

CS224n: Natural Language Processing with Deep Learning¹

Lecture Notes: Part V

Language Models, RNN, GRU and LSTM²

Winter 2019

¹ Course Instructors: Christopher Manning, Richard Socher

² Authors: Milad Mohammadi, Rohit Mundra, Richard Socher, Lisa Wang, Amita Kamath

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

1 Language Models

1.1 Introduction

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

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

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. $\{I\text{ have}, I\text{ had}, I\text{ has}, me\text{ have}, me\text{ had}\}$) 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" compared to "walking house after school".

1.2 n -gram Language Models

To compute the probabilities mentioned above, the count of each n -gram could be compared against the frequency of each word. This is

called an n-gram Language Model. 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.

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

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

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. But how long should the context be? In some cases, the window of past consecutive n words may not be sufficient to capture the context. For instance, consider the sentence "As the proctor started the clock, the students opened their ____". If the window only conditions on the previous three words "the students opened their", the probabilities calculated based on the corpus may suggest that the next word be "books" - however, if n had been large enough to include the "proctor" context, the probability might have suggested "exam".

This leads us to two main issues with n-gram Language Models: Sparsity and Storage.

1. Sparsity problems with n-gram Language models

Sparsity problems with these models arise due to two issues.

Firstly, note the numerator of Equation 3. If w_1, w_2 and w_3 never appear together in the corpus, the probability of w_3 is 0. To solve this, a small δ could be added to the count for each word in the vocabulary. This is called *smoothing*.

Secondly, consider the denominator of Equation 3. If w_1 and w_2 never occurred together in the corpus, then no probability can be calculated for w_3 . To solve this, we could condition on w_2 alone. This is called *backoff*.

Increasing n makes sparsity problems worse. Typically, $n \leq 5$.

2. Storage problems with n-gram Language models

We know that we need to store the count for all n-grams we saw in the corpus. As n increases (or the corpus size increases), the model size increases as well.

1.3 Window-based Neural Language Model

The "curse of dimensionality" above was first tackled by Bengio et al in A Neural Probabilistic Language Model, which introduced the

first large-scale deep learning for natural language processing model. This model learns a *distributed representation of words*, along with the probability function for word sequences expressed in terms of these representations. Figure 1 shows the corresponding neural network architecture. The input word vectors are used by both the hidden layer and the output layer.

Equation 4 represents Figure 1 and 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.

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

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).

A simplified version of this model can be seen in Figure 2, where the blue layer signifies concatenated word embeddings for the input words: $e = [e^{(1)}; e^{(2)}; e^{(3)}; e^{(4)}]$, the red layer signifies the hidden layer: $h = f(We + b_1)$, and the green output distribution is a softmax over the vocabulary: $\hat{y} = \text{softmax}(Uh + b_2)$.

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 3 introduces the RNN architecture where each vertical rectangular box is a hidden layer at a time-step, t . Each such layer holds a number of neurons, each of which performs a linear matrix operation on its inputs followed by a non-linear operation (e.g. tanh()). At each time-step, there are two inputs to the hidden layer: the output of the previous layer h_{t-1} , and the input at that timestep x_t . The former input is multiplied by a weight matrix $W^{(hh)}$ and the latter by a weight matrix $W^{(hx)}$ to produce output features h_t , which are multiplied with a weight matrix $W^{(S)}$ and run through a softmax over the vocabulary to obtain a prediction output \hat{y} of the next word (Equations 5 and 6). The inputs and outputs of each single neuron are illustrated in Figure 4.

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

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

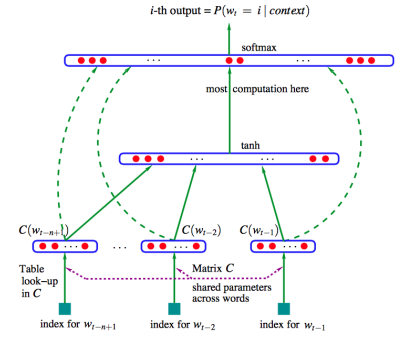


Figure 1: The first deep neural network architecture model for NLP presented by Bengio et al.

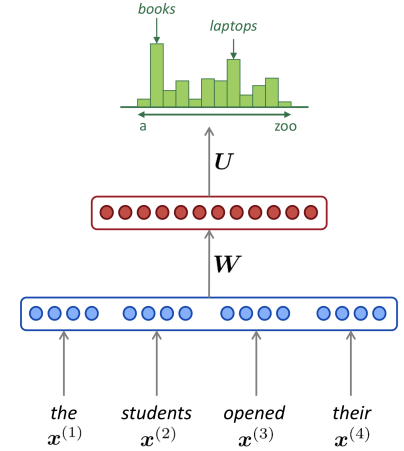


Figure 2: A simplified representation of Figure 1.

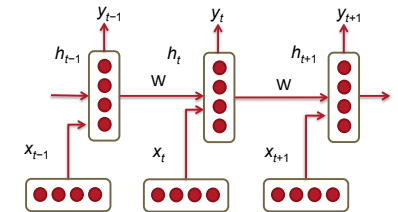


Figure 3: A Recurrent Neural Network (RNN). Three time-steps are shown.

What is interesting here is that the same weights $W^{(hh)}$ and $W^{(hx)}$ are applied repeatedly at each timestep. Thus, the number of parameters the model has to learn is less, and most importantly, is independent of the length of the input sequence - thus defeating the curse of dimensionality!

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

- $x_1, \dots, x_{t-1}, x_t, x_{t+1}, \dots, x_T$: 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 $t = 0$.
 - $\sigma(\cdot)$: the non-linearity function (sigmoid here)
- $\hat{y}_t = \text{softmax}(W^{(S)}h_t)$: 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.

An example of an RNN language model is shown in Figure 5. The notation in this image is slightly different: here, the equivalent of $W^{(hh)}$ is W_h , $W^{(hx)}$ is W_e , and $W^{(S)}$ is U . E converts word inputs $x^{(t)}$ to word embeddings $e^{(t)}$. The final softmax over the vocabulary shows us the probability of various options for token $x^{(5)}$, conditioned on all previous tokens. The input could be much longer than 4-5 tokens.

2.1 RNN Loss and Perplexity

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 .

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

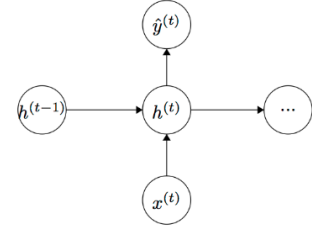


Figure 4: The inputs and outputs to a neuron of a RNN

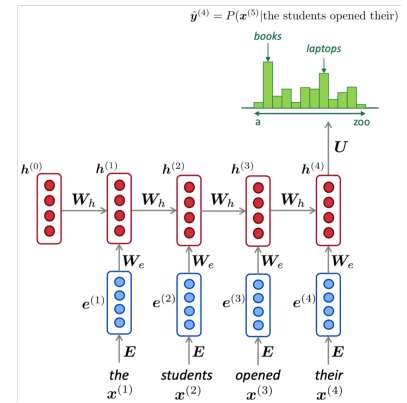


Figure 5: An RNN Language Model

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

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

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).

$$\text{Perplexity} = 2^J \quad (9)$$

2.2 Advantages, Disadvantages and Applications of RNNs

RNNs have several advantages:

1. They can process input sequences of any length
2. The model size does not increase for longer input sequence lengths
3. Computation for step t can (in theory) use information from many steps back.
4. The same weights are applied to every timestep of the input, so there is symmetry in how inputs are processed

However, they also have some disadvantages:

1. **Computation is slow** - because it is sequential, it cannot be parallelized
2. In practice, it is difficult to access information from many steps back due to problems like vanishing and exploding gradients, which we discuss in the following subsection

The amount of memory required to run a layer of RNN is proportional to the number of words in the corpus. We can consider a sentence as a minibatch, and 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. As aforementioned, 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.

RNNs can be used for many tasks, such as tagging (e.g. part-of-speech, named entity recognition), sentence classification (e.g. sentiment classification), and encoder modules (e.g. question answering,

machine translation, and many other tasks). In the latter two applications, we want a representation for the sentence, which we can obtain by taking the element-wise max or mean of all hidden states of the timesteps in that sentence.

Note: Figure 6 is an alternative representation of RNNs used in some publications. It represents the RNN hidden layer as a loop.

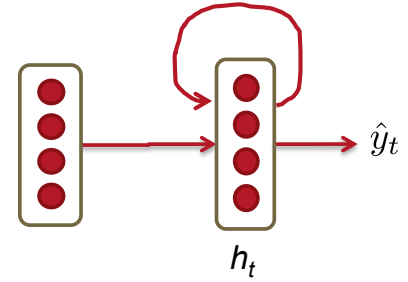


Figure 6: The illustration of a RNN as a loop over time-steps

2.3 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 timesteps, as we will show below. 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.

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

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.

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

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.

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

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

$$\frac{\partial h_j}{\partial h_{j-1}} = \left[\frac{\partial h_j}{\partial h_{j-1,1}} \dots \frac{\partial h_j}{\partial h_{j-1,D_n}} \right] = \begin{bmatrix} \frac{\partial h_{j,1}}{\partial h_{j-1,1}} & \cdot & \cdot & \cdot & \frac{\partial h_{j,1}}{\partial h_{j-1,D_n}} \\ \cdot & \cdot & & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & & \cdot & \cdot \\ \frac{\partial h_{j,D_n}}{\partial h_{j-1,1}} & \cdot & \cdot & \cdot & \frac{\partial h_{j,D_n}}{\partial h_{j-1,D_n}} \end{bmatrix} \quad (13)$$

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

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

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.

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

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.

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

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*. Due to vanishing gradients, we don't know whether there is no dependency between steps t and $t + n$ in the data, or we just cannot capture the true dependency due to this issue.

To gain practical intuition about the vanishing gradient problem, you may visit the following [example website](#).

2.4 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.

```

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

Figure 7 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 identity 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

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

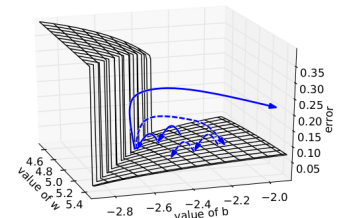


Figure 7: Gradient explosion clipping visualization

1. This way, gradients would flow through the neurons whose derivative is 1 without getting attenuated while propagating back through time-steps.

2.5 Deep Bidirectional RNNs

So far, we have focused on RNNs that condition on 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 8 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.

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

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

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

RNNs can also be multi-layered. Figure 9 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).

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

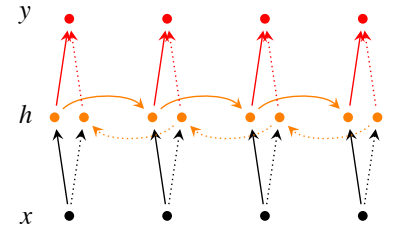


Figure 8: A bi-directional RNN model

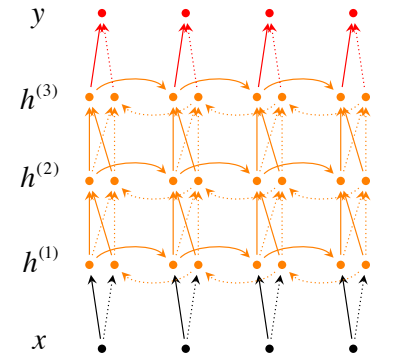


Figure 9: A deep bi-directional RNN with three RNN layers.

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

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

2.6 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 10; 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.

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

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

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

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.

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

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 $\phi(\cdot)$ 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 11)

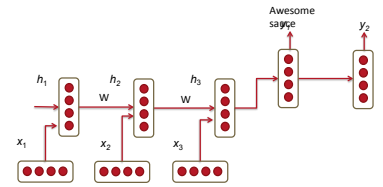


Figure 10: A RNN-based translation model. The first three RNN hidden layers belong to the source language model encoder, and the last two belong to the destination language model decoder.

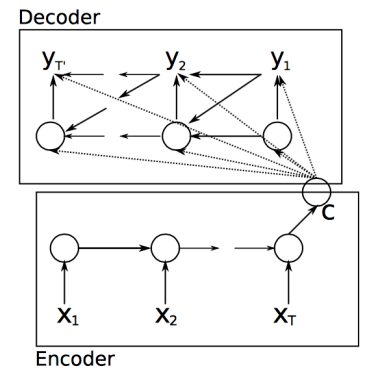


Figure 11: Language model with three inputs to each decoder neuron: (h_{t-1}, c, y_{t-1})

- 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 11 illustrates this model.


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

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 intuition 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 nonlinear . 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.

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

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 12):

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.

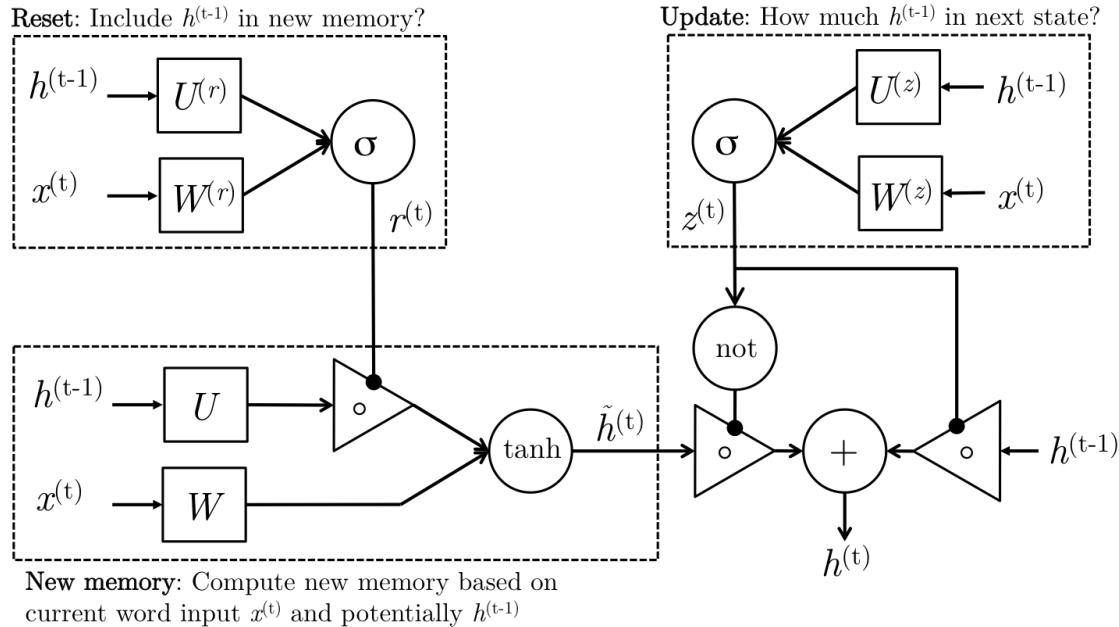


Figure 12: The detailed internals of a GRU

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

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1}) \quad (\text{Input gate})$$

$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1}) \quad (\text{Forget gate})$$

$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1}) \quad (\text{Output/Exposure gate})$$

$$\tilde{c}_t = \tanh(W^{(c)}x_t + U^{(c)}h_{t-1}) \quad (\text{New memory cell})$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \quad (\text{Final memory cell})$$

$$h_t = o_t \circ \tanh(c_t)$$

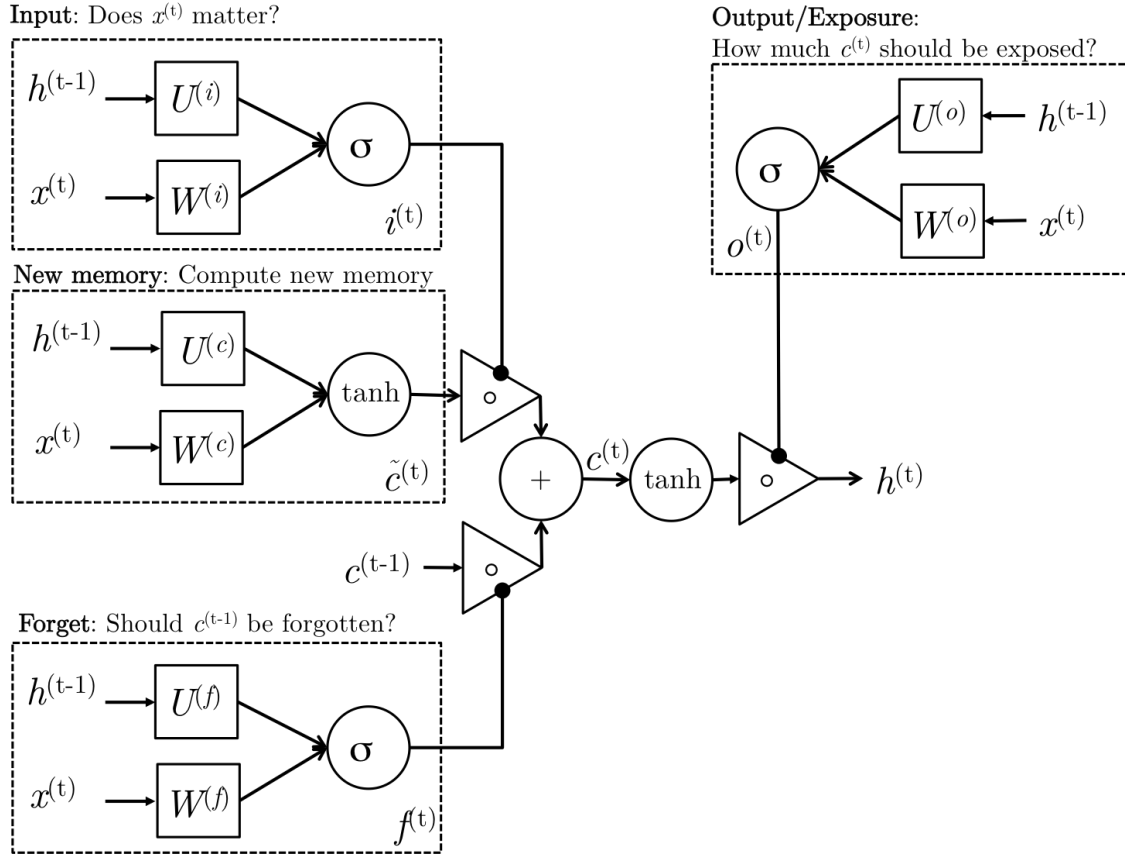


Figure 13: The detailed internals of a LSTM

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 – 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.