Recurrent Neural Networks

M. Şafak Bilici
safak@inzva.com

Contents

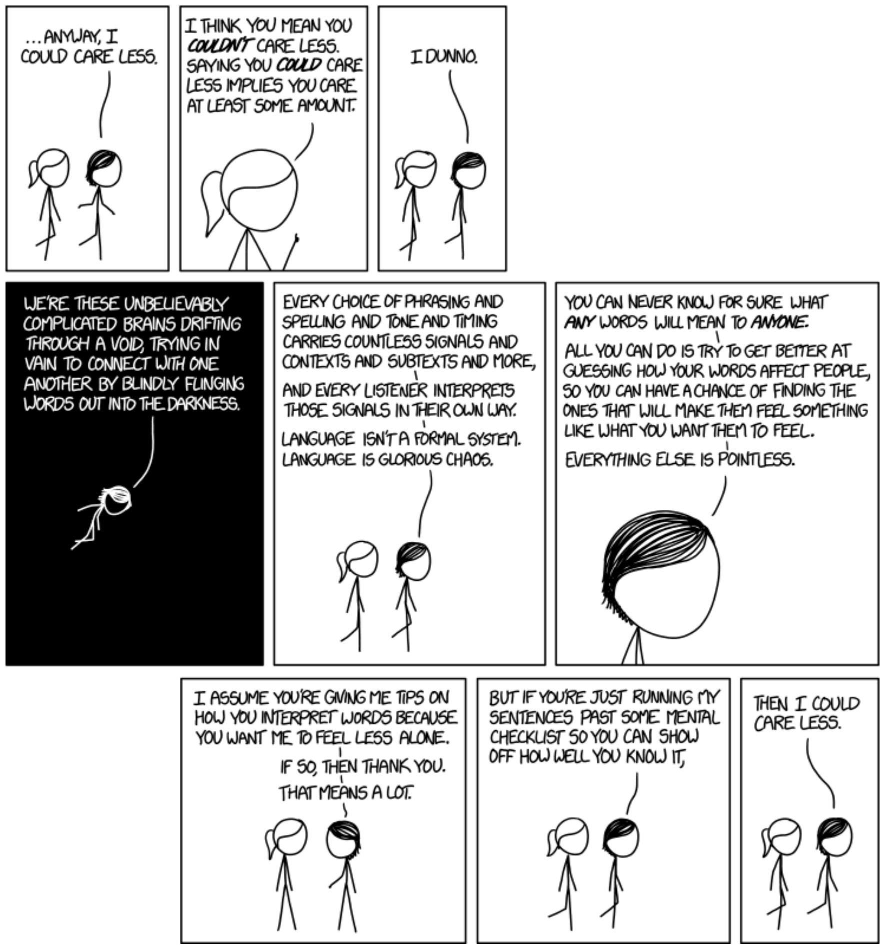

...ANYWAY, I COULD CARE LESS.
I THINK YOU MEAN YOU COULDN'T CARE LESS. SAYING YOU COULD CARE LESS IMPLIES YOU CARE AT LEAST SOME AMOUNT.
I DUNNO.
WE'RE THESE UNBELIEVABLY COMPLICATED BRAINS DRIFTING THROUGH A VOID, TRYING IN VAIN TO CONNECT WITH ONE ANOTHER BY BLINDLY FLINGING WORDS OUT INTO THE DARKNESS.
EVERY CHOICE OF PHRASING AND SPELLING AND TONE AND TIMING CARRIES COUNTLESS SIGNALS AND CONTEXTS AND SUBTEXTS AND MORE,
AND EVERY LISTENER INTERPRETS THOSE SIGNALS IN THEIR OWN WAY.
LANGUAGE ISN'T A FORMAL SYSTEM. LANGUAGE IS GLORIOUS CHAOS.
YOU CAN NEVER KNOW FOR SURE WHAT ANY WORDS WILL MEAN TO ANYONE.
ALL YOU CAN DO IS TRY TO GET BETTER AT GUESSING HOW YOUR WORDS AFFECT PEOPLE, SO YOU CAN HAVE A CHANCE OF FINDING THE ONES THAT WILL MAKE THEM FEEL SOMETHING LIKE WHAT YOU WANT THEM TO FEEL.
EVERYTHING ELSE IS POINTLESS.
I ASSUME YOU'RE GIVING ME TIPS ON HOW YOU INTERPRET WORDS BECAUSE YOU WANT ME TO FEEL LESS ALONE.
IF SO, THEN THANK YOU.
THAT MEANS A LOT.
BUT IF YOU'RE JUST RUNNING MY SENTENCES PAST SOME MENTAL CHECKLIST SO YOU CAN SHOW OFF HOW WELL YOU KNOW IT,
THEN I COULD CARE LESS.

Figure 1: https://xkcd.com/1576/

1 Motivation

Why Recurrent Neural Networks (RNNs)? If you have studied theoretical and practical aspects of Fully Connected Neural Networks, answer of this question may be intuitive for you. RNNs are a family of neural networks for processing and modeling "sequential" data. So, what is sequential?

- **Biological Data**: A biological data may have sequential structure, for example a single, continuous molecule of nucleic acid or protein.
- **Time Series**: A Time Series application can have specific practical case, for example Financial Forecasting. The ratio between income and expense can be modeled by time. The ratio at 3th month will affect the ratio at 5th month.
- **Language**: Language is position dependent. The word at position i may related with word j. Besides, its recursive and unbounded structure pushes us to use a sequential model like RNNs.

However; we will make our experiments, evaluations and examples in language perspective, mostly. Before introducing, let's explaing why not Fully Connected Networks. Take an input sentence, a traditional fully connected Fully Connected network would have separate parameters for each input feature, so it would need to learn all of the rules of the language separately at each position in the sentence [1].

2 Introducing Recurrent Neural Networks

Recurrent Neural Network is a function (f), which can be written recursively. Take and input $\mathbf{x}$, which can be decomposed as $\mathbf{x} : \{x_i, i \in 1, 2, ..., t\}$. Each x_i represents input element at time i. Introducing our function parameters as $\boldsymbol{\theta} \in \Theta$, our function can be written as

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}; \boldsymbol{\theta}) \tag{1}$$

where $\mathbf{h}^{(t)}$ is the hidden state of our function. If the input set is $\mathbf{x} : \{x_i, i \in 1, 2, 4\}$, the introduced function can be written as

$$\mathbf{h}^{(4)} = f(\mathbf{h}^{(3)}; \boldsymbol{\theta}) \tag{2}$$

$$= f(f(\mathbf{h}^{(2)}; \boldsymbol{\theta}); \boldsymbol{\theta})) \tag{3}$$

$$= f(f(f(\mathbf{h}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})); \boldsymbol{\theta}) \tag{4}$$

Same formulation is applied for all finite length inputs. The formulation tells us that, each hidden state can be written as previous hidden states with same parameters. This allows us to model sequential data. Let's illustrate this process. As seen in Figure 1, at time step 1, $x^{(1)}$ is passed to

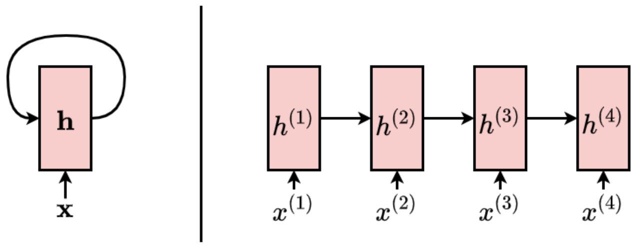


Figure 2: Left: Folded representation of RNN — Right: Unfolded representation of RNN.

a black block, it is processed and passed to time step 2. This time $x^{(2)}$ is processed with $h^{(2)}$, and with same parameters. This "prior" information from previous step allows us to model "sequential" structure of $\mathbf{x}$. At last, we process $x^{(4)}$ with given informations $x^{(1)} \rightarrow h^{(1)}, x^{(2)} \rightarrow h^{(2)}, x^{(3)} \rightarrow h^{(3)}$. Besides, this "shared parameter" prodecure is practical for processing variable length input with same architecture: for example $t = 5, 6, 7$ for $\mathbf{x} : \{x_i, i \in 1, 2, ..., t\}$. We will talk about this in later chapters.

3 Background: Representing Text

Before more on RNNs, it is significant to learn "how to represent text in computer" in simplest way. Normally, a text is a string. We have to represent it numerically to use in neural networks.

3.1 One-Hot Encoding

One-hot encoding is the most common, most basic way to turn a token into a vector. It consists in associating a unique integer index to every word, then turning this integer index i into a binary vector of size $\mid V \mid$, which is the size of our vocabulary, that would be all-zeros except for the i-th entry, which would be 1. For example, if we have an vocabulary V: {"i": 0, "we": 1, "go": 2, "school": 3}, the word "we" becomes

$$\begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \in \mathbb{N}^4 \tag{5}$$

and the sentence "i go school" becomes

$$\{\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 3 \end{bmatrix}\} \in \mathbb{N}^{(3\times4)} \tag{6}$$

4 Formulating Recurrent Neural Networks

We studied Recurrent Neural Networks as a mathematical function and representing text numerically. Now, we are going to write down the forward equation of the RNNs.

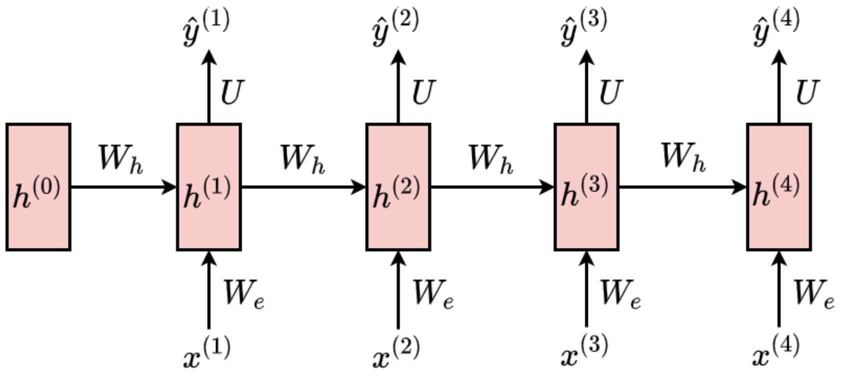


Figure 3: Representing forward equation of RNNs.

Figure 2 shows the forward phase of the RNNs. Let's write the algorithm with the dimensions of inputs, outputs and parameters.

Algorithm 1 Forward Algorithm for Recurrent Neural Network

```
Require: $\mathbf{x}, h^{(0)}, \{W_e, W_h, U\}, t$        ▷ $h^{(0)}$ can be learned or initialized with all-zeros.
1: for $i \in \{1, 2, 3, ..., t\}$ do
2:     $h^{(t)} \leftarrow \sigma(W_h \cdot h^{(t-1)} + W_e \cdot x^{(t)} + b_1)$        ▷ $x^{(t)} \in \mathbb{R}^{1\times|V|}, W_e \in \mathbb{R}^{d_h\times 1}, W_h \in \mathbb{R}^{d_h\times d_h}$
3:     $\hat{y}^{(t)} \leftarrow softmax(U \cdot h^{(t)} + b_2)$        ▷ $U \in \mathbb{R}^{1\times d_h}, \hat{y}^{(t)} \in \mathbb{R}^{1\times|V|}$
4: end for
```

4.1 Learning Embeddings Simultaneously

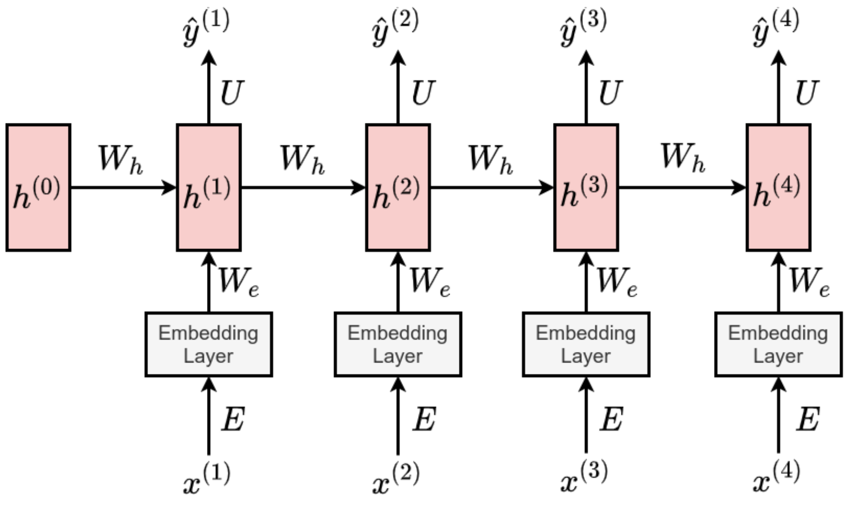


Figure 4: Dense embeddings with learning.

Not scope of the chapter, but word representations can be learned by Embedding Layer, which is a trainable matrix. It contains better representations than a single one-hot. This embeddings can be used as pretrained, or can be learned simultaneously. Let's write our equations again.

Algorithm 2 Forward Algorithm for Recurrent Neural Network (Embedding Layer)

```
Require: $\mathbf{x}, h^{(0)}, \{W_e, W_h, U, E\}, t$        ▷ $h^{(0)}$ can be learned or initialized with all-zeros.
1: for $i \in \{1, 2, 3, ..., t\}$ do
2:     $e^{(t)} \leftarrow E \cdot x^{(t)}$        ▷ $x^{(t)} \in \mathbb{R}^{1\times|V|}, E \in \mathbb{R}^{D\times 1}, e^{(t)} \in \mathbb{R}^{D\times|V|}$
3:     $h^{(t)} \leftarrow \sigma(W_h \cdot h^{(t-1)} + W_e \cdot e^{(t)} + b_1)$        ▷ $W_e \in \mathbb{R}^{d_h\times D}, W_h \in \mathbb{R}^{d_h\times d_h}$
4:     $\hat{y}^{(t)} \leftarrow softmax(U \cdot h^{(t)} + b_2)$        ▷ $U \in \mathbb{R}^{1\times d_h}, \hat{y}^{(t)} \in \mathbb{R}^{1\times|V|}$
5: end for
```

5 Backpropagation Through Time

The backward propagation for RNNs differs from Fully Connected or Convolutional Neural Networks. Since it is time dependent and parameter sharing, we have to formulate very careful.

5.1 Recap: Computational Graphs and Automatic Differentation

A computational graph is useful in visualizing dependency relations between intermediate variables; in other words, automatic differentation. In AD, all numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known, and combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition. In the first phase of AD, the original function code is run forward, populating intermediate variables v_i and recording the dependencies in the computational graph through a book-keeping procedure. In the second phase, derivatives are calculated by propagating adjoints v_i' in reverse, from the outputs to the inputs [2]. Let's define a neural network:

$$\mathbf{z}_1 = \mathbf{X} \cdot w_1^T + b_1 \tag{7}$$

$$\mathbf{h}_1 = \sigma(\mathbf{z}_1) \tag{8}$$

$$\mathbf{z}_2 = \mathbf{h}_1 \cdot w_2^T + b_2 \tag{9}$$

$$\mathbf{h}_2 = \sigma(\mathbf{z}_2) \tag{10}$$

$$\mathbf{z}_3 = \mathbf{h}_2 \cdot w_3^T + b_3 \tag{11}$$

$$\hat{\mathbf{y}} = \mathbf{h}_3 = \sigma(\mathbf{z}_3) \tag{12}$$

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{M} \sum (y - \hat{y})^2 \tag{13}$$

The computational graph of this forward equation would be

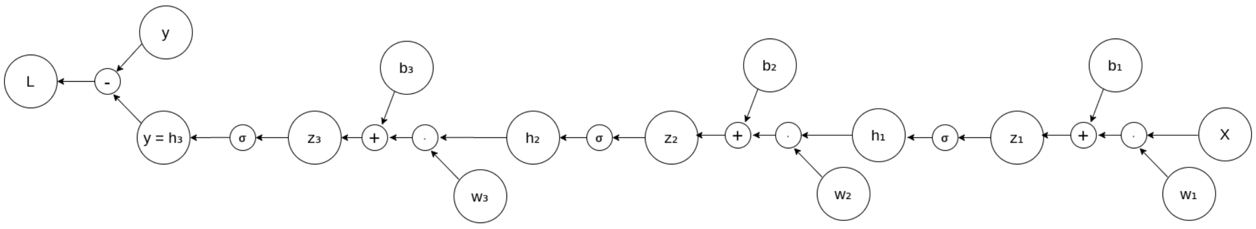


Figure 5: Computational Graph (Forward)

If we reverse the graph, the backward will be calculated by propagating adjoints in reverse, from the outputs to the inputs.

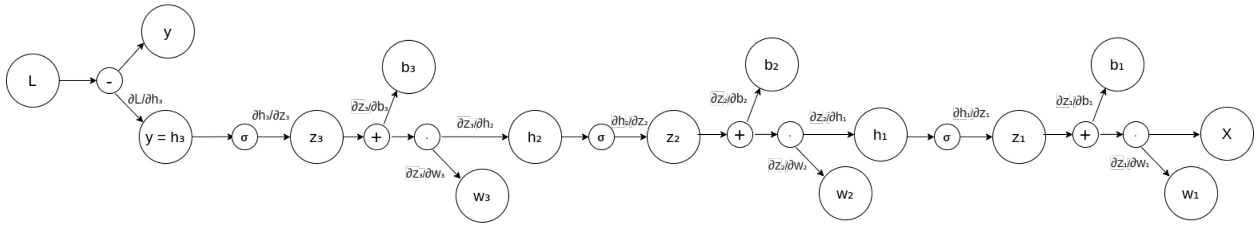


Figure 6: Computational Graph (Backward)

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_3} \tag{14}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2} \tag{15}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} \tag{16}$$

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial b_3} \tag{17}$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2} \tag{18}$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1} \tag{19}$$

We will talk about how to construct a computational graph for RNNs in later chapters.

5.2 Formulating Backpropagation Through Time

Suppose that we are calculating the loss for each output $y^{(t)}$. So loss is to be the cross-entropy loss:

$$\mathbb{E}_t[y_i^t, \hat{y}_i^t] = -\, y_i^t \cdot \log \hat{y}_i^t \tag{20}$$

$$\mathbb{E}_t[y^t, \hat{y}^t] = -\, y^t \cdot \log \hat{y}^t \tag{21}$$

Then the loss is average loss for each output,

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N} \cdot \sum_t^T y^t \cdot \log \hat{y}^t \tag{22}$$

5.2.1 Gradient for U

It is more clear for derivatives with denoting $q^t = U \cdot h^{(t)}$

$$\frac{\partial L^t}{\partial U_{ij}} = \frac{\partial L^t}{\partial \hat{y}_k^t} \cdot \frac{\partial \hat{y}_k^t}{\partial q_l^t} \cdot \frac{\partial q_l^t}{\partial V_{ij}} \tag{23}$$

You can clearly see that for $L^t = -y_k^t \log \hat{y}_k^t$, the derivative is

$$\frac{\partial L^t}{\partial \hat{y}_k^t} = -\frac{y_k^t}{\hat{y}_k^t} \tag{24}$$

Now, we will derive $\hat{y}_k^t = softmax(q_l^t)$ which is $\frac{\partial \hat{y}_k^t}{\partial q_l^t}$. First of all, we have to evaluate the derivation of softmax function.

$$\hat{y}_k^t = \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} \tag{25}$$

Here, softmax is a $\mathbb{R}^n \to \mathbb{R}^n$ mapping function:

$$\hat{y}_k^t : \begin{bmatrix} q_1^t \\ q_2^t \\ q_3^t \\ \vdots \\ q_N^t \end{bmatrix} \to \begin{bmatrix} \hat{y}_1^t \\ \hat{y}_2^t \\ \hat{y}_3^t \\ \vdots \\ \hat{y}_N^t \end{bmatrix} \tag{26}$$

Therefore, the Jacobian of the softmax will be:

$$\frac{\partial \hat{y}^t}{\partial q^t} = \begin{bmatrix} \frac{\partial \hat{y}_1^t}{\partial q_1^t} & \frac{\partial \hat{y}_1^t}{\partial q_2^t} & \cdots & \frac{\partial \hat{y}_1^t}{\partial q_N^t} \\ \frac{\partial \hat{y}_2^t}{\partial q_1^t} & \frac{\partial \hat{y}_2^t}{\partial q_2^t} & \cdots & \frac{\partial \hat{y}_2^t}{\partial q_N^t} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \hat{y}_N^t}{\partial q_1^t} & \frac{\partial \hat{y}_N^t}{\partial q_2^t} & \cdots & \frac{\partial \hat{y}_N^t}{\partial q_N^t} \end{bmatrix} \tag{27}$$

Let's compute the $\frac{\partial \hat{y}_k^t}{\partial q_l^t}$

$$\frac{\partial \hat{y}_k^t}{\partial q_l^t} = \frac{\partial}{\partial q_l^t} \cdot \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} \tag{28}$$

You all remember the quotient rule for derivatives from high school; for $f(x) = \frac{g(x)}{h(x)}$, the derivative of $f(x)$ is given by:

$$f'(x) = \frac{g'(x) \cdot h(x) - h'(x) \cdot g(x)}{h^2(x)} \tag{29}$$

Hence, in our case, $g_k = \exp(q_{t_k})$ and $h_k = \sum_n^N \exp(q_{t_n})$. The derivative, $\frac{\partial}{\partial q_l^t} \cdot h_k = \frac{\partial}{\partial q_l^t} \cdot \sum_n^N \exp(q_n^t)$ is equal to $\exp(q_l^t)$ because $\frac{\partial}{\partial q_l^t} \cdot \exp(q_n^t) = 0$ for $l \neq n$.
Derivative of g_k respect to q_l^t is $\exp(q_l^t)$ only if $k = l$, otherwise it is a constant 0. Therefore, **if we derive the gradient of the off-diagonal entries** of the Jacobian will yield:

$$\begin{aligned} \frac{\partial}{\partial q_l^t} \cdot \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} =& \frac{0 \cdot \sum_n^N \exp(q_n^t) - \exp(q_l^t) \cdot \exp(q_k^t)}{\left[\sum_n^N \exp(q_n^t)\right]^2} && (30) \\ =& - \frac{\exp(q_l^t)}{\sum_n^N \exp(q_n^t)} \cdot \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} && (31) \\ =& - \hat{y}_k^t \cdot \hat{y}_l^t && (32) \end{aligned}$$

Similiarly, **if we derive the gradient of the diagonal entries**, $diag(\partial \hat{y}_k / \partial q_l)$ for $k = l$, we have that:

$$\frac{\partial}{\partial q_l^t} \cdot \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} = \frac{\exp(q_k^t) \cdot \sum_n^N \exp(q_n^t) - \exp(q_l^t + q_k^t)}{\left[\sum_n^N \exp(q_n^t)\right]^2} \tag{33}$$

$$= \frac{\exp(q_k^t) \cdot \left(\sum_n^N \exp(q_n^t) - \exp(q_l^t)\right)}{\left[\sum_n^N \exp(q_n^t)\right]^2} \tag{34}$$

$$= \hat{y}_k^t (1 - \hat{y}_l^t) \tag{35}$$

Thus, we have

$$\frac{\partial \hat{y}_k^t}{\partial q_l^t} = \begin{cases} -\hat{y}_k^t \hat{y}_l^t, & \text{if } k \neq l \\ \hat{y}_k^t (1 - \hat{y}_l^t), & \text{if } k = l \end{cases} \tag{36}$$

If we put Equation 24 and 36 together, gives us a sum over all values of k to obtain $\frac{\partial L^t}{\partial q_l^t}$:

$$-\frac{y_l^t}{\hat{y}_l^t} \cdot \hat{y}_l^t + \sum_{k \neq l} \left(\frac{y_k^t}{\hat{y}_k^t}\right) \cdot (-\hat{y}_k^t \hat{y}_l^t) \tag{37}$$

$$= - y_l^t + y_l^t \cdot \hat{y}_l^t + \sum_{k \neq l} y_k^t \cdot \hat{y}_l^t \tag{38}$$

$$= - y_l^t + \hat{y}_l^t \cdot \sum_k y_k^t \tag{39}$$

If you recall that t^t are all one-hot vectors, then that sum is just equal to 1. So we have

$$\frac{\partial L^t}{\partial q_l^t} = \hat{y}_l^t - y_l^t \tag{40}$$

Recall that we defined $q^t = U \cdot h^{(t)}$, so we can say that

$$q_l^t = U_{lm} \cdot h_m^t \tag{41}$$

Then we have,

$$\frac{\partial q_l^t}{\partial U_{ij}} = \frac{\partial}{\partial U_{ij}} \cdot (U_{lm} h_m^t) \tag{42}$$

$$= \delta_{il} \cdot \delta_{jm} \cdot h_m^t \tag{43}$$

$$= \delta_{il} \cdot h_j^t \tag{44}$$

If we put Equation 40 and 44, we have

$$\frac{\partial L^t}{\partial U} = (\hat{y}^t - y^t) \cdot h_t \tag{45}$$

5.2.2 Gradient for W_h

It is clear that y^t depends on W_h both directly and indirectly. We can directly see the partial derivatives like:

$$\frac{\partial L^t}{\partial W_{h_{ij}}} = \frac{\partial L_t}{\partial \hat{y}_k^t} \cdot \frac{\partial \hat{y}_k^t}{\partial q_l^t} \cdot \frac{\partial q_l^t}{\partial h_m^t} \cdot \frac{\partial h_m^t}{\partial W_{h_{ij}}} \tag{46}$$

Yes this is the partial derivatives respect to $W_{h_{ij}}$ but note that at the last term, there is an implicit dependency of h^t on $W_{h_{ij}}$ through $h^{(t-1)}$. Hence, we have

$$\frac{\partial h_m^t}{\partial W_{h_{ij}}} = \frac{\partial h_m^t}{\partial W_{h_{ij}}} + \frac{\partial h_m^t}{\partial h_n^{(t-1)}} \cdot \frac{\partial h_n^{(t-1)}}{\partial W_{h_{ij}}} \tag{47}$$

We can apply this again and again

$$\frac{\partial h_m^t}{\partial W_{h_{ij}}} = \frac{\partial h_m^t}{\partial W_{h_{ij}}} + \frac{\partial h_m^t}{\partial h_n^{(t-1)}} \cdot \frac{\partial h_n^{(t-1)}}{\partial W_{h_{ij}}} + \frac{\partial h_m^t}{\partial h_n^{(t-1)}} \cdot \frac{\partial h_n^{(t-1)}}{\partial h_p^{(t-2)}} \cdot \frac{\partial h_p^{(t-2)}}{\partial W_{h_{ij}}} \tag{48}$$

This equation continues until h^0 is reached. h^0 is a vector of zeros.
Note that of these four terms we have already calculated first two derivatives. The third one is:

$$\begin{aligned} \frac{\partial q_l^t}{\partial h_m^t} &= \frac{\partial}{\partial h_m^t} \cdot (U_{lb} \cdot h_b^t) && (49) \\ &= U_{lb} \cdot \delta_{bm} && (50) \\ &= U_{lm} && (51) \end{aligned}$$

The last term at Equation 48 collapses to

$$\frac{\partial h_m^t}{\partial h_n^{(t-2)}} \cdot \frac{h_n^{(t-2)}}{\partial W_{h_{ij}}} \tag{52}$$

and we can turn the first item into

$$\frac{\partial h_m^t}{\partial h_n^t} \cdot \frac{\partial h_n^t}{\partial W_{h_{ij}}} \tag{53}$$

Then we have the compact form that is:

$$\frac{\partial h_m^t}{\partial W_{h_{ij}}} = \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{h_{ij}}} \tag{54}$$

And we rewrite this as sum over all values of p less than t, we have

$$\frac{\partial h_m^t}{\partial W_{h_{ij}}} = \sum_{p=0}^{t} \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{h_{ij}}} \tag{55}$$

If we combine all, we have

$$\frac{\partial L^t}{\partial W_{h_{ij}}} = (\hat{y}_l^t - y_l^t) \cdot U_{lm} \cdot \sum_{p=0}^{t} \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{h_{ij}}} \tag{56}$$

5.2.3 Gradient for W_e

It is easy to calculate the gradients W_e now.

$$\begin{aligned} \frac{\partial L^t}{\partial W_{e_{ij}}} &= \frac{\partial L^t}{\partial \hat{y}_k^t} \cdot \frac{\partial \hat{y}_k^t}{\partial q_l^t} \cdot \frac{\partial q_l^t}{\partial h_m^t} \cdot \frac{\partial h_m^t}{\partial W_{e_{ij}}} && (57) \\ \frac{\partial h_m^t}{\partial W_{e_{ij}}} &= \sum_{p=0}^{t} \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{e_{ij}}} && (58) \\ \frac{\partial L^t}{\partial W_{e_{ij}}} &= (\hat{y}_l^t - y_l^t) \cdot U_{lm} \cdot \sum_{p=0}^{t} \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{e_{ij}}} && (59) \end{aligned}$$

6 Multilayer Recurrent Neural Networks

A recurrent neural network can be made deep in many way. Multilayer Recurrent Neural Networks are stack of RNNs, which is designed to be more powerfull. However, if the number of stacks increased carelessly, the model tends to overfit the current dataset and perform poor on dev dataset.
Forward algorithm and backward algorithm is nearly the same as Vanilla Recurrent Neural Network. However, with deep architecture of it, it may require more computational power and memory for both activations, parameters and gradients. In practice, N layer RNN can be implemented with N single RNN networks but modern frameworks like PyTorch have `n_layers` parameter to build N layer RNN easily.

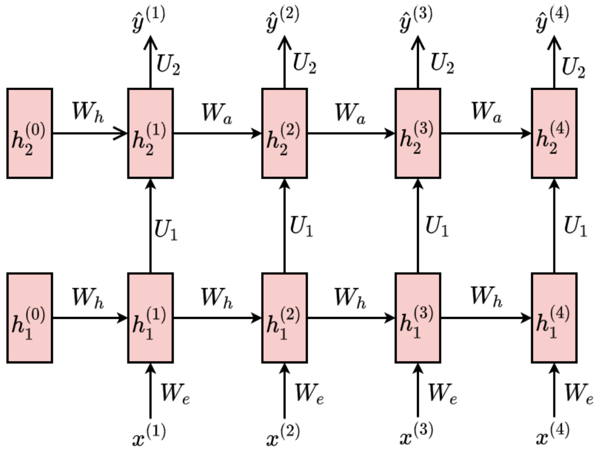


Figure 7: Multilayer Recurrent Neural Networks

Figure 6 shows 2 layer Multi RNN. In recent years power of Multi RNNs are shown: 2 layer RNNs are used to get contextual representation of words [3]. However, this is not the scope of this chapter.

7 Bidirectional Recurrent Neural Networks

Throughout the vanishing and exploding gradient problems (we will evaluate this situation on later chapters) and long-term dependencies of traditional Recurrent Neural Networks, yet there is an another problem for traditional Recurrent Networks. The characteristics of a sequential data could be unidirectional or bidirectional. Lack of traditional Recurrent Neural Networks is that the state at a particular time unit only has knowledge about the past inputs up to a certain point in a sequential data, but it has no knowledge about future states. What do I mean when I said unidirectional or bidirectional features of a sequential data? In certain applications like language modelling, the results are vastly improved with knowledge about both past and future states.
Let's give a very simple example based on a very simple sentence: "The rat ate cheese". The word "rat" and the word "cheese" are related in someway. There is a 'passive' advatage in using knowledge about both the past and future words. Cheese is generally eaten by rats and rats generally eat cheese. As I said this is a very simple example.

Given a sequence of N tokens $(t_1, t_2, ..., t_N)$, forward model computes the probability of the sequence by modeling the probability of token t_k, given the history $(t_1, t_2, ...t_{k-1})$:

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, t_2, ..., t_{k-1}) \tag{60}$$

At each position k, each RNN layer outputs a context-dependent representation $\vec{\mathbf{h}}_{k,j}$ where $j = 1, 2, ..., L$ (number of layers). Top layer RNN output, $\vec{\mathbf{h}}_{k,L}$ is used to predict next token t_{k+1} with softmax. The equations that we formulated above are for forward RNN. The backward RNN is similar to forward LM:

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1}, t_{k+2}, ..., t_N) \tag{61}$$

with each backward RNN layer j in L layer deep model producing representations $\overleftarrow{\mathbf{h}}_{k,j}$ of t_k, given $(t_{k+1}, t_{k+2}, ..., t_N)$. And the formulation jointly maximizes the log-likelihood of the forward and backward directions:

$$\sum_{k=1}^{N} \log p(t_k \mid t_1, t_2, ..., t_{k-1}; \Theta_x, \vec{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, t_{k+2}, ..., t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \tag{62}$$

7.1 Architecture of bi-RNN

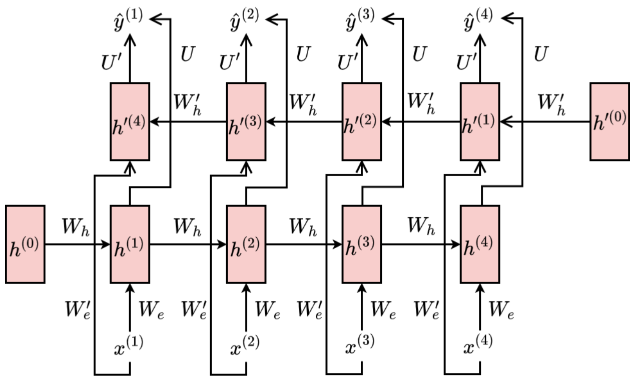


Figure 8: Bidirectional Recurrent Neural Network

In the bidirectional Recurrent Neural Networks, we have seperate hidden states/layers $h^{(t)}$ and $h'^{(t)}$ for the forward and backward directions. The forward hidden states interact with each other and the same is true for the backward hidden states. There is no multilayered connections between them. However, both $h^{(t)}$ and $h'^{(t)}$ receive input from the same vector $\hat{y}^{(t)}$. In general, any property of the current word can be predicted more effectively using this approach, because it uses the context on both sides. For example, the ordering of words in several languages is somewhat different depending on grammatical structure. Bidirectional Recurrent Neural Networks work well in tasks where the predictions are based on bidirectional context like handwrittings.
The outputs of both forward and backward directions can be averaged or concatenated to produce a single output. PyTorch framework applies this pooling strategy by concatenating.

8 Turing Completeness

Recurrent neural networks are known to be Turing complete. Turing completeness means that a recurrent neural network can simulate any algorithm, given enough data and computational re-

sources [4]. Define a formal language Σ^* for some finite alphabet Σ. Intuitively, a "grammar" is the finite set of rules that describes the structure of an infinite language. In formal language theory, a grammar can be seen as a computational system (automaton) describing a language. For example, a Deterministic Finite Automaton can recognize ab^*. However it cannot recognize $a^n b^n$, because we must "count" the n to recognize b^n after a^n. Finite-state automata represent the limits of computation with only finite memory available. With additional memory, a recognizer is capable of specifying more complex languages. One way to view more complicated formal grammars, then, is as finite-state automata augmented with various kinds of data structures, which introduce different kinds of unbounded memory [5].
With a stack automaton, we have a different way to create a recognizer for $a^n b^n$. We push each a onto the stack. For a b, we pop if the stack contains a, and raise an error if it contains a b or is empty. Finally, at the end of the string, we accept if the stack is empty. Also, a DFA cannot recognize parentheses languages. The set of languages recognizable by a stack automaton is called the deterministic context-free languages (DCFLs), and forms a strict subset of the context-free languages (CFLs) proper [5].

A big question in mathematical linguistics is the place of natural language within this hierarchy. Context-free formalisms describe much of the hierarchical structure of natural languages (or mildly context-sensitive).

A weighted finite-state automata uses a finite-state machine to encode a string into a number, rather than producing a boolean decision for language membership. A path score is seen like a basic RNN:

$$W(\pi) = \lambda(q_0) \otimes \left(\bigotimes_{i=1}^{t} \tau(q_{i-1,x_i,q_i}) \right) \otimes p(q_t) \tag{63}$$

If we introduce a Hankel Matrix H_f to a WFS, we can approximate any non-binary string with learning optimal $\hat{H}_f$ with SGD [5]. This definition is related to computational power of RNNs.

9 Practical Recurrent Neural Networks

In this section, we will evaluate what we can build with RNNs as a practical application. RNNs are designed to model sequential tasks like Sequence Classification, Sequence Labeling, Language Modeling. Besides, it can be used for Speech Recognition, Machine Translation, Image Generation etc. However, we do not include this topics in this chapter.

9.1 Special Tokens

In general, it is convenient to use "special tokens" in our tasks. For example, introducing `[START]` and `[END]` tokens might be helpful. To give an example

- **S1:** `[START]` fall fires burning neath black twisted boughs `[END]`
- **S2:** `[START]` a blaze so high it lights the night `[END]`

Assume that, we are tokenizing those sentences to get token ids. Then, pass it to a RNN model. This means batch size of 2. The token ids are

- **S1:** [0, 167, 137, 5141, 78, 77, 4561, 13, 1]
- **S1:** [0, 55, 744, 21, 44, 324, 879, 13596, 7, 1]

Generally in Deep Learning, we represent our data with tensors and the batch size would be a dimension of our tensor. However, as you can see, length of the introduced sentences are not equal. We are not able to represent it as a tensor, due to number of entries in a row must be equal.

9.2 Batching Variable Length Inputs

As we mentioned in previous section, length of sentences (sequences) might be different and we are not always able to generate batch tensors. So, what should we do?

If we use batch size of 1, this will not a real problem. Because, some of deep learning frameworks uses dynamic computational graphs. A dynamic computation graph is rebuilt at each training iteration. You can easily creat or remove a differentiable parameter at training, and this will not ruin your computational graph. Using variable sequences with batch size of 1 requires dynamic computational graph.

On the other side, static computational graphs, is rebuilt only once. So, you should pass your inputs as constant length.

Frameworks like PyTorch or DyNet [6] have excellent and efficient implementations for dynamic computational graphs, variable length batching etc. However, frameworks like Keras or Tensorflow still lack about it (even with introducing eager execution).

What if we use batch size of >1? We have to pad our sequences with padding token `[PAD]` to construct an appropriate tensor for batching.

- **S1:** `[START]` fall fires burning neath black twisted boughs `[PAD]` `[END]`
- **S2:** `[START]` a blaze so high it lights the night `[END]`

And,

- **S1:** [0, 167, 137, 5141, 78, 77, 4561, 13, 2, 1]
- **S1:** [0, 55, 744, 21, 44, 324, 879, 13596, 7, 1]

9.3 Sequence Classification

Sequence Classification is a task of classifying sentences by their sentiment or something else. Since a sentence is compisition of positional words, we can achieve this task with RNNs. So, how we can build a classifier with a RNN?

9.3.1 Building A Sequence Classifier

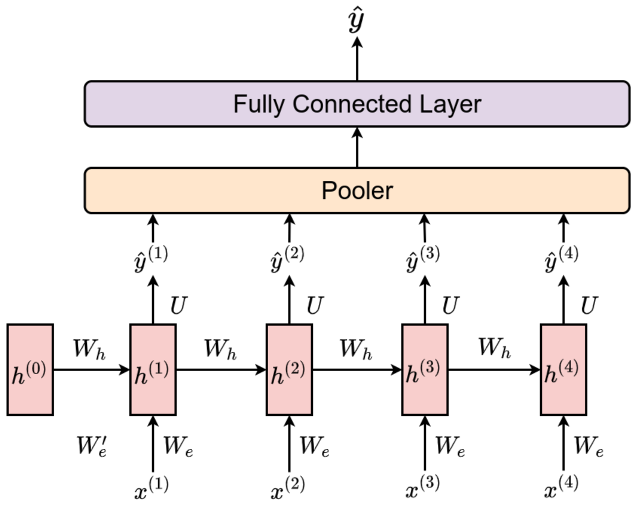


Figure 9: A sentence classifier with RNN.

Comparing to other tasks, this problem has very intuitive answer: adding a classifier layer (can be a Fully Connected Layer) to last output of the RNN. Since last output of RNN contains full information of the current sentence (because last position contains pervious information), it is proper to design this model.

9.4 Sequence Labeling

Sequence labeling has been one of the most discussed topics in Linguistics and Computational Linguistics history. Challenges like Dependency Parsing, Word Sense Disambiguation and Sequence Labeling, etc., arose with the formal definition of the syntax. Sequence labeling is a Natural Language Processing task. It aims to classify each token (word) in a class space **C**. This classification approach can be independent (each word is treated independently) or dependent (each word is dependent on other words).

Words are sequential building blocks of sentences. Each word contributes syntactic and semantic properties to a sentence. For example, a word can be an adjective, which can give positive semantics (delicate). By doing that, this Adjective can describe a noun (tender boy). This sequential relationship can go recursively and unbounded. This shows us words are related to each other. But how can we determine the role of the word? For example, how do we decide which word or phrase is a noun or which one is an adjective?

9.4.1 Part-Of-Speech-Tagging

Part-Of-Speech Tagging (POS Tagging) is a sequence labeling task. It is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech (syntactic tag), based on both its definition and its context. POS Tagging is a helper task for many tasks about NLP: Word Sense Disambiguation, Dependency Parsing, etc.

A sequence is a series of tokens where tokens are not independent of each other. Series in mathematics and sentences in linguistics are both sequences. Because; in both of them, the next token depends on the previous ones or vice versa.

Determining POS Tags is much more complicated than simply mapping words to their tags. Consider word **back**:

- Earnings growth took a **back/JJ** seat.
- A small building in the **back/NN**.
- A clear majority of senators **back/VBP** at the bill.

Each word back has a different role. The first back is Adjective; the second black is Noun, the third back is non-3rd person singular present Verb [7].

9.4.2 Named Entity Recognition

The first step in information extraction is to detect the entities in the text. A named entity is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. NER helps us to identify and extract critical elements from the text.

NER can be used for customer services, medical purposes, document categorization. For example, extracting aspects from customer reviews with NER helps identification of semantics. Or extracting medical diseases and drugs from a text helps identification of treatment.

9.4.3 Building A Token Classifier

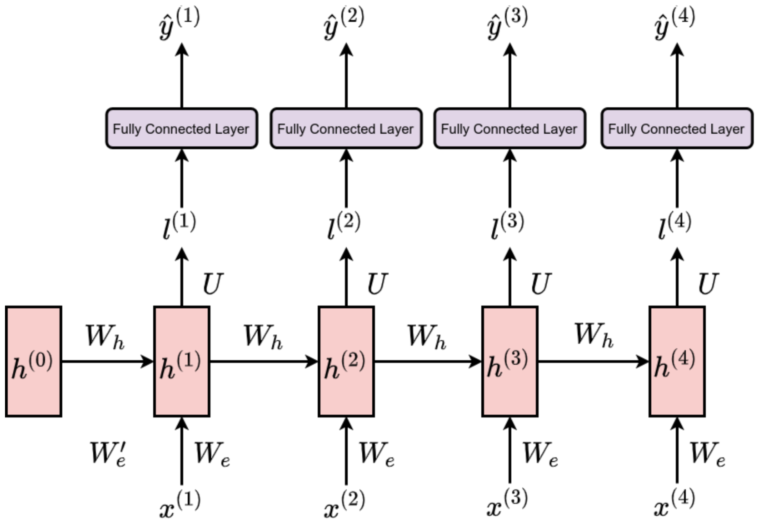


Figure 10: A token classifier with RNN.

If we put a classifier layer (for example fully connected layer), we are able to classify each token. A POS task or NER task can be done with RNNs like that.

9.5 Language Modeling

A (generative) language model is "autoregressive". A language model is called autoregressive if it predicts future tokens based on past tokens.
Training is autoregressive language model is similiar to POS tagging. We are calculating loss between input and shifted input:

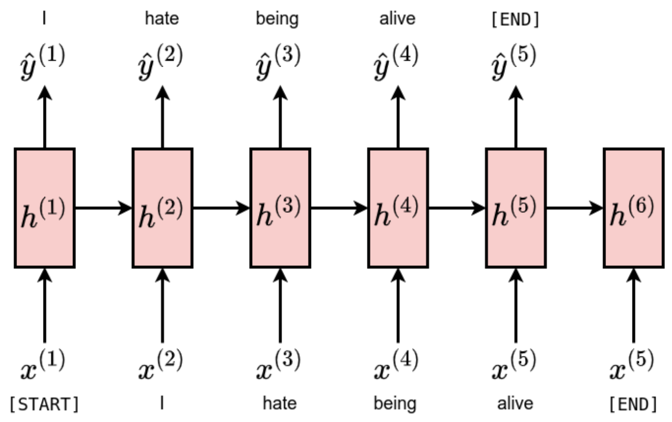


Figure 11: Training a autoregressive language model.

However, inference time differs from training procedure. We actually, generate tokens/words autoregressively.

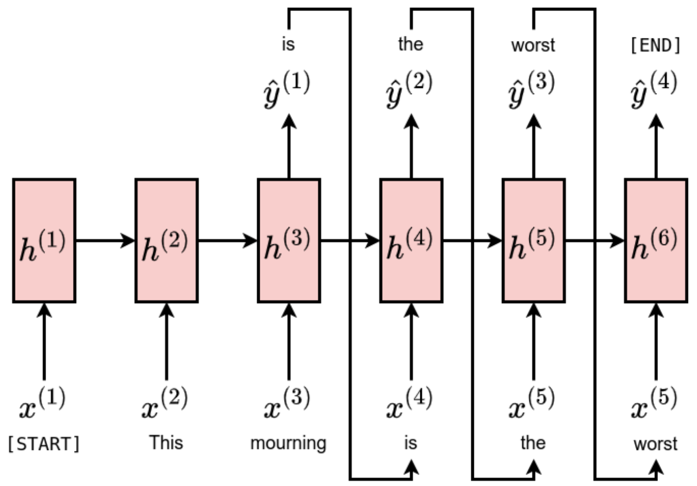


Figure 12: Generating text with RNN LM.

10 Vanishing Gradients

Vanishing gradients is a crucial problem in RNNs. It is mainly caused by long dependencies (high sequence lengths). If our sentence length is around ∼10, then it is easy to capture full information from text. However, applications like text summarization, the length of the text may be long. If the length of our sentence converges to ∼10000, it is not always easy to capture full information: i.e., last hidden layer may not generalize the first words of our sentence. To make a formal definition of vanishing gradients, recall that from RNNs,

$$h^{(t)} = \sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1) \tag{64}$$

Therefore,

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1))W_h \tag{65}$$

Consider the gradient of loss $J^{(i)}(\theta)$ on step i, with respect to the hidden state $h^{(j)}$ on the same previous step j:

$$\frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \cdot \prod_{j<t\leq i} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \tag{66}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \cdot W_h^{(i-j)} \prod_{j<t\leq i} diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1)) \tag{67}$$

So, if W_h is small, then this term gets vanishingly small as$i$ and j get further apart. Consider L2 matrix norms,

$$\left\| \frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} \right\| \leq \left\| \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \right\| \cdot \|W_h\|^{(i-j)} \cdot \prod_{j<t\leq i} \left\| diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1)) \right\| \tag{68}$$

If the $\max_{\lambda_i} \lambda_i \leq 1$ then the gradient will shrink exponentially. The problem of vanishing gradients cause what we call "syntactic recency" vs "sequential recency". As in the Vanishing Gradient, the problem of recency is getting important while sentence length is increasing:

the writer of the books is . VS the writer of the books are .

10.1 Gradient Clipping

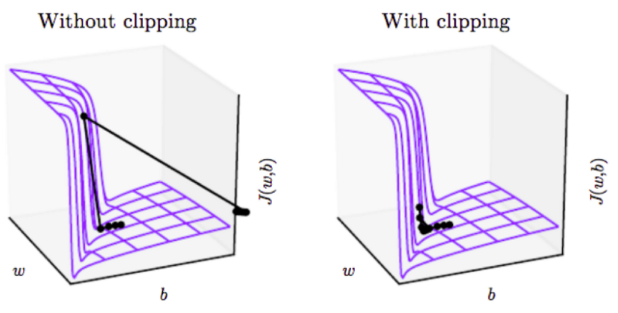


Figure 13: without clipping vs with clipping [8]

Algorithm 3 Gradient Clipping Algorithm [8]

```
Require: threshold
1: $\hat{\mathbf{g}} \leftarrow \frac{\partial E}{\theta}$
2: if $\|\hat{\mathbf{g}}\| \geq threshold$ then
3:     $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
4: end if
```

From the Deep Learning textbook [1]: the basic idea is to recall that the gradient does not specify the optimal step size, but only the optimal direction within an infinitesimal region. When the traditional gradient descent algorithm proposes to make a very large step, the gradient clipping heuristic intervenes to reduce the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent [1].

11 Long Short-Term Memory

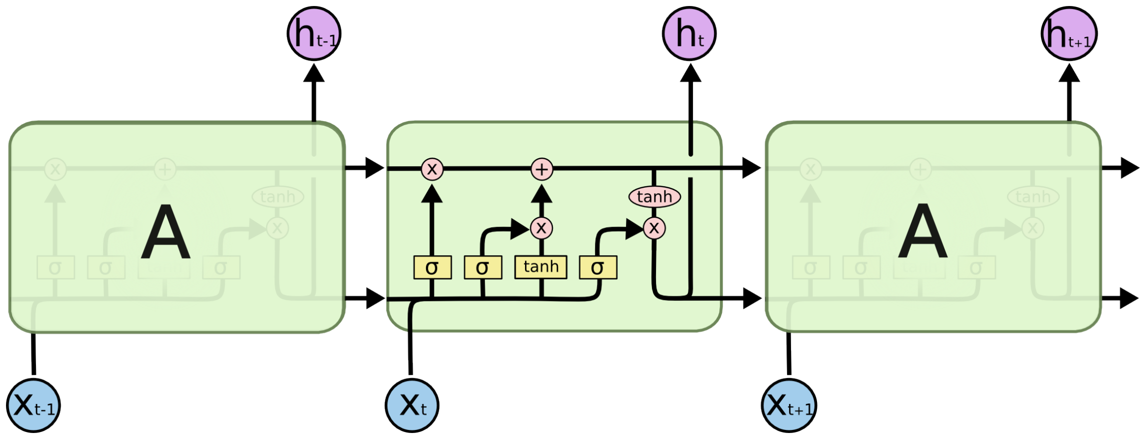


Figure 14: Long short-term memory [9].

We are lucky that there is an amazing explanation and definition for Long Short-Term Memory from Christopher Olah's LSTM Blog Post [9]. We are going to study his detailed work:

Long Short-Term Memory (LSTMs) are a special kind of RNNs, capable of learning long-term dependencies. They were proposed in [10]. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

LSTMs are fancy version of RNNs. Instead of a unit that simply applies an elementwise non-linearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information [1].

11.0.1 Cell State

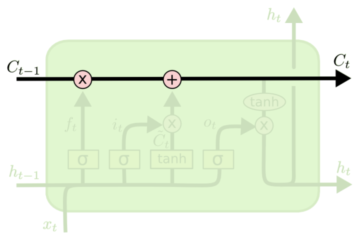


Figure 15: Cell state [9].

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation

11.0.2 Forget Gate

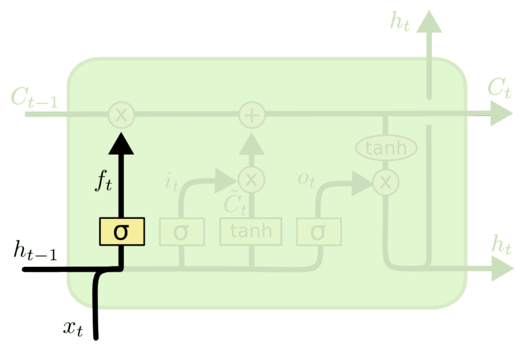


Figure 16: Forget gate [9].

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the forget gate. Let's consider a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject [9].
The formal definition of forget gate can be done:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \tag{69}$$

where $\mathbf{x}^{(t)}$ is current input vector and $\mathbf{h}^{(t)}$ is the current hidden layer vector. $\mathbf{b}^f$ is biases, $\mathbf{U}^f$, $\mathbf{W}^f$ are trainable input and recurrent weights for forget gate.

11.0.3 External Input Gate

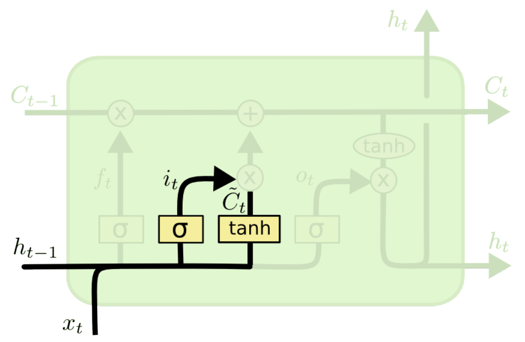


Figure 17: External input gate [9].

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the external input gate decides which values we'll update.

Next, a *tanh* layer creates a vector of new candidate values, $\tilde{\mathbf{C}}^{(t)}$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting [9]. The formal definition of external input gate can be done:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right) \tag{70}$$

$$\tilde{C}_i^{(t)} = \tanh\left(b_i^C + \sum_j U_{i,j}^C x_j^{(t)} + \sum_j W_{i,j}^C h_j^{(t-1)}\right) \tag{71}$$

11.0.4 Updating Cell State

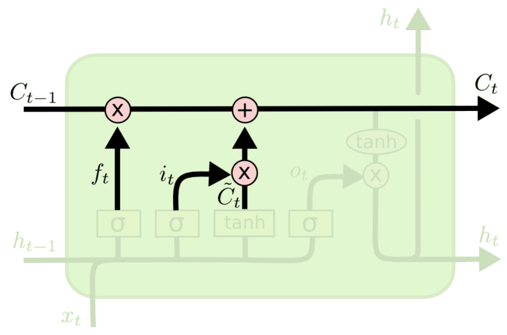


Figure 18: Cell state updating [9].

It's now time to update the old cell state: $\mathbf{C}^{(t-1)} \rightarrow \mathbf{C}^{(t)}$. We multiply the old state by $\mathbf{f}^{(t)}$ forgetting the things we decided to forget earlier. Then we add

$$\mathbf{i}^{(t)} \odot \tilde{\mathbf{C}}^{(t)} \tag{72}$$

$$C_i^{(t)} = f_i^{(t)} C_i^{(t-1)} + g_i^{(t)} \tilde{C}_i^{(t)} \tag{73}$$

This is the new candidate values, scaled by how much we decided to update each state value. In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps [9].

11.0.5 Output Gate

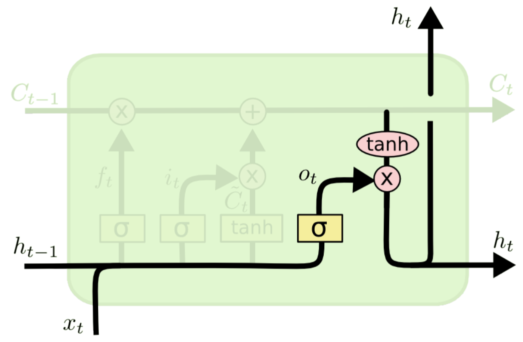


Figure 19: Output gate [9].

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to [9].

$$\mathbf{C}^{(t)} \odot \mathbf{o}^{(t)} \tag{74}$$

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next [9]. The formal definition of output gate can be done:

$$o_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t)}\right) \tag{75}$$

$$h_i^t = \tanh(C_i^{(t)}) o_i^{(t)} \tag{76}$$

12 Gated Recurrent Unit

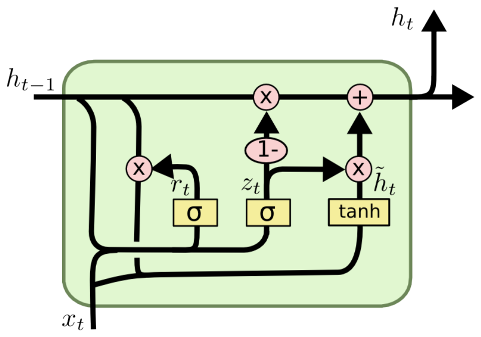


Figure 20: Gated Recurrent Unit.

A different modification of LSTM is Gated Recurrent Unit (GRU) [11]. It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular [9].

$$z_i^{(t)} = \sigma\left(b_i^z + \sum_j W_{i,j}^z x_j^{(t)} + \sum_j W_{i,j}^z h_j^{(t-1)}\right) \tag{77}$$

$$r_i^{(t)} = \left(b_i^r + \sum_j W_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t-1)}\right) \tag{78}$$

$$\tilde{h}_i^{(t)} = \tanh\left(b_{hh}^r + r^{(t)} \sum_j W_{i,j}^{hh} h_j^{(t-1)} + \sum_j W_{i,j}^{hh} x_j^{(t)}\right) \tag{79}$$

$$h_i^{(t)} = (1 - z^{(t)}) * h^{(t-1)} + z^{(t)} \tilde{h}^{(t)} \tag{80}$$

13 Recursive Neural Networks

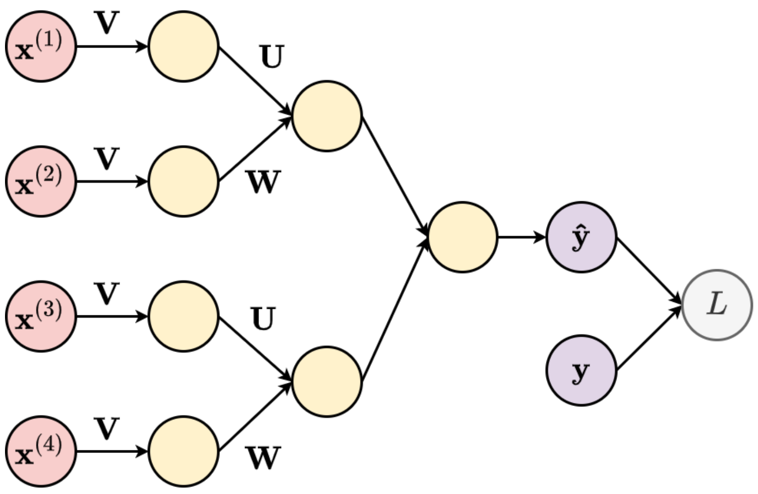
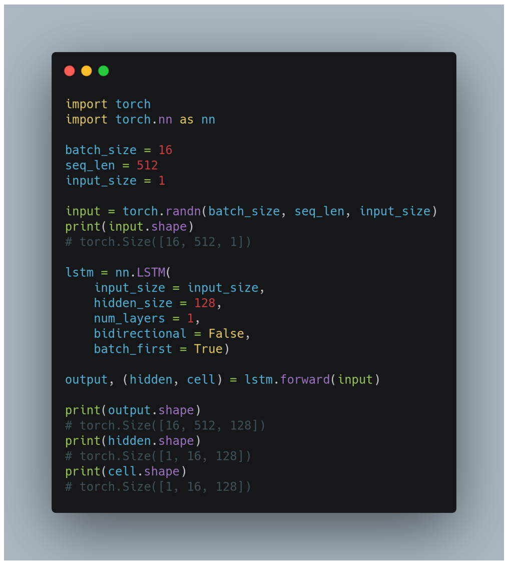


Figure 21: Recursive Neural Network.

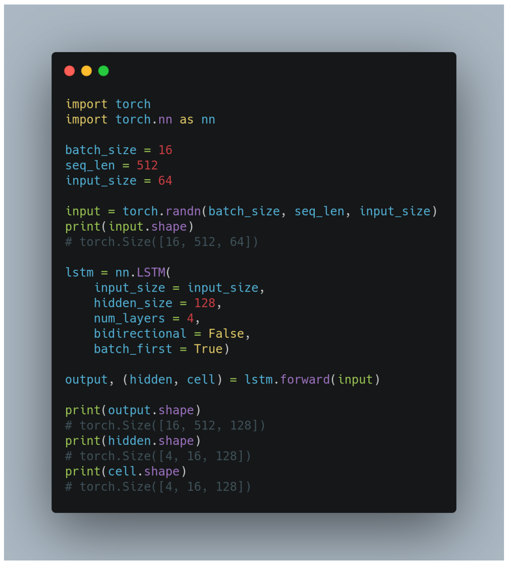
Recursive Neural Networks are special design of neural networks to represent and process a sentence in the tree form rather than the chain-like structure of RNNs. Normally, almost every linguistics approach includes tree data structure. In some applications in Natural Language Processing, choosing a Recursive Neural Network might be best to represent the task, i.e., Recursive Neural Networks can easily handle with parse trees, morphological trees etc.

14 LSTMs with PyTorch



```
import torch
import torch.nn as nn

batch_size = 16
seq_len = 512
input_size = 1

input = torch.randn(batch_size, seq_len, input_size)
print(input.shape)
# torch.Size([16, 512, 1])

lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = 128,
    num_layers = 1,
    bidirectional = False,
    batch_first = True)

output, (hidden, cell) = lstm.forward(input)

print(output.shape)
# torch.Size([16, 512, 128])
print(hidden.shape)
# torch.Size([1, 16, 128])
print(cell.shape)
# torch.Size([1, 16, 128])
```

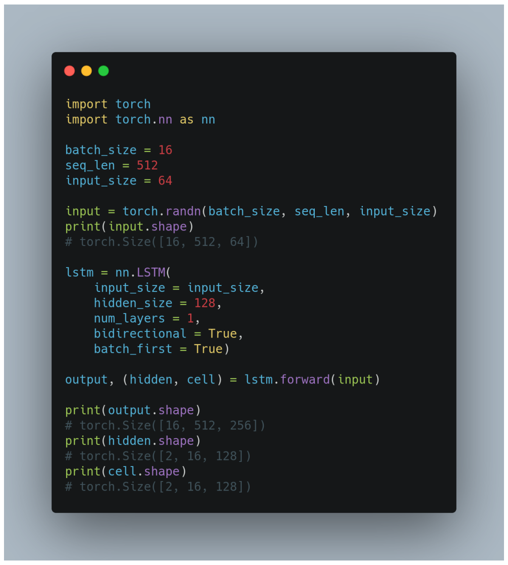
(a) One Layer Unidirectional LSTM



```
import torch
import torch.nn as nn

batch_size = 16
seq_len = 512
input_size = 64

input = torch.randn(batch_size, seq_len, input_size)
print(input.shape)
# torch.Size([16, 512, 64])

lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = 128,
    num_layers = 4,
    bidirectional = False,
    batch_first = True)

output, (hidden, cell) = lstm.forward(input)

print(output.shape)
# torch.Size([16, 512, 128])
print(hidden.shape)
# torch.Size([4, 16, 128])
print(cell.shape)
# torch.Size([4, 16, 128])
```

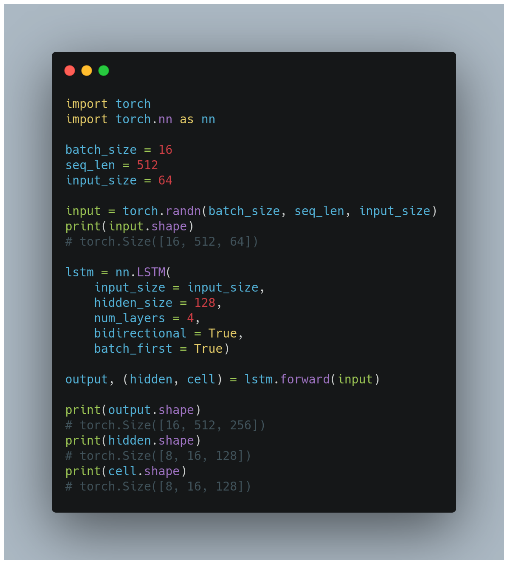
(b) Multilayer Unidirectional LSTM

Figure 22



```
import torch
import torch.nn as nn

batch_size = 16
seq_len = 512
input_size = 64

input = torch.randn(batch_size, seq_len, input_size)
print(input.shape)
# torch.Size([16, 512, 64])

lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = 128,
    num_layers = 1,
    bidirectional = True,
    batch_first = True)

output, (hidden, cell) = lstm.forward(input)

print(output.shape)
# torch.Size([16, 512, 256])
print(hidden.shape)
# torch.Size([2, 16, 128])
print(cell.shape)
# torch.Size([2, 16, 128])
```

(a) One Layer Bidirectional LSTM



```
import torch
import torch.nn as nn

batch_size = 16
seq_len = 512
input_size = 64

input = torch.randn(batch_size, seq_len, input_size)
print(input.shape)
# torch.Size([16, 512, 64])

lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = 128,
    num_layers = 4,
    bidirectional = True,
    batch_first = True)

output, (hidden, cell) = lstm.forward(input)

print(output.shape)
# torch.Size([16, 512, 256])
print(hidden.shape)
# torch.Size([8, 16, 128])
print(cell.shape)
# torch.Size([8, 16, 128])
```

(b) Multilayer Bidirectional LSTM

Figure 23

15 Neural Machine Translation

One of the main usage of RNNs is (was...) Neural Machine Translation (NMT). In NMT, our task is translating source language to target language, by using encoder-decoder RNNs. Encoder-decoder RNNs generally noted as sequence-to-sequence models, which is just a combination of 2 RNN model. The first one is used for encoding the source language to a hidden vector, the latter one is used for decoding it to the target language. The encoder of the seq2seq NMT model might be bidirectional or unidirectional, however, the decoder must be used as a autoregressive model. Hence, it must be a unidirectional RNN. At training time, our model acts like a POS tagging scheme. We calculate each token's loss and sum them, then backpropagate. For example $J^{(1)}(\theta)$ is negative log-probability of "hayatın" (end-to-end training). Or, $J^{(1)}(\theta)$ is negative log-probability of special token "<END>".
NMT directly calculates $p(y \mid x)$:

$$p(y \mid x) = p(y^{(1)} \mid x) \cdot p(y^{(2)} \mid y^{(1)}, x) \cdot p(y^{(3)} \mid y^{(2)}, y^{(1)}, x) \cdot ... \cdot p(y^{(T)} \mid y^{(T-1)}, y^{(T-2)}, ..., y^{(1)}, x) \tag{81}$$

Each component is probability of next target word given target words so far and source sentence x. The objective is, as can be seen,

$$\arg\max_y p(y \mid x)$$

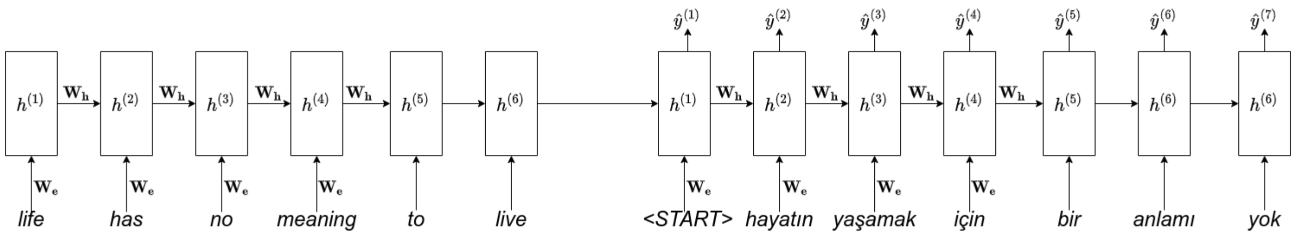


Figure 24: NMT

However, at inference time, we pass the source sentence to translate. Then, we generate the words in autoregressive fashion. Each generated word is passed to $t+1$th step as an input, then produce the $t+1$th generated sentence.

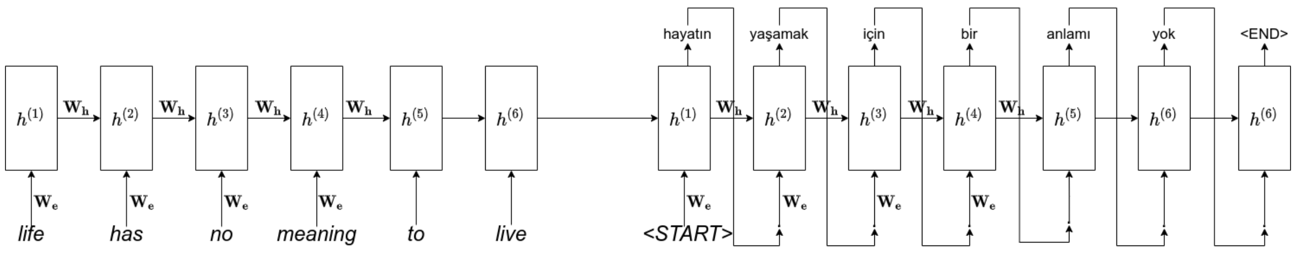


Figure 25: NMT

We call it, this is **greedy decoding**. Taking the most probable word on each step. There are several decoding strategies in natural language processing. So, what is wrong with greedy decoding?

15.1 Decoding Strategies

15.1.1 Greedy Decoding

Greedy decoding has no way to undo decisions. Also we are not guaranteed that, there will be <END> token at decoding state. On the other hand, if we get a <END> token, there is still a possibility that it may not be the most optimal solution.

input: life has no meaning to live.
$\rightarrow$ hayatın ......
$\rightarrow$ hayatın yaşamak ......
$\rightarrow$ hayatın yaşamak için ......
$\rightarrow$ hayatın yaşamak için turtası ...... (???? no going back now)

What we are doing in greedy decoding actually is taking $\arg\max$ of each output, so we have one output with highest probability, but this does not mean that it will converge to highest overall probability.

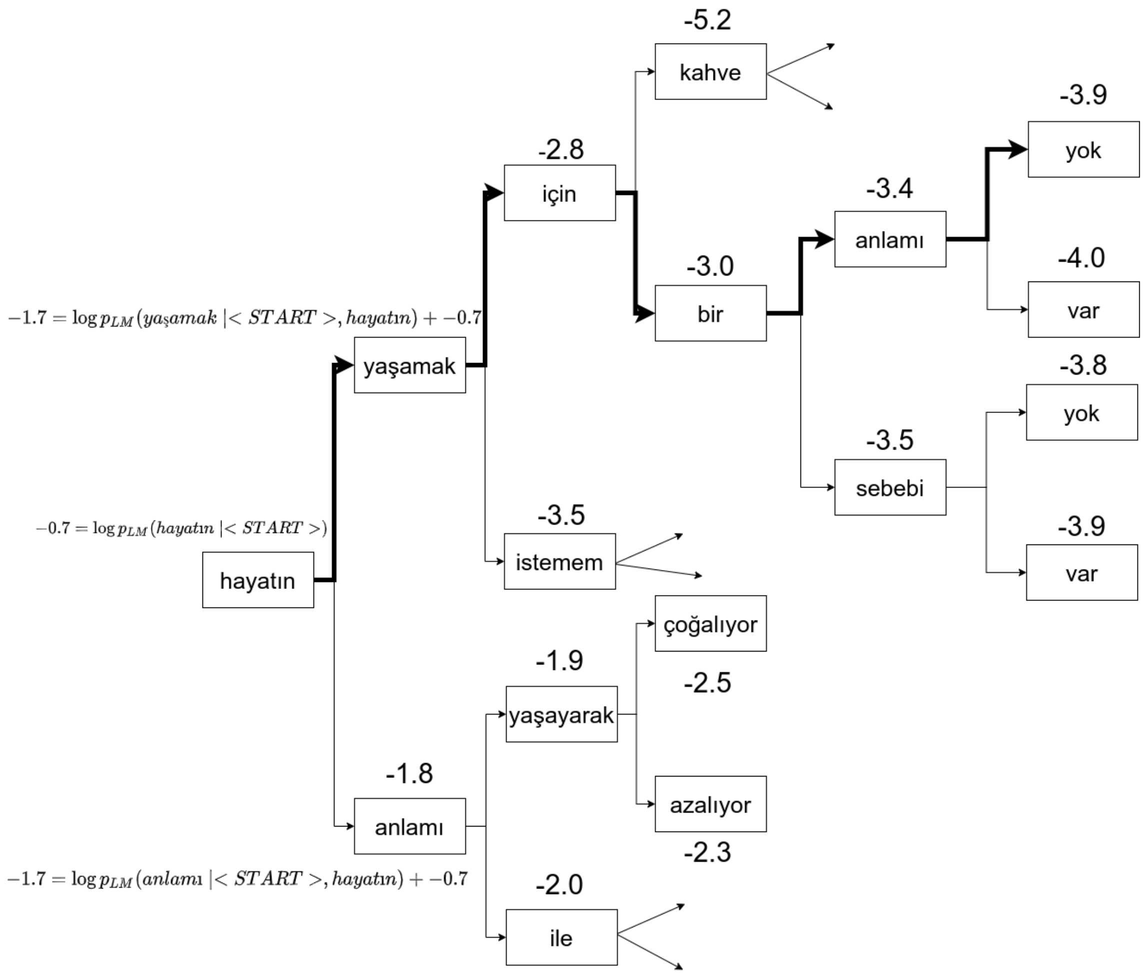
15.1.2 Exhaustive Search

Could we try computing all possible sentences y? Well yes but actually no. This has $O(\mid V \mid^T)$ complexity.

15.1.3 Beam Search

On each step of decoder, keep track of the k most probable partial translations (which we call hypotheses).

- k is the beam size (in practice: 5-10).
- A hypothesis $y_1, ..., y_t$ has a score which is log probability.
  - We search for high-scoring hypotheses, tracking top k on each step.
- Beam search is not guaranteed to find optimal solution.



-5.2
kahve
-3.9
yok
-2.8
için
-3.4
anlamı
-3.0
bir
-4.0
var
$-1.7 = \log p_{LM}(yaşamak \mid < START >, hayatın) + -0.7$
yaşamak
-3.8
yok
-3.5
sebebi
-3.5
istemem
-3.9
var
$-0.7 = \log p_{LM}(hayatın \mid < START >)$
hayatın
çoğalıyor
-1.9
yaşayarak
-2.5
-1.8
anlamı
azalıyor
-2.3
$-1.7 = \log p_{LM}(anlamı \mid < START >, hayatın) + -0.7$
-2.0
ile

Figure 26: Beam Search

As you can see, Greedy Decoding chooses [”hayatın”, ”yaşamak”, ”için”, ”bir”, ”anlamı”, ”yok”] with score of -3.9. But it is not the most probable sentence. With Beam Search, with beam size = 2, we can select the most probable sentence (in the space of beam size = 2), [”hayatın”, ”yaşamak”, ”için”, ”bir”, ”sebebi”, ”yok”].

Beam Search: Stopping Criterion

- In greedy decoding, we usually decode until the model produces <END> token.
- In Beam Search, different hypotheses may produce <END> tokens on different timesteps.
  - When a hypothesis produces <END>, that hypothesis is complete.
  - Place it aside and continue exploring other hypotheses via Beam Search.
- Usually we continue Beam Search until:
  - We reach timestep T (pre-defined) or,
  - We reach at least n completed hypotheses (pre-defined).

Beam Search: Finishing Up

- We have our list of completed hypotheses.

- How to select top one with highest score?
- **Choosing shortest one is a problem**: longer hypotheses have lower scores.
- So, normalize by length!

$$\frac{1}{t}\sum_{i=1}^{t}\log p_{\mathrm{LM}}(y_i \mid y_{i-1}, ..., y_1, x)$$

15.2 BLEU Score

Turkish: kedi paspasın üzerinde
Reference 1: the cat is on the mat
Reference 2: there is a cat on the mat
Candidate: the cat the cat on the mat

15.2.1 Unigram Precision

Unigram	**Shown In References?**
the	1
cat	1
the	1
cat	1
on	1
the	1
mat	1

Then, merge the unigram counts:

Unique Unigram	**Count**
the	3
cat	2
on	1
mat	1

The total number of counts for the unique unigrams in the candidate sentence is 7, and the total number of unigrams in the candidate sentence is 7. The unigram precision is 7/7 = 1.0

15.2.2 Bigram Precision

Bigram	**Shown In References?**
the cat	1
cat the	0
the cat	1
cat on	1
on the	1
the mat	1

Then, merge the bigram counts:

Unique Bigram	**Count**
the cat	2
cat the	0
cat on	1
on the	1
the mat	1

The total number of counts for the unique bigrams in the candidate sentence is 5, and the total number of bigrams in the candidate sentence is 6. The bigram precision is $5/6 = 0.833$.

15.2.3 Calculating BLEU

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

and

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r \end{cases}$$

Where p_n is the precision of n-gram, with natural logarithm, w_n is the weight between 0 and 1, with constraint of $\sum_{i=1}^{n} w_i = 1$, and BP is the brevity penalty to penalize short machine translations. Where c is the number of unigrams (length) in all the candidate sentences, and r is the best match lengths for each candidate sentence in the corpus. BLEU generally uses $N = 4$ and $w_n = \frac{1}{n}$.

15.2.4 Proof of BLEU $\in [0, 1]$

$$\begin{aligned} \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) &= \prod_{n=1}^{N} \exp\left(w_n \log p_n\right) \\ &= \prod_{n=1}^{N} \Big[\exp\left(\log p_n\right)\Big]^{w_n} = \prod_{n=1}^{N} {p_n}^{w_n} \\ &\in [0, 1] \end{aligned}$$

16 Attention for Recurrent Layers

Attention is invented for mainly alignment of source-target language. In Statistical Machine Translation (SMT), the problem of alignment is done by hand-crafted methods or phrase-to-phrase methods. With neural attention, models can learn the alignment by gradient based methods.

16.1 The Problem of Alignment

Since Statistical Machine Translation, alignment between source sentence's words and target sentence's words is a main problem in research. There are many alingment types and some of them are complex.

16.1.1 one-to-one Alignment

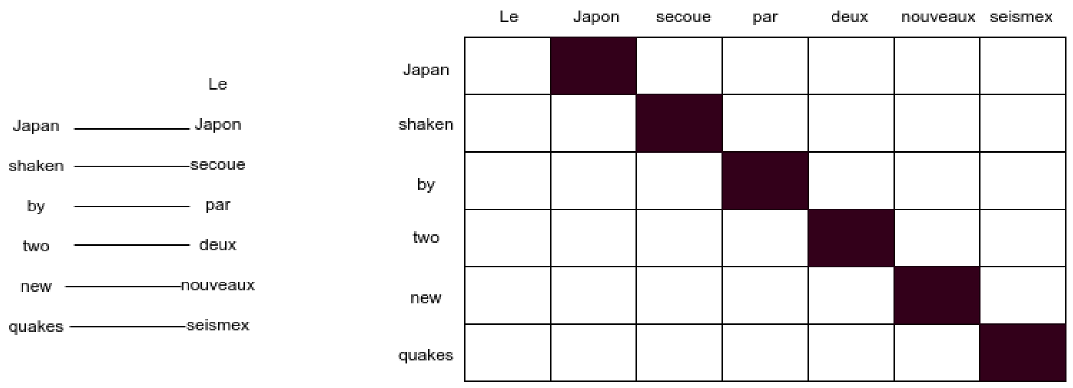


Figure 27: one-to-one

We call "Le" as "spurios" word.

16.1.2 many-to-one Alignment

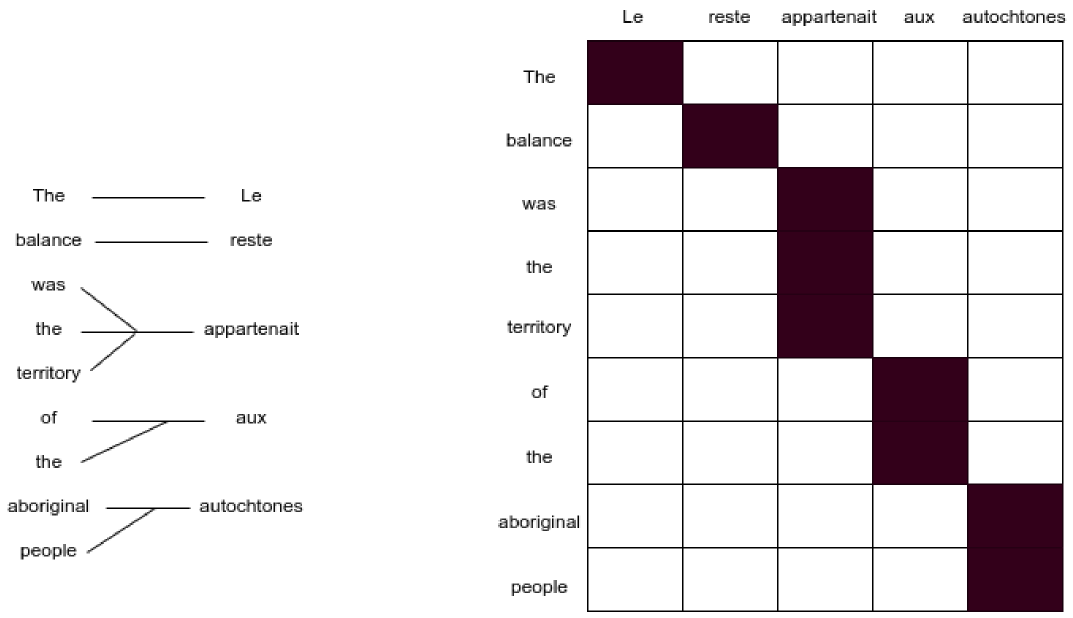


Figure 28: many-to-one

16.1.3 one-to-many Alignment

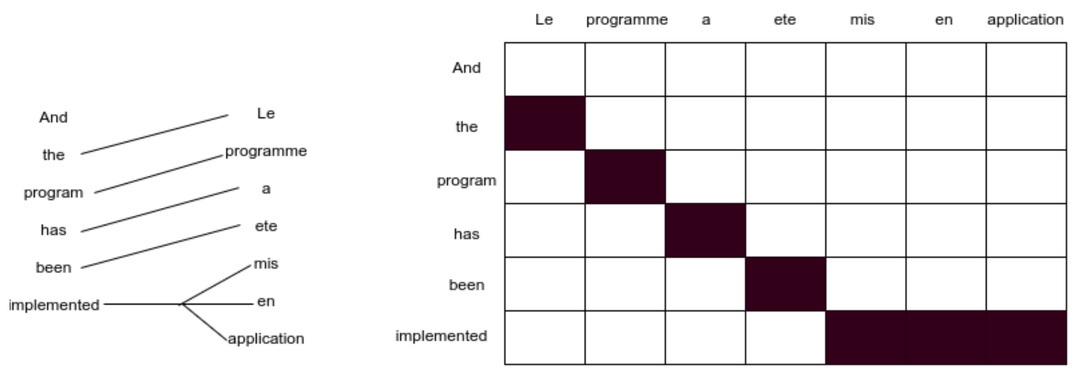


Figure 29: one-to-many

We call "implemented" as "fertile" word.

16.1.4 many-to-many Alignment

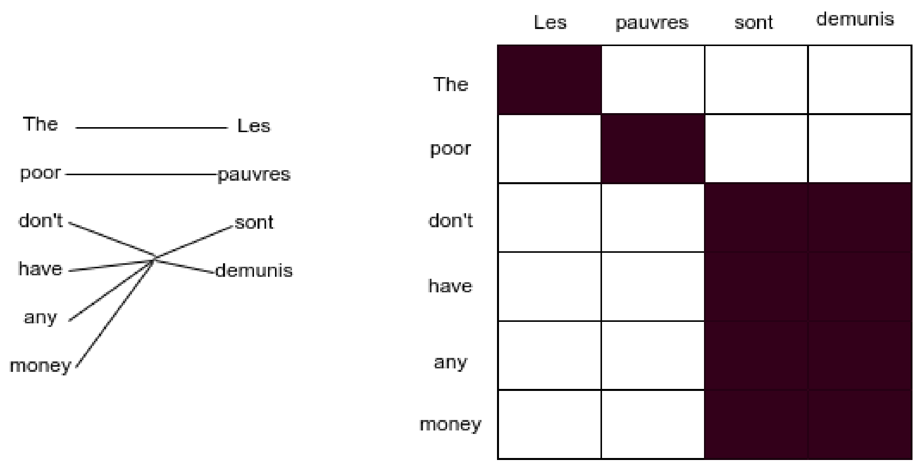


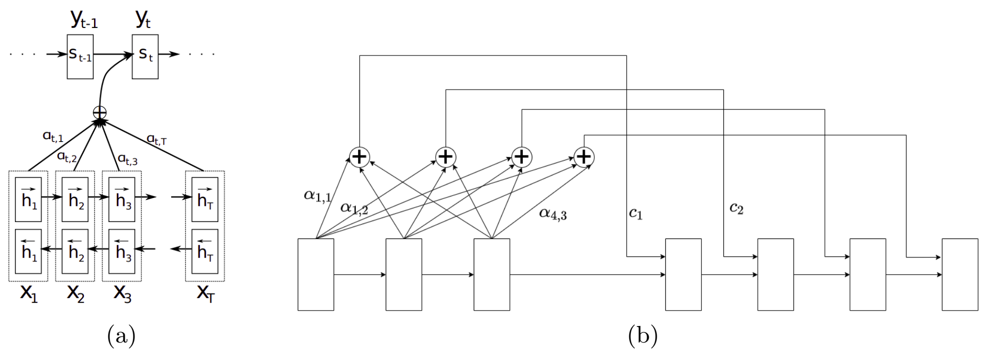
Figure 30: many-to-many

16.2 Fixing Problems with Attention Mechanism

The attention mechanism was born to help memorize long source sentences in neural machine translation (NMT). Rather than building a single context vector out of the encoder's last hidden state, the secret sauce invented by attention is to create shortcuts between the context vector and the entire source input. The weights of these shortcut connections are customizable for each output element. While the context vector has access to the entire input sequence, we don't need to worry about forgetting. The alignment between the source and target is learned and controlled by the context vector. Essentially the context vector consumes three pieces of information; encoder hidden states, decoder hidden states, alignment between source and target. [12]

16.3 Neural Machine Translation by Jointly Learning to Align and Translate

Neural Machine Translation with attention mechanism allows us to learn alignments insted of defining phrase based alignments as in Statistical Machine Translation. Attention mechanism is not only for NMT, it can be applied into Language Modeling, Question Answering and other tasks in NLP. Also, since we showed alignments in Statistical Machine Translation, the attentions are binary, not weighted. In linguistics, words are related and dependent, and dependencies may have importance. Learning attention in NMT also allows us to learn those weighted dependencies. In Neural Machine Translation by Jointly Learning to Align and Translate [12], the additive attention is proposed. The illustration of this mechanism can be shown,



y_{t-1}
y_t
s_{t-1}
s_t
$\alpha_{t,1}$
$\alpha_{t,2}$
$\alpha_{t,3}$
$\alpha_{t,T}$
$\overrightarrow{h}_1$ $\overrightarrow{h}_2$ $\overrightarrow{h}_3$ $\overrightarrow{h}_T$
$\overleftarrow{h}_1$ $\overleftarrow{h}_2$ $\overleftarrow{h}_3$ $\overleftarrow{h}_T$
x_1 x_2 x_3 x_T
(a)
$\alpha_{1,1}$
$\alpha_{1,2}$
$\alpha_{4,3}$
c_1
c_2
(b)

Figure 31: Additive Attention

To make a formal definition, we have input sequence $\mathbf{x}$ with length of n and output sequence $\mathbf{y}$ with length of m,

$$\mathbf{x} = [x_1, x_2, ..., x_n] \tag{82}$$
$$\mathbf{y} = [y_1, y_2, ..., x_m] \tag{83}$$

Bi-directional encoder state is

$$\boldsymbol{h}_i = [\overrightarrow{\boldsymbol{h}}_i^\top ; \overleftarrow{\boldsymbol{h}}_i^\top]^\top, i = 1, \dots, n \tag{84}$$

Let's denote decoder hidden states as s_t which is a function of

$$\boldsymbol{s}_t = f(\boldsymbol{s}_{t-1}, y_{t-1}, \mathbf{c}_t)$$

The context vector c_t is defined as

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{t,i} \mathbf{h_i} \tag{85}$$

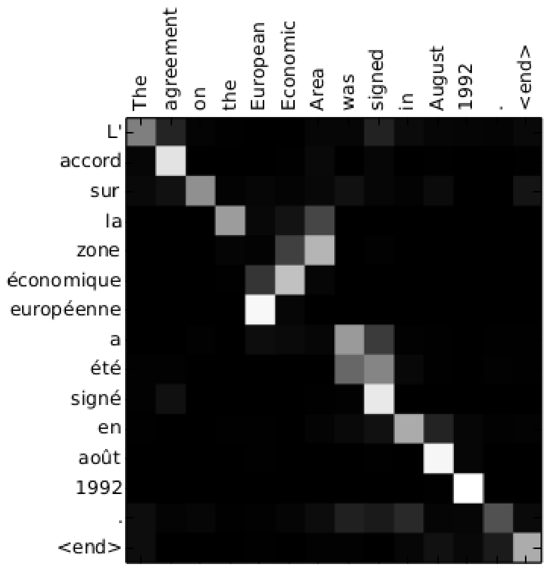
The term $\alpha_{t,i}$ represents "How well two words y_t and x_i are aligned", and defined as

$$\alpha_{t,i} = \text{align}(y_t, x_i) = \frac{\exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i))}{\sum_{j=1}^{n} \exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_j))} \tag{86}$$

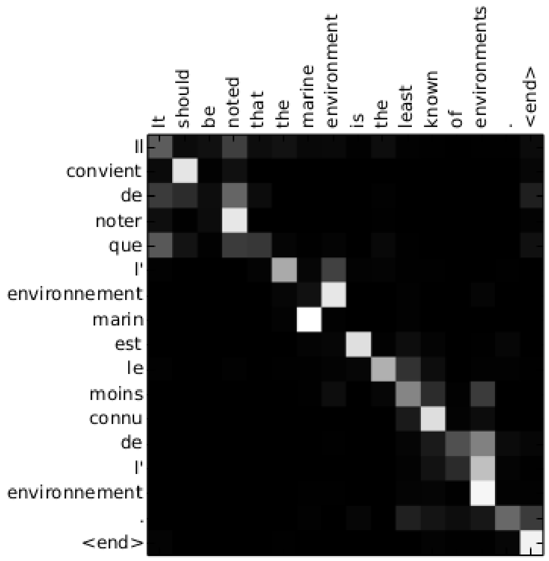
The alignment model assigns a score $\alpha_{t,i}$ to the pair of input at position i and output at position t, (y_t, x_i), based on how well they match. The set of $\alpha_{t,i}$ are weights defining how much of each source hidden state should be considered for each output. Since this alignment is built with softmax, it can be seen as a classification between pairs. Score function is a scoring function between (y_t, x_i) pairs.

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i]) \tag{87}$$

where both $\mathbf{v}$ and $\mathbf{W}_a$ are weight matrices to be learned in the alignment model. The matrix of alignment scores is a nice byproduct to explicitly show the correlation between source and target words.

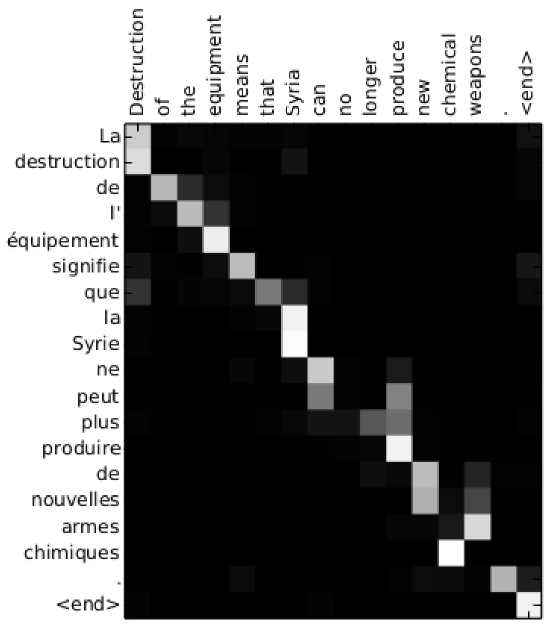


(a)

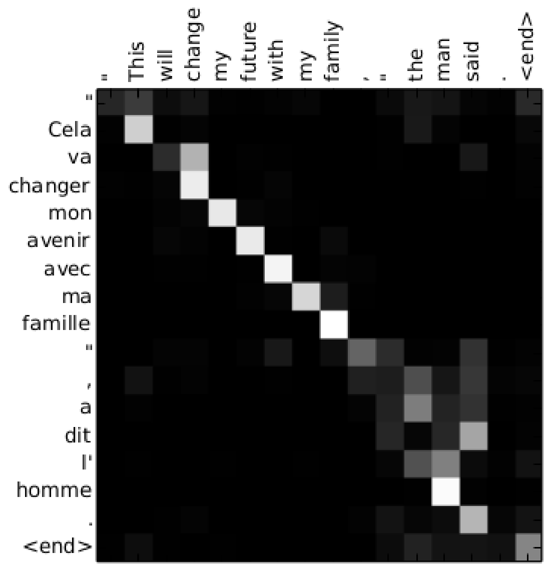


(b)

Figure 32: illustrated attention [12]



(c)



(d)

Figure 33: illustrated attention [12]

16.4 Types Of Attention Mechanism (source)

- **Content-base attention**

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$$

- **Location-Base**:

$$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$$

- **General**:
$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$$
- **Dot-Product**:
$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$$
- **Scaled Dot-Product**:
$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$$

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[2] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: 1502.05767 [cs.SC].

[3] Matthew E. Peters et al. *Deep contextualized word representations*. 2018. arXiv: 1802.05365 [cs.CL].

[4] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer International Publishing, 2018.

[5] William Merrill. *Formal Language Theory Meets Modern NLP*. 2021. arXiv: 2102.10094 [cs.CL].

[6] Graham Neubig et al. *DyNet: The Dynamic Neural Network Toolkit*. 2017. arXiv: 1701.03980 [stat.ML].

[7] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 1st. USA: Prentice Hall PTR, 2000. ISBN: 0130950696.

[8] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG].

[9] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[10] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[11] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL].

[12] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL].