Sequence Models

Ji Yang

Department of Computing Science, University of Alberta

February 14, 2018

This is a note mainly based on Prof. Andrew Ng's MOOC **Sequential Models**. I also include materials (equations, figures, and tables) from other websites, blogs, and articles. It is used as my personal study & research index only. The last updated time is shown in the title section.

1 Recurrent Neural Network (RNN)

1.1 Vanilla RNN

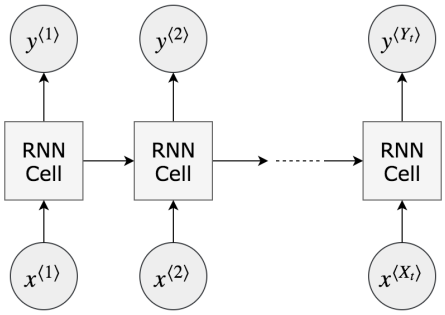

Figure 1: An unrolled RNN architecture, inspired by [1]

The problems with naive neural network include (but not limited to):

- the input size and the output size can be different in different samples
- doesn't share features learned from various location of text
- when vocabulary size is large, the weight matrix of the first layer could be enormous
- for an input $x^{\langle t \rangle}$ (t-th element in an example), it can only be benefited from the previous context but the later

1.1.1 Forward Propagation

Given an example $x = (x^{\langle 1 \rangle}, ..., x^{\langle T_x \rangle})$, the forward propagation can be briefly represented by the two equations below:

$$\begin{aligned} a^{\langle t \rangle} &= g(W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle} + b_a) \\ \hat{y}^{\langle t \rangle} &= g(W_{ya} a^{\langle t \rangle} + b_y) \end{aligned} \tag{1}$$

where function g could usually be a tanh or ReLU activation. A simplified version of Eq. 1 is provided as well:

$$\begin{aligned} a^{\langle t \rangle} &= g(W_a \left[a^{\langle t-1 \rangle}, x^{\langle t \rangle} \right] + b_a) \\ \hat{y}^{\langle t \rangle} &= g(W_y a^{\langle t \rangle} + b_y) \end{aligned} \tag{2}$$

where $W_a = \begin{bmatrix} W_{aa} & W_{ax} \end{bmatrix}$ and $\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle} \right] = \begin{bmatrix} a^{\langle t-1 \rangle} \\ x^{\langle t \rangle} \end{bmatrix}$. The dimension of parameters in the equations above is shown in Table 1.

	$x^{\langle t \rangle}$	$a^{\langle t \rangle}$	$a^{\langle t-1 \rangle}$	W_{ax}	W_{aa}	W_{ya}	b_a	b_y	$y^{\langle t \rangle}$
Dim.	(n_x, m)	(n_a, m)	(n_a, m)	(n_a, n_x)	(n_a, n_a)	(n_y, n_a)	$(n_a, 1)$	$(n_y, 1)$	(n_y, m)

Table 1: The dimension of used parameters in a vanilla RNN forward pass. We have a dimension size of m because we are doing vectorization calculation for m inputs simultaneously.

1.1.2 Backpropagation Through Time (BPTT)

Given a cross-entropy loss function for a multi-label classification task:

$$\begin{aligned} L^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}) &= -y^{\langle t \rangle} \log \hat{y}^{\langle t \rangle} - (1 - y^{\langle t \rangle}) \log(1 - \hat{y}^{\langle t \rangle}) \\ L(\hat{y}, y) &= \sum_{t=1}^{T_y} J^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}) \end{aligned} \tag{3}$$

We then are able to do backpropagation based on L, start from the last input $x^{\langle T_x \rangle}$. In this section, for the sake of illustrating the idea of BPTT a slightly simplified version of derivation is given.

We know the forward pass in the vanilla RNN is defined as

$$a^{\langle t \rangle} = \tanh(W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle} + b_a) \tag{4a}$$

$$\hat{y}^{\langle t \rangle} = g(W_{ya} a^{\langle t \rangle} + b_y) \tag{4b}$$

Now, we first consider the gradients inside an RNN cell. Compute the gradient of the hidden state at time t, $a^{\langle t \rangle}$ with respect to weights, W_{aa} and W_{ax}, and the bias, b_a.

$$\frac{\partial a^{\langle t \rangle}}{\partial W_{aa}} = (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b_a)) \cdot a^{\langle t-1 \rangle T} \tag{5a}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial W_{ax}} = (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b_a)) \cdot x^{\langle t \rangle T} \tag{5b}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial b_a} = \sum_{\text{batch}} (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b_a)) \tag{5c}$$

and also the gradient of $a^{\langle t \rangle}$ with respect to the input at time t, $x^{\langle t \rangle}$, and the previous hidden state, $a^{\langle t-1 \rangle}$.

$$\frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}} = W_{ax}^T \cdot (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b_a)) \tag{6a}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}} = W_{aa}^T \cdot (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b_a)) \tag{6b}$$

Note that "$\cdot$" means dot product.

It's time to derive the complete backpropagation, start from the derivation of $\frac{\partial L}{\partial W_{aa}}$. The following derivation is based on [2] with modifications and extra explanations. As $L^{\langle t \rangle}$ is contributed by $\hat{y}^{\langle t \rangle}$, $\hat{y}^{\langle t \rangle}$ is contributed by $a^{\langle t \rangle}$ and $a^{\langle t \rangle}$ is contributed by W_{aa}, by using chain rule, we now derive the gradient of the loss at time t with respect to W_{aa} as

$$\frac{\partial L^{\langle t \rangle}}{\partial W_{aa}} = \frac{\partial L^{\langle t \rangle}}{\partial \hat{y}^{\langle t \rangle}} \frac{\partial \hat{y}^{\langle t \rangle}}{\partial a^{\langle t \rangle}} \frac{\partial a^{\langle t \rangle}}{\partial W_{aa}} \tag{7}$$

We know from Eq. 4a that $a^{\langle t \rangle}$ partially depends on $a^{\langle t-1 \rangle}$, also think about the recursive definition in Eq. 4a, we now have an updated Eq. 7.

$$\frac{\partial L^{\langle t \rangle}}{\partial W_{aa}} = \frac{\partial L^{\langle t \rangle}}{\partial \hat{y}^{\langle t \rangle}} \frac{\partial \hat{y}^{\langle t \rangle}}{\partial a^{\langle t \rangle}} \frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}} \frac{\partial a^{\langle t-1 \rangle}}{\partial W_{aa}} \tag{8}$$

As mentioned during the course exercise, if we are given $\frac{\partial L^{\langle t \rangle}}{\partial a^{\langle t \rangle}}$, we can then shorten Eq. 8 to

$$\frac{\partial L^{\langle t \rangle}}{\partial W_{aa}} = \frac{\partial L^{\langle t \rangle}}{\partial a^{\langle t \rangle}} \frac{\partial a^{\langle t \rangle}}{\partial W_{aa}} \tag{9}$$

and then compute the result $\frac{\partial L^{\langle t \rangle}}{\partial W_{aa}}$ with Eq. 5a. The exact same idea works for $\frac{\partial L^{\langle t \rangle}}{\partial W_{ax}}$ and $\frac{\partial L^{\langle t \rangle}}{\partial b_a}$. The derivation of these two gradients leaves as exercises.

1.1.3 RNN Architectures

Think that the input length and the output length could be different. Suppose $x = (x^{\langle 1 \rangle}, ..., x^{\langle T_x \rangle})$, and $y = (y^{\langle 1 \rangle}, ..., y^{\langle T_y \rangle})$. Given that $T_x = T_y$, we then have our input and output with the same size. This is called a *Many-to-Many* architecture. Now, suppose we want to do a binary semantic classification, the output T_y is then equal to 1 because we need to have an output that represents the probability of the input example being positive. This is called a *Many-to-One* architecture. There is also an *One-to-Many* architecture, it can be used for tasks like music generation, text generation and so on. However, when we talk about *Many-to-Many* architectures, we also want to consider the case where $T_x \neq T_y$, which is very common in NLP like machine translation.

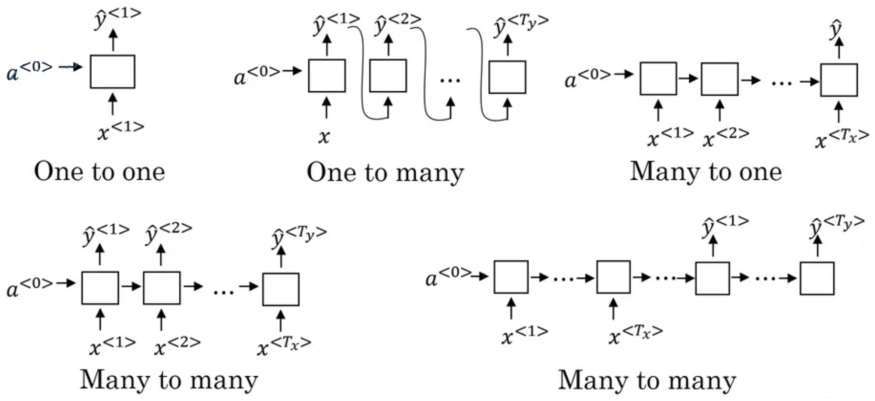

Figure 2: RNN architectures, adopted from Andrew Ng's Deep Learning course

1.1.4 Case Study: Language Model

To build a language model, you need a large training set, for example, a large corpus of English text. Then you tokenize the words, punctuations, and any other content in the corpus at the character, word or sentence level (depends on use case and performance). chen2016gentle

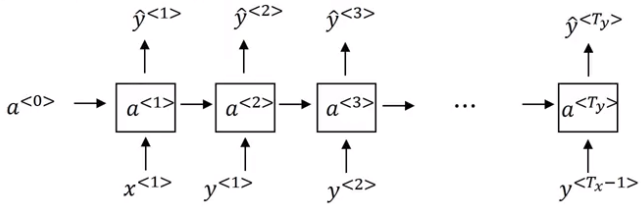

Figure 3: How to train a RNN language model, adopted from Andrew Ng's Deep Learning course

We can refer to the bottom-left architecture in Figure 2 to understand that how to build a language model. Just use the notation in the figure, $a^{\langle 0 \rangle}$ and $x^{\langle 0 \rangle}$ would be zero vectors, and other $x^{\langle t \rangle}$ would be $y^{\langle t-1 \rangle}$. The output $\hat{y}^{\langle t \rangle}$ is actually a softmax output, that is a probability distribution that represents the probability of $y^{\langle t \rangle}$ being some words at time step t. Note that $p(y^{\langle 1 \rangle}, ..., y^{\langle t \rangle}) = p(y^{\langle 1 \rangle})p(y^{\langle 2 \rangle}|y^{\langle 1 \rangle})...p(y^{\langle t \rangle}|y^{\langle 1 \rangle}, ..., y^{\langle t-1 \rangle})$

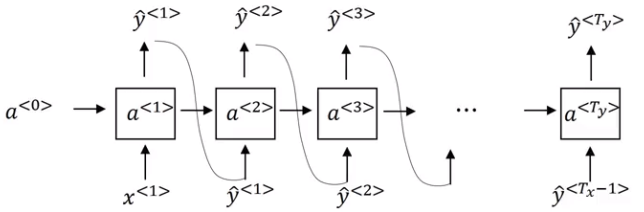

Figure 4: Sampling from a trained RNN, adopted from Andrew Ng's Deep Learning course

1.1.5 Vanishing Gradients with RNNs

Consider we have two sentences: "The cat, which already ate ...(many food names), was full" and "The cats, which already ate ...(many food names), were full". The only difference is whether the word "cat" is singular or plural. As in English, the "many food names" part in both sentences can be arbitrarily long, the network has to be able to remember something way earlier in the sentence to inference whether we need a "was" or "were" in the later part. This is an essential problem with the vanilla RNN architectures we've seen so far: the gradient will vanish in very deep recurrent architecture. Therefore, for example, a word $y^{\langle 3 \rangle}$ can only be influenced effectively by nearby words $y^{\langle 1 \rangle}$ and $y^{\langle 2 \rangle}$ through backpropagation. The same thing happens in the forward pass, vanilla RNNs can hardly remember long-term information and could perform poorly in the example we mentioned above. One may recall we also encounter gradient exploding problem, but it can be solved with gradient clipping so it's not a very big problem.

1.2 Gated Recurrent Unit (GRU)

We start from a simplified version GRN. Let $c^{\langle t \rangle}$ represent t-th memory cell, and Γ_u represent an update gate (see the later explanation). Again, we use the sentence "The cat, which already ate ...(many food names), was full" as our example, then $c^{\langle 2 \rangle} = 1$ (say, 1 represent "is singular"), then $c^{\langle 2 \rangle}$ will be somehow remembered until the position of word "was". The job of the gate is to decide when to apply this remembered $c^{\langle 2 \rangle}$. Our desired result is that at only at the positions of word "cat" and "was", $\Gamma_u = 1$, otherwise $\Gamma_u = 0$. The mathematical formulation of a GRU is

$$
\begin{aligned}
c^{\langle t-1 \rangle} &= a^{\langle t-1 \rangle} \\
\tilde{c}^{\langle t \rangle} &= \tanh(W_c \left[c^{\langle t-1 \rangle}, x^{\langle t \rangle} \right] + b_c) \\
\Gamma_u &= \sigma(W_u \left[c^{\langle t-1 \rangle}, x^{\langle t \rangle} \right] + b_u) \\
c^{\langle t \rangle} &= \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle} \\
a^{\langle t \rangle} &= c^{\langle t \rangle}
\end{aligned}
$$

where $*$ means element-wise multiplication.

One may observe that the formula of calculating Γ_u is very similar to a logistic regression, where I consider these two things share similar ideas. And this is also why the Γ_u is activated by a sigmoid function (result in a value between 0 and 1). Now, let's look at the equations for Full GRU.

$$
\begin{aligned}
c^{\langle t-1 \rangle} &= a^{\langle t-1 \rangle} \\
\tilde{c}^{\langle t \rangle} &= \tanh(W_c \left[\Gamma_r * c^{\langle t-1 \rangle}, x^{\langle t \rangle} \right] + b_c) \\
\Gamma_u &= \sigma(W_u \left[c^{\langle t-1 \rangle}, x^{\langle t \rangle} \right] + b_u) \\
\Gamma_r &= \sigma(W_r \left[c^{\langle t-1 \rangle}, x^{\langle t \rangle} \right] + b_r) \\
c^{\langle t \rangle} &= \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle} \\
a^{\langle t \rangle} &= c^{\langle t \rangle}
\end{aligned}
$$

1.3 Long Short Term Memory (LSTM)

Another important recurrent architecture in the literature is LSTM. We can see it as a modified version of GRU though LSTM comes way earlier than GRU. There are 3 gates in LSTM: update gate, forget gate and output gate, which are represented by Γ_u, Γ_f, Γ_o in the formula below. Also, note that we now heavily use $a^{\langle t-1 \rangle}$ in the calculation rather than $c^{\langle t-1 \rangle}$ as we did in GRU.

Parameters	Dimension
W_f, W_u, W_c, W_o	$(n_a, n_a + n_x)$
W_y	(n_y, n_a)
b_f, b_u, b_c, b_o	$(n_a, 1)$
b_y	$(n_y, 1)$
$a^{\langle t \rangle}, a^{\langle t-1 \rangle}, c^{\langle t \rangle}, c^{\langle t-1 \rangle}$	(n_a, m)
$\Gamma_f, \Gamma_u, \Gamma_o$	(n_a, m)
$\tilde{c}^{\langle t \rangle}$	(n_a, m)

Table 2: Dimensions of parameters used in an LSTM forward pass. We have a dimension size of m because we are doing vectorization calculation for m inputs simultaneously.

1.3.1 Forward Pass in LSTM

Note that the flow constructed by $c^{\langle t \rangle}$s in each cell provides LSTM the ability to remember long-term information. One can refer to Figure 5 and Table 2 to get a better understanding.

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

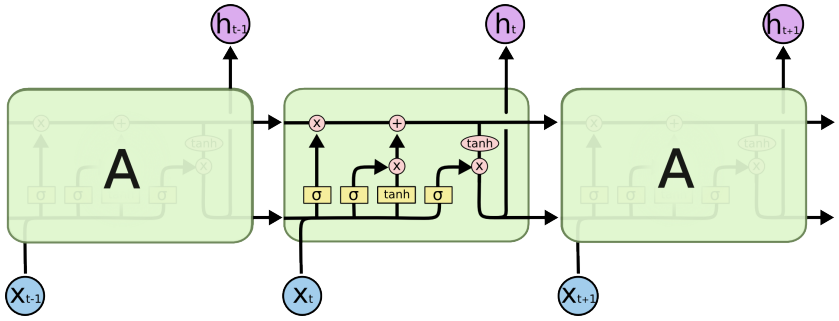

Figure 5: An LSTM cell visualization, adopted from [1]

1.3.2 Backpropagation in LSTM

Please kindly refer to https://wiseodd.github.io/techblog/2016/08/12/lstm-backprop/.

1.3.3 About the Gates[1]

- **Forget Gate**: For the sake of this illustration, lets assume we are reading words in a piece of text, and want use an LSTM to keep track of grammatical structures, such as whether the subject is singular or plural. If the subject changes from a singular word to a plural word, we need to find a way to get rid of our previously stored memory value of the singular/plural state. In an LSTM, the forget gate lets us do this:

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

  Here, W_f are weights that govern the forget gate's behavior. We concatenate $[a^{\langle t-1 \rangle}, x^{\langle t \rangle}]$ and multiply by W_f. The equation above results in a vector $\Gamma_f^{\langle t \rangle}$ with values between 0

[1]Directly copied & pasted from the course assignment instruction with a few modifications

and 1. This forget gate vector will be multiplied element-wise by the previous cell state $c^{\langle t-1 \rangle}$. So if one of the values of $\Gamma_f^{\langle t \rangle}$ is 0 (or close to 0) then it means that the LSTM should remove that piece of information (e.g. the singular subject) in the corresponding component of $c^{\langle t-1 \rangle}$. If one of the values is 1, then it will keep the information.

- **Update Gate**: Once we forget that the subject being discussed is singular, we need to find a way to update it to reflect that the new subject is now plural. Here is the formulat for the update gate:

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

Similar to the forget gate, here $\Gamma_u^{\langle t \rangle}$ is again a vector of values between 0 and 1. This will be multiplied element-wise with $\tilde{c}^{\langle t \rangle}$, in order to compute $c^{\langle t \rangle}$.

- **Updating the Cell**: To update the new subject we need to create a new vector of numbers that we can add to our previous cell state. The equation we use is:

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

Finally, the new cell state is:

$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} * c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} * \tilde{c}^{\langle t \rangle}$$

- **Output Gate**: To decide which outputs we will use, we will use the following two formulas:

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

$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} * \tanh(c^{\langle t \rangle})$$

Where in the first equation you decide what to output using a sigmoid function and in the second you multiply that by the tanh of the previous state.

1.4 Other RNNs

Bidirectional RNN is another type of RNN that is very popular due to one of its intuitive natures.

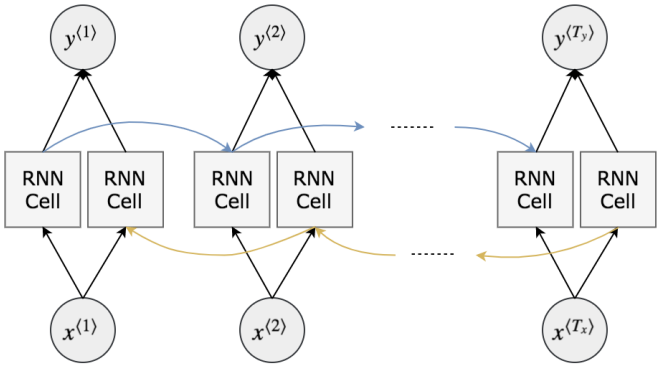

Figure 6: A bidirectional RNN architecture

When walking through a sentence, we may want to get information from future as we can get previous context. The key idea is that we calculate activations start from both sides, then for each $x^{\langle t \rangle}$ we will have two activations $\overrightarrow{a}^{\langle t \rangle}$ and $\overleftarrow{a}^{\langle t \rangle}$. Our new version of $\hat{y}^{\langle t \rangle}$ where

$$\hat{y}^{\langle t \rangle} = g(W_y \left[\overrightarrow{a}^{\langle t \rangle}, \overleftarrow{a}^{\langle t \rangle}\right] + b_y) \tag{10}$$

In addition, we can also build a deep RNN by stacking more RNN cells, that is, having more layers just like we did in deep neural networks. The way we stacking them is very simple, consider use a grid like structure to understand this, refer to Figure 7.

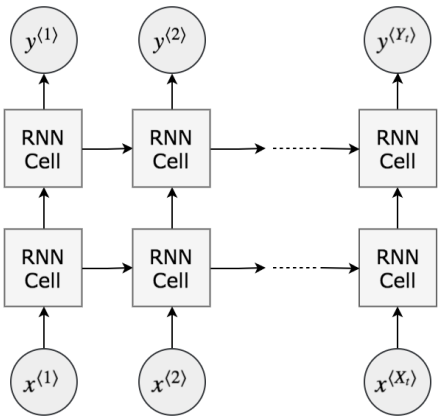

Figure 7: A deep RNN architecture

2 Natural Language Processing and Word Embeddings

2.1 Word Embeddings

A model cannot directly understand the word, or the text. We'd like to build word representations based on some techniques. One of the most intuitive way to do that is use one-hot vectors to represent words.

Suppose we have two sentence "I want a glass of orange juice" and "I want a glass of apple juice". Then we can build one-hot vectors with respect to our entire vocabulary. For the sake of saving spaces, the word vectors are written in a row fashion. We could now have the word "I" be represented by a vector $(1, 0, 0, 0, 0, 0, 0, 0)^T$, the word "am" be represented by a vector $(0, 1, 0, 0, 0, 0, 0, 0)^T$, and so on so forth.

However, there is a "harmful" weakness with representing words with one-hot vectors: each word is treated (or say, is represented) independently to other words. For example, there is no relation between the one-hot vector of "apple" and "orange". Imagine you already have a trained language model that can easily predict the next word of "I want a glass of orange" is "juice". The relationship between "apple" and "orange" is not closer than "orange" with any other words, like "king". So its ability of generalization will be poor. Also, since the dot product of two one-hot vectors will be zero, you cannot apply any distance-like metrics to evaluate the similarity.

Instead, a feature-like representation would be appreciate. The most successful recent break change in learning featurized representation is what so-called "Word2Vec", introduced in [3, 4]. A good way to visualize the word embeddings and see their similarity relationships is to plot them, for simplicity, t-SNE algorithm can be used here for dimensionality reduction. After the magic work by this algorithm, suppose we have all the vectors in 2D space. We would likely to have a plot looks like Figure 8. As you may observe, "man" and "woman" are closer than "man" and "cat" are.

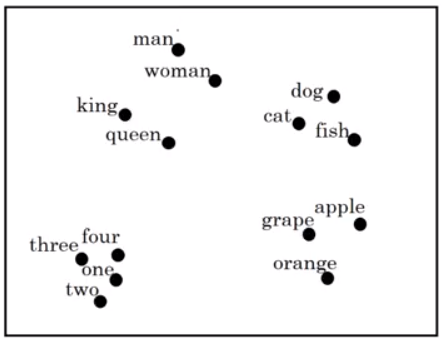

Figure 8: A featurized word vector representation after dimensionality reduction

The word embeddings also bring us the benefits of using transfer learning. You can either (1) train word embeddings on a large corpus (usually 1-100B words), (2) download pre-trained em-

beddings (such as Word2Vec, GloVe), or (3) fine-tune pre-trained embeddings based on specific corpus. Using such featurized embeddings can reduce the dimensionality of the word vectors (from sparse vectors to dense vectors).

2.1.1 Properties of Word Embeddings

- Analogy: with well-trained word embeddings, we can obtain vectors look like in Table 3. It's easy to observe that $vec(\text{man}) - vec(\text{woman}) \approx vec(\text{king}) - vec(\text{woman})$. This kind of relation can also be understood as Eq. 11. Find a word w that

$$\arg\max_{w} \ \text{sim}(vec_w, vec(\text{king}) - vec(\text{man}) + vec(\text{woman})) \tag{11}$$

  Note that Eq. 11 is only valid when you play with the original word embeddings rather than embeddings after performing PCA or t-SNE.

- Cosine similarity: we can also borrow an equation used for calculating the angle between two vectors from linear algebra.

$$\text{sim}(u, v) = \frac{u^T v}{\|u\|_2 \|v\|_2} = cos(\theta) \tag{12}$$

  We know that if θ is close to 0 then the two word vectors are similar, and $\cos(\theta) \approx 1$

- Eculidean similarity: this is more a measurement of unsimilarity, we can add a negative sign before the RHS to obtain a similarity measurement.

$$\text{sim}(u, v) = \|u - v\|_2 \tag{13}$$

	Man	Woman	King	Queen
Royal	0.01	0.01	0.98	0.99
Gender	0.97	-0.99	0.97	-0.98
Food	0.02	0.01	0.02	0.02
Temperature	0.01	0.02	0.01	0.01

Table 3: Why word embeddings work with analogy tasks (assume 4D word vectors)

2.2 Learning Word Embeddings

There are two basic types of the Word2Vec algorithm: one called skip-gram and another one is continuous bag-of-word (CBOW) model, as shown in Figure 9. I've done a very detailed investigation in one of my another literature review, only a brief description will be here.

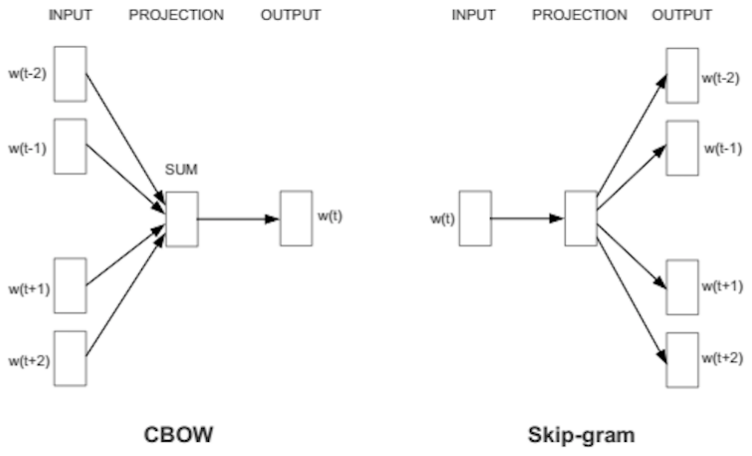

Figure 9: Two word2vec architectures, adopted from the deeplearning4j documentation [5]

We can see that the only difference between these two architectures is that in CBOW, the model predicts a target word by given context whereas in skip-gram case, the model predicts context words by a give target. These two architecture both provides decent word embeddings but may have significantly different performance on some specific tasks. Intuitively, since the skip-gram model is trained to "understand" the surrounding words, or say, the context, it is somehow better at understanding a sentence or guessing the context.

2.2.1 Learning Word Embeddings with Word2Vec

Let's use the CBOW model for the sake of easier illustration. The entire process of the model is:

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

where O_c is the one-hot vector, E is the lookup table, e_c is the featurized word vector we found in E. The output will be a softmax probability, of the target word:

$$p(t|c) = \frac{e^{\theta_e^T e_c}}{\sum_{j=1}^{V} e^{\theta_j^T e_c}} \tag{14}$$

where θ_e is word c's corresponding column in the lookup table matrix, V is the size of the vocabulary. However, the basic form of this softmax output has a weakness that can make the model be trained very slow in large corpus. At the denominator part, we need to sum up tons of times! The loss function is then:

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

A pretty straightforward way to avoid massive amount of summation at the denominator of the softmax function is to use hierarchical softmax. It is a little bit similar to binary search. Intuitively, suppose we have a vocabulary size of 1000, by using hierarchical softmax, we first use a binary classifier to tell whether the target word is in the first 500 words or not. If it's in the first 500, then we use the second classifier to tell us whether it's in the first 250 or the second 250, so on so forth. This method works with $O(\log V)$

2.2.2 Negtive Sampling

In a large corpus, we can easily have words like "to", "is" or "and" occurs hundreds of thousands times, and these words means almost nothing without the context. In fact, we need the words like "man", "woman", "orange", "apple" or "cat" to be learned well, but compared with the frequent words mentioned earlier, their count of occurrence will be relatively small.

By defining a new learning algorithm called negative sampling, we can solve this problem easily. In short, we create k negative samples from the corpus, as shown in Table 4.

Context	Target	Ture Target?
orange	juice	1
orange	man	0
orange	of	0
orange	book	0

Table 4: A simple example of creating k (= 3) negative samples by random selecting words from the vocabulary, where "orange juice" is indeed in the corpus but the rest combinations are not.

The learning objective is then, given a pair of words (context + target), predict whether they are a true combination. A good starting point of choosing k is that, for smaller dataset, use $k = 5 \sim 20$, for larger corpuse, use $k = 2 \sim 5$ [3]. Then, we can now understand why this method can reduce the computation significantly. Previously, we need calculate tons of probability and sum them up (check Eq. 14), now instead, only 1 + k logistic probabilities need to be calculated by Eq. 15.

$$\hat{y} = P(y = 1|c, t) = \sigma(\theta_t^T e_c) \tag{15}$$

Moreover, if the vocabulary size is 10000, then we have 10000 logistic regression units in the output layer, but only $k + 1$ of them need to be update every time.

In [3], the authors also give a heuristic way of sampling negative samples rather than just uniformly selecting negative samples. The probability of sample a word w_i as the negative sample is

$$P(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_{j=1}^{V} f(w_j)^{\frac{3}{4}}} \tag{16}$$

2.2.3 GloVe

The training process of GloVe is a little bit different. Let X_{ij} be # of times word i appears in context of word j. Refer to our previous notation, $X_{ij} = X_{ct}$ where c means the context word and t means the target word. The learning objective of the GloVe model is then

$$\text{minimize} \sum_{i}^{V} \sum_{j}^{V} f(X_{ij})(\theta_i^T e_j + b_i - b'_j - \log X_{ij})^2 \tag{17}$$

where $f(X_{ij})$ is a weighting term that guarantees we won't have $\log 0$ when X_{ij} is zero ($f(X_ij) = 0$ if $X_{ij} = 0$) . The term, $\theta_i^T e_j - \log X_{ij}$, in Eq. 17 is more like asking how related are words i and j as measured by how often they occur with each other.

2.2.4 Debiasing Word Embeddings

A main problem with the pre-trained word embeddings is that, due to the fact that training word embeddings is supervised, those embeddings can reflect gender, age, sexual orientation or other biases in the training set. For example, we can say "man" is close to "king", "woman" is close to "queen". But if "man" is close to "computer programmer", "woman" is close to "homekeeper", that might not be a good thing to have in our trained embeddings. There are 3 steps to debias word embeddings [6]:

1. identify bias directions
2. neutralize: for every word that is not definitional, project to non-bias direction to get rid of bias
3. equalize pairs

Note this section is still incomplete, equations from the course exercise should be added.

2.3 Sentiment Analysis

Assume we are building a model that can help to classify whether a comment of a YouTube video is positive or negative. A simple way to build the model is that we can use a pre-trained word embeddings to get embeddings for every word in the sentence and we use a softmax layer to classify the average/sum of all the embeddings. However, this method cannot handle the case when the order of the words play an important role to help understand the meaning. Given a sentence "Completely lacking of good content, good video quality, good description." The sentence could be classified as positive since the multiple occurrences of word "good", which is obviously wrong.

Instead, we should take advantage of the sequence models. The "Many-to-One" model is exactly the one we want to use for sentiment classification, as shown in Figure 2. You can end up with a decent classification model with using the "Many-to-One" model.

3 Sequence-to-Sequence Models

Sequence-to-sequence models (seq2seq) are significantly helpful on tasks that both the input and the output are sequential, such as machine translation and speech recognition. As shown in Figure 10, the idea behind it is that we use T_x RNN cells to encode the input sequence and get an encoded embedding vector. Later, a decoder, which consists of T_y RNN cells, is used to decode the encoded embedding. Note that T_x and T_y are not necessarily the same with this encoder-decoder architecture.

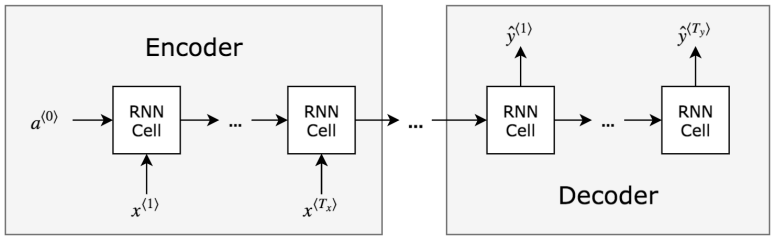

Figure 10: A sequence-to-sequence architecture, which consists of an encode r followed by a decoder, where T_x is not necessarily equal to T_y

Using the seq2seq architecture to build a model for machine translation is actually building a conditional language model. A language model can tell what is the probability of an input sentence to occur, $P(x)$, whereas a conditional language model, like the one in Figure 10, tells the probability of an output sentence to occur given an input sentence, $P(\hat{y}^{\langle 1 \rangle}, ..., \hat{y}^{\langle T_y \rangle} | x)$. The objective of a conditional language model is then:

$$\underset{\hat{y}^{\langle 1 \rangle}, ..., \hat{y}^{\langle T_y \rangle}}{\arg\max} \; P(\hat{y}^{\langle 1 \rangle}, ..., \hat{y}^{\langle T_y \rangle} | x) \tag{18}$$

To save some paper space, let's assume $\hat{y}^{\langle t \rangle}$ is actually the sampled word rather than the probability distribution at time step t in Eq 18.

3.1 Beam Search

Given input sentence x and generated part in the target sentence $y^{\langle 1 \rangle}, ..., y^{\langle t \rangle}$, use a simple greedy strategy to sample the next word $y^{\langle t+1 \rangle}$ is usually only suboptimal. That is,

$$\underset{y^{\langle t+1 \rangle}}{\arg\max} \, P(y^{\langle t+1 \rangle} | x, y^{\langle 1 \rangle}, ..., y^{\langle t \rangle}) \tag{19}$$

is suboptimal. Consider the case that the $(t+1)$-th word could be "beat" or "hit" where "hit" will gives a better translation but the probability, $P(y^{\langle t+1 \rangle} = \text{"hit"} | x, y^{\langle 1 \rangle}, ..., y^{\langle t \rangle})$ is slightly smaller than $P(y^{\langle t+1 \rangle} = \text{"beat"} | x, y^{\langle 1 \rangle}, ..., y^{\langle t \rangle})$. A greedy strategy will just throw the word "hit" away and use the word "beat" as a sample result. Then the rest of the output sentence will be influenced by this suboptimal decision.

Beam search can solve this issue, and its idea is easy to illustrate and understand. The simplest beam search only needs 1 parameter, namely, B, the beam width. Let $B = 3$, and assume our vocabulary size V is 1000. We will first sample $y^{\langle 1 \rangle}$ from $\hat{y}^{\langle 1 \rangle}$. With $B = 3$, we (a) select the top-3 samples, and (b) build the next cell of the "decoder" component based on these 3 samples to obtain 3×1000 possible $y^{\langle 1 \rangle}, y^{\langle 2 \rangle}$ combinations. We then repeat the two steps above.

There are latent issues behind the beam search algorithm. The target of the conditional language model is to find y, that

$$\underset{y}{\arg\max} \prod_{t=1}^{T_y} P(y^{\langle t \rangle} | x, y^{\langle 1 \rangle}, ..., y^{\langle t-1 \rangle}) \tag{20}$$

However, this target equation has two potential problems. We first talk about the underflow problem and give a simple solution. In Eq. 20, $\prod_{t=1}^{T_y} P(y^{\langle t \rangle} | x, y^{\langle 1 \rangle}, ..., y^{\langle t-1 \rangle})$ is actually a product of several probabilities that are smaller than 1 (and most likely close to 0). Such a product could easily obtain a extremely small result and even underflow. By adding a log term to it, we obtained the following new form of the target,

$$\underset{y}{\arg\max} \sum_{t=1}^{T_y} \log P(y^{\langle t \rangle} | x, y^{\langle 1 \rangle}, ..., y^{\langle t-1 \rangle}) \tag{21}$$

In Eq. 21, we solved the problem of underflow with the simple logarithm trick, but another issue with both Eq. 20 and Eq. 21 is that both equations will prefer a shorter output, in general. Because the more terms you have in the output, the less the product term in Eq. 20, or the

more negative the sum term in Eq. 21 is. By normalizing the result by the length of the output (this is a heuristic!), we now have

$$\arg\max_{y} \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{\langle t \rangle} | x, y^{\langle 1 \rangle}, ..., y^{\langle t-1 \rangle}) \tag{22}$$

where α is usually to be around 0.7 (set as a heuristic).

Here is an extra note on the beam width, B. The setting of B basically depends on your need. 3 is a small value for B, production system could use values like 10 or 20, some researchers just use 3000 or even 5000 to get the best result though large value could meet diminishing return problem and significantly more computation time. So choose the beam width wisely, based on the application case. Overall, in beam search, if you increase the beam width B, beam search will run more slowly, use up more memory, and generally find better solutions.

The speech recognition system (RNN + beam search) could be problematic. Given an input audio clip, your algorithm outputs the transcript $\hat{y}$ = "I'm building an A Eye system in Silly con Valley.", whereas a human gives a much superior transcript y^* = "I'm building an AI system in Silicon Valley." Now, according to your model, $P(\hat{y} \mid x) = 1.09 * 10^-7$, $P(y^* \mid x) = 7.21 * 10^-8$, because $P(y^* \mid x) \leq P(\hat{y} \mid x)$ indicates the error should be attributed to the RNN rather than to the search algorithm.

3.2 Attention Mechanism

When a human translates a very long sentence, say French to English, generally he/she will split the sentence to small segments and translate them one by one. However, a RNN model cannot do so (at least with our current knowledge) because the model will try to remember the entire input. Attention mechanism, is designed to handle this problem.

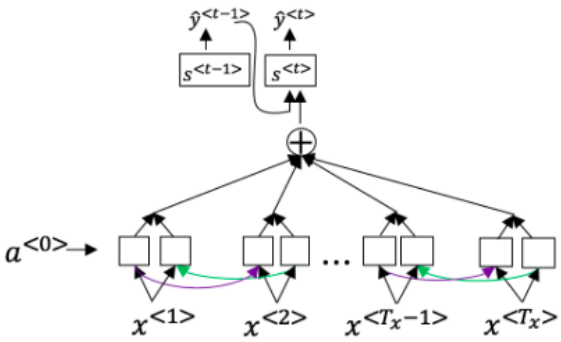

Figure 11: A basic illustration of attention model.

The basic attention model architecture is shown in Figure 11. The attention value, for each attention cell, $\alpha^{\langle t,t' \rangle}$, is calculated by

$$\alpha^{\langle t,t' \rangle} = \frac{\exp(e^{\langle t,t' \rangle})}{\sum_{t'=1}^{T_x} \exp(e^{\langle t,t' \rangle})} \tag{23}$$

We expect $\alpha^{\langle t,t' \rangle}$ to be generally larger for values of $a^{\langle t' \rangle}$ that are highly relevant to the value the network should output for $y^{\langle t \rangle}$. And $\sum_{t'} \alpha^{\langle t,t' \rangle} = 1$. The network learns where to "pay attention" by learning the values $e^{\langle t,t' \rangle}$, which are computed using a small neural network: We can't replace $s^{\langle t-1 \rangle}$ with $s^{\langle t \rangle}$ as an input to this neural network. This is because $s^{\langle t \rangle}$ depends on $\alpha^{\langle t,t' \rangle}$ which in turn depends on $e^{\langle t,t' \rangle}$; so at the time we need to evalute this network, we haven't computed $s^{\langle t \rangle}$ yet.

Note that in Figure 12 we can see that[1]:

- There are two separate LSTMs in this model. Because the one at the bottom of the picture is a Bi-directional LSTM and comes *before* the attention mechanism, we will call it *pre-attention* Bi-LSTM. The LSTM at the top of the diagram comes *after* the attention

[1] directly copied from the assignment notebook

mechanism, so we will call it the *post-attention* LSTM. The pre-attention Bi-LSTM goes through T_x time steps; the post-attention LSTM goes through T_y time steps.

- The post-attention LSTM passes $s^{\langle t \rangle}, c^{\langle t \rangle}$ from one time step to the next. In the lecture videos, we were using only a basic RNN for the post-activation sequence model, so the state captured by the RNN output activations $s^{\langle t \rangle}$. But since we are using an LSTM here, the LSTM has both the output activation $s^{\langle t \rangle}$ and the hidden cell state $c^{\langle t \rangle}$. However, unlike previous text generation examples, in this model the post-activation LSTM at time t does will not take the specific generated $y^{\langle t-1 \rangle}$ as input; it only takes $s^{\langle t \rangle}$ and $c^{\langle t \rangle}$ as input. We have designed the model this way, because (unlike language generation where adjacent characters are highly correlated) there isn't as strong a dependency between the previous character and the next character in a YYYY-MM-DD date.
- We use $a^{\langle t \rangle} = [\overrightarrow{a}^{\langle t \rangle}; \overleftarrow{a}^{\langle t \rangle}]$ to represent the concatenation of the activations of both the forward-direction and backward-directions of the pre-attention Bi-LSTM.

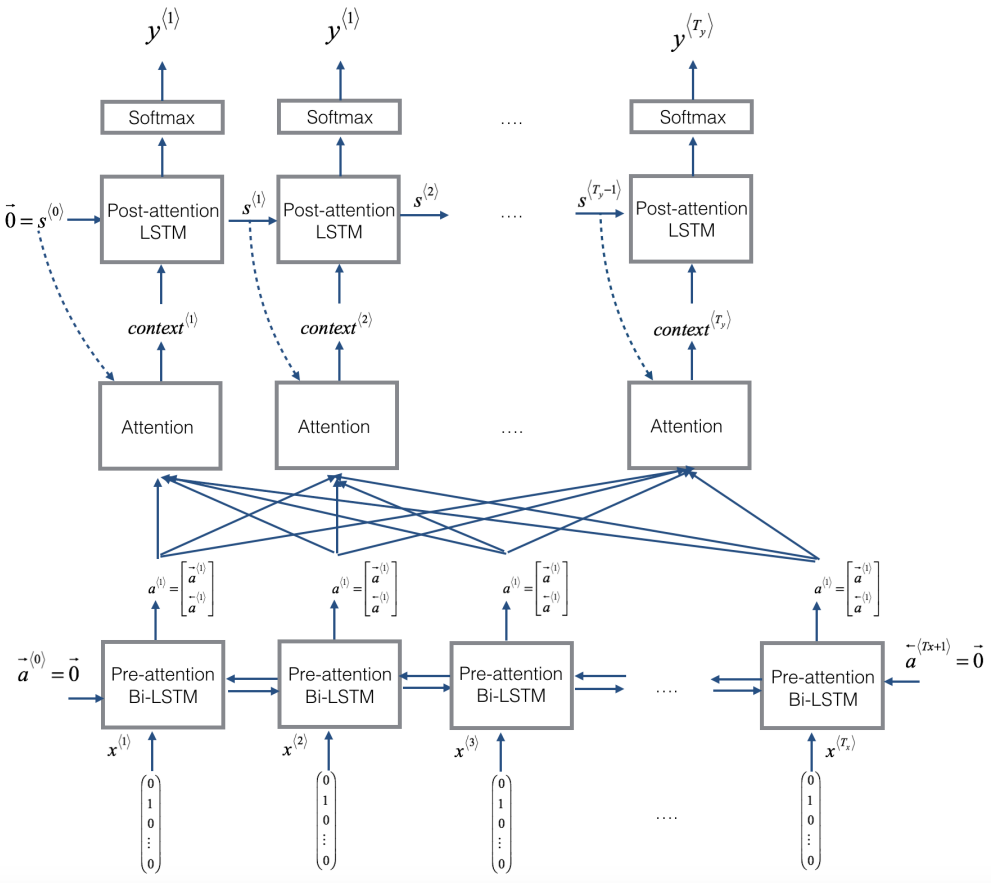

Figure 12: A complete attention model, with two separate LSTMs used as a pre-attention model to calculate the attention and a post-attention model to generate the result.

And as a conclusion[2]:

- Machine translation models can be used to map from one sequence to another. They are useful not just for translating human languages (like French->English) but also for tasks like date format translation.
- An attention mechanism allows a network to focus on the most relevant parts of the input when producing a specific part of the output.
- A network using an attention mechanism can translate from inputs of length T_x to outputs of length T_y, where T_x and T_y can be different.
- You can visualize attention weights $\alpha^{\langle t,t' \rangle}$ to see what the network is paying attention to while generating each output.

[2]directly copied from the assignment notebook

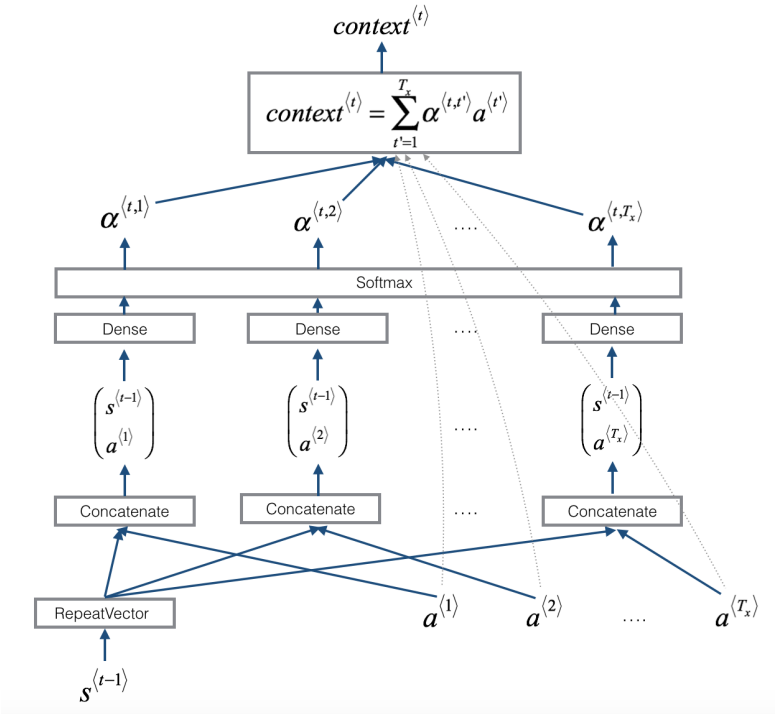

Figure 13: some caption

References

[1] C. Olah, "Understanding lstm networks," 2015.

[2] G. Chen, "A gentle tutorial of recurrent neural network with error backpropagation," *arXiv preprint arXiv:1610.02583*, 2016.

[3] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[5] deeplearning4j, "Word2vec: Neural word embeddings in java and scala," 2018.

[6] T. Bolukbasi, K.-W. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai, "Man is to computer programmer as woman is to homemaker? debiasing word embeddings," in *Advances in Neural Information Processing Systems*, pp. 4349–4357, 2016.