CS224n: Natural Language Processing with Deep Learning[1]

Lecture Notes: Part V[2]

Winter 2017

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Authors: Milad Mohammadi, Rohit Mundra, Richard Socher, Lisa Wang

Keyphrases: Language Models. RNN. Bi-directional RNN. Deep RNN. GRU. LSTM.

1 Language Models

Language models compute the probability of occurrence of a number of words in a particular sequence. The probability of a sequence of m words $\{w_1, ..., w_m\}$ is denoted as $P(w_1, ..., w_m)$. Since the number of words coming before a word, w_i, varies depending on its location in the input document, $P(w_1, ..., w_m)$ is usually conditioned on a window of n previous words rather than all previous words:

$$P(w_1, ..., w_m) = \prod_{i=1}^{i=m} P(w_i | w_1, ..., w_{i-1}) \approx \prod_{i=1}^{i=m} P(w_i | w_{i-n}, ..., w_{i-1}) \quad (1)$$

Equation 1 is especially useful for speech and translation systems when determining whether a word sequence is an accurate translation of an input sentence. In existing language translation systems, for each phrase / sentence translation, the software generates a number of alternative word sequences (e.g. *{I have, I had, I has, me have, me had}*) and scores them to identify the most likely translation sequence.

In machine translation, the model chooses the best word ordering for an input phrase by assigning a *goodness* score to each output word sequence alternative. To do so, the model may choose between different word ordering or word choice alternatives. It would achieve this objective by running all word sequence candidates through a probability function that assigns each a score. The sequence with the highest score is the output of the translation. For example, the machine would give a higher score to *"the cat is small"* compared to *"small the is cat"*, and a higher score to *"walking home after school"* compare do *"walking house after school"*. To compute these probabilities, the count of each n-gram would be compared against the frequency of each word. For instance, if the model takes bi-grams, the frequency of each bi-gram, calculated via combining a word with its previous word, would be divided by the frequency of the corresponding uni-gram. Equations 2 and 3 show this relationship for

bigram and trigram models.

$$p(w_2|w_1) = \frac{count(w_1, w_2)}{count(w_1)} \tag{2}$$

$$p(w_3|w_1, w_2) = \frac{count(w_1, w_2, w_3)}{count(w_1, w_2)} \tag{3}$$

The relationship in Equation 3 focuses on making predictions based on a fixed window of context (i.e. the n previous words) used to predict the next word. In some cases, the window of past consecutive n words may not be sufficient to capture the context. For instance, consider a case where an article discusses the history of Spain and France and somewhere later in the text, it reads "The two countries went on a battle"; clearly the information presented in this sentence alone is not sufficient to identify the name of the two countries. Bengio et al. introduced the first large-scale deep learning for natural language processing model that enables capturing this type of context via *learning a distributed representation of words*; Figure 1 shows the corresponding neural network architecture. In this model, input word vectors are used by both the hidden layer and the output layer. Equation 4 shows the parameters of the softmax() function consisting of the standard tanh() function (i.e. the hidden layer) as well as the linear function, $W^{(3)}x + b^{(3)}$, that captures all the previous n input word vectors.

$$\hat{y} = softmax(W^{(2)}tanh(W^{(1)}x + b^{(1)}) + W^{(3)}x + b^{(3)}) \tag{4}$$

Note that the weight matrix $W^{(1)}$ is applied to the word vectors (solid green arrows in Figure 1), $W^{(2)}$ is applied to the hidden layer (also solid green arrow) and $W^{(3)}$ is applied to the word vectors (dashed green arrows).

In all conventional language models, the memory requirements of the system grows exponentially with the window size n making it nearly impossible to model large word windows without running out of memory.

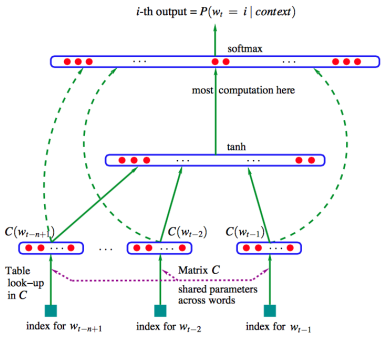

Figure 1: The first deep neural network architecture model for NLP presented by Bengio et al.

2 *Recurrent Neural Networks (RNN)*

Unlike the conventional translation models, where only a finite window of previous words would be considered for conditioning the language model, Recurrent Neural Networks (RNN) are capable of conditioning the model on *all* previous words in the corpus.

Figure 2 introduces the RNN architecture where rectangular box is a hidden layer at a time-step, t. Each such layer holds a number of neurons, each of which performing a linear matrix operation on

its inputs followed by a non-linear operation (e.g. tanh()). At each time-step, the output of the previous step along with the next word vector in the document, x_t, are inputs to the hidden layer to produce a prediction output $\hat{y}$ and output features h_t (Equations 5 and 6). The inputs and outputs of each single neuron are illustrated in Figure 3.

$$h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_{[t]}) \tag{5}$$

$$\hat{y}_t = softmax(W^{(S)}h_t) \tag{6}$$

Below are the details associated with each parameter in the network:

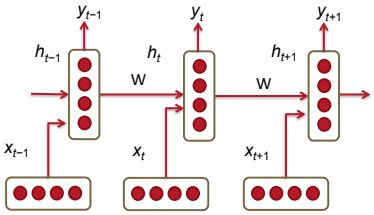

Figure 2: A Recurrent Neural Network (RNN). Three time-steps are shown.

- $x_1, ..., x_{t-1}, x_t, x_{t+1}, ...x_T$: the word vectors corresponding to a corpus with T words.
- $h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$: the relationship to compute the hidden layer output features at each time-step t
  - $x_t \in \mathbb{R}^d$: input word vector at time t.
  - $W^{hx} \in \mathbb{R}^{D_h \times d}$: weights matrix used to condition the input word vector, x_t
  - $W^{hh} \in \mathbb{R}^{D_h \times D_h}$: weights matrix used to condition the output of the previous time-step, h_{t-1}
  - $h_{t-1} \in \mathbb{R}^{D_h}$: output of the non-linear function at the previous time-step, $t-1$. $h_0 \in \mathbb{R}^{D_h}$ is an initialization vector for the hidden layer at time-step $t = 0$.
  - $\sigma()$: the non-linearity function (sigmoid here)
- $\hat{y}_t = softmax(W^{(S)}h_t)$: the output probability distribution over the vocabulary at each time-step t. Essentially, $\hat{y}_t$ is the next predicted word given the document context score so far (i.e. h_{t-1}) and the last observed word vector $x^{(t)}$. Here, $W^{(S)} \in \mathbb{R}^{|V| \times D_h}$ and $\hat{y} \in \mathbb{R}^{|V|}$ where $|V|$ is the vocabulary.

The loss function used in RNNs is often the cross entropy error introduced in earlier notes. Equation 7 shows this function as the sum over the entire vocabulary at time-step t.

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j}) \tag{7}$$

The cross entropy error over a corpus of size T is:

$$J = -\frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j}) \tag{8}$$

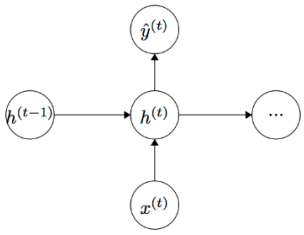

Figure 3: The inputs and outputs to a neuron of a RNN

Equation 9 is called the *perplexity* relationship; it is basically 2 to the power of the negative log probability of the cross entropy error function shown in Equation 8. Perplexity is a measure of confusion where lower values imply more confidence in predicting the next word in the sequence (compared to the ground truth outcome).

$$Perplexity = 2^J \tag{9}$$

The amount of memory required to run a layer of RNN is proportional to the number of words in the corpus. For instance, a sentence with k words would have k word vectors to be stored in memory. Also, the RNN must maintain two pairs of W, b matrices. While the size of W could be very large, it does not scale with the size of the corpus (unlike the traditional language models). For a RNN with 1000 recurrent layers, the matrix would be 1000×1000 regardless of the corpus size.

Figure 4 is an alternative representation of RNNs used in some publications. It representsRNNs the RNN hidden layer as a loop.

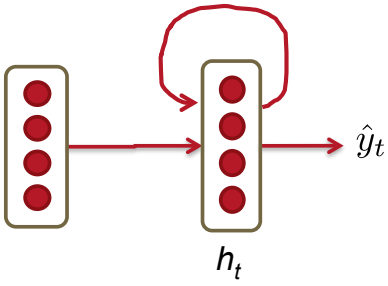

Figure 4: The illustration of a RNN as a loop over time-steps

2.1 *Vanishing Gradient & Gradient Explosion Problems*

Recurrent neural networks propagate weight matrices from one time-step to the next. Recall the goal of a RNN implementation is to enable propagating context information through faraway time-steps. For example, consider the following two sentences:

Sentence 1

"Jane walked into the room. John walked in too. Jane said hi to ___"

Sentence 2

"Jane walked into the room. John walked in too. It was late in the day, and everyone was walking home after a long day at work. Jane said hi to ___"

In both sentences, given their context, one can tell the answer to both blank spots is most likely "John". It is important that the RNN predicts the next word as "John", the second person who has appeared several time-steps back in both contexts. Ideally, this should be possible given what we know about RNNs so far. In practice, however, it turns out RNNs are more likely to correctly predict the blank spot in Sentence 1 than in Sentence 2. This is because during the back-propagation phase, the contribution of gradient values gradually vanishes as they propagate to earlier time-steps. Thus, for long sentences, the probability that "John" would be recognized as the next word reduces with the size of

the context. Below, we discuss the mathematical reasoning behind the vanishing gradient problem.

Consider Equations 5 and 6 at a time-step t; to compute the RNN error, dE/dW, we sum the error at each time-step. That is, dE_t/dW for every time-step, t, is computed and accumulated.

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W} \tag{10}$$

The error for each time-step is computed through applying the chain rule differentiation to Equations 6 and 5; Equation 11 shows the corresponding differentiation. Notice dh_t/dh_k refers to the partial derivative of h_t with respect to *all* previous k time-steps.

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W} \tag{11}$$

Equation 12 shows the relationship to compute each dh_t/dh_k; this is simply a chain rule differentiation over all hidden layers within the $[k, t]$ time interval.

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \times diag[f'(j_{j-1})] \tag{12}$$

Because $h \in \mathbb{R}^{D_n}$, each $\partial h_j/\partial h_{j-1}$ is the Jacobian matrix for h:

$$\frac{\partial h_j}{\partial h_{j-1}} = [\frac{\partial h_j}{\partial h_{j-1,1}} \cdots \frac{\partial h_j}{\partial h_{j-1,D_n}}] = \begin{bmatrix} \frac{\partial h_{j,1}}{\partial h_{j-1,1}} & \cdot & \cdot & \cdot & \frac{\partial h_{j,1}}{\partial h_{j-1,D_n}} \\ \cdot & \cdot & & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & & \cdot & \cdot \\ \frac{\partial h_{j,D_n}}{\partial h_{j-1,1}} & \cdot & \cdot & \cdot & \frac{\partial h_{j,D_n}}{\partial h_{j-1,D_n}} \end{bmatrix} \tag{13}$$

Putting Equations 10, 11, 12 together, we have the following relationship.

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} (\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}) \frac{\partial h_k}{\partial W} \tag{14}$$

Equation 15 shows the norm of the Jacobian matrix relationship in Equation 13. Here, β_W and β_h represent the upper bound values for the two matrix norms. The norm of the partial gradient at each time-step, t, is therefore, calculated through the relationship shown in Equation 15.

$$\| \frac{\partial h_j}{\partial h_{j-1}} \| \leq \| W^T \| \| diag[f'(h_{j-1})] \| \leq \beta_W \beta_h \tag{15}$$

The norm of both matrices is calculated through taking their L2-norm. The norm of $f'(h_{j-1})$ can only be as large as 1 given the sigmoid non-linearity function.

$$\| \frac{\partial h_t}{\partial h_k} \| = \| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \| \leq (\beta_W \beta_h)^{t-k} \tag{16}$$

The exponential term $(\beta_W \beta_h)^{t-k}$ can easily become a very small or large number when $\beta_W \beta_h$ is much smaller or larger than 1 and $t - k$ is sufficiently large. Recall a large $t - k$ evaluates the cross entropy error due to faraway words. The contribution of faraway words to predicting the next word at time-step t diminishes when the gradient vanishes early on.

During experimentation, once the gradient value grows extremely large, it causes an overflow (i.e. NaN) which is easily detectable at runtime; this issue is called the *Gradient Explosion Problem*. When the gradient value goes to zero, however, it can go undetected while drastically reducing the learning quality of the model for far-away words in the corpus; this issue is called the *Vanishing Gradient Problem*.

To gain practical intuition about the vanishing gradient problem, you may visit the following example website.

2.2 *Solution to the Exploding & Vanishing Gradients*

Now that we gained intuition about the nature of the vanishing gradients problem and how it manifests itself in deep neural networks, let us focus on a simple and practical heuristic to solve these problems.

To solve the problem of exploding gradients, Thomas Mikolov first introduced a simple heuristic solution that *clips* gradients to a small number whenever they explode. That is, whenever they reach a certain threshold, they are set back to a small number as shown in Algorithm 1.

Algorithm 1: Psudo-code for norm clipping in the gradients whenever they explode

$\hat{g} \leftarrow \frac{\partial E}{\partial W}$
if $\| \hat{g} \| \geq threshold$ **then**
  $\hat{g} \leftarrow \frac{threshold}{\| \hat{g} \|} \hat{g}$
end if

Figure 5 visualizes the effect of gradient clipping. It shows the decision surface of a small recurrent neural network with respect to its W matrix and its bias terms, b. The model consists of a single unit of recurrent neural network running through a small number of time-steps; the solid arrows illustrate the training progress on each gradient descent step. When the gradient descent model hits the high error wall

in the objective function, the gradient is pushed off to a far-away location on the decision surface. The clipping model produces the dashed line where it instead pulls back the error gradient to somewhere close to the original gradient landscape.

To solve the problem of vanishing gradients, we introduce two techniques. The first technique is that instead of initializing $W^{(hh)}$ randomly, start off from an identify matrix initialization.

The second technique is to use the Rectified Linear Units (ReLU) instead of the sigmoid function. The derivative for the ReLU is either 0 or 1. This way, gradients would flow through the neurons whose derivative is 1 without getting attenuated while propagating back through time-steps.

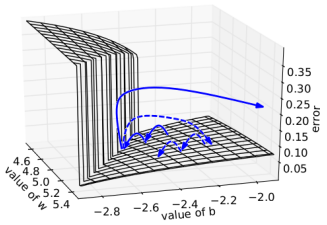

Figure 5: Gradient explosion clipping visualization

2.3 *Deep Bidirectional RNNs*

So far, we have focused on RNNs that look into the past words to predict the next word in the sequence. It is possible to make predictions based on future words by having the RNN model read through the corpus backwards. Irsoy et al. shows a bi-directional deep neural network; at each time-step, t, this network maintains two hidden layers, one for the left-to-right propagation and another for the right-to-left propagation. To maintain two hidden layers at any time, this network consumes twice as much memory space for its weight and bias parameters. The final classification result, $\hat{y}_t$, is generated through combining the score results produced by both RNN hidden layers. Figure 6 shows the bi-directional network architecture, and Equations 17 and 18 show the mathematical formulation behind setting up the bi-directional RNN hidden layer. The only difference between these two relationships is in the direction of recursing through the corpus. Equation 19 shows the classification relationship used for predicting the next word via summarizing past and future word representations.

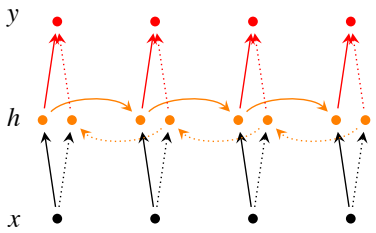

Figure 6: A bi-directional RNN model

$$\overrightarrow{h}_t = f(\overrightarrow{W}x_t + \overrightarrow{V}\overrightarrow{h}_{t-1} + \overrightarrow{b}) \tag{17}$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b}) \tag{18}$$

$$\hat{y}_t = g(Uh_t + c) = g(U[\overrightarrow{h}_t; \overleftarrow{h}_t] + c) \tag{19}$$

Figure 7 shows a multi-layer bi-directional RNN where each lower layer feeds the next layer. As shown in this figure, in this network architecture, at time-step t each intermediate neuron receives one set of parameters from the previous time-step (in the same RNN layer), and two sets of parameters from the previous RNN hidden layer; one

input comes from the left-to-right RNN and the other from the right-to-left RNN.

To construct a Deep RNN with L layers, the above relationships are modified to the relationships in Equations 20 and 21 where the input to each intermediate neuron at level i is the output of the RNN at layer $i-1$ at the same time-step, t. The output, $\hat{y}$, at each time-step is the result of propagating input parameters through all hidden layers (Equation 22).

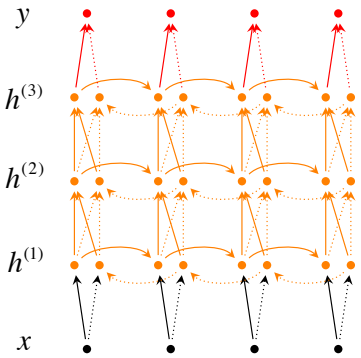

Figure 7: A deep bi-directional RNN with three RNN layers.

$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)}) \tag{20}$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)}) \tag{21}$$

$$\hat{y}_t = g(Uh_t + c) = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c) \tag{22}$$

2.4 *Application: RNN Translation Model*

Traditional translation models are quite complex; they consist of numerous machine learning algorithms applied to different stages of the language translation pipeline. In this section, we discuss the potential for adopting RNNs as a replacement to traditional translation modules. Consider the RNN example model shown in Figure 8; here, the German phrase *Echt dicke Kiste* is translated to *Awesome sauce*. The first three hidden layer time-steps *encode* the German language words into some language word features (h_3). The last two time-steps *decode* h_3 into English word outputs. Equation 23 shows the relationship for the Encoder stage and Equations 24 and 25 show the equation for the Decoder stage.

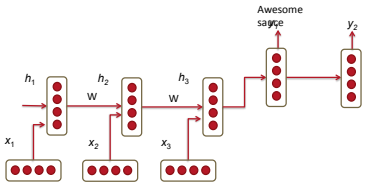

Figure 8: A RNN-based translation model. The first three RNN hidden layers belong to the source language model encoder, and the last two belong to the destination language model decoder.

$$h_t = \phi(h_{t-1}, x_t) = f(W^{(hh)} h_{t-1} + W^{(hx)} x_t) \tag{23}$$

$$h_t = \phi(h_{t-1}) = f(W^{(hh)} h_{t-1}) \tag{24}$$

$$y_t = softmax(W^{(S)} h_t) \tag{25}$$

One may naively assume this RNN model along with the cross-entropy function shown in Equation 26 can produce high-accuracy translation results. In practice, however, several extensions are to be added to the model to improve its translation accuracy performance.

$$max_\theta \frac{1}{N} \sum_{n=1}^{N} \times log(p_\theta(y^{(n)}|x^{(n)})) \tag{26}$$

Extension I: train different RNN weights for encoding and decoding. This decouples the two units and allows for more accuracy prediction of each of the two RNN modules. This means the $\phi()$ functions in Equations 23 and 24 would have different $W^{(hh)}$ matrices.

Extension II: compute every hidden state in the decoder using three different inputs:

- The previous hidden state (standard)
- Last hidden layer of the encoder ($c = h_T$ in Figure 9)
- Previous predicted output word, $\hat{y}_{t-1}$

Combining the above three inputs transforms the ϕ function in the decoder function of Equation 24 to the one in Equation 27. Figure 9 illustrates this model.

$$h_t = \phi(h_{t-1}, c, y_{t-1}) \tag{27}$$

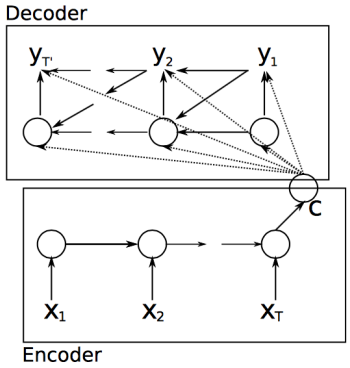

Figure 9: Language model with three inputs to each decoder neuron: (h_{t-1}, c, y_{t-1})

Extension III: train deep recurrent neural networks using multiple RNN layers as discussed earlier in this chapter. Deeper layers often improve prediction accuracy due to their higher learning capacity. Of course, this implies a large training corpus must be used to train the model.

Extension IV: train bi-directional encoders to improve accuracy similar to what was discussed earlier in this chapter.

Extension V: given a word sequence $A\ B\ C$ in German whose translation is $X\ Y$ in English, instead of training the RNN using $A\ B\ C \rightarrow X\ Y$, train it using $C\ B\ A \rightarrow X\ Y$. The intutition behind this technique is that A is more likely to be translated to X. Thus, given the vanishing gradient problem discussed earlier, reversing the order of the input words can help reduce the error rate in generating the output phrase.

3 *Gated Recurrent Units*

Beyond the extensions discussed so far, RNNs have been found to perform better with the use of more complex units for activation. So far, we have discussed methods that transition from hidden state h_{t-1} to h_t using an affine transformation and a point-wise nonlinearity. Here, we discuss the use of a gated activation function thereby modifying the RNN architecture. What motivates this? Well, although RNNs can theoretically capture long-term dependencies, they are very hard

to actually train to do this. Gated recurrent units are designed in a manner to have more persistent memory thereby making it easier for RNNs to capture long-term dependencies. Let us see mathematically how a GRU uses h_{t-1} and x_t to generate the next hidden state h_t. We will then dive into the intuition of this architecture.

$$
\begin{aligned}
z_t &= \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) && \text{(Update gate)} \\
r_t &= \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) && \text{(Reset gate)} \\
\tilde{h}_t &= \tanh(r_t \circ Uh_{t-1} + Wx_t) && \text{(New memory)} \\
h_t &= (1 - z_t) \circ \tilde{h}_t + z_t \circ h_{t-1} && \text{(Hidden state)}
\end{aligned}
$$

The above equations can be thought of a GRU's four fundamental operational stages and they have intuitive interpretations that make this model much more intellectually satisfying (see Figure 10):

1. **New memory generation:** A new memory $\tilde{h}_t$ is the consolidation of a new input word x_t with the past hidden state h_{t-1}. Anthropomorphically, this stage is the one who knows the recipe of combining a newly observed word with the past hidden state h_{t-1} to summarize this new word in light of the contextual past as the vector $\tilde{h}_t$.

2. **Reset Gate:** The reset signal r_t is responsible for determining how important h_{t-1} is to the summarization $\tilde{h}_t$. The reset gate has the ability to completely diminish past hidden state if it finds that h_{t-1} is irrelevant to the computation of the new memory.

3. **Update Gate:** The update signal z_t is responsible for determining how much of h_{t-1} should be carried forward to the next state. For instance, if $z_t \approx 1$, then h_{t-1} is almost entirely copied out to h_t. Conversely, if $z_t \approx 0$, then mostly the new memory $\tilde{h}_t$ is forwarded to the next hidden state.

4. **Hidden state:** The hidden state h_t is finally generated using the past hidden input h_{t-1} and the new memory generated $\tilde{h}_t$ with the advice of the update gate.

It is important to note that to train a GRU, we need to learn all the different parameters: $W, U, W^{(r)}, U^{(r)}, W^{(z)}, U^{(z)}$. These follow the same backpropagation procedure we have seen in the past.

4 *Long-Short-Term-Memories*

Long-Short-Term-Memories are another type of complex activation unit that differ a little from GRUs. The motivation for using these is similar to those for GRUs however the architecture of such units does differ.

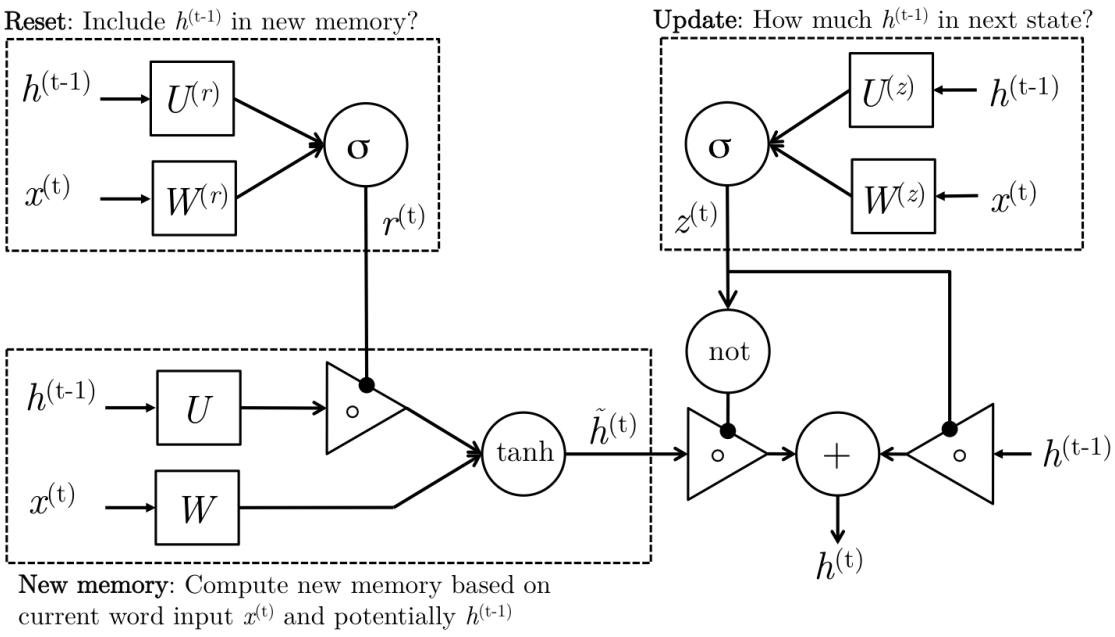

Figure 10: The detailed internals of a GRU

Let us first take a look at the mathematical formulation of LSTM units before diving into the intuition behind this design:

$$
\begin{aligned}
i_t &= \sigma(W^{(i)}x_t + U^{(i)}h_{t-1}) && \text{(Input gate)} \\
f_t &= \sigma(W^{(f)}x_t + U^{(f)}h_{t-1}) && \text{(Forget gate)} \\
o_t &= \sigma(W^{(o)}x_t + U^{(o)}h_{t-1}) && \text{(Output/Exposure gate)} \\
\tilde{c}_t &= \tanh(W^{(c)}x_t + U^{(c)}h_{t-1}) && \text{(New memory cell)} \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t && \text{(Final memory cell)} \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

We can gain intuition of the structure of an LSTM by thinking of its architecture as the following stages:

1. **New memory generation:** This stage is analogous to the new memory generation stage we saw in GRUs. We essentially use the input word x_t and the past hidden state h_{t-1} to generate a new memory $\tilde{c}_t$ which includes aspects of the new word $x^{(t)}$.

2. **Input Gate:** We see that the new memory generation stage doesn't check if the new word is even important before generating the new memory – this is exactly the input gate's function. The input gate uses the input word and the past hidden state to determine whether or not the input is worth preserving and thus is used to gate the new memory. It thus produces i_t as an indicator of this information.

3. **Forget Gate:** This gate is similar to the input gate except that it does not make a determination of usefulness of the input word –

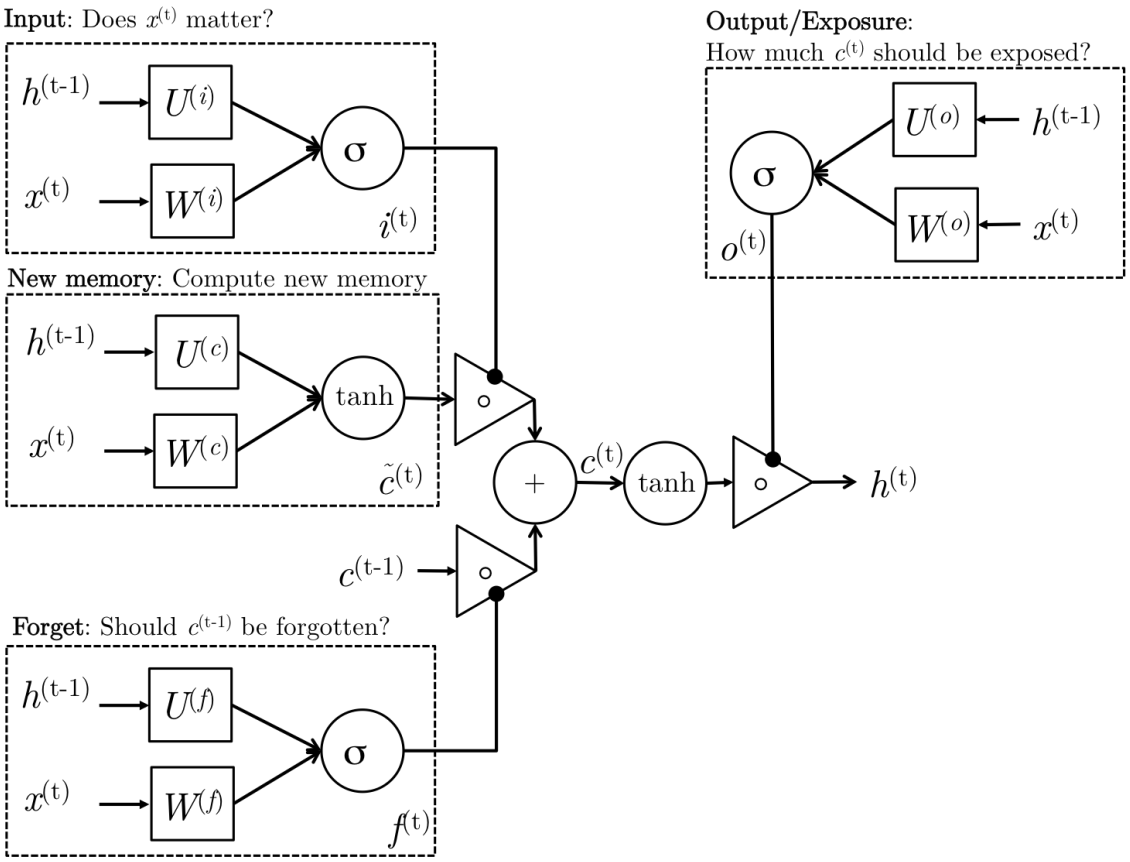

Figure 11: The detailed internals of a LSTM

instead it makes an assessment on whether the past memory cell is useful for the computation of the current memory cell. Thus, the forget gate looks at the input word and the past hidden state and produces f_t.

4. **Final memory generation:** This stage first takes the advice of the forget gate f_t and accordingly forgets the past memory c_{t-1}. Similarly, it takes the advice of the input gate i_t and accordingly gates the new memory $\tilde{c}_t$. It then sums these two results to produce the final memory c_t.

5. **Output/Exposure Gate:** This is a gate that does not explicitly exist in GRUs. It's purpose is to separate the final memory from the hidden state. The final memory c_t contains a lot of information that is not necessarily required to be saved in the hidden state. Hidden states are used in every single gate of an LSTM and thus, this gate makes the assessment regarding what parts of the memory c_t needs to be exposed/present in the hidden state h_t. The signal it produces to indicate this is o_t and this is used to gate the point-wise tanh of the memory.