Traditional Language Models

J'ai vu le match de foot $\longrightarrow$

Sequence	$P(\text{Sequence})$
I saw the game of soccer	4.5 e-5
I saw the soccer game	6.0 e-5
I saw the soccer match	4.6 e-5
Saw I the game of soccer	2.6 e-9

Highest Probability

N-grams

$$P(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)} \longrightarrow \text{Bigrams}$$

$$P(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)} \longrightarrow \text{Trigrams}$$

$$\boxed{P(w_1, w_2, w_3) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_2)}$$

- Large N-grams to capture dependencies between distant words
- Need a lot of space and RAM

Advantages of RNNs

Nour was supposed to study with me. I called her but she did not ___have___

want
respond
choose
want
have
ask
attempt
answer
know

$\longrightarrow$ Similar probabilities with trigram

Advantages of RNNs

Nour was supposed to study with me. I called her but she did not answer

- want
- respond
- choose
- want
- have
- ask
- attempt
- answer
- know

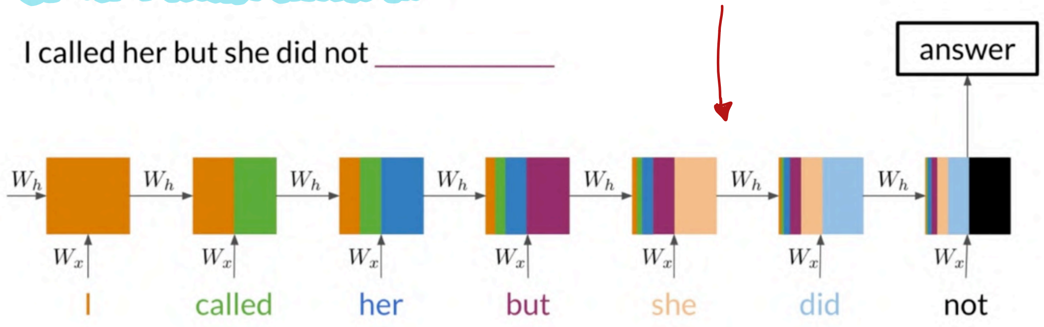
→ Similar probabilities with trigram

RNNs look at every previous word

As a better alternative, RNNs aren't limited to looking at just the previous n words. They propagate information from the beginning of the sentence to the end. So if you trained an RNN for this task, you'd get a better prediction, the word answer, even though that word is less probable. If you wanted to have a n-gram capable of completing sentences like this one, you would have to account for six-word-long sequences, which is pretty impractical. To see how recurrent neural networks work, take the second sentence from the last example. I called her but she did not blank. A plain RNN propagates information from the beginning of the sentence through to the end, starting with the first word of the sequence, the hidden value at the far left and the first values are computed here. Then it propagates some of the computed information, takes the second word in the sequence, and gets new values. You can see this process illustrated here. The orange area denotes the first computed values and the green denotes the second word. The second values are computed using the older values in orange and the new word in green. After that, it takes the third word and the propagated values from the first and second words, and computes another set of values from both of those and so on. Each of the boxes in this diagram represents the computations made at each step and the colors represent the information that is used for every computation. As you can see, the computations made at the last step have information from all the words in the sentence. At the final step, the recurrent neural network is able to predict the word answer.

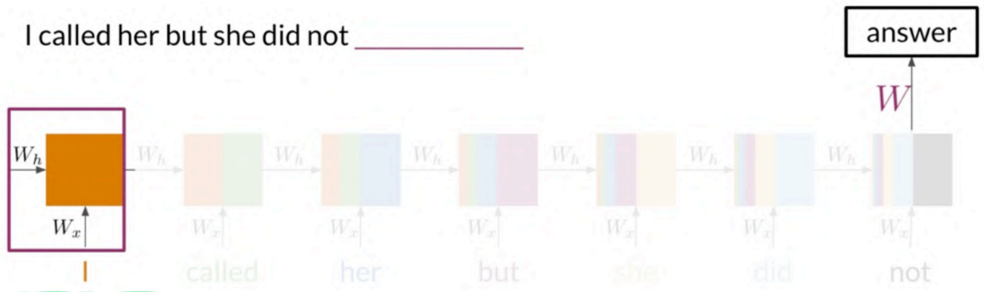
RNNs Basic Structure

I called her but she did not ________



RNNs Basic Structure

I called her but she did not ________

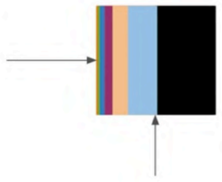


Learnable parameters

The magic of recurrent neural networks is that the information from every word in the sequence is multiplied by the same weight, W subscript of X, The information propagates it from the beginning to the end. It is multiplied by W subscript H. In other words, this block is repeated for every word in the sequence. So the only learnable parameters are the ones in W subscript X, W subscript H, and W - the weights used to make the final prediction. That's why they're called recurrent neural networks. They compute values that are fed over and over again to themselves until a prediction is made.

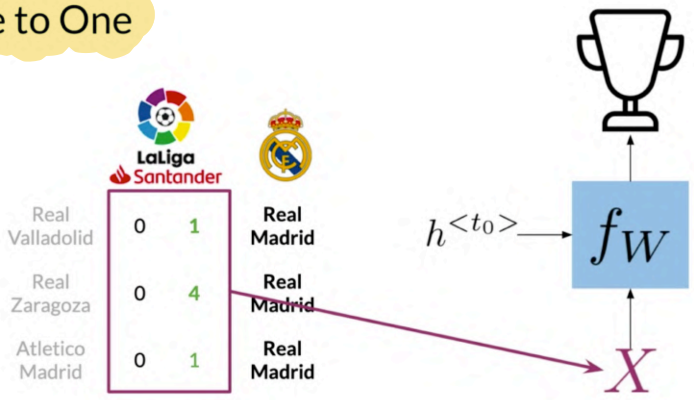
Summary

- RNNs model relationships among distant words
- In RNNs a lot of computations share parameters



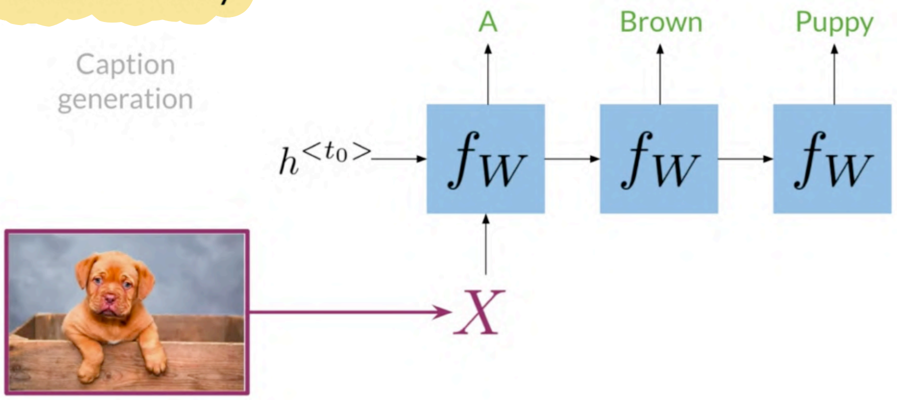
The principle advantage of RNNs is that they propagate information within sequences and the computations share most of the parameters. Here, you saw an example of an RNN that's propagated information from the beginning to the end of a word sequence to make a single prediction at the end.

One to One

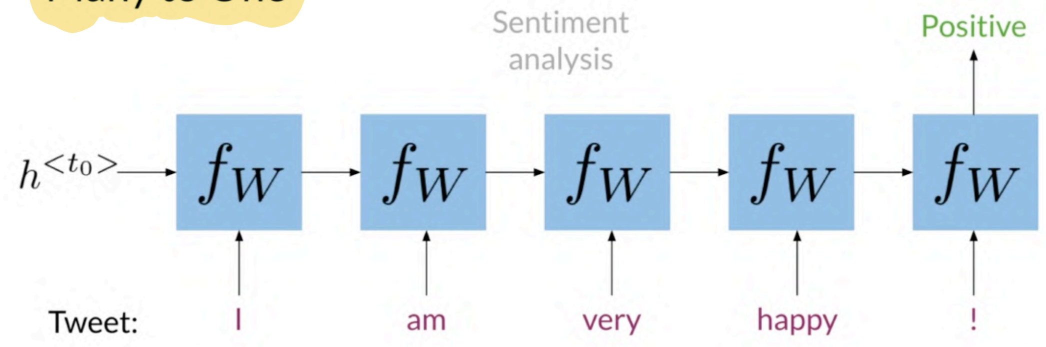


One to Many

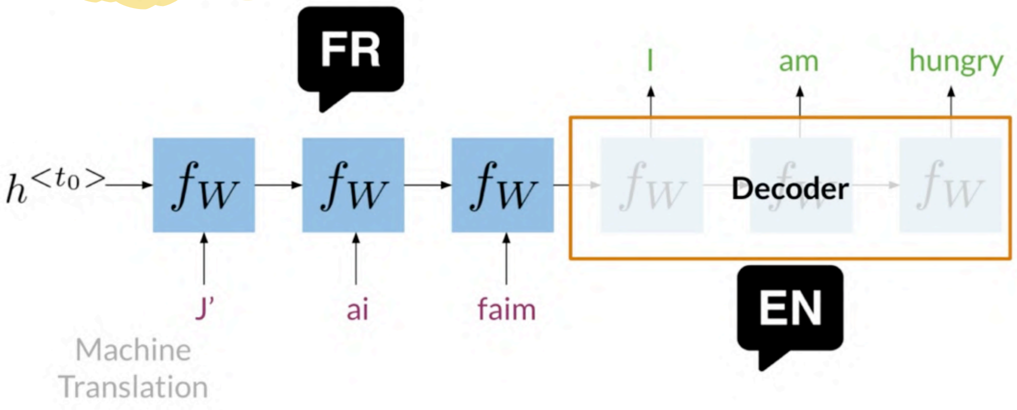
Caption generation



Many to One



Many to Many

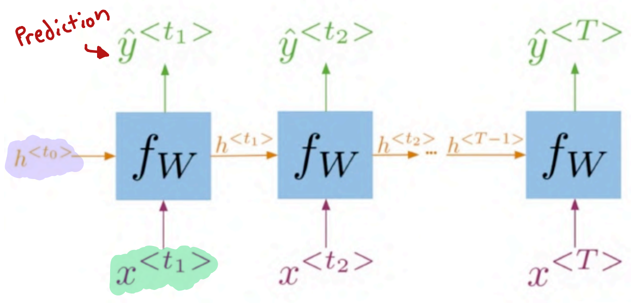


Summary

- RNNs can be implemented for a variety of NLP tasks
- Applications include Machine translation and caption generation

A Vanilla RNN

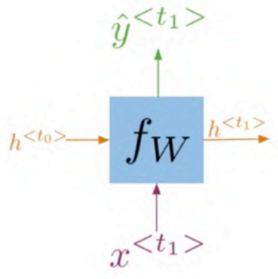
Many-to-Many



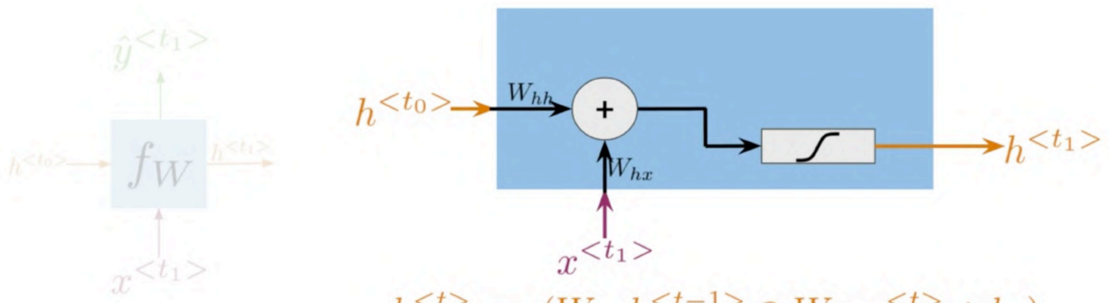
$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$
$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

$$\boxed{h^{<t>} = g(W_{hh}h^{<t-1>} \oplus W_{hx}x^{<t>} + b_h)}$$

A Vanilla RNN

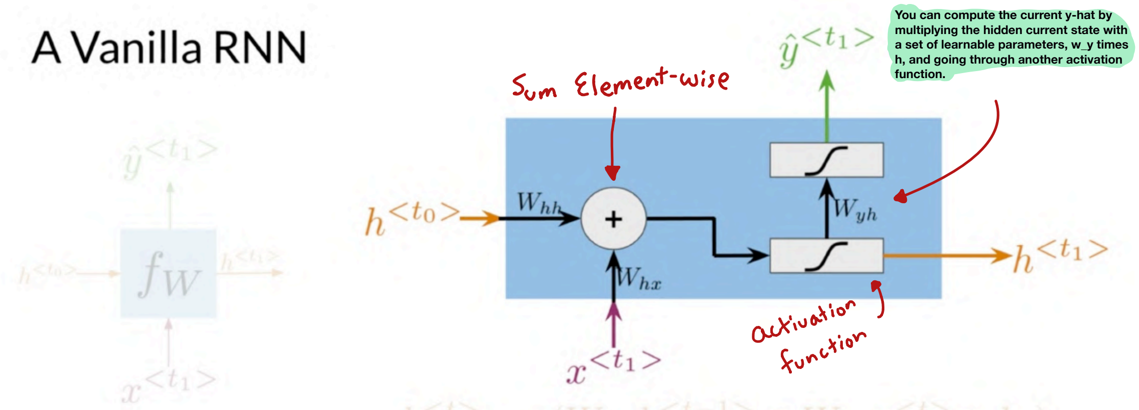


A Vanilla RNN



$$h^{<t>} = g(W_{hh}h^{<t-1>} \oplus W_{hx}x^{<t>} + b_h)$$

A Vanilla RNN



You can compute the current y-hat by multiplying the hidden current state with a set of learnable parameters, w_y times h, and going through another activation function.

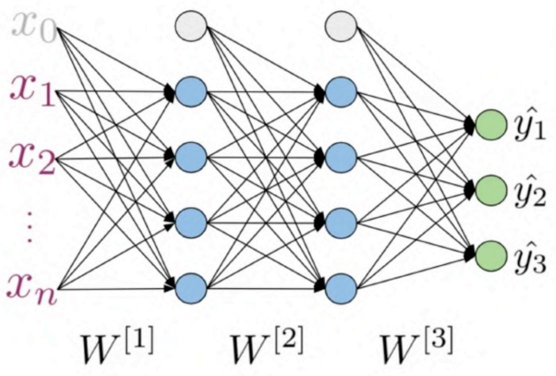
$$h^{<t>} = g(W_{hh}h^{<t-1>} \oplus W_{hx}x^{<t>} + b_h)$$

$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

Summary

- Hidden states propagate information through time
- Basic recurrent units have two inputs at each time: $h^{<t-1>}$, $x^{<t>}$

Cross Entropy Loss



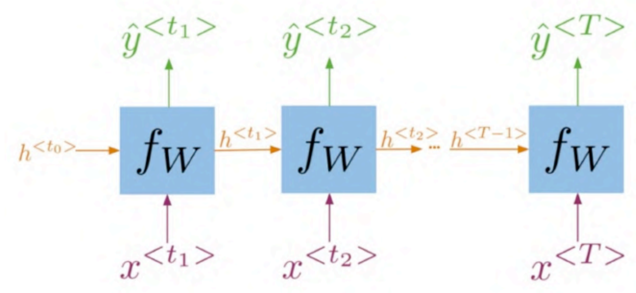
K - classes or possibilities

Either 0 or 1

$$J = -\sum_{j=1}^{K} \boxed{y_j} \log \hat{y}_j$$

Looking at a single example (x, y)

Cross Entropy Loss

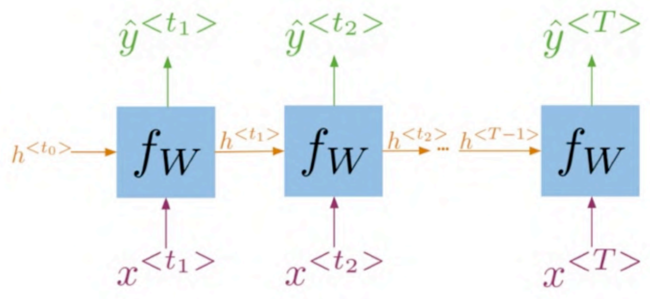


$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{K} y_j^{<t>} \log \hat{y}_j^{<t>}$$

Cross Entropy Loss



$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{K} y_j^{<t>} \log \hat{y}_j^{<t>}$$

Average with respect to time

Summary

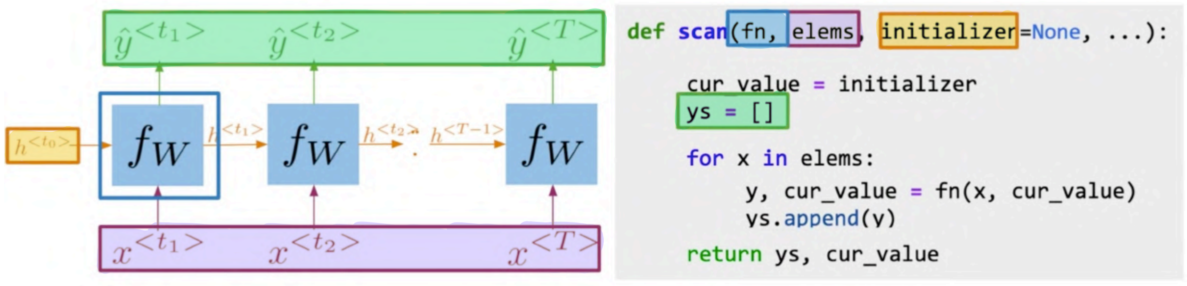
For RNNs the loss function is just an average through time!

Outline

- scan() function in tensorflow
- Computation of forward propagation using abstractions

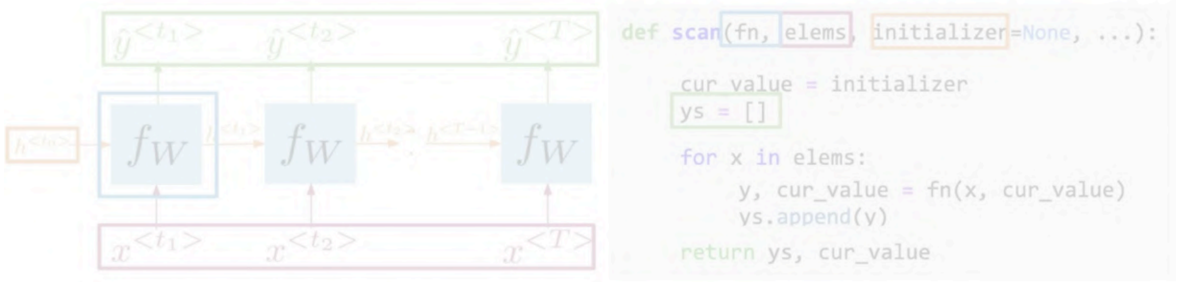
The scan function is designed to take a function fn, and apply it to all of the elements from the beginning to the end in the list elems. Initializer is an optional variable that could be used in the first computation of fn. Now take this RNN where fn is equivalent to fw. Elems is the list with all the inputs x superscript t, and the initializer is the hidden state as h superscript t subscript 0. The scan function first initializes the hidden states as h superscript t subscript 0, and sets the ys which store the prediction values as an empty list. Then for every x in the list of elements, fn is called with x, and the value of the last hidden state is arguments. So this four loop computes every time step of the RNN, and stores the values of the prediction and hidden states. Finally, the function returns the list of predictions, and the last hidden state. You might think this function is unnecessary, because it is essentially a four loop through every time step of the RNN. However, Frameworks like Tensorflow need this type of abstraction in order to perform parallel computations, and run on GPUs.

tf.scan() function



```
def scan(fn, elems, initializer=None, ...):

    cur_value = initializer
    ys = []

    for x in elems:
        y, cur_value = fn(x, cur_value)
        ys.append(y)
    return ys, cur_value
```

tf.scan() function



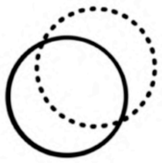
```
def scan(fn, elems, initializer=None, ...):

    cur_value = initializer
    ys = []

    for x in elems:
        y, cur_value = fn(x, cur_value)
        ys.append(y)
    return ys, cur_value
```

Frameworks like Tensorflow need this type of abstraction

Parallel computations and GPU usage

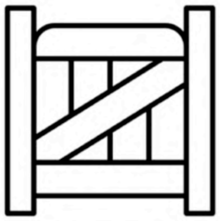
Summary

- Frameworks require abstractions
- tf.scan() mimics RNNs



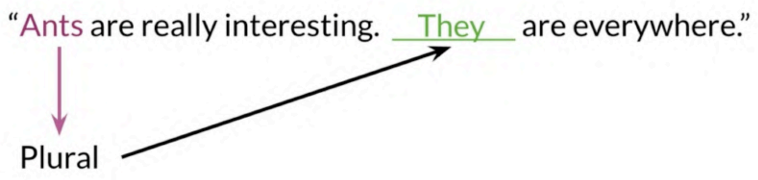
Outline

- Gated recurrent unit (GRU) structure
- Comparison between GRUs and vanilla RNNs



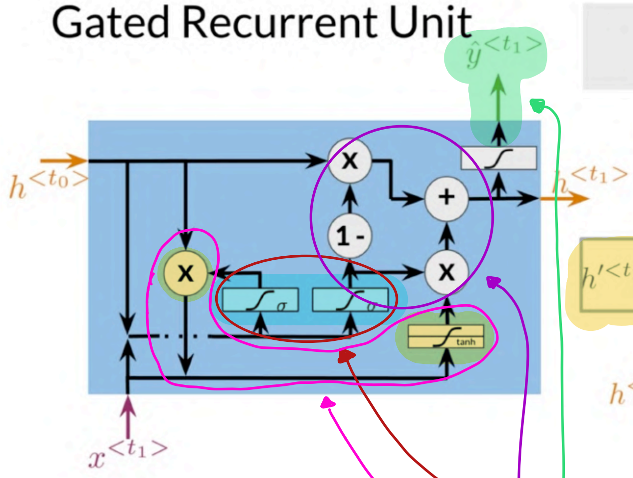
This type of model has some parameters which allow you to control how much information to forget from the past and how much information to extract from the current input

Gated Recurrent Units



Relevance and update gates to remember important prior information

Gated Recurrent Unit



Gates to keep/update relevant information in the hidden state

first Computations

$$\Gamma_r = \sigma(W_r[h^{<t_0>}, x^{<t_1>}] + b_r)$$

$$\Gamma_u = \sigma(W_u[h^{<t_0>}, x^{<t_1>}] + b_u)$$

$$h'^{<t_1>} = \tanh\left(W_h[\Gamma_r * h^{<t_0>}, x^{<t_1>}] + b_h\right)$$

Hidden state candidate

$$h^{<t_1>} = \Gamma_u * h^{<t_0>} + (1 - \Gamma_u) * h'^{<t_1>}$$

$$\hat{y}^{<t_1>} = g(W_y h^{<t_1>} + b_y)$$

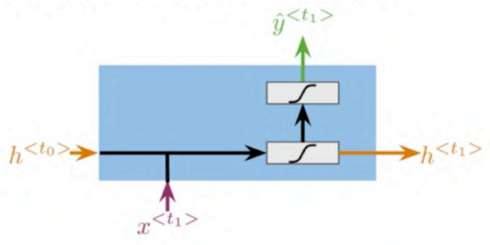
The first two computations made in GRU are the relevance gates. Gamma sub r and the update gate Gamma sub u. These gates compute the sigmoid activation function, so their result is a vector of values which have been squeezed to fits between zero and one.
The update and relevance gates in GRUs are the most important computations. Their outputs help determine which information from the previous hidden state is relevant and which values should be updated with currents information.

After the relevance gates is computed, a candidate's h prime for the hidden state is found. Its computation takes as parameters the previous hidden state times the relevance gates, and the variable x for the current time. This value stores all the candidates for information that's got overwrites the one contained in the previous hidden states.

After that, a new value for the hidden state is calculated using the information from the previous hidden state, the candidate hidden state and the update gate.
The update gate determines how much of the information from the previous hidden state will be overwritten.

Finally, a prediction y hat is computed using the current hidden state.

Vanilla RNN vs GRUs

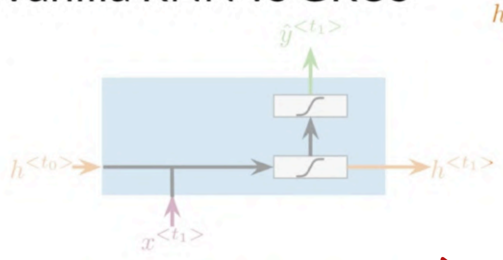


$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

$$\hat{y}^{<t>} = g(W_{yh} h^{<t>} + b_y)$$

On the other hand, GRUs compute significantly more operations, which can cause longer processing times and memory usage. The relevance and update gates determine which information from the previous hidden state is relevant and what information should be updated. The hidden state candidate stores the information that could be used to overwrite the one passed from the previous hidden state. The current hidden state is computed and updates some of the information from the last hidden state. Any prediction y hat is made with the updated hidden states. All of these computations allow the network to learn what type of information to keep and when to overwrite it.

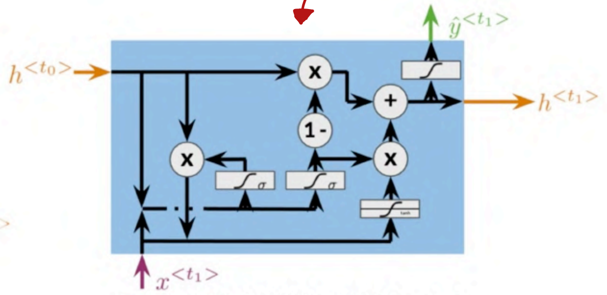
Vanilla RNN vs GRUs



$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

$$\hat{y}^{<t>} = g(W_{yh} h^{<t>} + b_y)$$

Remember that a vanilla RNN such as this one computes an activation function with the previous hidden state and current's variable x as parameters to get the current hidden state. With the current hidden state, another activation function is computed to get the current prediction y hat. This architecture is updating the hidden state at every time step. So for a long sequences, the information tends to vanish. This is one cause of the so-called vanishing gradients problem.



$$\Gamma_u = \sigma(W_u[h^{<t_0>}, x^{<t_1>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[h^{<t_0>}, x^{<t_1>}] + b_r)$$

$$h'^{<t_1>} = \tanh\left(W_h[\Gamma_r * h^{<t_0>}, x^{<t_1>}] + b_h\right)$$

$$h^{<t_1>} = \Gamma_u * h^{<t_0>} + (1 - \Gamma_u) * h'^{<t_1>}$$

$$\hat{y}^{<t_1>} = g(W_y h^{<t_1>} + b_y)$$

Summary

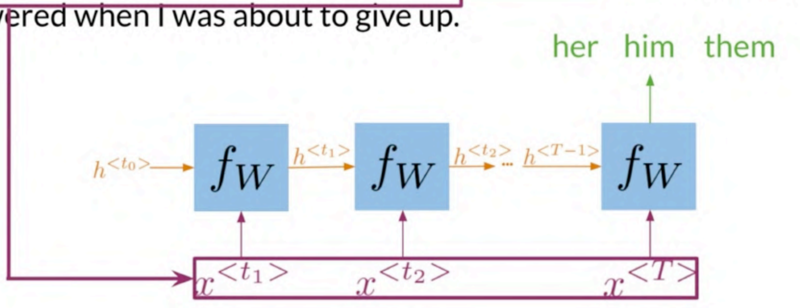
- GRUs "decide" how to update the hidden state
- GRUs help preserve important information

Outline

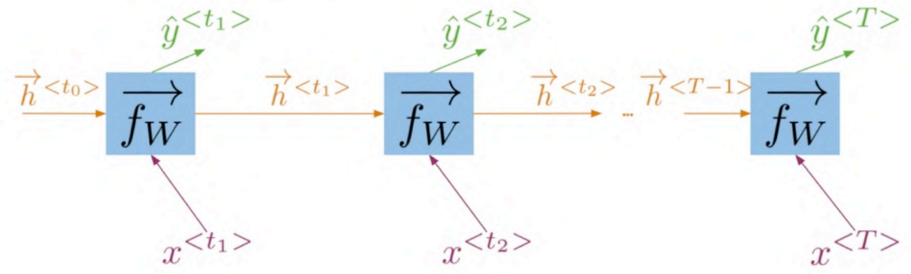
- How bidirectional RNNs propagate information
- Forward propagation in deep RNNs

Bi-directional RNNs

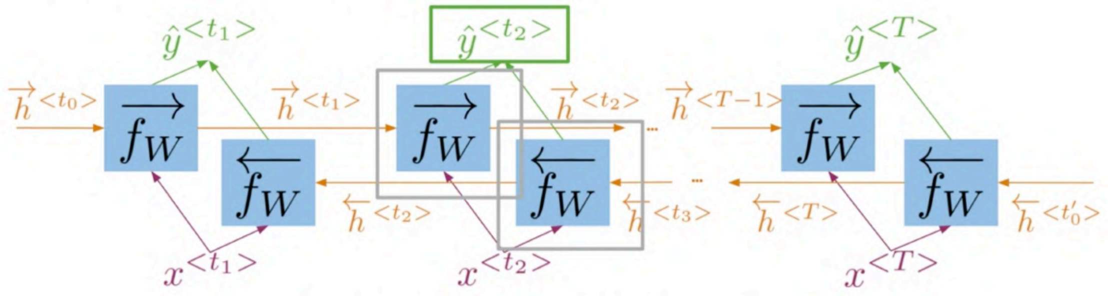
I was trying really hard to get a hold of ________. **Louise,** finally answered when I was about to give up.



Bi-directional RNNs

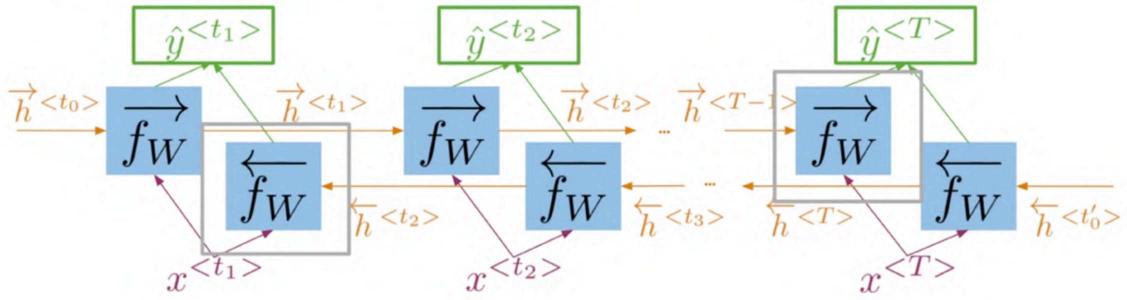


Bi-directional RNNs



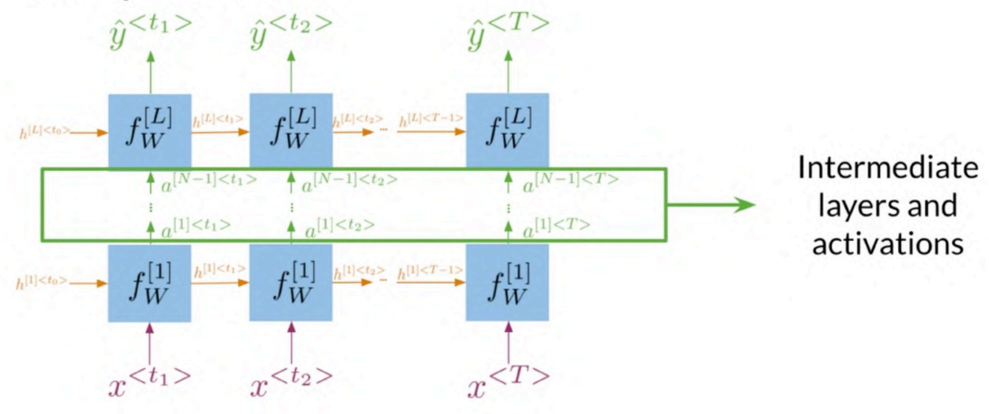
Information flows from the past and from the future **independently**

Bi-directional RNNs



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

Deep RNNs



Deep RNNs

$$h^{[l]<t>} = f^{[l]}(W_h^{[l]}[h^{[l]<t-1>}, a^{[l-1]<t>}] + b_h^{[l]})$$

$$a^{[l]<t>} = f^{[l]}(W_a^{[l]} h^{[l]<t>} + b_a^{[l]})$$

1. Get hidden states for current layer
2. Pass the activations to the next layer

Summary

- In bidirectional RNNs, the outputs take information from the past and the future
- Deep RNNs have more than one layer, which helps in complex tasks