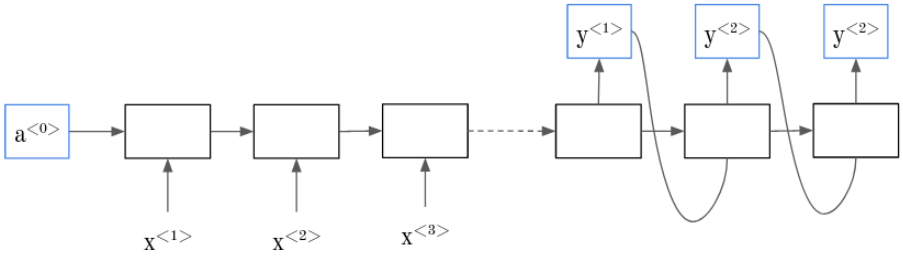
Sequence to Sequence models (`seq2seq`) run everything from machine translation (MT) to speech recognition. Consider the following French sentence and its English equivalent.

Jane visite l'Afrique en Septembre
Jane is visiting Africa in September

Pennington (2014) "Glove: Global vectors for word representation", ACL https://www.aclweb.org/anthology/D14-1162.pdf

Sutskevar (2014), "Sequence to Sequence Learning with Neural Networks", https://arxiv.org/abs/1409.3215

Kyunghyun Cho (2014), "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation", https://arxiv.org/abs/1406.1078



To translate such sentences, we should begin with an encoder RNN network which ingests the input sequence. Then we could use a decoder network. Given enough pairs of English and French, the network can learn translations well.
A very similar concept is used in visual domain in the form of *encoding*. Pre-trained CNN can hold representation of an image. In such cases, the representations can be designed to become an encoder and it could be used in conjunction with RNN as a decoder for image captioning.

Picking likely sentences

In sequence models, machine translation is building a conditional language model.

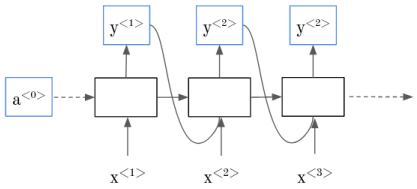


Figure 1: Language model

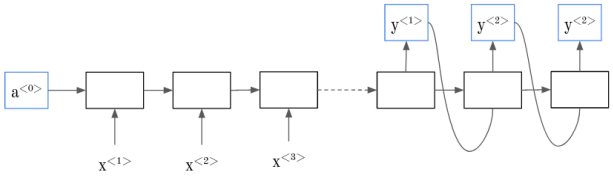


Figure 2: MT model

In a language model, we are quite plainly using the first prediction chained to the second, and so on, to find the most likely word. In general, it is an estimate of of the probability of a valid sentence $P(y^{<1>}, y^{<2>}, \ldots)$ In MT model, we have a decoder network which is identical to a Language model, but is preceded by a encoder network. It is a *conditional* language model, because the outputs are constrained by the encoded sentence i.e. $P(y^{<1>}, y^{<2>}, \ldots, y^{<2>}|x^{<1>} \ldots x^{<M>})$.
When we have $P(y^{<1>}, y^{<2>}, \ldots, y^{<2>}|x^{<1>} \ldots x^{<M>})$ to solve, we do not want the outputs at random. Although there could be various linguistic equivalents between source and target language, we are trying to maximize the log likelihood probability at the output. *Greedy search* picks whatever is the most likely word based on the conditional model, and then keeps picking the most likely word in similar fashion. This may not be very useful, as depending on the underlying language model the transcript can go out of context very easily. We want a sentence that maximizes the joint probability of the tokens $P(y^{<1>}, y^{<2>}, \ldots, y^{<2>}|x^{<1>} \ldots x^{<M>})$.

Beam Search

The Beam search is a systematic approach to finding the best pair of words which can maximize the joint probability.

Mao (2014), "Deep Captioning with Multimodal Recurrent Neural Networks", https://arxiv.org/abs/1412.6632
Vinyals (2014), "Show and Tell: A Neural Image Caption Generator", https://arxiv.org/abs/1411.4555
Karpathy (2014),"Deep Visual-Semantic Alignments for Generating Image Descriptions" https://arxiv.org/abs/1412.2306

1. Find the top words with the highest probability given the input sentence. The number of most likely words are based on the beam width (**B**)

$$P(\hat{y}^{<1>}|x^{<1>} \dots x^{<M>})$$

   - Input the encoded input sentence to the decoder; the decoder will then apply softmax function to all the 10,000 words in the vocabulary. From 10,000 possibilities, we will select only the top B words with the highest probability. B could be a small number e.g. 3.
   - Store these words in memory.

2. For each of the B choices, it will further consider B choices each from the 10,000 word vocabulary.

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

   - Take the first B selected words from [1] as input to the second step. Apply the softmax function to all the 10,000 words in the vocabulary to find the three best alternatives for the second word. While doing this, we will figure out the combination of the first and second words that are most likely to form a pair using conditional probability.
   - To find the B pairs for the first and second words, we will take the first word, apply the softmax function to all the 10,000 words in the vocabulary.
   - Evaluate the probabilities for the other (B-1) first words.
   - Run B-words $\times$ 10000 different combinations to choose top B pairs of *first-second* combination. [Drop a first word if necessary, i.e. if it no longer figures in the top-B first-second word combination. Another situation is when a first word is found in the second step, with an alternate first word i.e. from (B-1) candidates]
   - At every step, instantiate copies of the encoder-decoder network to evaluate these partial sentence fragments and the output. Number of copies of the network is the same as the size of the beam width

3. Carry on the process unless the translation is exhausted with a sequence of words having the highest joint probability.

If the beam width is set at 1, the beam search reduces to greedy search algorithm. With a wide beam width, it is possible to acquire a good translation of the input sentence.

Refinements

Length Normalization

Beam search aims at maximizing the conditional probability of words, given the input sentence.

$$arg\ \underset{y}{max} \prod_{t=1}^{T_y} P\left(\hat{y}^{<t>}|\hat{y}^{<1>} \dots y^{<t-1>}, x\right)$$

To avoid numerical underflow coming from a product of probabilities, this matter is handled computationally by taking logarithm

$$arg\ \underset{y}{max} \sum_{t=1}^{T_y} log\, P\left(\hat{y}^{<t>}|\hat{y}^{<1>} \dots y^{<t-1>}, x\right)$$

If we have a very long sentence, we end up with a small probability term. This in turn gives the method an incentive to look for shorter sentences, which may not be great translations. One way to overcome is by normalizing by the a factor proportional to number of words.

$$arg\ \max_{y} \frac{1}{(T_y)^{\alpha}} \sum_{t=1}^{T_y} log\ P\left(\hat{y}^{<t>} | \hat{y}^{<1>} \dots y^{<t-1>}, x\right)$$

Usually $\alpha = 0.7$ is a good working value. If $\alpha = 0$, then the expression becomes non-normalized. We pick sentences which score high on *Normalized log likelihood*.

Beam width

Unlike exact search methods such as BFS and DFS, beam search runs faster but is not guaranteed to find the exact maximum for the normalized conditional probability. It is heuristic. Larger value of beam width guarantees more possibilities, at the expense of slow computation. Smaller values are faster, but accuracy becomes tradeoff. To troubleshoot errors while using beam search, it is notable to compare the machine translation against human translation. If we compute the machine translation's joint probability against the one of reference truth (derived from softmax of LM), if the human/ground truth score is higher then beam search needs to be expanded. If the machine scores higher than ground truth, then it is usually a problem with the decoder.

BLEU score

Between two languages, there are many different ways of translation, all resulting in the same meaning.

French: *Le chat est sur le tapis*
English: *The cat is on the mat* English: *There is a cat on the mat*

BLEU (Bilingual Evaluation Under-study)score evaluates the quality of machine translation. So long as the translation is close to the ground truth, the algorithm scores high.

- `Precision`: It measures what fraction of MT output words also occur in all the reference sentences (without counting uniqueness).
- `Modified Precision`: Measures a ratio where MT words appearing in reference sentences, where the words are given credit up to maximum number of times they could appear in any one of the sentences.

Tp illustrate, if the MT turned out to be *"the the the the the the the"*, then precision would be $\frac{7}{7}$, since all 7 words of translation were present, which are copies of *the*. In Modified Precision scheme, the same would be $\frac{2}{7}$, because *the* appears at maximum twice in any of the reference sentences, from the 7 words MT generated.

BLEU on n-Grams

We can create the simplest n-gram as the *bi-gram* by selecting two tokens at a time. To calculate the modified precision on bi-grams, we compute the number of occurring valid bi-grams over the bi-grams possible from MT.
Generalizing, for n-grams:

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

Papineni (2002), "BLEU", ACL Anthology, https://www.aclweb.org/anthology/P02-1040.pdf

If MT is similar to references, then the P_n values will be 1. Else they will always be a fraction. Another metric which is often cited is a combined BLEU score defined as

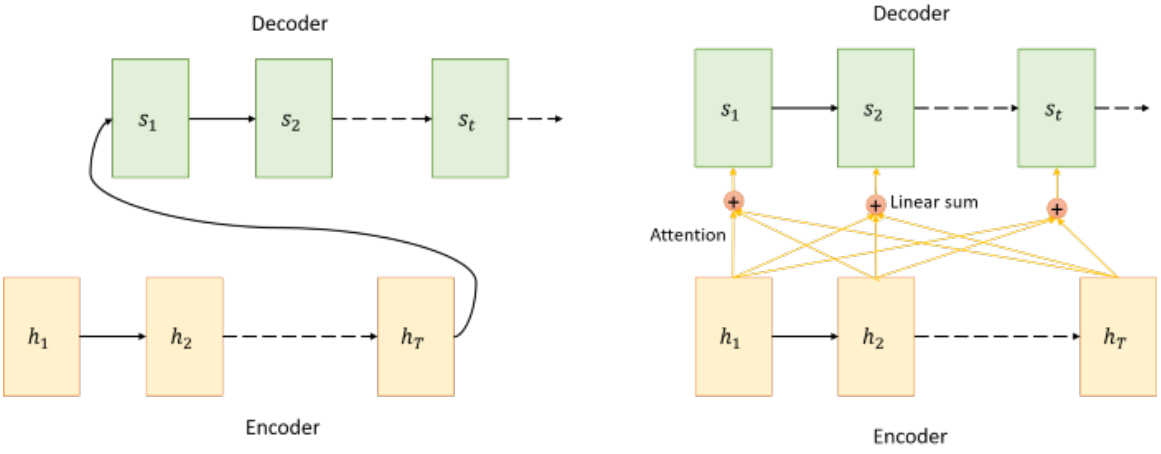
$$S = exp\left(\frac{1}{4}\sum_{n=1}^{4} P_n\right) \times BP$$

BP is *Brevity Penalty*, which penalizes the model from making short sentences to reach higher precision (If number of relevant tokens generated is smaller, it may be opportunistic for model to reach modified precision scores by deflating the denominator).

$$BP = \begin{cases} 1 & \text{if } MT > \text{REF} \\ e^{(1-\frac{REF}{MT})} & \text{if } MT < REF \end{cases}$$

Attention Models

The trouble with long sequences is that we digest the input sequences, store the activations and then decode the translation. In human translation however, this process is done word by word, and thus possibly leads to better quality of semantic output. *Attention Model* is close to the human model in this respect. It works on a part of the input at any particular time. It looks at a word and tries to learn the context by looking at the surrounding words. This is achieved by assigning attention and alignment of words in the sequence. It is proposed as a solution to the limitation of the Encoder Decoder model encoding the input sequence to one fixed length vector.
The main assumption in sequence modelling networks so far is that the current state holds information for the whole of input seen so far. Hence the final state of a RNN after reading the whole input sequence should contain complete information about that sequence. This assumption is a stretch and hence the source of imperfections. Attention mechanisms relax this assumption and propose that we should look at the hidden states corresponding to the whole input sequence in order to make any prediction.

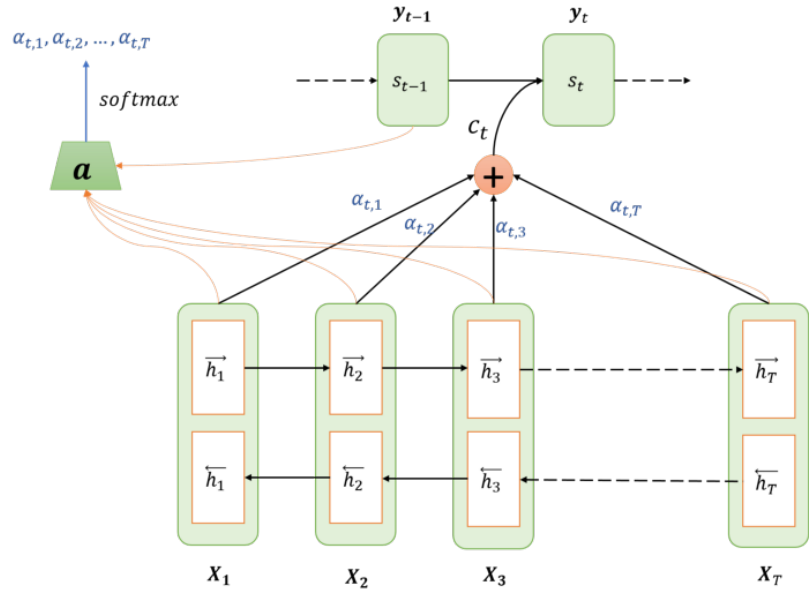


(a) Vanilla Encoder Decoder Architecture **(b) Attention Mechanism**

The following figure shows the model state where the network has computed the hidden states $\overrightarrow{h}, \overleftarrow{h}$ from each input X_i. The decoder has also run for $t-1$ steps, and about to compute the output for

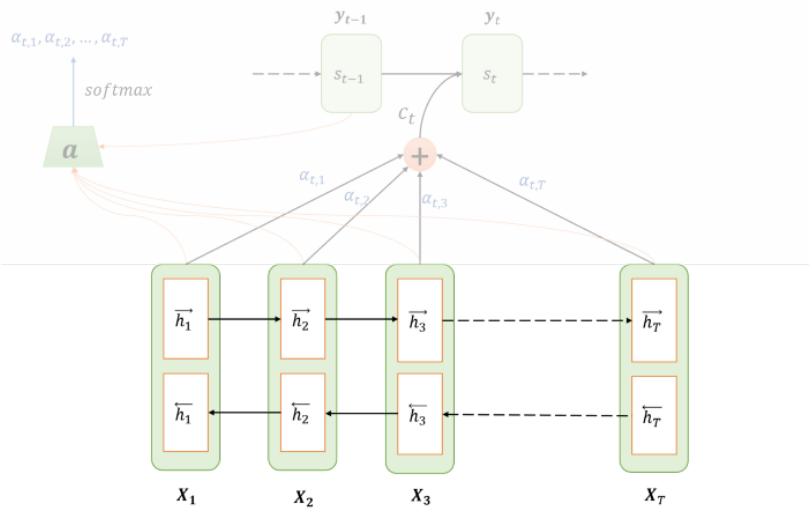
Bahdanau (2014), "Neural Machine Translation by Jointly Learning to Align and Translate, https://arxiv.org/abs/1409.0473

step t.



Encoding

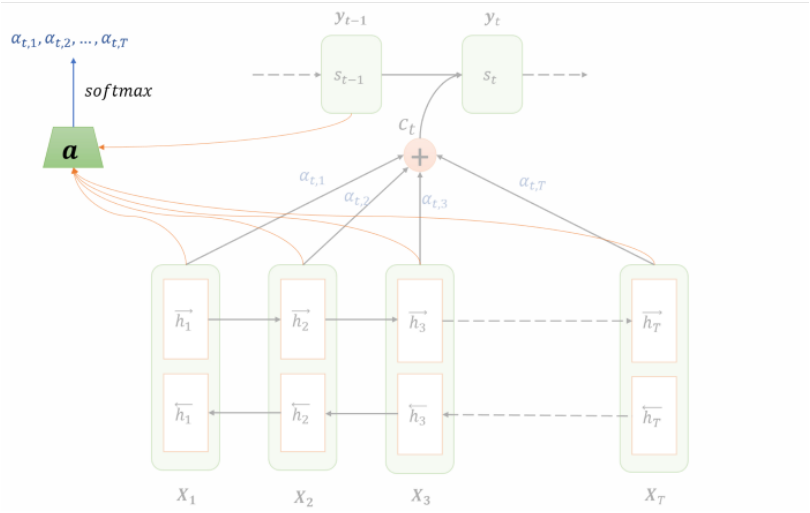
From previous discussion of bidirectional RNN, each hidden state is a concatenation of the forward and backward directional RNN. We will use a weighted linear combination of all of these h_j to make predictions at each step of the decoder. The decoder output length could be same or different from that of encoder.



Attention Computation

At each time step t of the decoder the amount of attention to be paid to the hidden encoder unit h_j is denoted by $\alpha_{t,j}$and calculated as a function of both h_j and previous hidden state of decoder s_{t-1}.

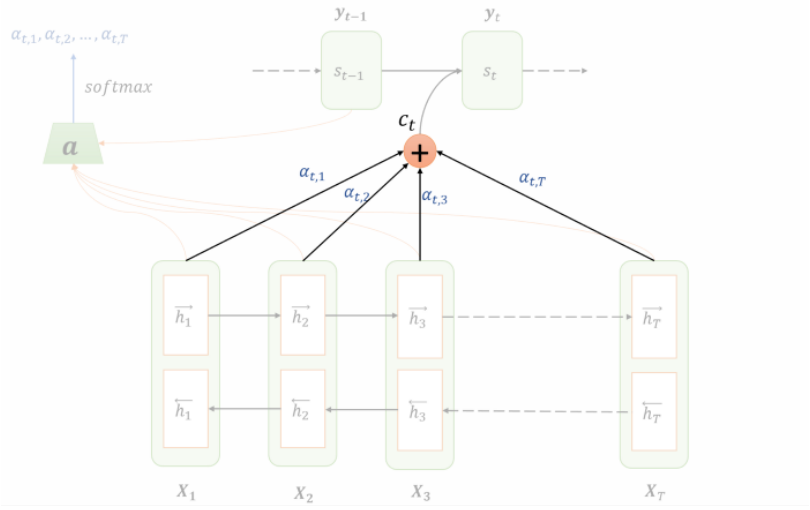
$$E_{t,j} = A(h_j, s_{t-1})$$
$$\alpha_{t,j} = \frac{e^{E_{t,j}}}{\sum_{k=1}^{T} e^{E_{t,j}}}$$



In the paper A is parameter depicted as a feed-forward neural network that runs for all j at the decoding time step t. Note the $0 \leq \alpha t, j \leq 1$ and $\sum \alpha t, j = 1$ because of the softmax on $E_{t,j}$.

Context Vector

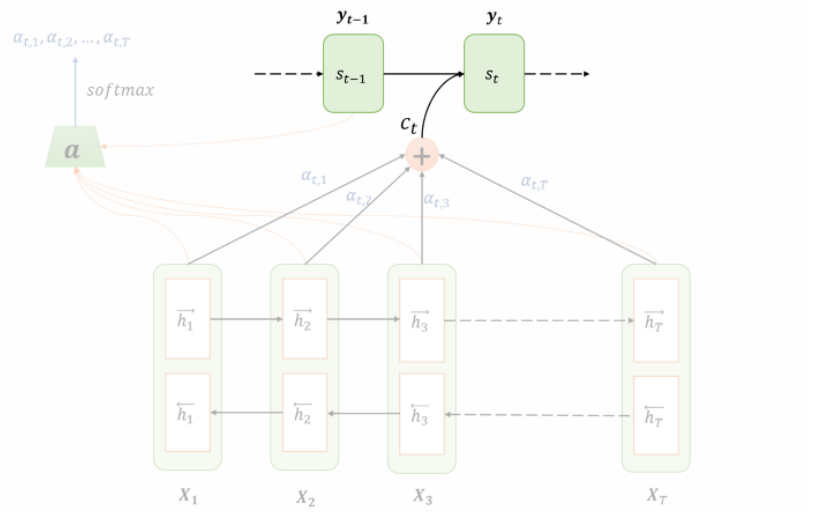
The context vector is simply a linear combination of the hidden weights h_j weighted by the attention values $\alpha_{t,j}$ that we computed,



$$c_t = \sum_{j=1}^{T} \alpha_{t,j} \cdot h_j$$

Decoding

All there remains is to use the context vector c_t along with the previous hidden state of the decoder s_{t-1} & previous output y_{t-1} and use all of them to compute the new hidden state & decoder output, $[s_t, y_t]$.



$$s_t = f(s_{t-1}, y_{t-1}, c_t)$$
$$P(y_t|y_1, y_2 \ldots, y_{t-1}, x) = g(y_{t-1}, s_t,)$$

The only downside of this method is that it is $\Omega(n^2)$ in time complexity. The quadratic time makes the process slow and reliant on fast hardware to compute efficiently.

Speech Recognition & Trigger Word Detection

Previously, speech recognition systems relied on using *phonemes*, which were considered basic units of auditory sounds. With end-to-end deep learning, hand engineering has been replaced by feeding digitized audio and its transcript.
One fundamental advance was the *Connectionist temporal classification* (CTC) which was able to reduce the noise. Usually audio units are more dense (or numerous) in comparison to their text transcript. CTC function allows collapsing repeated characters *not separated by blank* to one.

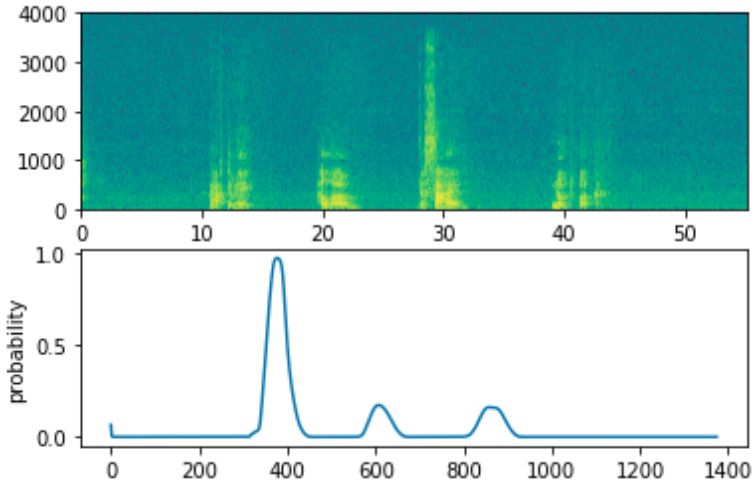
ttt_hh_ee qqq_uu_ii_cc_kk ffff_oo_xxx → the quick fox

Trigger word detection is a good application of deep learning based speech detection. We look for a particular pattern in the Mel-spectrum and estimate the probability of the phonemes.

Xu et al. (2015), "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", https://arxiv.org/abs/1502.03044

Graves (2006), "Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks", ICML

```
filename = "./raw_data/dev/1.wav"
prediction = detect_triggerword(filename)
chime_on_activate(filename, prediction, 0.5)
IPython.display.Audio("./chime_output.wav")
```



Since a positive occurrence is a small fraction of the actual spectrum, while training we keep the label hot for a few moments after the trigger word to boost the sample quality.