Natural Language Processing

End-term Report

Aniket Bhoy

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.

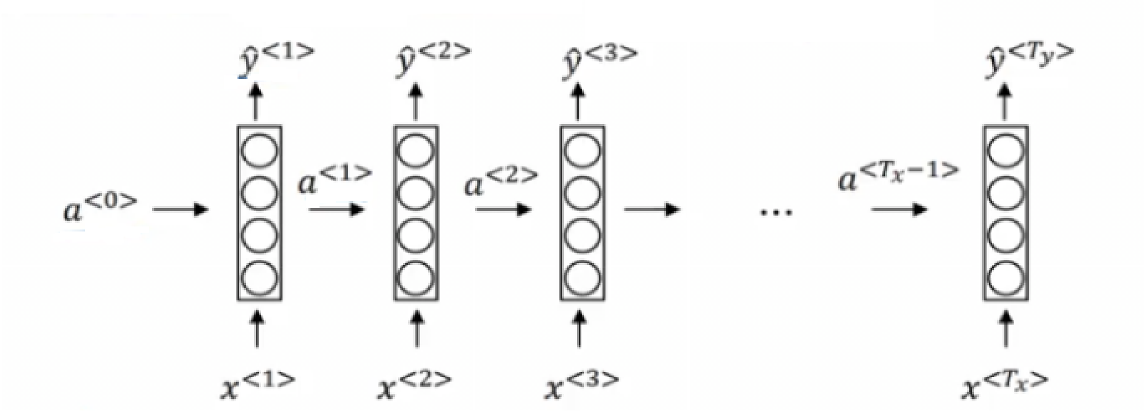


Figure 1: Model.

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:

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

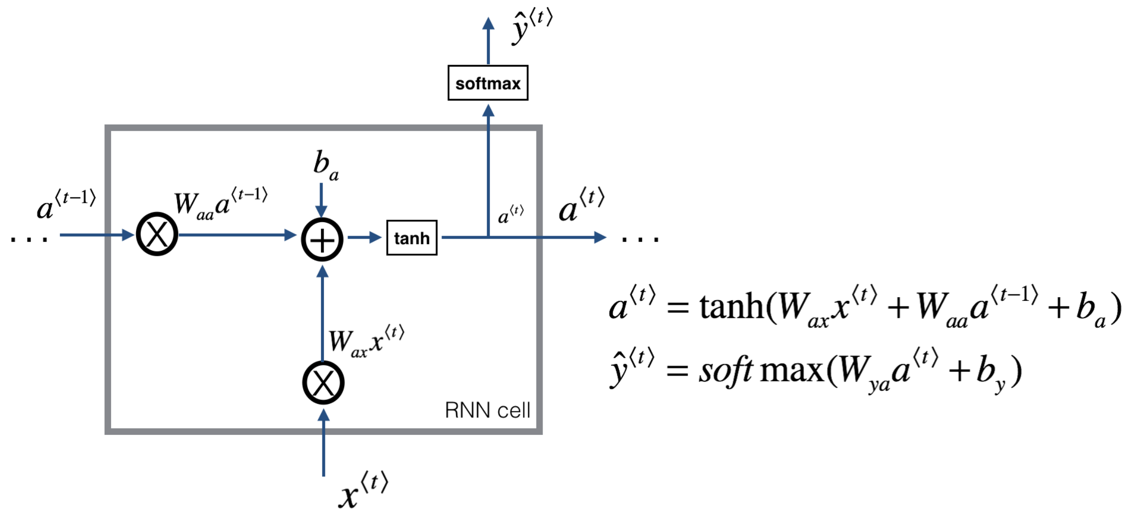

$\hat{y}^{\langle t \rangle}$
softmax
b_a
$a^{\langle t-1 \rangle}$
$W_{aa}a^{\langle t-1 \rangle}$
tanh
$a^{\langle t \rangle}$
$a^{\langle t \rangle}$
$W_{ax}x^{\langle t \rangle}$
RNN cell
$x^{\langle t \rangle}$
$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle}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

Figure 2: Basic Unit.

1.3 Backward propagation for the basic RNN unit

Computing the loss using cross-entropy loss function:

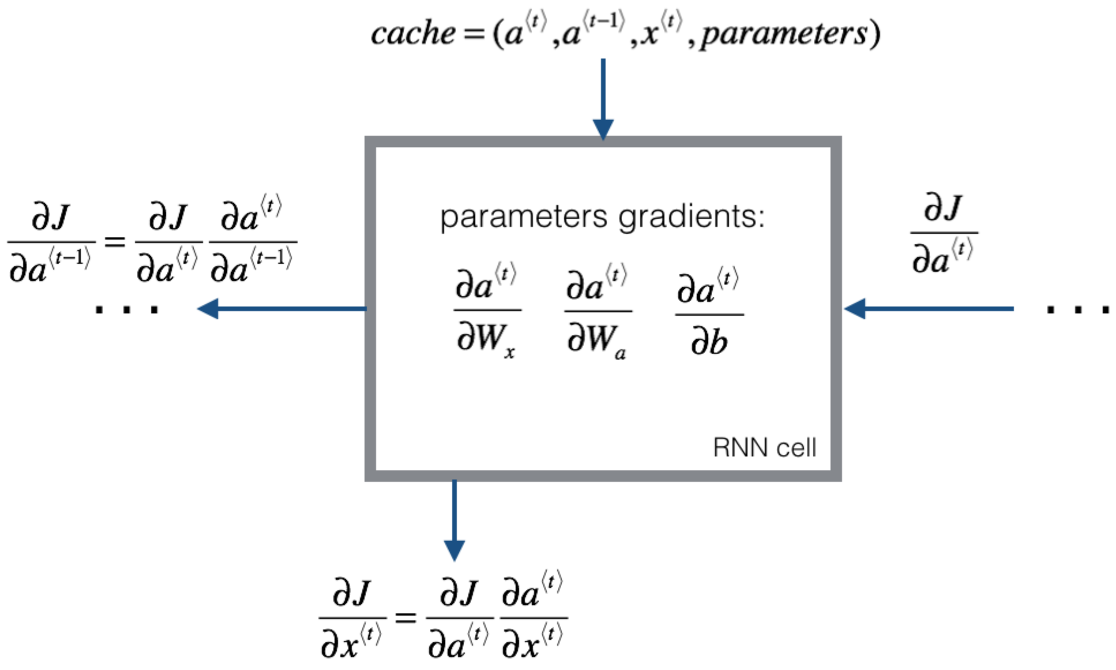
$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log \hat{y}^{<t>} - (1 - \hat{y}^{<t>}) \log(1 - \hat{y}^{<t>})$$

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 $\dfrac{\partial J}{\partial a^{<t>}}$ 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 $\dfrac{\partial J^{<t>}}{\partial a^{<t-1>}}$ to previous layer.
Value of $a^{<t>} = f(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$ and derivative of tanh is:

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



$cache = (a^{\langle t \rangle}, a^{\langle t-1 \rangle}, x^{\langle t \rangle}, parameters)$
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

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

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

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

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

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

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

1.4 Different types of RNN

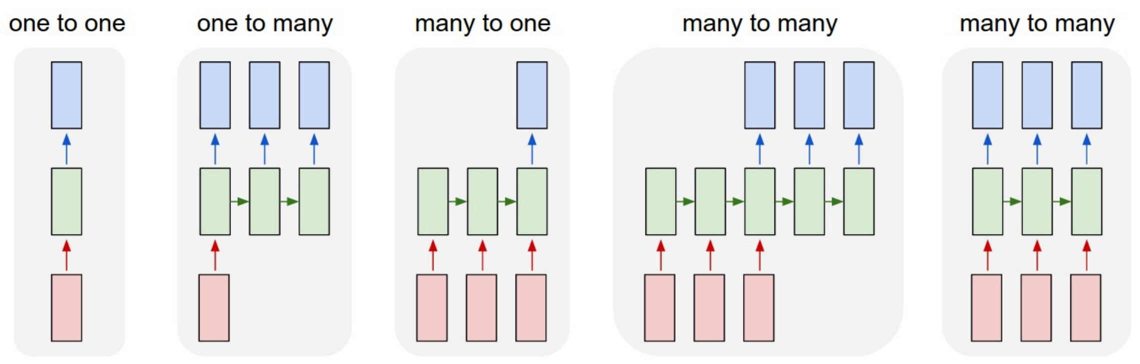


Figure 4: Different types of RNN.

Many to one:

- Input: Sequence like sentence or time series
- Output: Non sequential (numbers, scalers or 0,1)
- Used in sentiment analysis, Customer rating prediction

One to many:

- Input: Non sequential like an image
- Output: Sequential (words)
- Image captioning, Music generation

Many to many:

- Input: Sequence
- Output: Non sequential
- Input and output sequences can have same or different length
- Same length used in Parts of Speech tagging and Named entity recognition
- Different length used in machine translation (translating one language to another)

One to one:

- Input: Non sequential
- Output: Non sequential
- Used in image classification

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.

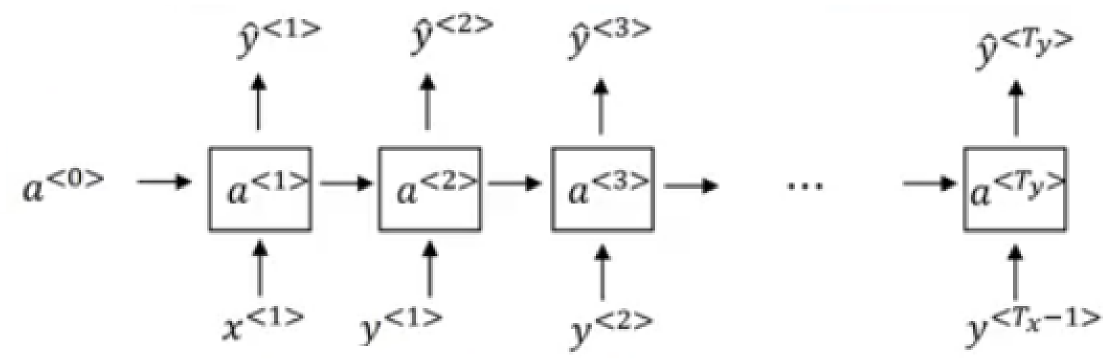


Figure 5:

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:

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

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.

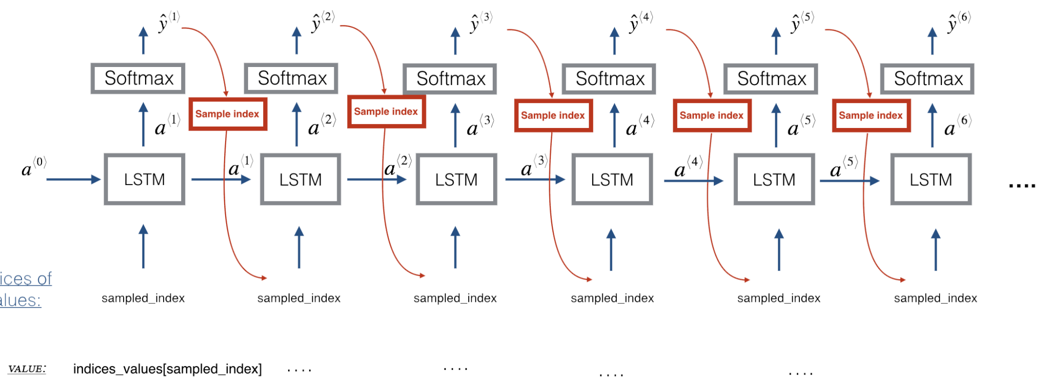


Figure 6: RNN architecture.

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.

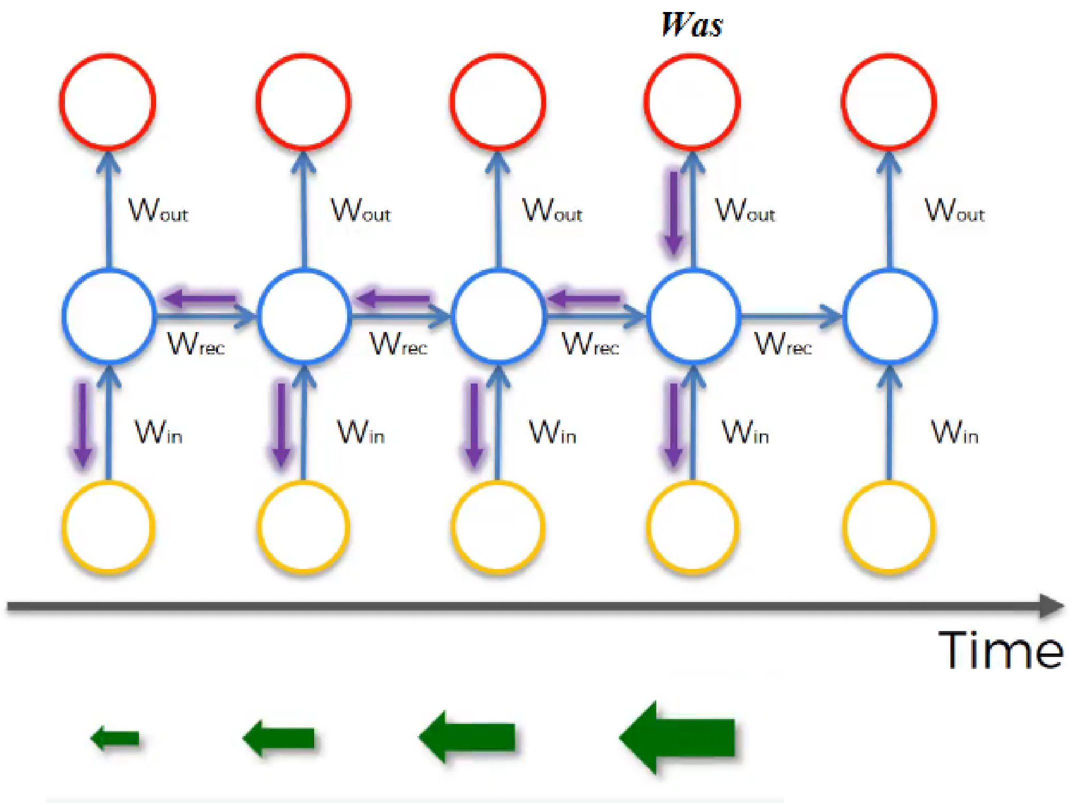


Figure 7: Vanishing Gradient.

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.

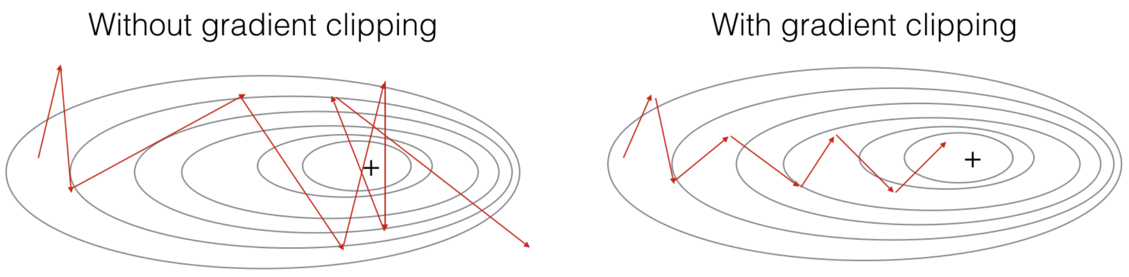


Figure 8: Gradient clip.

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 $a^{<t>}$

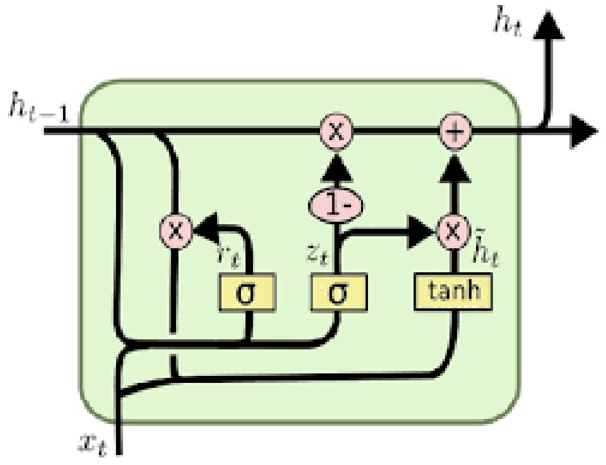


Figure 9: GRU cell.

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:

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

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

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 $\tilde{C}^{<t>}$.

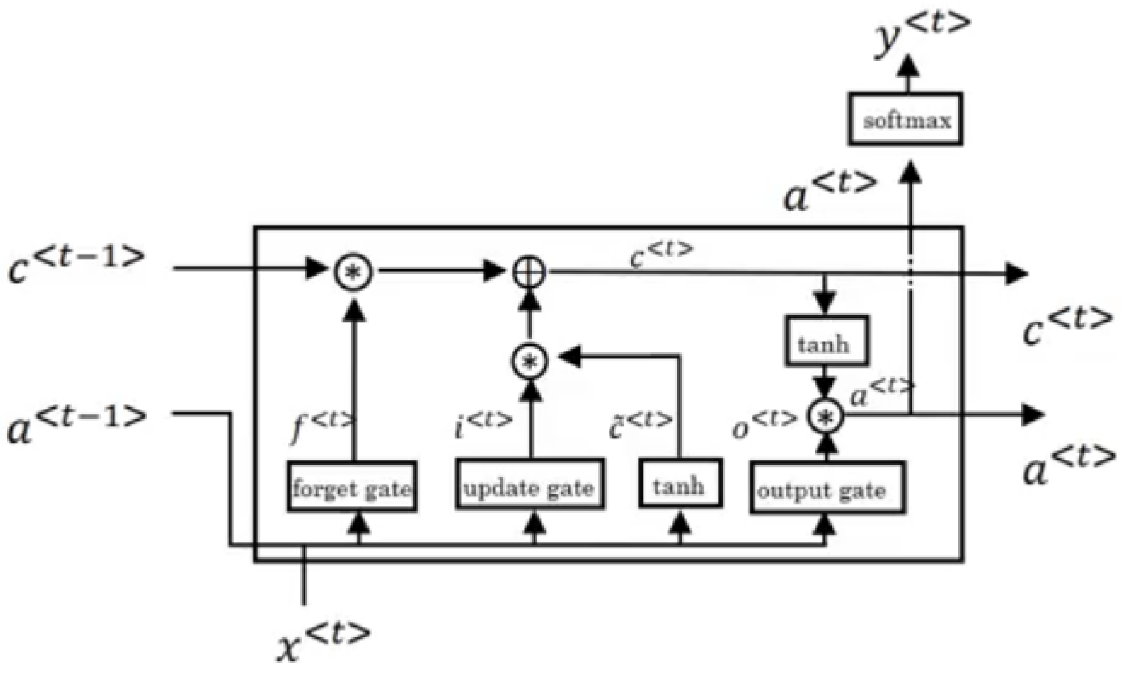


Figure 10: LSTM cell.

1.10 Bidirectional RNN

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

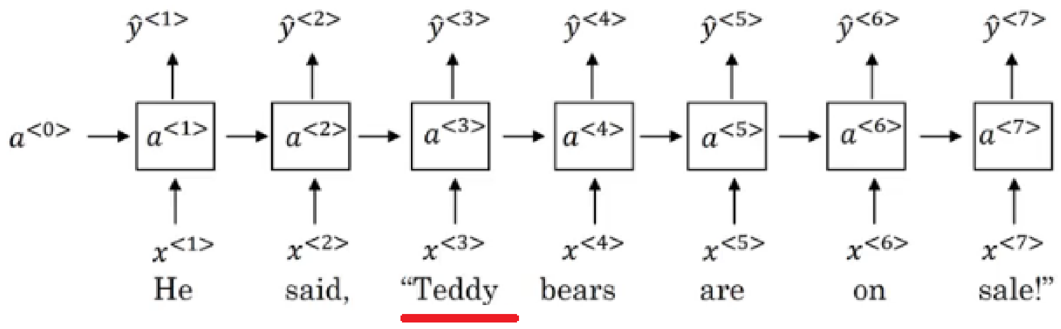


Figure 11: Name Entity recognition.

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.

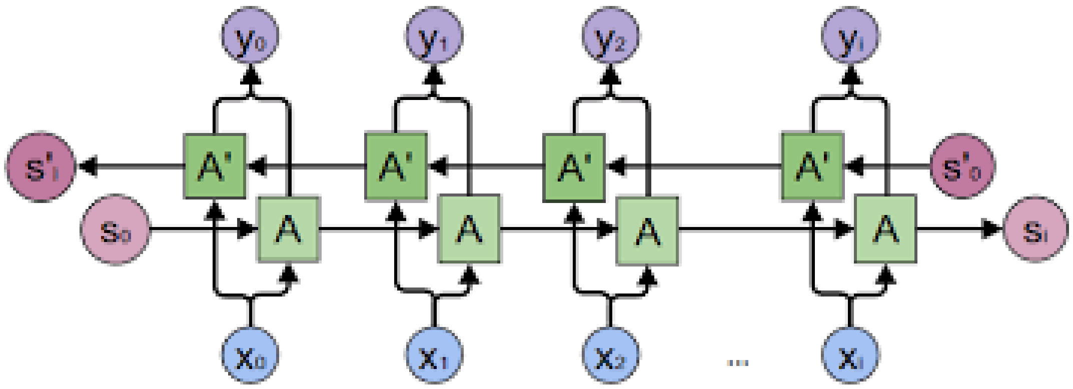


Figure 12: BiRNN.

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

- 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.

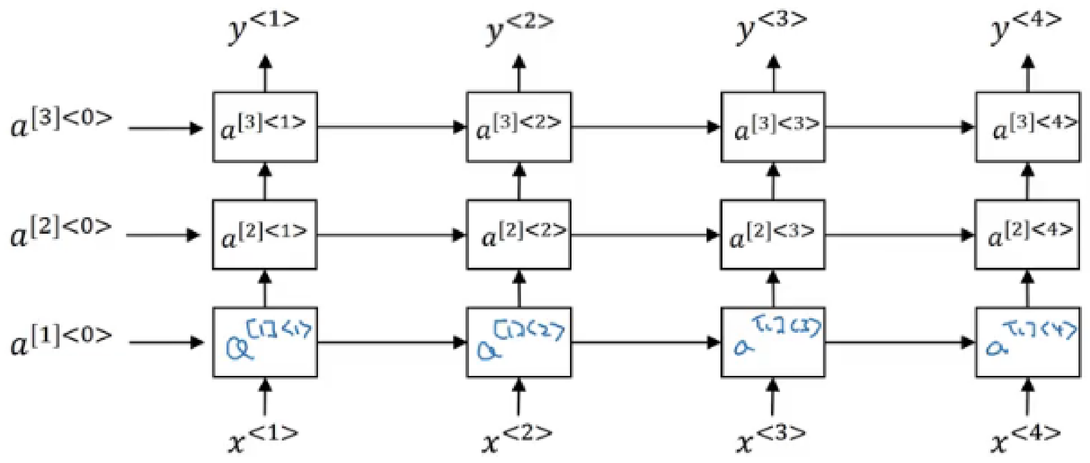


Figure 13: DeepRNN.

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):

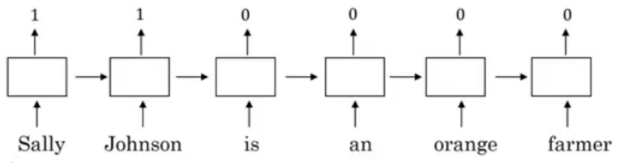


Figure 14:

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
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

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:

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

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.

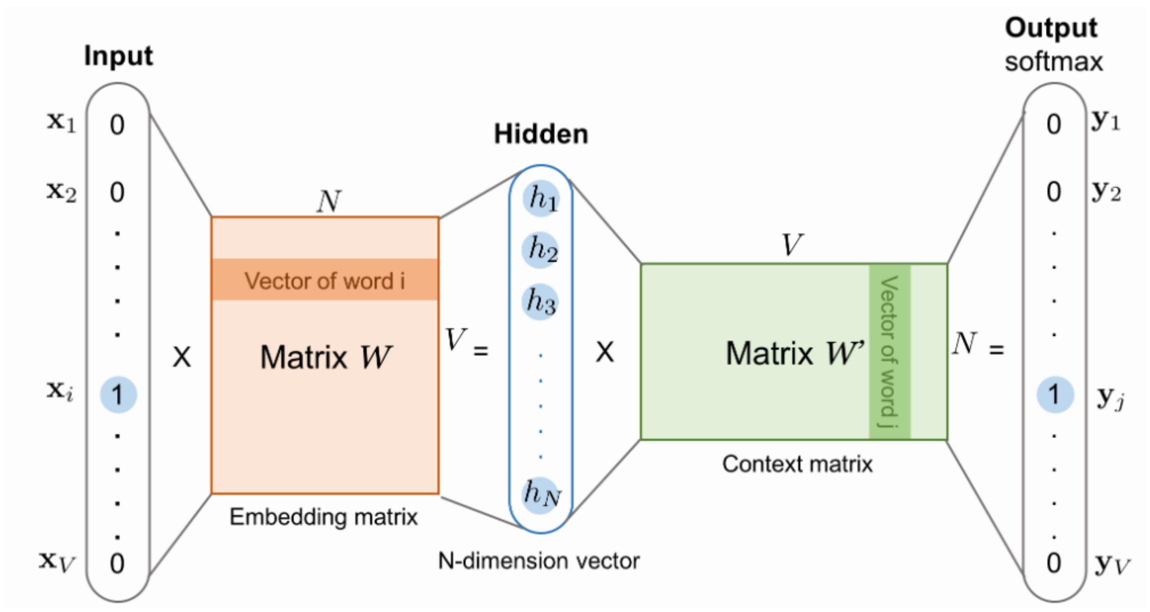


Figure 15: Skipgram

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.

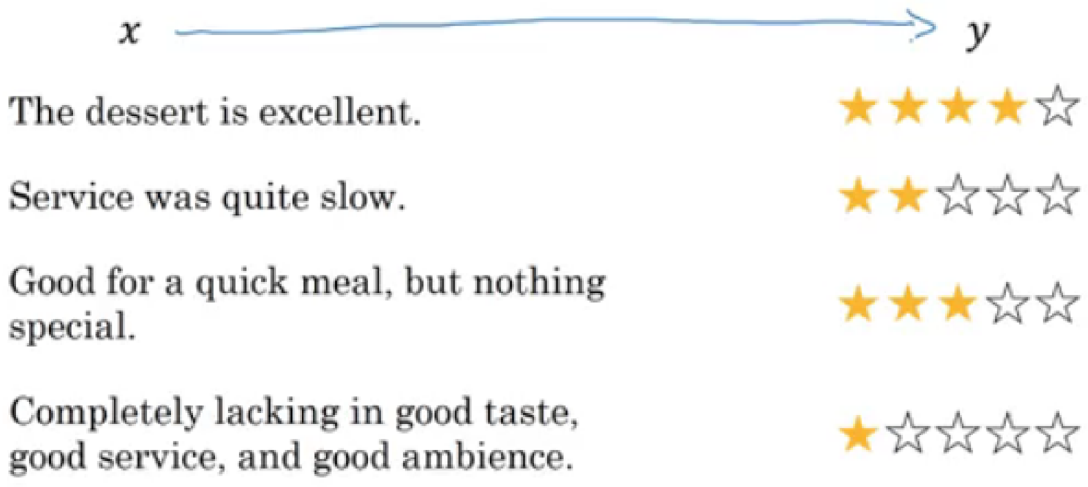

x
y
The dessert is excellent.
Service was quite slow.
Good for a quick meal, but nothing special.
Completely lacking in good taste, good service, and good ambience.

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:

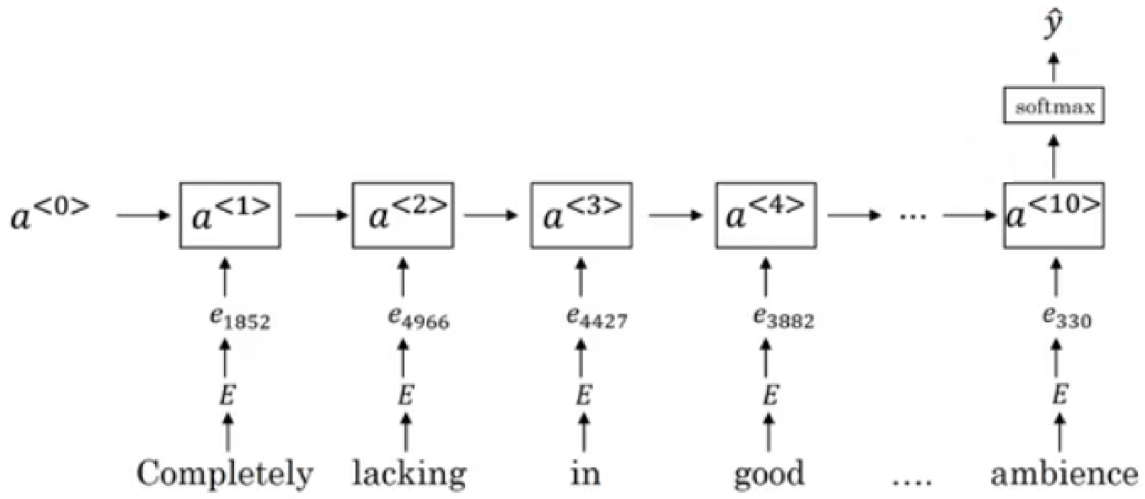


Figure 17: Many-to-one

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=$\rightarrow$ Doctor as Women=$\rightarrow$?

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:

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

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

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

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

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

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

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

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

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

3 Sequence Models and Attention Mechanism

3.1 Machine Translation

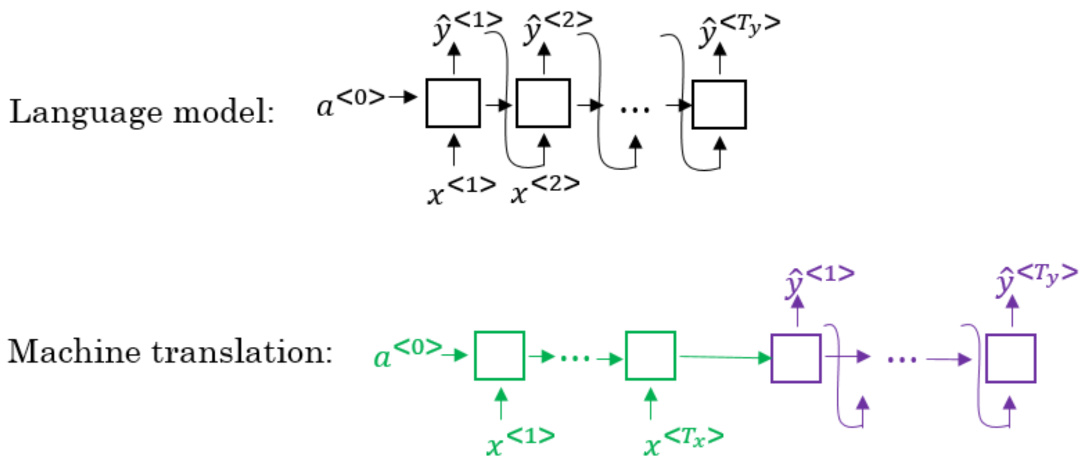


Figure 18: Translation

Machine Translation is similar to language model however instead of always starting with vector of all zeros, it has an encoded network.
For example, for French to English translation:
$P(y^{<1>}, \cdots, y^{<T_y>} | x^{<1>}, \cdots, x^{<T_x>})$ will tell us what is the probability of different English sentences, given a French sentence. Our model should choose the sentence with maximum probability. That is,

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

Why not Greedy Search?
It generates first most likely word according to conditional language model. Then picks whatever is the second word that seems most likely. It continues till entire sentence is completed. It may not be efficient because what we want is pick the entire sequence of words $y^{<1>}, \cdots, y^{<T_y>}$ that maximizes joint probability of that whole thing.

3.1.1 Beam Search Algorithm

This algorithm has a parameter called beam width (B). Suppose B = 3
We run the input French sentence through this encoder network and then first step

will decode the network, and through softmax, output overall 10000 probabilities ($P(y^{<1>}|x)$). Out of 10000 outputs keep in memory which were the top three.
Example: “Jane visite l’Afrique en septembre.”
Y may be:

- Jane is visiting Africa in September.
- Jane is going to be visiting Africa in September.
- In September, Jane will visit Africa.

The three choices for first word are: [in, jane, september]
For each of these 3 choices consider what should be the second word. Take top three values of $P(y^{<2>}|x, \text{'in'})$ and store them in memory. Similarly, do the same for $P(y^{<2>}|x, \text{'jane'})$ and $P(y^{<2>}|x, \text{'september'})$. After the second output we’ll have 9 possible values for the first two words. Take best 3 combinations among them that maximize $P(y^{<1>}|x)P(y^{<2>}|x, \text{'in'})$. Se we’ll have [“in september”, “jane is”, “jane visit”]. Find the next 3 words for “september”, “is” and “visit”. Keep repeating the process for entire sentence.

3.1.2 Refinements to Beam Search

Length Normalization: In beam search we are trying to optimize:

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

$\prod_{t=1}^{T_y} P(y^{<t>}|x, y^{<1>}, \cdots, y^{<t-1>})$ is same as:

$$P(y^{<t>}, \cdots, y^{<t>}|x) = P(y^{<1>}|x)P(y^{<2>}|x, y^{<1>}) \cdots P(y^{<T_y>}|x, y^{<1>}, y^{<2>}, \cdots, y^{<T_y-1>})$$

Probabilities are numbers less than one, multiplying a lot of numbers less than one results in a tiny number, which can cause numerical underflow. To avoid that, we’ll take logarithm of $P(y|x)$. Our objective becomes:

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

However, this function will prefer small sequences rather than long ones. Because long sequences will lead to multiplying more fractions and probability of that sequence will become small. Since our model will select sentences with maximum probabilities, small sequences will be preferred.
To avoid that, we'll scale our function by $1/{T_y}^{\alpha}$. So our objective function becomes:

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

3.1.3 Error Analysis in Beam Search

Beam search is an approximate search algorithm, also called heuristic search algorithm. So we can carry out error analysis to improve our algorithm. e.g.

- Jane visite l'Afrique en Septembre.
- Human: Jane visits Africa in September. (y^*)
- Algorithm: Jane visited Africa last September. $(\hat{y})$

Our model has 2 main components: RNN model and Beam Search
We can attribute this error to one of the 2 components. Increasing the beam width by itself might not get us where we want to go. Most useful thing to do is compute $P(y^*|x)$ and $P(\hat{y}|x)$.

Case I: $P(y^*|x) > P(\hat{y}|x)$
Beam search chose $\hat{y}$. RNN was computing $P(y|x)$ and beam search's job was to try to find a value of y that gives maximum value of our objective function. But $y^*|x$ attains higher $P(y|x)$.
Therefore, Beam search is failing to give us the value of y that maximizes $P(y|x)$.
Conclusion: Beam search is at fault.

Case II: $P(y^*|x) <= P(\hat{y}|x)$
It means that $y^*|x$ is a better translation than $\hat{y}$. But according to RNN $P(y^*|x)$ is less than $P(\hat{y}|x)$. RNN described $y^*|x$ in lower probability than the inferior translation.
Conclusion: RNN is at fault.
For the entire development set, find out who is at fault, what fraction of errors are due to beam search versus the RNN model.
If we find that beam search is responsible for a lot of errors, then maybe increase

beam width.

If RNN is at fault, we you could go to a deeper layer of analysis to try to figure out if we want to add regularization or get more training data or try a different architecture or something else.

3.1.4 BLEU Score

BLEU stands for Bilingual Evaluation Understudy.
Given a French sentence, there could be multiple English translations that are equally good. To evaluate a machine translation system if there are multiple equally good answers, Bleu score is used.
BLEU Score for Unigram:
We'll be finding precision by looking at one word at a time. For example,

- French: Le chat est sur le tapis.
- Reference 1: The cat is on the mat.
- Reference 2: There is a cat on the mat.
- MT output: the the the the the the the ($\hat{y}$)

$$Precision = \frac{2}{7}$$

The precision is defined as a fraction the denominator is count of number of times the word 'the' appears of 7 words in total. And numerator is count of maximum number of times the word 'the' appears in reference.

BLEU Score for Bigram:
We'll be finding precision by looking at two words at a time. For example,

- French: Le chat est sur le tapis.
- Reference 1: The cat is on the mat.
- Reference 2: There is a cat on the mat.
- MT output: The cat the cat on the mat. ($\hat{y}$)

Bigrams	**count**	**count clip**
the cat	2	1
cat the	1	0
cat on	1	1
on the	1	1
the mat	1	1

$$Precision = \frac{sum(count\ clip)}{sum(count)} = \frac{4}{6}$$

Formally, for unigram:

$$p_1 = \frac{\sum_{unigram \in \hat{y}} count_{clip}(unigram)}{\sum_{unigram \in \hat{y}} count(unigram)}$$

For n-gram:

$$p_n = \frac{\sum_{n-gram \in \hat{y}} count_{clip}(n-gram)}{\sum_{n-gram \in \hat{y}} count(n-gram)}$$

If the output is exactly the same as either Reference 1 or Reference 2 then $p_1 = p_2 = \cdots = p_n = 1.0$

Combined BLEU Score:

$$p_n = BP\ exp(\frac{1}{n}\sum_{n=1}^{n} p_n)$$

If the outputs are very short sentences, it's easier to get high precision because most of the words appear in the references.
BP is an adjustment factor that penalizes too short outputs.

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

3.1.5 Attention Model

Intuition

The problem of long sequences

Human translator would read the first part of it, maybe generate part of the translation, look at the second part, generate a few more words, look at a few more words, generate a few more words and so on. You kind of work part by part through the sentence, because it's just really difficult to memorize the whole long sentence like that. What you see for the Encoder-Decoder architecture above is that, it works quite well for short sentences, so we might achieve a relatively high Bleu score, but for very long sentences, maybe longer than 30 or 40 words, the performance comes down.
For long sentences, our model doesn't do well because it's just difficult to get in your network to memorize a super long sentence. So we use attention model which translates a bit more like humans do.

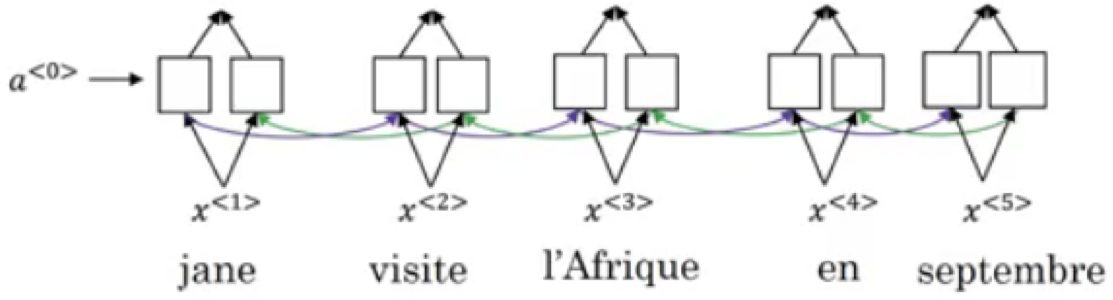


Figure 19: Translation

What we've done is for each word, for each of the five positions into sentence, we can compute a very rich set of features about the words in the sentence and maybe surrounding words in every position.
We're going to use another RNN to generate the English translations.

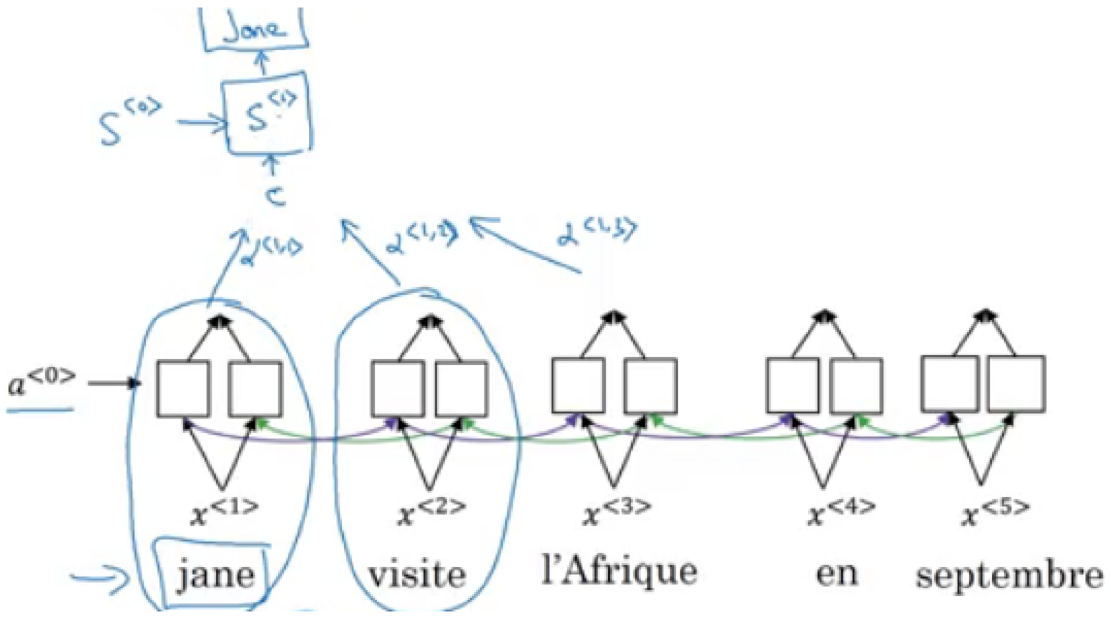


Figure 20: Translation

When we're trying to generate the first word we should be looking primarily at first word (in French sentence) maybe a few other words close by but we don't need to be looking at the end of sentence.
For outputting first word 'Jane', what the attention model would be computing is a set of attention weights $\alpha^{<1,1>}, \alpha^{<1,2>}, \alpha^{<1,3>}$. These attention weights determine how much attention we are paying to each individual input words.

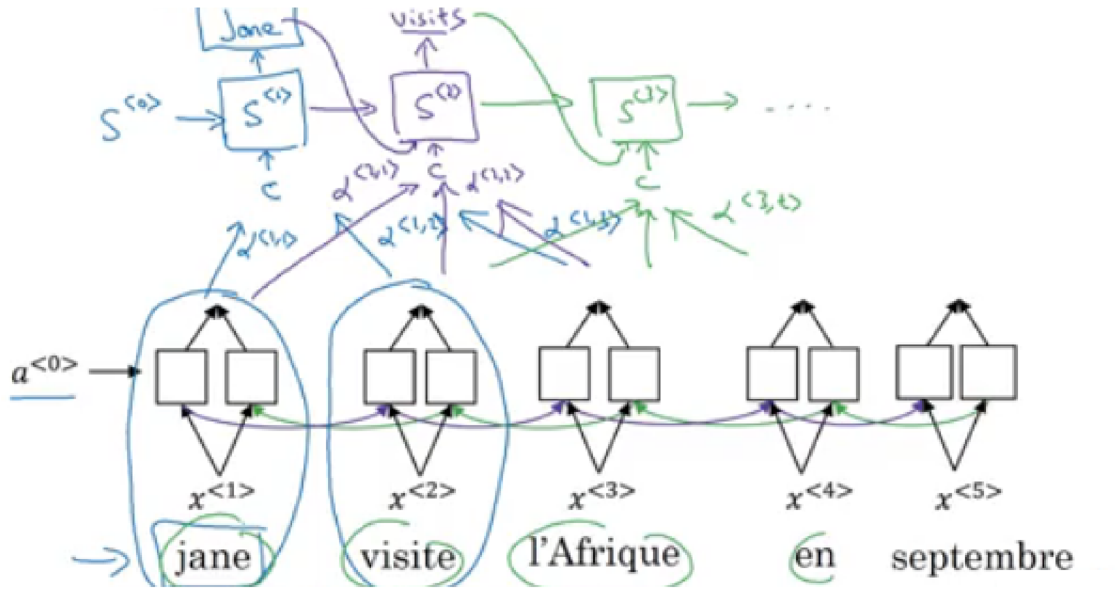


Figure 21: Translation

For the next step, we output the second word. The first word we generate 'Jane' is also an input to this along with some context c. The attention weights will be $\alpha^{<2,1>}, \alpha^{<2,2>}, \alpha^{<2,3>}$.

The single direction RNN with state S. At the first time step it should generate $\hat{y}^{<1>}$, it will have some context c. The weights $\alpha^{<2,1>}, \alpha^{<2,2>}, \alpha^{<2,3>}$ will tell us how much the context would depend on the features we're getting or the activations we're getting from different time steps. Sum of attention weight for each element: $\sum \alpha^{<1,t'>} = 1$
The way we define context is a weighted sum of features from the different time steps weighted by these attention weights.

$$c^{<1>} = \sum_{t'} \alpha^{<1,t'>} a^{t'}$$

where, $a^{t'} = (\overrightarrow{a}^{<t'>}, \overleftarrow{a}^{<t'>})$ Computing attention $\alpha^{<t,t'>}$:
$\alpha^{<t,t'>}$ = amount of attention $y^{<t>}$ should pay to $a^{<t'>}$

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

For every value of t, the weights sum to 1. To compute $e^{<t,t'>}$ we use a small NN as follows:

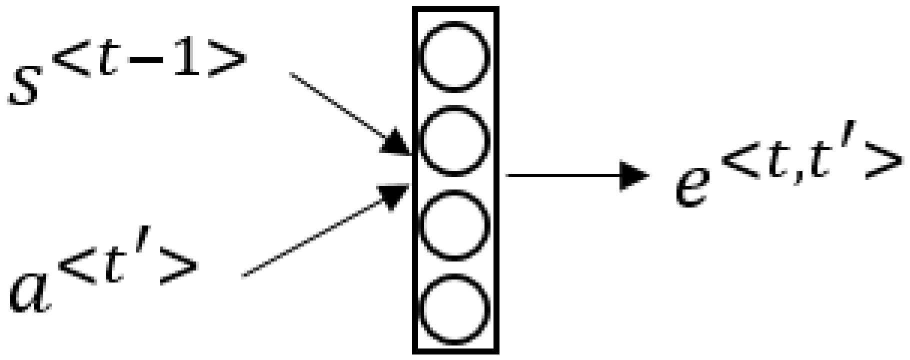


Figure 22: Small NN of one layer

$s^{<t-1>}$ is the neural network state from previous time step.
One downside of this algorithm is that it runs in quadratic time.

3.2 Speech Recognition - Audio Data

3.2.1 Speech Recognition Problem

We're given an audio clip, X, and our job is to automatically find a text transcript, Y. Most common preprocessing step in audio data is to run our audio clip and generate a spectrogram.
An audio recording is a long list of numbers measuring the little air pressure changes detected by the microphone and we hear voice because our ear detects little changes in air pressure.

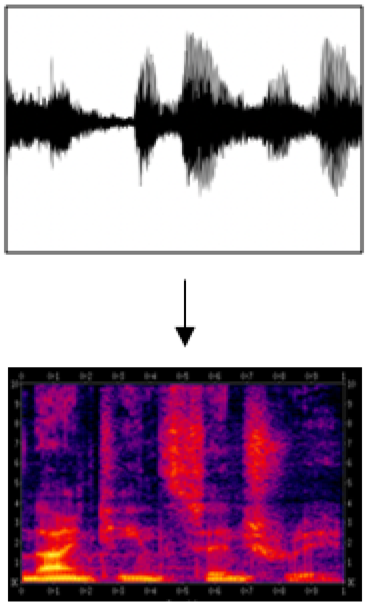


Figure 23: Plot of an audio clip

Vertical axis is frequency and Horizontal axis is time and intensity of different colors show the amount of energy.

CTC Cost for Speech Recognition:

CTC stands for Connectionist Temporal Classification. It is one of the models that work well on speech recognition.
Let's say the audio clip is someone saying: "the quick brown fox"
We're going to use a NN structured like below with an equal number of input and output. In speech recognition usually the number of time steps is much bigger than the number of output time steps.

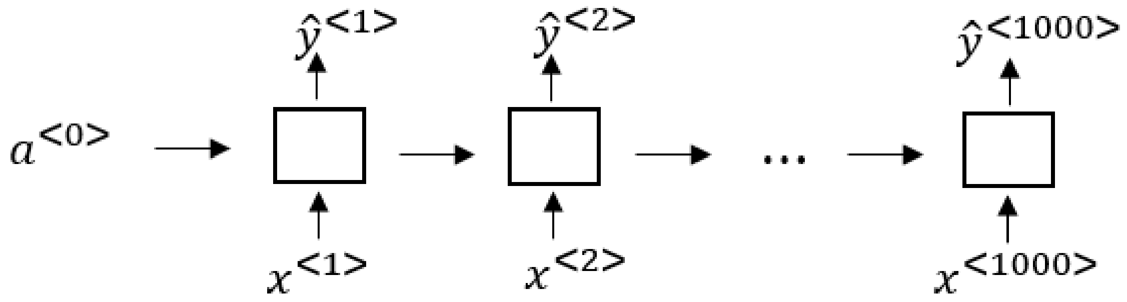


Figure 24: Plot of an audio clip

If we have 10 seconds of audio and our feature came at 100 Hz(100 sample/s) then a 10 second audio clip would end up with 1000 inputs. But output might not have a thousand characters.
The CTC cost function allows the RNN to generate and output like
ttt_h_eee___ <SPC> ___qqq__
which covers "the q". _ is a special character "blank".
And this is considered a correct output for first parts of space.
Basic Rule of CTC: Collapse repeated characters not separated by blank.
This allows our network to have a 1000 output by repeating character multiple times. So by inserting a bunch of blank characters it still ends up with a much shorter output text transcript.
The phrase "the quick brown fox" including spaces actually has 19 characters, by allowing the network to insert blanks and repeated characters it can still represent this 19 character output with 1000 values of Y.

3.2.2 Trigger word detection

Examples of trigger word systems include the Amazon Echo, which is woken up with the word *Alexa*, the Baidu DuerOS powered devices woken up with the phrase *xiaodunihao*, Apple Siri woken up with *Hey, Siri*, and Google Home woken up with *OK, Google.*
And so all that remains to be done is to define the target labels, Y.

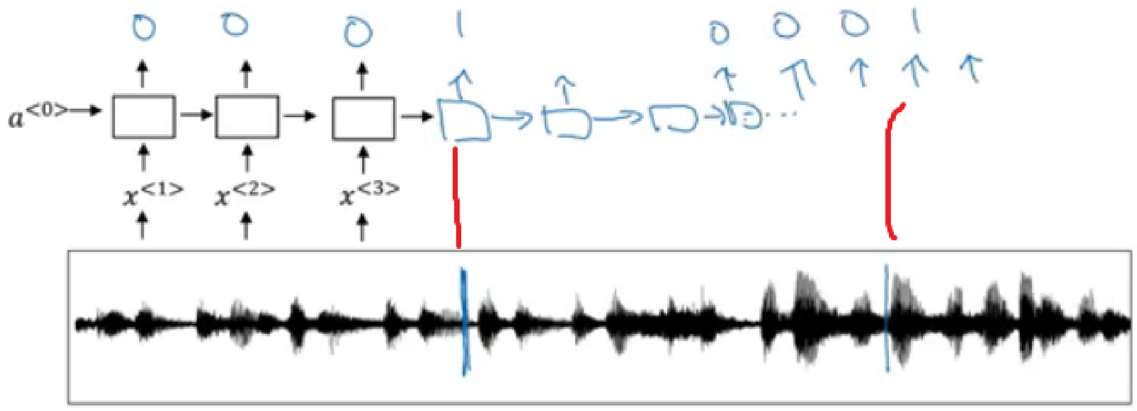


Figure 25: Plot of an audio clip

If the blue line in the audio clip is when someone just finished saying the trigger word, such as Alexa or xiaodunihao or hey, Siri, or okay, Google, then in the training sets, we can set the target labels to be 0 for everything before that point and right after that set the target label to 1. Then if a little bit later on, the trigger word was said again and the trigger word was said at this point, then you can again set the target label to be 1 right after that.
One slight disadvantage of this is it creates a very imbalanced training set to have a lot more 0s than 1s. So one other thing we could do, can make the model a little bit easier to train is instead of setting only a single time step output 1, we can make it output a few 1s for several times or for a fixed period of time before reverting back to 0. So that slightly evens out the ratio of 1s to 0s.

4 Transformers

4.1 Motivation

While GRU and LSTM improved control over the flow of information they also increased complexity. The major innovaton of transformer architecture is combining the use of attention based representations and a CNN based style of processing.

And for all of these models to compute the output of final unit, we first have to compute the outputs of all of the units that come before. The transformer architecture allows us to run a lot more of these computations for an entire sequence in parallel. So we can ingest an entire sentence all at the same time, rather than just processing it one word at a time from left to right. The major innovation of the transformer architecture is combining the use of attention based representations and a convolutional neural network style of processing. An RNN is a very sequential way of processing tokens, and we might contrast this with a CNN that can take input a lot of pixels or maybe a lot of words and can compute representations for them in parallel.

Two key ideas are required to understand the attention network.
The first is **self attention**. The goal of self attention is, if we have, say, a sentence of five words it will end up computing five representations for these five words, $A^{<1>}$, $A^{<2>}$, $A^{<3>}$, $A^{<4>}$ and $A^{<5>}$. And this will be an attention based way of computing representations for all the words in your sentence in parallel.
Then **multi headed attention** is basically for loop over the self attention process. So we end up with multiple versions of these representations. And it turns out that these representations, which will be very rich representations, can be used for machine translation or other NLP tasks to great effectiveness.

4.2 Self Attention

We create attention based vector representations for each of the words in the input sentence.
$A(q, k, v)$ = attention based vector representation of a word.

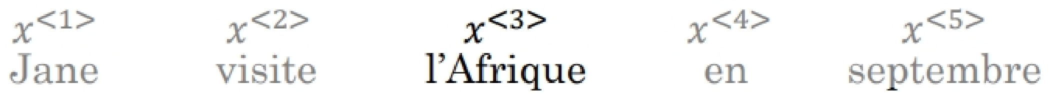


Figure 26: Representation

It will look a surrounding words to figure out how we're talking about 'Africa' in this sentence, and find the most appropriate representation for this.

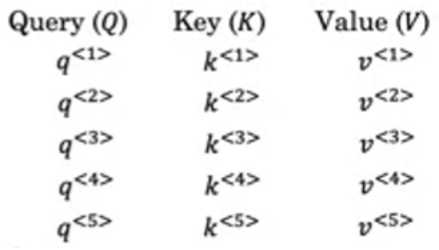
We'll compute these representations in parallel for all five words in a sentence.
With the self attention mechanism the used is:

$$A(q, k, v) = \sum_i \frac{exp(q \cdot k^{<i>})}{\sum_j exp(q \cdot k^{<j>})} v^{<i>}$$

For every word we have 3 values: the query(q), key(k) and value(v). These values are inputs to computing the attention value for each words. $q^{<3>}$ is computed as a learned matrix:

$$q^{<3>} = W^Q x^{<3>}$$
$$k^{<3>} = W^k x^{<3>}$$
$$v^{<3>} = W^v x^{<3>}$$

Matrices W^Q, W^k, and W^v are parameters to our learning algorithm.



Query (Q)	Key (K)	Value (V)
$q^{<1>}$	$k^{<1>}$	$v^{<1>}$
$q^{<2>}$	$k^{<2>}$	$v^{<2>}$
$q^{<3>}$	$k^{<3>}$	$v^{<3>}$
$q^{<4>}$	$k^{<4>}$	$v^{<4>}$
$q^{<5>}$	$k^{<5>}$	$v^{<5>}$

Figure 27:

We then calculate the inner product between $q^{<3>}$ and $k^{<i>}$, where $i = 1, \cdots, 5$
The goal is to pull up the most information that's needed to help us compute the most useful representation $A^{<3>}$
$k^{<1>}$ represents that Jane is a person and $k^{<2>}$ represents that visite is an action.

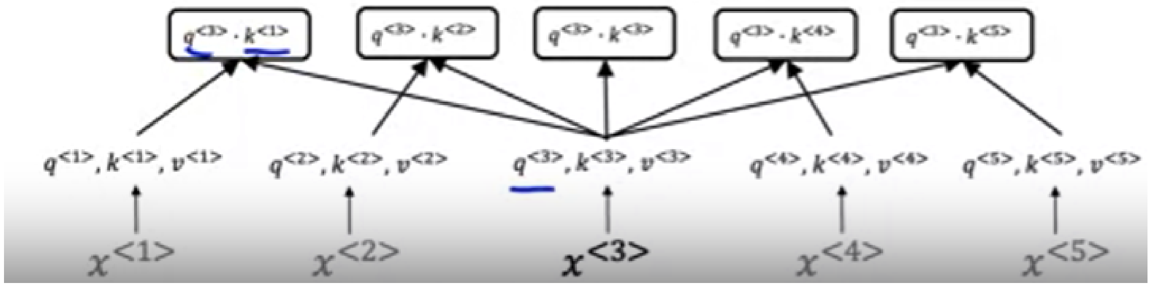


Figure 28:

We may find that $q^{<3>} \cdot k^{<2>}$ has highest value of inner products. This suggests that 'visite' gives us most relevant contents to $q^{<3>}$ in Africa, which is that it's viewed as a destination for a visit.

We'll compute softmax over these 5 values. Then take the 5 softmax values and multiply them with $v^{<1>}, \cdots, v^{<5>}$ respectively. Finally we sum it all up to get $A^{<3>}$.

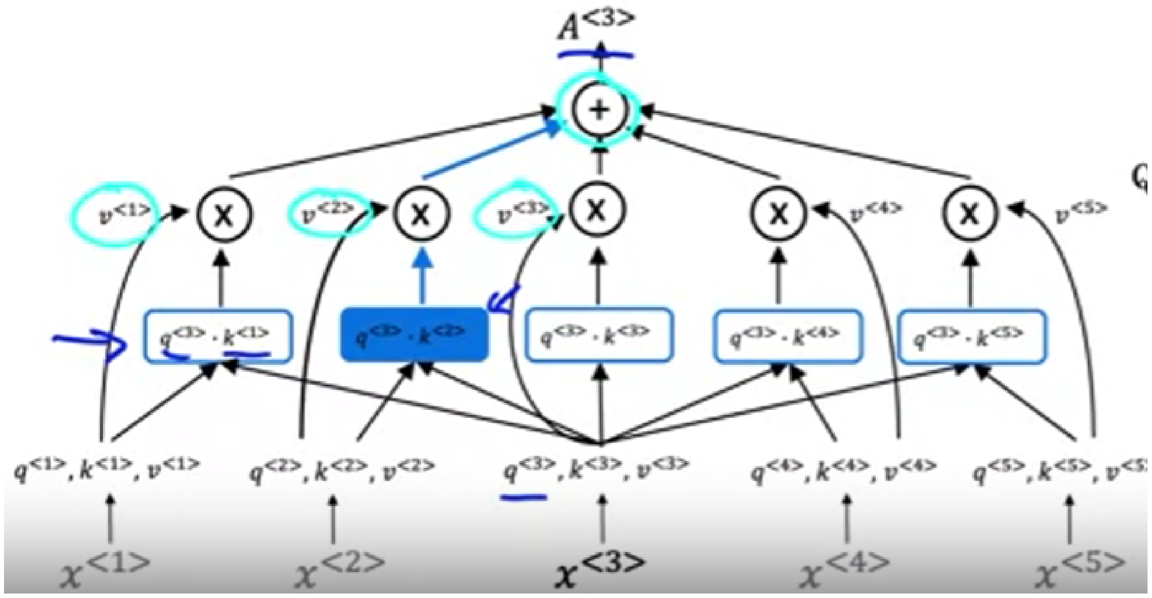


Figure 29:

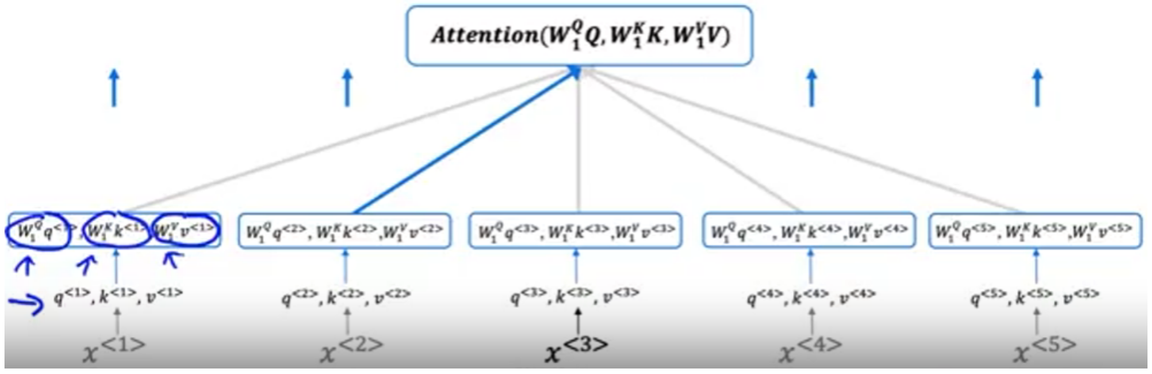
If we put all of these 5 computations together, the notation used is:

$$Attention(Q, k, v) = softmax(\frac{Qk^{\intercal}}{\sqrt{d_k}})v$$

where, $\sqrt{d_k}$ is to scale the dot product to avoid it from exploding.

4.3 Multi Head Attention

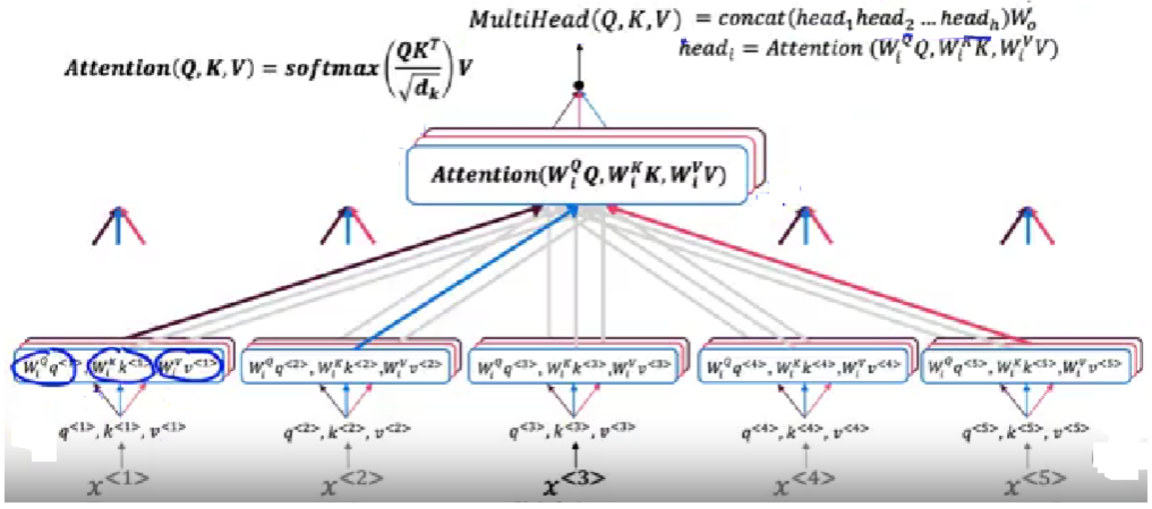
The notation gets a little bit complicated, but the thing to keep in mind is it's basically just a big for-loop over the self attention mechanism.
Each time we calculate the value of self attention it's called a head. With multi head we take the same set of query, key and value vectors as inputs & calculate multiple self attentions.



$Attention(W_1^Q Q, W_1^K K, W_1^V V)$
$W_1^Q q^{<1>}, W_1^K k^{<1>}, W_1^V v^{<1>}$
$W_1^Q q^{<2>}, W_1^K k^{<2>}, W_1^V v^{<2>}$
$W_1^Q q^{<3>}, W_1^K k^{<3>}, W_1^V v^{<3>}$
$W_1^Q q^{<4>}, W_1^K k^{<4>}, W_1^V v^{<4>}$
$W_1^Q q^{<5>}, W_1^K k^{<5>}, W_1^V v^{<5>}$
$q^{<1>}, k^{<1>}, v^{<1>}$
$q^{<2>}, k^{<2>}, v^{<2>}$
$q^{<3>}, k^{<3>}, v^{<3>}$
$q^{<4>}, k^{<4>}, v^{<4>}$
$q^{<5>}, k^{<5>}, v^{<5>}$
$x^{<1>}$
$x^{<2>}$
$x^{<3>}$
$x^{<4>}$
$x^{<5>}$

Figure 30:

We multiply the k,q,v matrices with weight matrices W_Q^1, W_k^1, W_v^1 and so these 3 values give you a new set of query, key and value vectors for first word. And so with this computation, the word visite gives the best answer to context in which $x^{<3>}$ is used.
This is how we get the representation for l'Afrique. Do the same for Jane, visite, en and septembre. So we end up with 5 vectors to represent the 5 words in the sequence. So this is a computation we carry out for the first of the several heads we use in multi head.



$MultiHead(Q,K,V) = concat(head_1 head_2 \dots head_h)W_o$
$head_i = Attention(W_i^Q Q, W_i^K K, W_i^V V)$
$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
$Attention(W_i^Q Q, W_i^K K, W_i^V V)$
$W_i^Q q^{<1>}, W_i^K k^{<1>}, W_i^V v^{<1>}$
$W_i^Q q^{<2>}, W_i^K k^{<2>}, W_i^V v^{<2>}$
$W_i^Q q^{<3>}, W_i^K k^{<3>}, W_i^V v^{<3>}$
$W_i^Q q^{<4>}, W_i^K k^{<4>}, W_i^V v^{<4>}$
$W_i^Q q^{<5>}, W_i^K k^{<5>}, W_i^V v^{<5>}$
$q^{<1>}, k^{<1>}, v^{<1>}$
$q^{<2>}, k^{<2>}, v^{<2>}$
$q^{<3>}, k^{<3>}, v^{<3>}$
$q^{<4>}, k^{<4>}, v^{<4>}$
$q^{<5>}, k^{<5>}, v^{<5>}$
$x^{<1>}$
$x^{<2>}$
$x^{<3>}$
$x^{<4>}$
$x^{<5>}$

Figure 31:

For the other words step through exactly the same calculation that we had just now for l'Afrique. The attention equation is same as:

$$A(q,k,v) = \sum_i \frac{exp(q \cdot k^{<i>})}{\sum_j exp(q \cdot k^{<j>})} v^{<i>}$$

Now do this computation with the second head. It will have new set of weight matrices W_Q^2, W_k^2, W_v^2. May be the inner product between septembre query and l'Afrique query will have highest value. So the final value is concatenation of all these h heads multiplied by a matrix W_o. In practice we can actually compute these different head values in parallel because one head values does not depend on the other.

$$head_i = Attention(W_i^Q Q, W_i^k k, W_i^v v)$$
$$Multihead(Q,k,v) = concat(head_1 head_2 head_3 \cdots head_n)W_o$$

4.4 Transformer Network

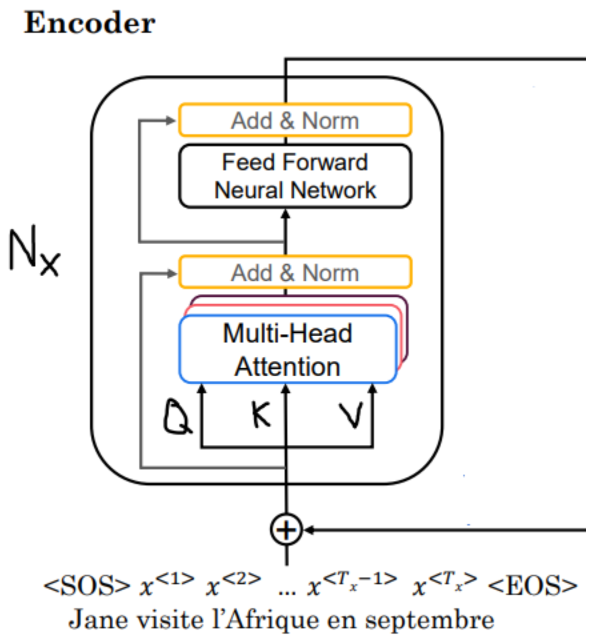

Encoder
Add & Norm
Feed Forward Neural Network
Nx
Add & Norm
Multi-Head Attention
Q
K
V
<SOS> $x^{<1>}$ $x^{<2>}$ ... $x^{<T_x-1>}$ $x^{<T_x>}$ <EOS>
Jane visite l'Afrique en septembre

Figure 32:

The first step in the transformer is, these embeddings get fed into an encoder block which has a multi-head attention layer. Where we feed in the values Q, K and V computed from the embeddings and the weight matrices W. This layer then produces a matrix that can be passed into a feed-forward neural network which helps determine what interesting features there are in the sentence. In the transformer paper, this encoding block is repeated n times and a typical value for n is six. After maybe about six times through this block, we will then feed the output of the encoder into a decoder block.

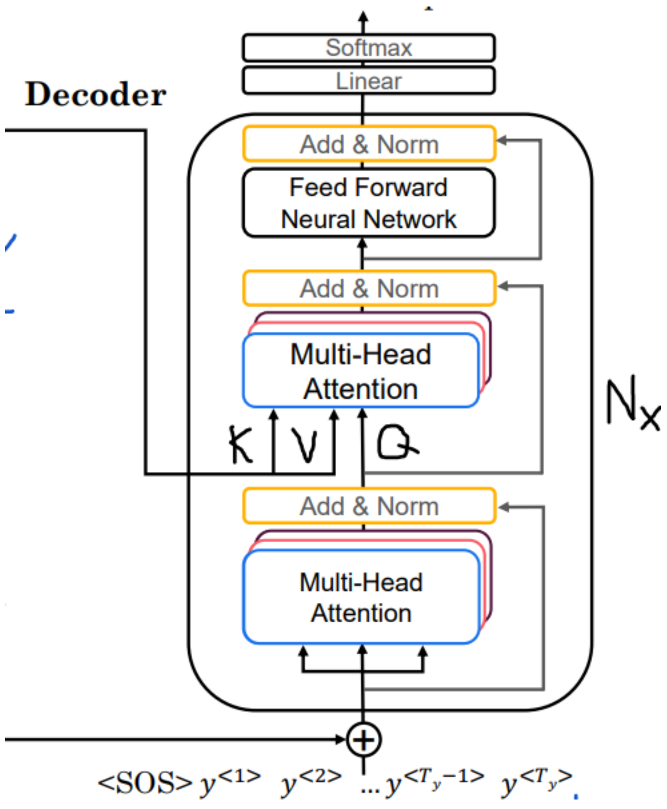


Figure 33:

The decoders block's job is to output the English translation. The first output will be the start of sentence token. At every step, the decoder block will input the first few words, whatever we've already generated of the translation. When we're just getting started, the only thing we know is that the translation will start with a start of sentence token($< SOS >$). The start of sentence token gets fed in to this multi-head attention block and just this one token, the $< SOS >$ token, is used to compute Q, K and V for this multi-head attention block.

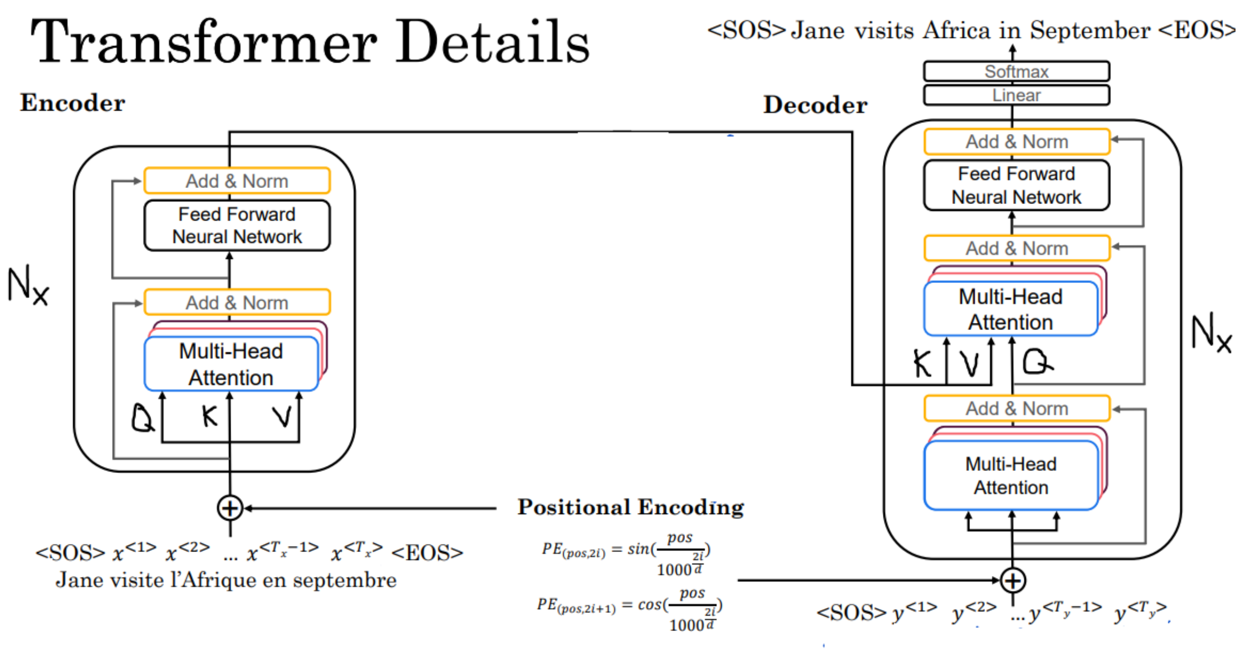

Transformer Details
Encoder
Decoder
<SOS> Jane visits Africa in September <EOS>
Softmax
Linear
Add & Norm
Feed Forward Neural Network
Multi-Head Attention
Q K V
K V Q
Nx
Positional Encoding
$PE_{(pos,2i)} = sin(\frac{pos}{1000^{\frac{2i}{d}}})$
$PE_{(pos,2i+1)} = cos(\frac{pos}{1000^{\frac{2i}{d}}})$
<SOS> $x^{<1>}$ $x^{<2>}$ ... $x^{<T_x-1>}$ $x^{<T_x>}$ <EOS>
Jane visite l'Afrique en septembre
<SOS> $y^{<1>}$ $y^{<2>}$ ... $y^{<T_y-1>}$ $y^{<T_y>}$

Figure 34:

This first block's output is used to generate the Q matrix for the next multi-head attention block and the output of the encoder is used to generate K and V. The second multi-head attention block input Q, K and V as before. It is structured this way because the input down here is what we've translated of the sentence so far. It will then pull context from K and V, which is translated from the French version of the sentence to then try to decide what is the next word in the sequence to generate. To finish the description of the decoded block, the multi-head attention block outputs the values which are fed to a feed forward neural network. This decoder block is also going to be repeated n times, maybe six times, where we take the output, feed it back to the input, and have this go through, say, half a dozen times.
The job of this neural network is to predict the next word in the sentence. Hopefully, it will decide that the first word in the English translation is Jane. What we do is then feed Jane to the input as well. Let's find the right key and the right value, then lets us generate the most appropriate next word, which hopefully will generate visite.
Then running this neural network again generates 'Africa'. Then we feed Africa back

into the input. It then generates ‘in’ and then ‘September’, and with this input, it generates $< EOS >$ and then we’re done.
These encoder and decoder blocks, and how they’re combined to perform a sequence to sequence translation tasks are the main ideas behind the transformer architecture. But beyond these main ideas, there are a few things.
The first of these is positional encoding of the input. If we recall the self attention equations, there’s nothing that indicates the position of a word. But the position within a sentence can be extremely important to translation. The way we encode the position of elements in the input is that we use a combination of these sine and cosine equations. Let’s say, for example, that our word embedding is a vector with four values. In this case, the dimension d of the word embedding is 4, so $x^{<1>}, x^{<2>}, x^{<3>}, x^{<4>}$, let’s say those are four dimensional vectors. In this example, we’re going to then create a positional embedding vector of the same dimension, also four dimensional. $p^{<1>}$ is for the position embedding of the first word Jane.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{1000^{2i/d}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{1000^{2i/d}}\right)$$

In this equation above, pos denotes the numerical position of the word. For the word Jane, pos = 1, and i over here refers to the different dimensions of encoding. The first element corresponds to i = 0. Variables pos and i, go into these equations. i goes from 0 to 1, and d = 4, is the dimension of this vector. What the position encoding does with the sine and cosine is create a unique positional encoding vector. One of these vectors that is unique for each word. The vector $p^{<3>}$ that encodes the position of ‘l’Afrique’, the third word will be a set of four values that’ll be different than the four values used in code position of the first word of ‘Jane’.

4.4.1 Transformer Application: Question Answering

Extractive Question Answering

Question Answering (QA) is a task of natural language processing that aims to automatically answer questions. The goal of extractive QA is to identify the portion of the text that contains the answer to a question. For example, when tasked with answering the question ‘When will Jane go to Africa?’ given the text data ‘Jane visits Africa in September’, the question answering model will highlight ‘September’. We’ll be using the bAbl datasets, which is one of the bAbI datasets generated by Facebook AI Research. Following is the code for the same:

Load dataset

```
from datasets import load_from_disk
babi_dataset = load_from_disk(‘data/’)
```

Data Preprocessing

```
type_set = set()
for story in babi_dataset['train']:
    if str(story['story']['type'] )not in type_set:
        type_set.add(str(story['story'][‘type'] ))

flattened_babi = babi_dataset.flatten()

def get_question_and_facts(story):
    dic = {}
    dic['question'] = story['story.text'][2]
    dic['sentences'] = ' '.join([story['story.text'][0], story['story.text'][1]])
    dic['answer'] = story['story.answer'][2]
    return dic

processed = flattened_babi.map(get_question_and_facts)

def get_start_end_idx(story):
    str_idx = story['sentences'].find(story['answer'])
    end_idx = str_idx + len(story['answer'])
    return {'str_idx':str_idx,
          'end_idx': end_idx}

processed = processed.map(get_start_end_idx)
```

Tokenize and Align with Hugging Face Library's DistilBERT fast tokenizer

To feed text data to a Transformer model, we will need to tokenize our input using a Transformer tokenizer. It is crucial that the tokenizer we use must match the Transformer model type we are using. In this exercise, we will use the DistilBERT fast tokenizer, which standardizes the length of our sequence to 512 and pads with zeros.

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('tokenizer/')
```

We align the start and end indices with the tokens associated with the target answer word with the below function:

```
def tokenize_align(example):
    encoding = tokenizer(example['sentences'], example['question'],
     truncation=True, padding=True, max_length=tokenizer.model_max_length)
    start_positions = encoding.char_to_token(example['str_idx'])
    end_positions = encoding.char_to_token(example['end_idx']-1)
    if start_positions is None:
        start_positions = tokenizer.model_max_length
    if end_positions is None:
        end_positions = tokenizer.model_max_length
    return {'input_ids': encoding['input_ids'],
          'attention_mask': encoding['attention_mask'],
          'start_positions': start_positions,
          'end_positions': end_positions}

qa_dataset = processed.map(tokenize_align)

qa_dataset = qa_dataset.remove_columns(['story.answer', 'story.id',
'story.supporting_ids', 'story.text', 'story.type'])
```

Training

```
train_ds = qa_dataset['train']
test_ds = qa_dataset['test']

from transformers import TFDistilBertForQuestionAnswering
model = TFDistilBertForQuestionAnswering.from_pretrained("model/tensorflow",
return_dict=False)

import tensorflow as tf

columns_to_return = ['input_ids','attention_mask', 'start_positions',
'end_positions']

train_ds.set_format(type='tf', columns=columns_to_return)
```

```
train_features = {x: train_ds[x] for x in ['input_ids', 'attention_mask']}
train_labels = {"start_positions": tf.reshape(train_ds['start_positions'],
shape=[-1,1]), 'end_positions': tf.reshape(train_ds['end_positions'],
shape=[-1,1])}


train_tfdataset = tf.data.Dataset.from_tensor_slices((train_features,
train_labels)).batch(8)

EPOCHS = 3
loss_fn1 = tf.keras.losses.SparseCategoricalCrossentropy( from_logits=True)
loss_fn2 = tf.keras.losses.SparseCategoricalCrossentropy( from_logits=True)
opt = tf.keras.optimizers.Adam(learning_rate=3e-5)

losses = []
for epoch in range(EPOCHS):
    print("Starting epoch: %d"% epoch )
    for step, (x_batch_train, y_batch_train) in enumerate(train_tfdataset):
        with tf.GradientTape() as tape:
            answer_start_scores, answer_end_scores = model(x_batch_train)
            loss_start = loss_fn1(y_batch_train['start_positions'],
            answer_start_scores)
            loss_end = loss_fn2(y_batch_train['end_positions'],
            answer_end_scores)
            loss = 0.5 * (loss_start + loss_end)
        losses.append(loss)
        grads = tape.gradient(loss, model.trainable_weights)
        opt.apply_gradients(zip(grads, model.trainable_weights))

        if step % 20 == 0:
            print("Training loss (for one batch) at step %d: %.4f"(step,
            float(loss_start)))


import matplotlib.pyplot as plt
plt.plot(losses)
plt.show()
```

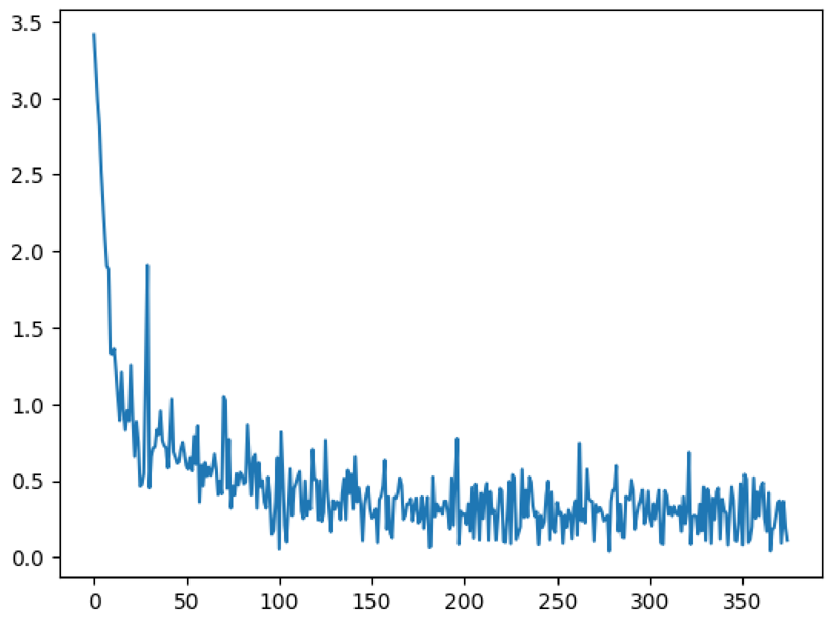


Figure 35:

NLP Revised POA

Deadlines:

- 27th May - Basic RNN and types
- 3rd June - Language Model, GRU, LSTM, Backpropagation
- 12th June - Word Embeddings, BiRNN
- 22nd June - Deep RNN, Sentiment classification and de-biasing
- 25th June - Midterm report submission
- 6th July - Self Attention, Multi-head attention
- 16th July - Transformers Network
- 21st July - Question Answering Model

Resources:

- Coursera Specialization on Natural Language Processing
- GloVe: Global Vectors for Word Representation
- Hugging Face Transformers Tokenizer Documentation
- DistilBERT Fast Tokenizer Documentation