Natural Language Processing

Aniket Bhoy

June 25, 2024

Contents

1 Recurrent Neural Networks

1.1 Notation

Suppose we want our model to recognize names of people in a sentence(Name Entity Recognition). Consider,

- X : "Harry Potter and Hermoine Granger invented a new spell."
- Y : 1 1 0 1 1 0 0 0 0

where, 1 means its a name, while 0 means its not a name.
Let T_x be the size of the input sequence and T_y be the size of the output sequence.
$T_x = T_y = 9$ in the above example although they can be different in other problems.
$x^{(i)<t>}$ is the element t of the input vector i. Similarly $y^{(i)<t>}$ means the t^{th} element in the output sequence of the i^{th} training example.
$T_x^{(i)}$ the input sequence length for training example i. It can be different across the examples. Similarly for $T_y^{(i)}$ will be the length of the output sequence in the i^{th} training example.

1.1.1 Representing Words

We need a vocabulary list that contains all the words in our target sets.
Example: [a,...,app, apple, ...,ball, ...,zoo].
Our vocabulary contains words in alphabetical order and each word will have a unique index that it can be represented with.
We will represent every word in our vocabulary by a one-hot vector.
We can add a token in the vocabulary with name $< UNK >$ which stands for unknown text and use its index for your one-hot vector.
For example in the sentence, "Harry Potter and Hermoine Granger invented a new spell." The one-hot representation of few words are:

- $x^{<1>}$ = Harry = $(0, 0, \cdots, 1, \cdots, 0)^\intercal$, 1 is at 4075^{th} index.
- $x^{<2>}$ = Potter = $(0, 0, \cdots, 1, \cdots, 0)^\intercal$, 1 is at 6830^{th} index.
- $x^{<3>}$ = and = $(0, 0, \cdots, 1, \cdots, 0)^\intercal$, 1 is at 367^{th} index.
- $x^{<7>}$ = a = $(0, 0, \cdots, 1, \cdots, 0)^\intercal$, 1 is at 1^{st} index.

1.2 RNN Model

In this problem $T_x = T_y$. In other problems where they aren't equal, the RNN architecture may be different.
$a^{<0>}$ is usually initialized with zeros, but may be initialized randomly in some cases.

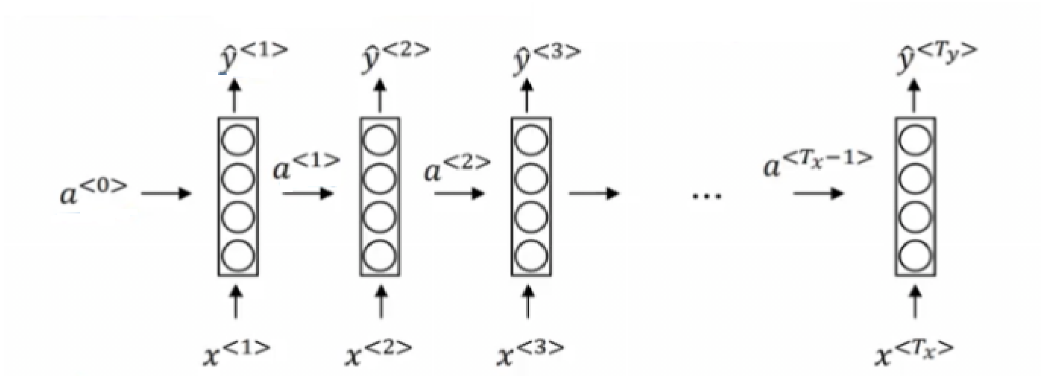


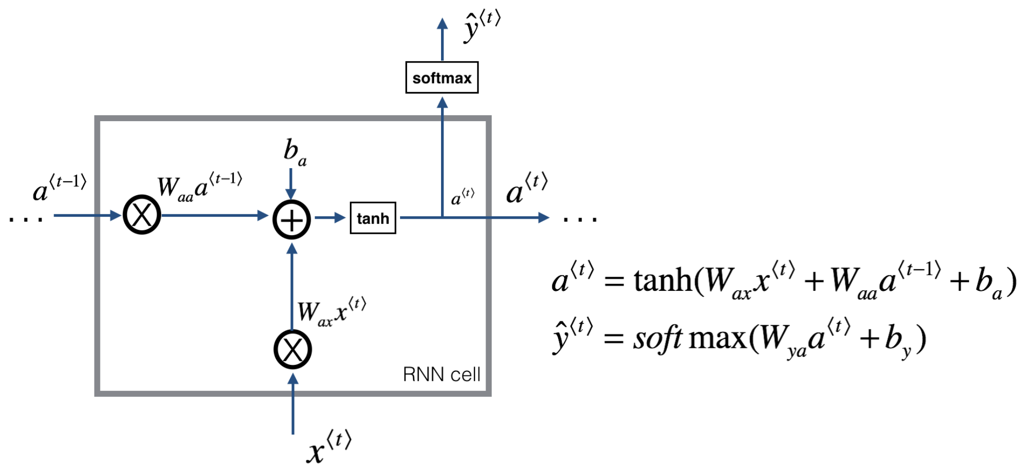
Figure 1: Model.

For each layer we are outputting $\hat{y}^{<t>}$ and $a^{<t>}$ to the next layer by taking $x^{<t>}$ and $a^{<t-1>}$ from previous layer as input. This way we have information from all the past layers.
There are three weight matrices here: W_{ax}, W_{aa}, and W_{ya} with shapes:

- W_{ax}: (NoOfHiddenNeurons, n_x)
- W_{aa}: (NoOfHiddenNeurons, NoOfHiddenNeurons)
- W_{ay}: (n_y, NoOfHiddenNeurons)

Forward propagation equations are:

- $a^{<0>} = \vec{0}$
- $a^{<t>} = f(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$
- $\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$



$\hat{y}^{\langle t \rangle}$
softmax
b_a
$a^{\langle t-1 \rangle}$
$W_{aa}a^{\langle t-1 \rangle}$
tanh
$a^{\langle t \rangle}$
$a^{\langle t \rangle}$
$W_{ax}x^{\langle t \rangle}$
RNN cell
$x^{\langle t \rangle}$
$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$
$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$

Figure 2: Basic Unit.

1.3 Backward propagation for the basic RNN unit

Computing the loss using cross-entropy loss function:

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

For the whole sequence cost is given by the summation over all the calculated single example losses:

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

We have values of $a^{<t>}$, $a^{<t>}$, weight and biases and $x^{<t>}$ from forward propagation and $\frac{\partial J}{\partial a^{<t>}}$ from next layer. Our goal is to compute partial derivative of cost function$J$ w.r.t. W_{ax}, W_{aa}, b, $x^{<t>}$, $a^{<t-1>}$ and pass the value of $\frac{\partial J^{<t>}}{\partial a^{<t-1>}}$ to previous layer.
Value of $a^{<t>} = f(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$ and derivative of tanh is:

$$\frac{\partial \tanh x}{\partial x} = 1 - (\tanh x)^2$$

Using given equations all the required partial derivatives can be calculated as:

$$\frac{\partial a^{<t>}}{\partial W_{ax}} = (1 - (\tanh(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a))^2)x^{<t>\intercal}$$

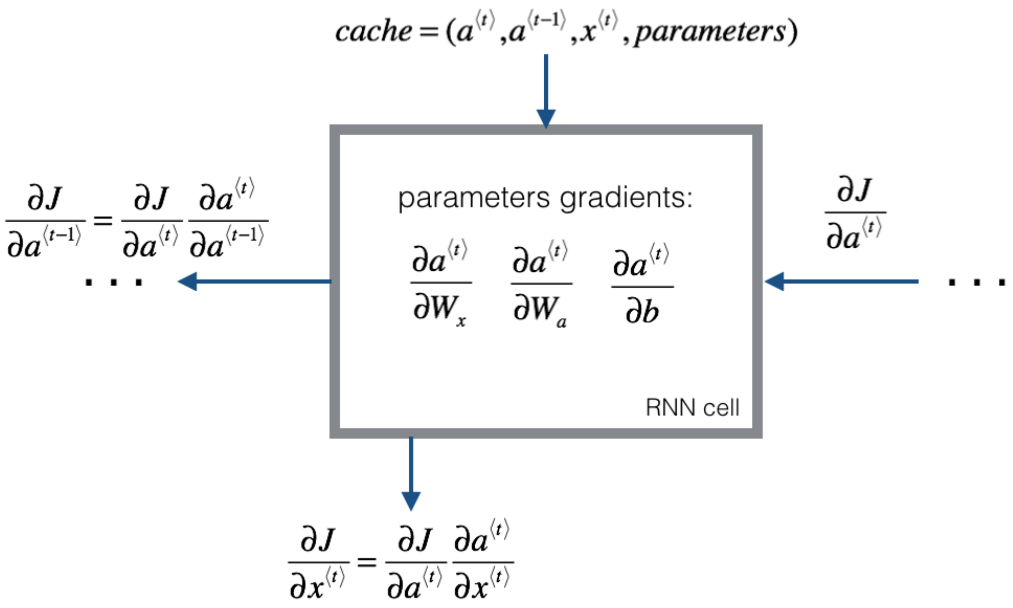

$cache = (a^{\langle t \rangle}, a^{\langle t-1 \rangle}, x^{\langle t \rangle}, parameters)$
parameters gradients:
$\frac{\partial a^{\langle t \rangle}}{\partial W_x}$ $\frac{\partial a^{\langle t \rangle}}{\partial W_a}$ $\frac{\partial a^{\langle t \rangle}}{\partial b}$
RNN cell
$\frac{\partial J}{\partial a^{\langle t \rangle}}$
$\frac{\partial J}{\partial a^{\langle t-1 \rangle}} = \frac{\partial J}{\partial a^{\langle t \rangle}} \frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}}$
$\frac{\partial J}{\partial x^{\langle t \rangle}} = \frac{\partial J}{\partial a^{\langle t \rangle}} \frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}}$

Figure 3: Backward propagation

$$\frac{\partial a^{<t>}}{\partial W_{aa}} = (1 - (\tanh(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a))^2)a^{<t-1>\intercal}$$

$$\frac{\partial a^{<t>}}{\partial b} = \sum_{batch}(1 - (\tanh(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)))^2$$

$$\frac{\partial a^{<t>}}{\partial x^{<t>}} = W_{ax}^{\intercal}(1 - (\tanh(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a))^2)$$

$$\frac{\partial a^{<t>}}{\partial a^{<t-1>}} = W_{aa}^{\intercal}(1 - (\tanh(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a))^2)$$

1.4 Different types of RNN

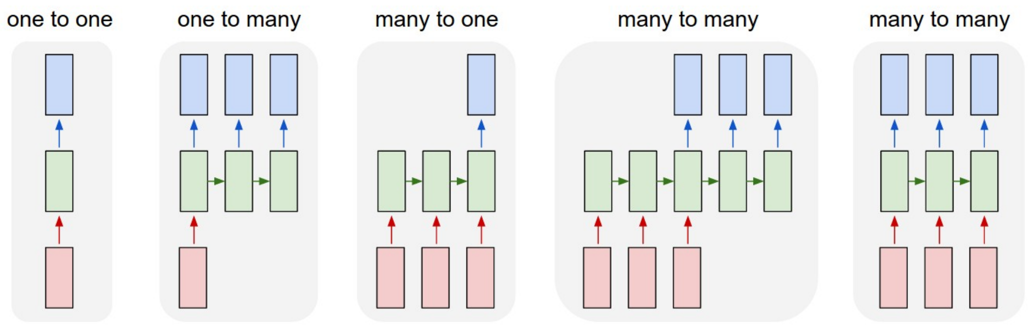


Figure 4: Different types of RNN.

For music generation, a One to Many architecture application would be useful Many To Many architecture finds applications in machine translation where inputs and outputs sequences have different lengths in most of the cases.

1.5 Language Model and Sequence Generation

While solving speech recognition problems we want our language model to give a probable next word in a sentence. We can do that by first getting a training set consisting of many language texts. Then, we can tokenize the words of these texts through our vocabulary and convert them into one-hot vectors.

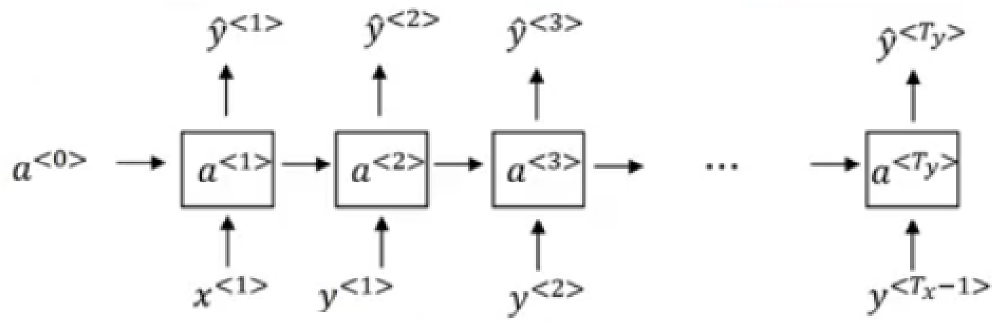


Figure 5:

For predicting the probability of next word, we feed a sentence to RNN and compute $y^{<t>}$ and then arrange in order of decreasing probability.

Probability of a sentence can be computed as:

$P(y^{<1>}, y^{<2>}, y^{<3>}, \dots, y^{<t>}) = P(y^{<1>}) * P(y^{<2>}) * P(y^{<3>}) * \dots * P(y^{<t>})$

The loss function is defined by cross-entropy loss:

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

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

1.6 Sampling Novel sequences

The purpose of novel sequence is to check what the sequence model has learned.

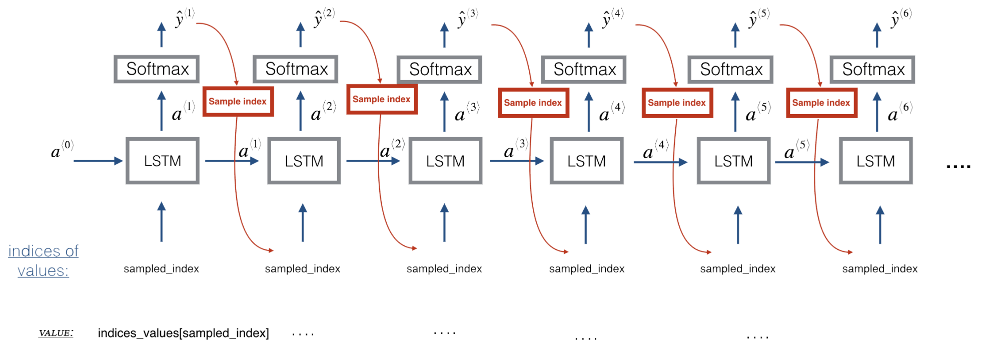


Figure 6: RNN architecture.

1. Initialize $a^{<0>} = \vec{0}$, and $x^{<1>} = \vec{0}$.
2. To get a random beginning of a sentence after every run we randomly choose a prediction obtained by $\hat{y}^{<1>}$.
3. Pass the last predicted word to the next unit.
4. Then we repeat the 2 steps above for a preferred length or until we reach $< EOS >$ token.

We can also implement a character-level language model in which the vocabulary contains letters and numbers [a-z,A-Z,0-9], punctuation, special characters and tokens.

1.7 Vanishing gradients with RNNs

While computing gradients at a particular unit, we need to multiply fractions since we have to compute gradients for previous units. Multiplication of large numbers leads to exploding gradients while small fractions leads to vanishing gradients. Problem of vanishing gradient is more common with basic RNN architectures that are trained on language models
We can identify exploding gradients easily as the value of weights become NaN.

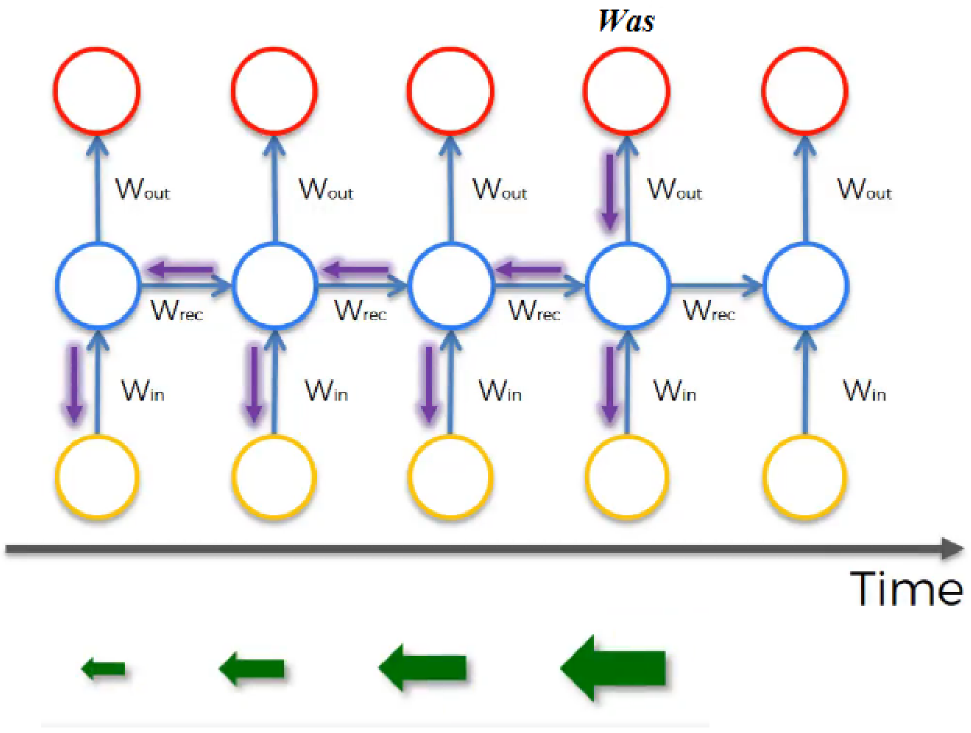


Figure 7: Vanishing Gradient.

To solve exploding gradient we apply gradient clipping in which we re-scale our gradient vector to make it clipped according to some maximum value.

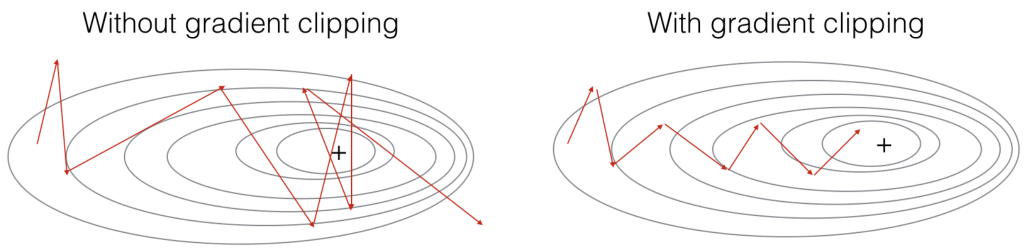


Figure 8: Gradient clip.

1.8 Gated Recurrent Unit(GRU)

GRU is implemented to solve the vanishing gradient problem and remembering long-term dependencies.
To decide whether to memorize something or not, GRU has a memory cell C which stores information. If the update gate (Γ_u) value becomes 1, memory cell discards the older value and stores a new one. $\tilde{C}$ is the candidate cell.
In GRUs:

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

Equations for the simplified version of GRUs:

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

GRU avoid vanishing gradient problem because Γ_u tends to zero usually. In the equation this makes $C^{<t>} = C^{<t-1>}$ in a lot of cases. Value of memory cell is based on the update gate and the previous cell.
Shapes:

- $a^{<t>}$ shape is (NoOfHiddenNeurons, 1)
- $C^{<t>}$ is the same as $a^{<t>}$
- $\tilde{C}^{<t>}$ is the same as $a^{<t>}$
- Γ_u is the same as $a^{<t>}$

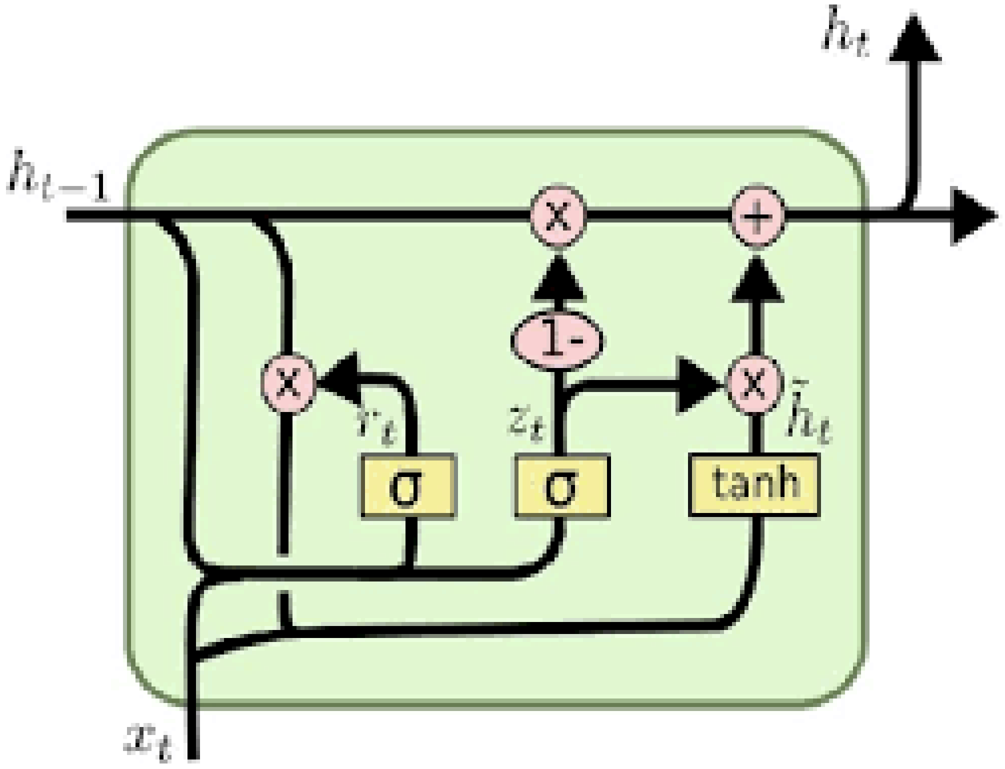


Figure 9: GRU cell.

The full GRU contains an additional gate (Γ_r) that decide how relevant is $C^{<t-1>}$ to $C^{<t>}$.
Equations:

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

1.9 Long Short Term Memory(LSTM)

LSTM can also remember long-term dependencies just like GRU. To decide whether to memorize something or not, GRU has a memory cell C which

stores information. Though it is more widely used since it is more powerful and more general than GRU.
In LSTM ,

$$C^{<t>}! = a^{<t>}$$

Equations of an LSTM unit:

- $\tilde{C}^{<t>} = \tanh(W_c[a^{<t-1>}, X^{<t>}] + b_c)$
- $\Gamma_u = \sigma(W_u[a^{<t-1>}, X^{<t>}] + b_u)$
- $\Gamma_f = \sigma(W_f[a^{<t-1>}, X^{<t>}] + b_f)$
- $\Gamma_o = \sigma(W_o[a^{<t-1>}, X^{<t>}] + b_o)$
- $C^{<t>} = \Gamma_u * \tilde{C}^{<t>} + \Gamma_f * \tilde{C}^{<t-1>}$
- $a^{<t>} = \Gamma_o * \tanh C^{<t>}$

In LSTM we have an update gate Γ_u (also known as input gate I), a forget gate Γ_f, an output gate Γ_o, and a candidate cell variable $\tilde{C}^{<t>}$.

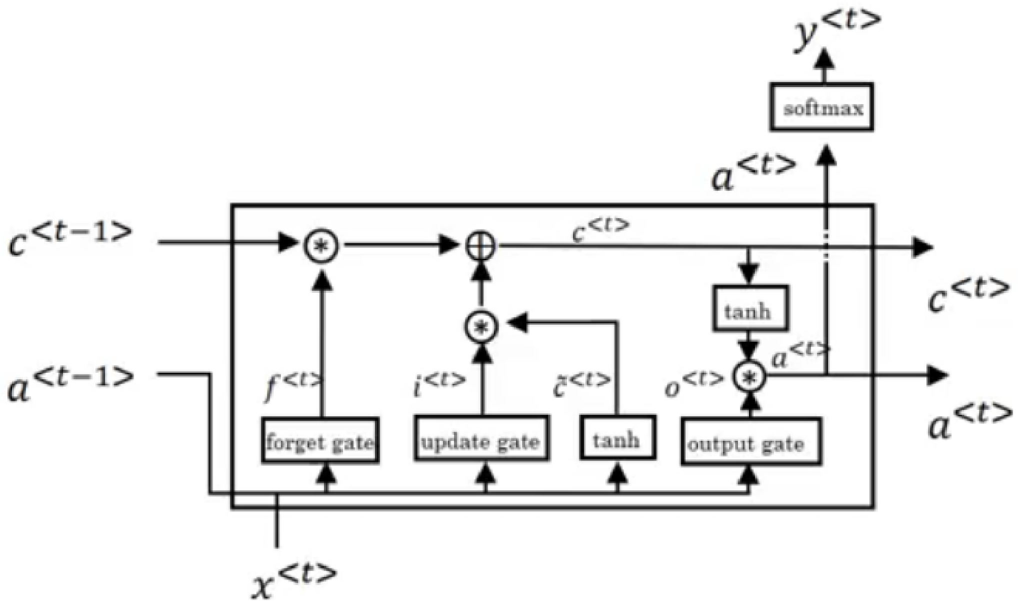


Figure 10: LSTM cell.

1.10 Bidirectional RNN

To illustrate the importance of BiRNN, we shall look at an example of the Name entity recognition task:

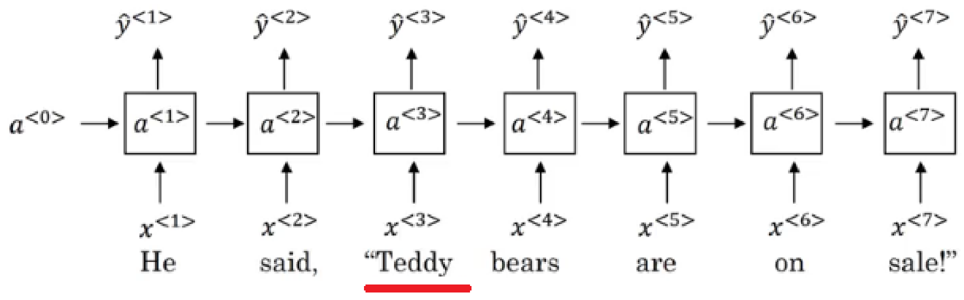


Figure 11: Name Entity recognition.

To know whether the word **Teddy** is a name or not, we have to process words from both front and back, this where BiRNN can help us.

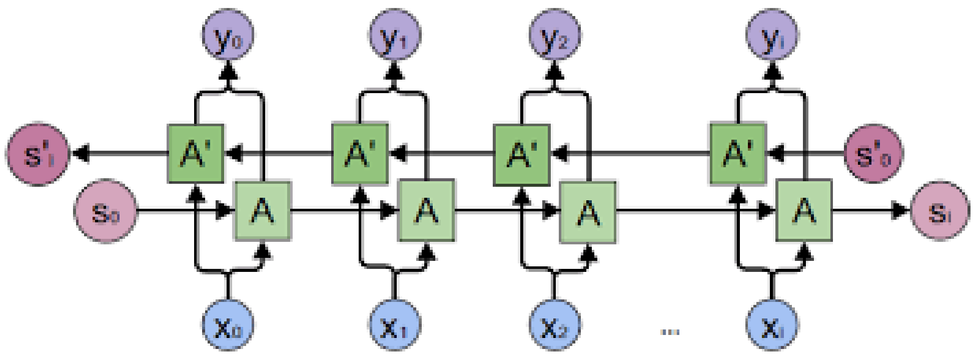


Figure 12: BiRNN.

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

- Unlike basic RNN, in BiRNN we have two parallel flow of blocks.
- One of them is responsible for forward propagation from left to right, while the other from right to left.

- To compute $\hat{y}^{<t>}$ at a particular layer we combine inputs from blocks of both sides.
- For text processing problems a BiRNN with LSTM blocks is preferred.

1.11 Deep RNN

- Standard RNNs are enough to solve a lot of basic NLP problems, however to get better results in more complex problems use of deeper layers might be useful.
- In feed forward deep RNNs, number of layers ranging from 100 to 200 are rare to see. As even 3 layer has a lot of parameters and is computationally expensive to train.

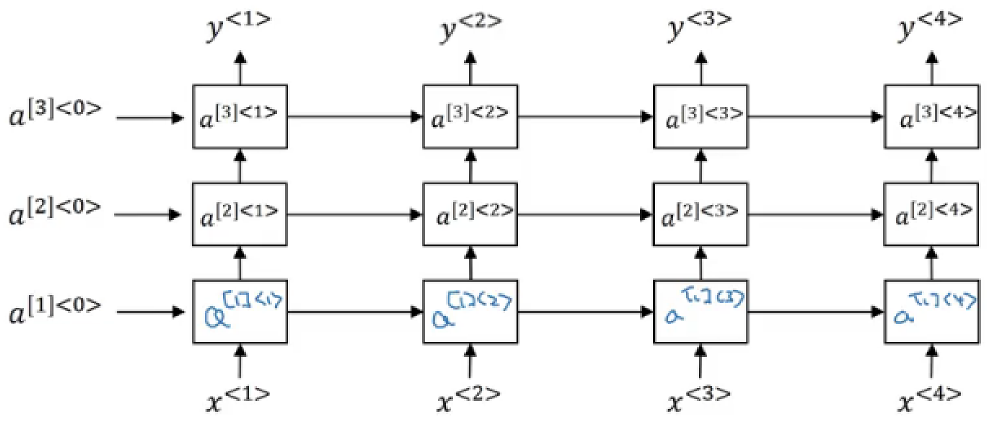


Figure 13: DeepRNN.

Sometimes feed-forward network layers are connected at the end of recurrent cell to get better results.

2 Natural Language Processing and Word Embeddings

2.1 Introduction to Word Embedding

2.1.1 Word Representation

In order to process language we need to represent words in form of vectors, which is essential to perform calculations. Word embeddings is one of the ways in which we can do that. It helps our algorithm to make analogies between different words.

Let O_{idx} represent the one-hot vector of the idx^{th} word in our vocabulary. We can have a featurized representation of these words: man, woman, king, queen, apple, and orange.

	Man	**Women**	**King**	**Queen**	**Apple**	**Orange**
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Food	0.04	0.01	0.02	0.01	0.95	0.97
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Age	0.03	0.02	0.7	0.69	0.03	-0.02

We'll represent each of the 10000 word in terms of 300 features with a floating point number ranging from -1 to 1. The benefit of doing this is that we will be able to group together words that fall under same category, given that that category exists as one of the 300 features that we have chosen. This way we can generalize between them. Each word has 300 features with a type of float point number.
For example "I want a glass of orange ", a model should predict the next word as juice.
We will use the notation e_j to represent featurized representation of j^{th} word in our vocabulary.

Given this example (from named entity recognition):

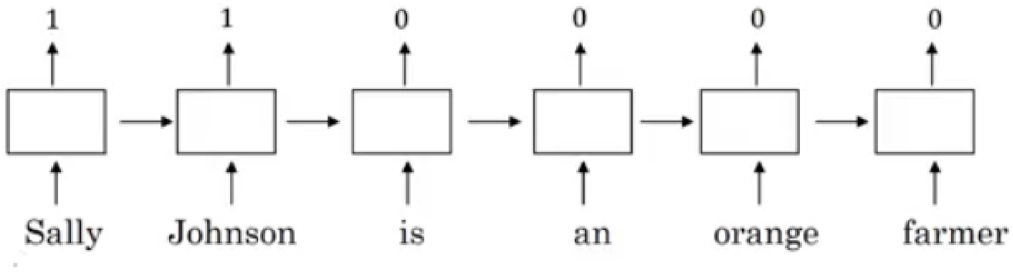


Figure 14:

If we allow our model to train on this sentence, it will be able to identify James Demper as farmer in the sentence "James Demper is an apple farmer" by making analogy between orange and apple.

Transfer learning and word embeddings:

- We train our model from large text corpus available in the internet and learn word embeddings (1-100 billion words). This way we can cover all the most commonly used words.
- Then we transfer the word embedding to our NLP problem with a smaller training set (say about 100 thousand words).

Analogy Reasoning:

	Man	**Women**	**King**	**Queen**	**Apple**	**Orange**
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Food	0.04	0.01	0.02	0.01	0.95	0.97
Age	0.03	0.02	0.7	0.69	0.03	-0.02
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Given above word embeddings table, we can conclude this relation:

- Man $\Longrightarrow$ Woman
- King $\Longrightarrow$?

We can write the vectors as:

- $e_{Man} = \begin{bmatrix} -1 & 0 & 0 & 0 \end{bmatrix}$
- $e_{Woman} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$
- $e_{King} = \begin{bmatrix} -1 & 1 & 0 & 1 \end{bmatrix}$
- $e_{Queen} = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$

We can subtract e_{Man} from e_{Woman} to get $e_{Woman} - e_{Man} = \begin{bmatrix} -2 & 0 & 0 & 0 \end{bmatrix}$.
Also, $e_{Queen} - e_{King} = \begin{bmatrix} -2 & 0 & 0 & 0 \end{bmatrix}$
Since both of these vectors are similar if we want our algorithm to find $e_?$, we can reformulate the problem to solve: $e_{Man} - e_{Woman} \approx e_{King} - e_?$
It can also be represented mathematically by:

$$\underset{w}{\operatorname{argmax}}\ sim(u, v)$$

where, u = e_{Woman}, v = $e_{Man} - e_{King} + e_?$
& $sim(u, v)$ is the cosine similarity function:

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

On calculating $sim(u, v)$ with for all words, we'll find that it is maximum for $e_? = e_{Queen}$.

2.1.2 Embedding Matrix

The matrix generated after performing word embedding to all the words is embedding matrix(E). Its shape is (number of features, number of words in vocabulary) = (300, 10000). Relation between O_j and e_j is simply, $E{\cdot}O_j = e_j$.

2.2 Word2vec & GloVe

2.2.1 Word2vec

The models rely on the idea that, words that appear in the same context have similar representations.
It can capture semantic meaning of words.
Skip-grams: It predicts words within a certain window size before and after the current word in the same sentence. This model can help us maximize the probability of finding the context word for a given target word.
Example: For window size=3: The (target,context) pair for the sentence "The wide road shimmered in the hot sun." will be:

X	**Y**
wide	The, road
road	wide, shimmered
shimmered	road, in
the	in, hot
hot	the, sun

- We have one-hot vector of all the words in the sentence.
- Our aim is to represent each vector in terms of N features.
- We will do this by training a NN model on (X,Y). The input layer will contain all the target words. The hidden layer has number of nodes equal to N(same as number of features). The output layer outputs probabilities of the context words using softmax classifier.
- Then by comparing actual word with output we'll get loss function then we can back-propagate to update the weights.
- After training we will obtain weights corresponding to each input word. These weights are the actual representation of the input words.

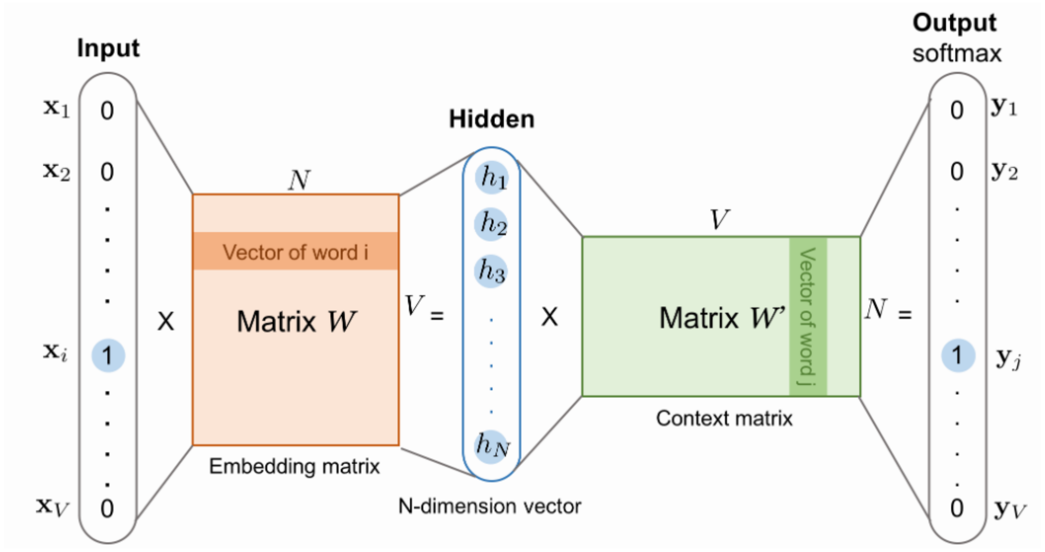


Figure 15: Skipgram

We use softmax layer to calculate $P(t|c)$.
Also, $e_c = E \cdot O_c$

$$P(t|c) = \frac{e^{\theta_t^\intercal e_c}}{\sum_{j=1}^{10000} e^{\theta_j^\intercal e_c}}$$

2.2.2 Negative Sampling

Negative sampling allows us approximate the softmax.
Maximising this ratio $P(t|c)$ ensures, words that appear closer together in text have more similar vectors than words that do not. However, computing this can be expensive, because there are lot of contexts c. In negative sampling we just select a bunch of context words c. Result is that if word w1 appears in the context of w2, then the vector of w2 is more similar to the vector of w1 than the vectors of several other chosen words. In the equation the denominator is expensive to compute:

$$P(t|c) = \frac{e^{\theta_t^\intercal e_c}}{\sum_{j=1}^{10000} e^{\theta_j^\intercal e_c}}$$

To generate a positive sample we pick a positive context by using skip-grams technique.

To generate k negative samples we pick random words from our vocabulary.

We get positive example by using the same skip-grams technique, with a fixed window that goes around.
To generate a negative example, we pick a word randomly from the vocabulary.
Therefore, we have $k : 1$ ratio of negative to positive samples in the data we are collecting.

How to select negative samples:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10000} f(w_j)^{3/4}}$$

where $f(w_i)$ is frequency of word i.

2.2.3 GloVe word vectors

Global vectors for word representations.
It creates a co-occurrence matrix X using a large corpus of words in which X_{ij} represents number of times word j appeared in context of i. Let,

$$P(j|i) = P(j \text{ is in the context of } i) = \frac{X_{ij}}{X_i}$$

where, $X_i = \sum_k X_{ik}$

	k=solid	**k=liquid**	**k=gas**	**k=random**
$P(k\|ice)$	high	low	high	low
$P(k\|steam)$	low	high	high	low
$\frac{P(k\|ice)}{P(k\|steam)}$	> 1	< 1	~ 1	~ 1

Define:

$$F(w_i, w_j, \tilde{w_k}) = \frac{P(k|i)}{P(k|j)}$$

$$\Rightarrow F((w_i - w_j)^\intercal \cdot \tilde{w_k}) = \frac{P(k|i)}{P(k|j)}$$

Assuming homomorphism:

$$F(w_i^\intercal \cdot \tilde{w_k} - w_j^\intercal \cdot \tilde{w_k}) = \frac{F(w_i^\intercal \cdot \tilde{w_k})}{F(w_j^\intercal \cdot \tilde{w_k})} = \frac{P(k|i)}{P(k|j)}$$

$$\Rightarrow F(w_i^\intercal \cdot \tilde{w_k}) = cP(k|i)$$

$F(x) = e^x$ is a solution. Therefore,

$$w_i^\intercal \cdot \tilde{w_k} = \ln P(k|i) = \ln X_{ik} - \ln X_i$$

By introducing bias, X_i term can be absorbed.

$$w_i^\intercal \cdot \tilde{w_k} + b_i + \tilde{b_k} = \ln X_{ik}$$

Our objective is to minimize difference between $\theta_i^\intercal \cdot e_j$ and $\log(X_{ij})$.
Also, $X_{ij} = X_{ji}$, if we choose a window pair.
The loss function is defined like this:

$$\min \sum_{i=1}^{10000} \sum_{j=1}^{10000} f(X_{ij})(\theta_i^\intercal e_j + b_i + b'_j - \log X_{ij})^2$$

where, θ and e are symmetric & $f(x)$ is the weighting function.

2.3 Applications using word embeddings

2.3.1 Sentiment classification

Sentiment classification helps to identify the sentiment involved in texts, whether it is positive, negative or neutral.

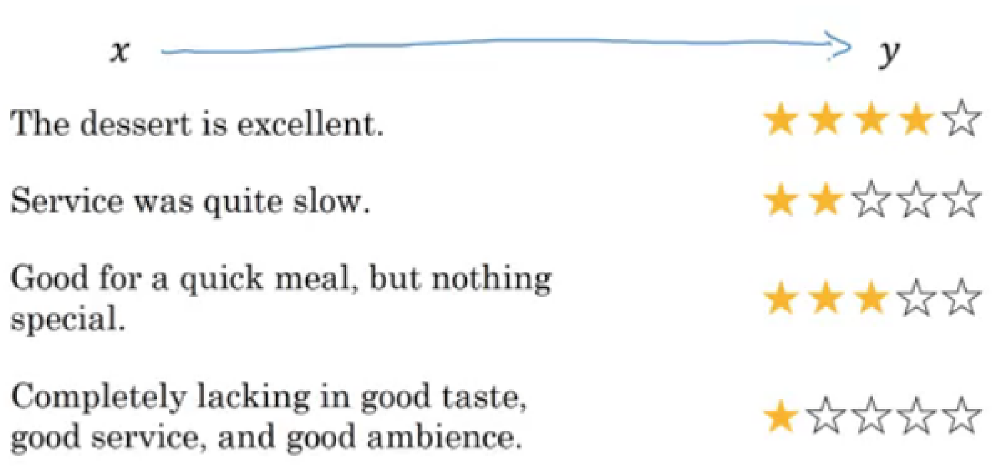

x
y
The dessert is excellent.
Service was quite slow.
Good for a quick meal, but nothing special.
Completely lacking in good taste, good service, and good ambience.

Figure 16:

In the above examples we are rating customer reviews from 1 star to 5 star.

We can use word embeddings to create huge labelled training data, say 100 billion words.
If we simply average all the word vectors to get an overall value and decide the sentiment, then it can cause misleading reviews. For example "Completely lacking in good taste, good service, and good ambience" has the word good 3 times but its a negative review.
To avoid that we use a better model like RNN:

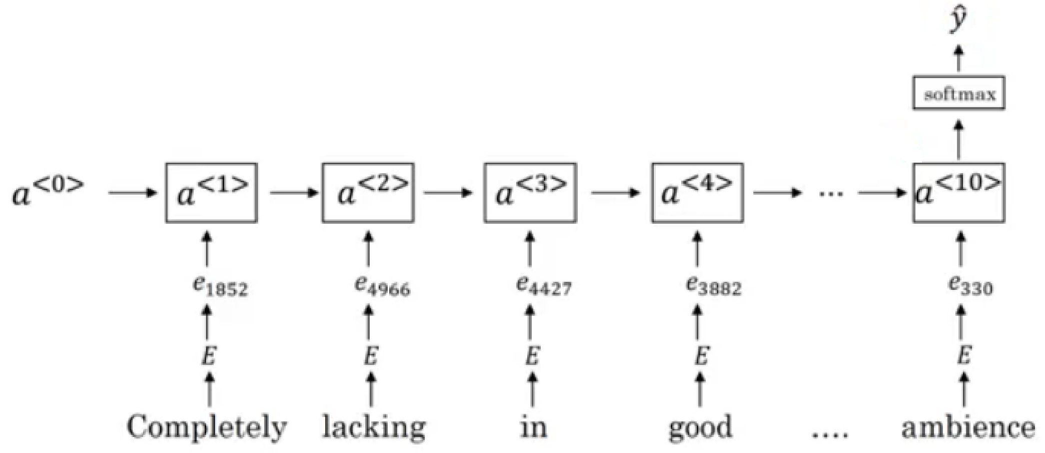


Figure 17: Many-to-one

2.3.2 Debiasing word embeddings

To avoid word embeddings from reflecting undesirable forms of bias, such as gender, age, race we have to de-bias it. For e.g. our model might say nurse in the analogy:

- Men=→ Doctor as Women=→ ?

This can be offensive, as it is being gender biased.

Neutralizing bias for non-gender specific words: If we're using a 50-dimensional word embedding, the 50 dimensional space can be split into two parts: The bias-direction g, and the remaining 49 dimensions, which we'll call $g_{\perp}$. The neutralization step takes a vector such as e_{nurse} and zeros out the component in the direction of g, giving us $e_{nurse}^{debiased}$.
$e^{bias\_component}$ is the projection of e onto the direction of g.

$$e^{bias\_component} = \frac{e \cdot g}{\|g\|_2^2} * g$$

$$e^{debiased} = e - e^{bias\_component}$$

Equalization algorithm for gender-specific words: Equalization is applied to pair of words that you might want to have differ only through the gender property.
Equalization is to make sure that a particular pair of words are equi-distant from the 49-dimensional $g_{\perp}$.
Suppose e_{w1} & e_{w2} are the word vector pairs we want to equalize, then we have to perform following calculations:

$$\mu = \frac{e_{w1} + e_{w2}}{2}$$

$$\mu_B = \frac{\mu \cdot bias\_axis}{\|bias\_axis\|_2^2} * bias\_axis$$

$$\mu_{\perp} = \mu - \mu_B$$

$$e_{w1B} = \frac{e_{w1} \cdot bias\_axis}{\|bias\_axis\|_2^2} * bias\_axis$$

$$e_{w2B} = \frac{e_{w2} \cdot bias\_axis}{\|bias\_axis\|_2^2} * bias\_axis$$

$$e_{w1B}^{corrected} = \sqrt{|1 - \|\mu_{\perp}\|_2^2} * \frac{e_{w1B} - \mu_B}{\|(e_{w1} - \mu_{\perp}) - \mu_B\|}$$

$$e_{w2B}^{corrected} = \sqrt{|1 - \|\mu_{\perp}\|_2^2} * \frac{e_{w2B} - \mu_B}{\|(e_{w2} - \mu_{\perp}) - \mu_B\|}$$

$$e_1 = e_{w1B}^{corrected} + \mu_{\perp}$$

$$e_2 = e_{w2B}^{corrected} + \mu_{\perp}$$

NLP Revised POA

- 27th May Basic RNN
- 3rd June Language Model, GRU, LSTM
- 12th June Word Embedding
- 22nd June Sentiment classification and de-biasing

Resources:

- https://www.coursera.org/specializations/natural-language-processing
- https://nlp.stanford.edu/pubs/glove.pdf