IIT Madras
Centre For Innovation

in collaboration with

Analytics Club IITM
Data Science and Deep Learning

&

COMPUTER VISION AND
INTELLIGENCE GROUP

presents

Summer School 2021

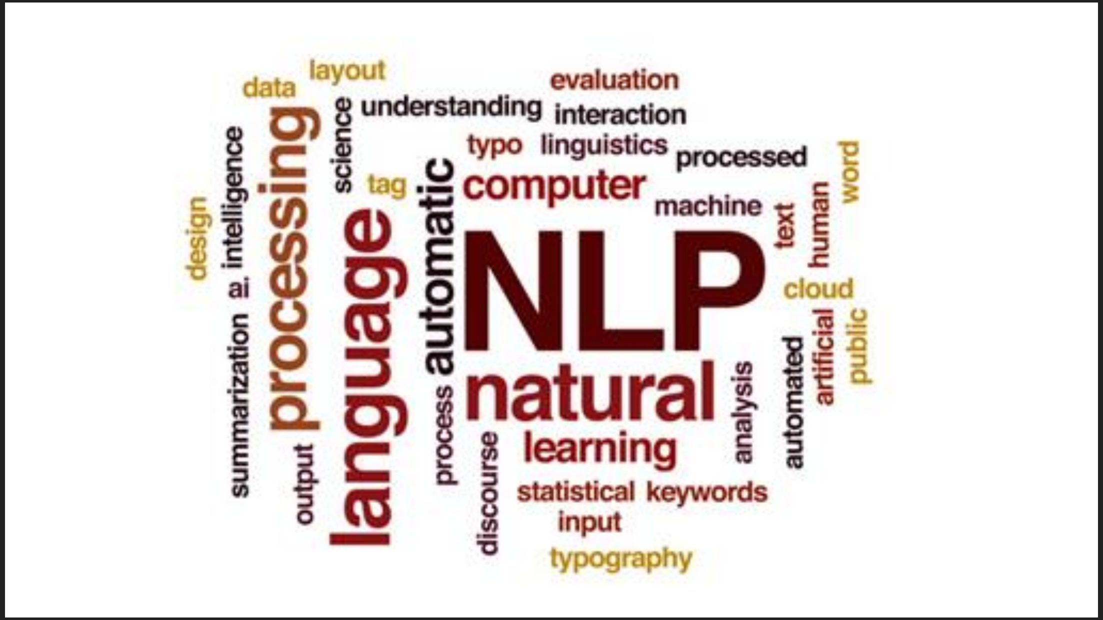
Analytics Club IITM
Data Science and Deep Learning

Deep Learning Master-Class

COMPUTER VISION AND INTELLIGENCE GROUP

Week 3 Session 2

Deep Learning for Natural Language Processing



Natural Language Processing: Motivation

Natural Language Processing, or NLP for short, is broadly defined as the automatic manipulation of natural language, like speech and text, by software.

- Text and Speech Processing
    - Optical Character Recognition
    - Speech Recognition
- Syntactic Analysis
- Lexical Semantics
- Machine Translation
- Question Answering

Deep Learning for NLP

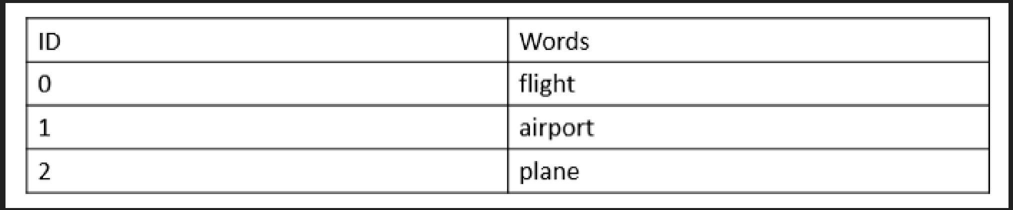
- Embedding Layers
- Convolutional Neural Networks
- Recurrent Neural Networks
  - Vanilla RNNs
  - GRUs
  - LSTMs
- Transformers

Word Representation: Dictionary Lookup

The Naïve idea is a word ID lookup in the Dictionary.

- Step 1: Identify the corpus.
- Step 2: Preprocess the word – One way is to use the root word, i.e., talk for talking.
- Step 3: Create a mapping between the processed vocabulary and the dictionary.

For each word, the algorithm returns the corresponding integer representation in the dictionary.



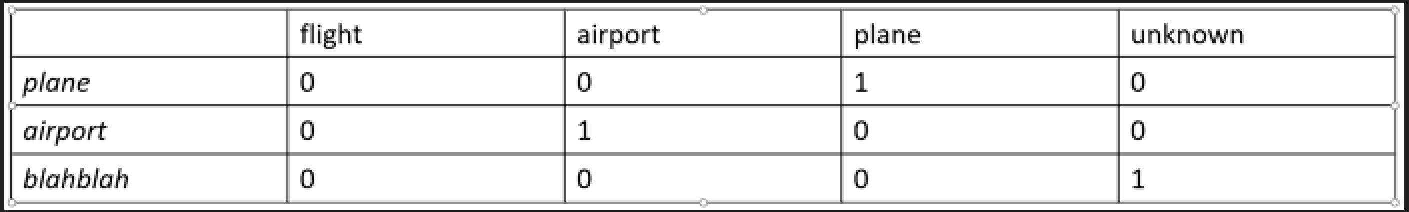
ID	Words
0	flight
1	airport
2	plane

Drawbacks: Finite Corpus, Ordering.

Word Representation: One Hot Encoding

A better approach would be to use one-hot encoding vectors in place of integer mappings.

For a single word, the encoded token consists of a $1 \times (N+1)$ dimensional vector. The corresponding column is filled with 1 whereas, the other columns are zeros.



	flight	airport	plane	unknown
plane	0	0	1	0
airport	0	1	0	0
blahblah	0	0	0	1

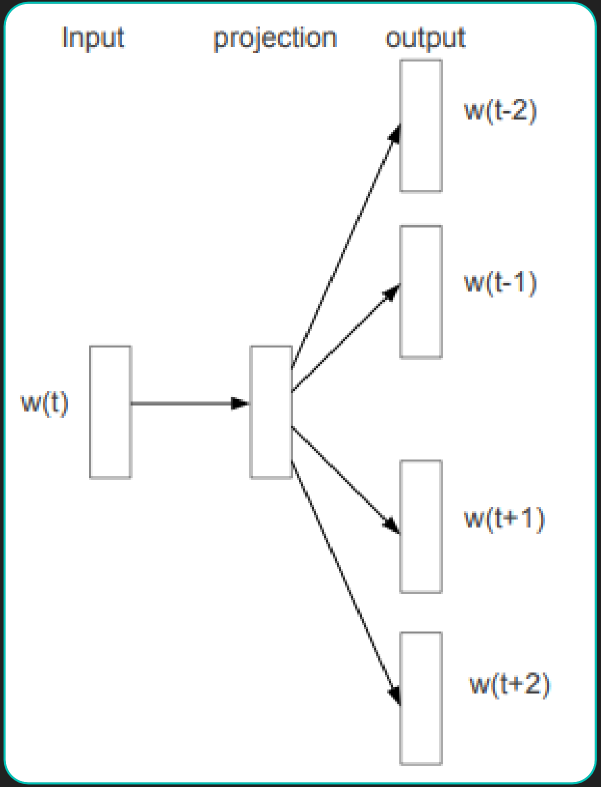
Disadvantages: Every vector is sparse and requires lot of memory for computation.

Word Representation: Word2Vec

The main idea is to represent every word by means of its neighbors. The objective is to find word representations that can predict surrounding words.

Given the sequence of words, $w_1, w_2, \ldots, w_T$ the objective is to maximize $\max \frac{1}{T} \sum_{t=1}^{T} \sum_{-c \le j \le c, j \ne 0} \log(p(w_{t+j} | w_t))$ where c is the size of training context.

The probabilities are calculated using techniques like Hierarchical SoftMax, where the words are represented in a binary tree and the output layer of W, v_w as its nodes.



Check out the paper for more details.

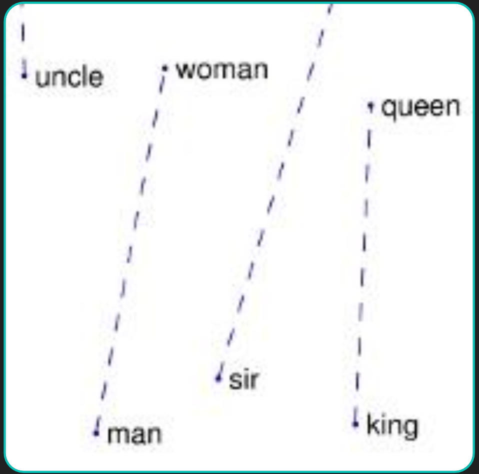
Word Representation: GloVe

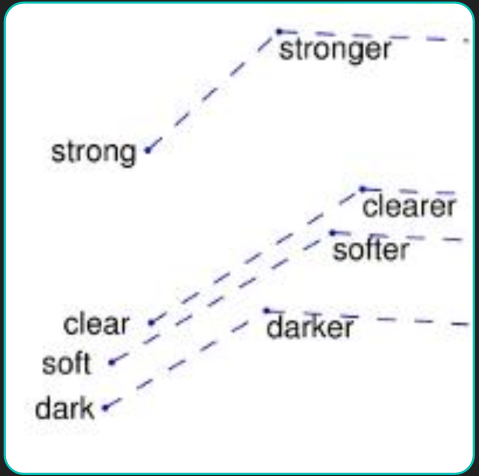
Word2Vec captures the local context window while GloVe exploits the overall co-occurrence statistics of the word corpus.

Both GloVe and Word2Vec learn useful features such as semantically similar words.

- The resulting representations showcase interesting linear substructures of the word vector space.
  - The Euclidean distance between two-word vectors provides an effective method for measuring the linguistic or semantic similarity of the corresponding words.

Check out the website or paper for more details.





Question

What is a major drawback of using GloVe embeddings?

Answer

GloVe fails to learn the positional importance and therefore fails to capture the meaning of the same word used in different contexts.

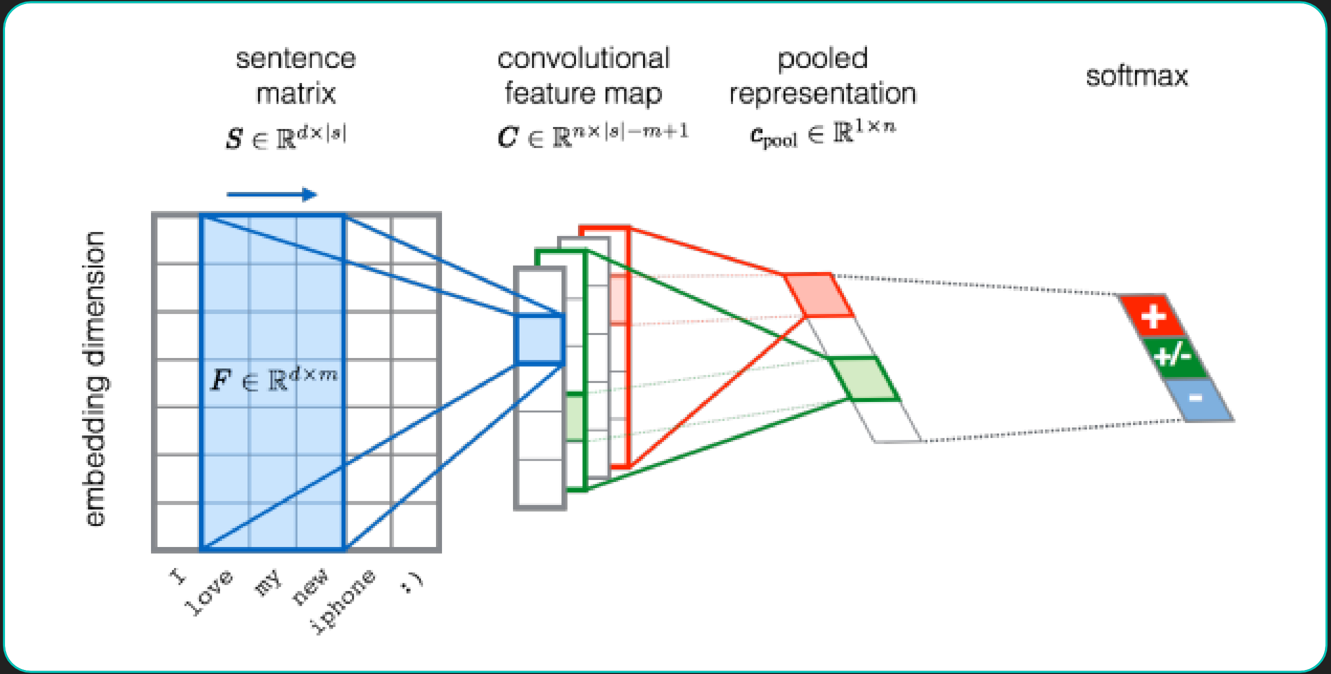
For example, the word bark in “dog barking” and “bark of a tree” have completely different meaning.

Convolutional Neural Networks

Now that we have represented each word as a vector, we can apply filters over a length to create a convolutional feature map.

Using classical techniques, we can train our CNN model to predict sentiment, etc.

Drawbacks: In most cases, CNNs for NLP performs poorly as it lacks to learn features beyond the context.


sentence matrix
$S \in \mathbb{R}^{d \times |s|}$
convolutional feature map
$C \in \mathbb{R}^{n \times |s|-m+1}$
pooled representation
$c_{pool} \in \mathbb{R}^{1 \times n}$
softmax
embedding dimension
$F \in \mathbb{R}^{d \times m}$
I
love
my
new
iphone
:)
+
+/-
-

Check out the paper for more details.

Sequence Models

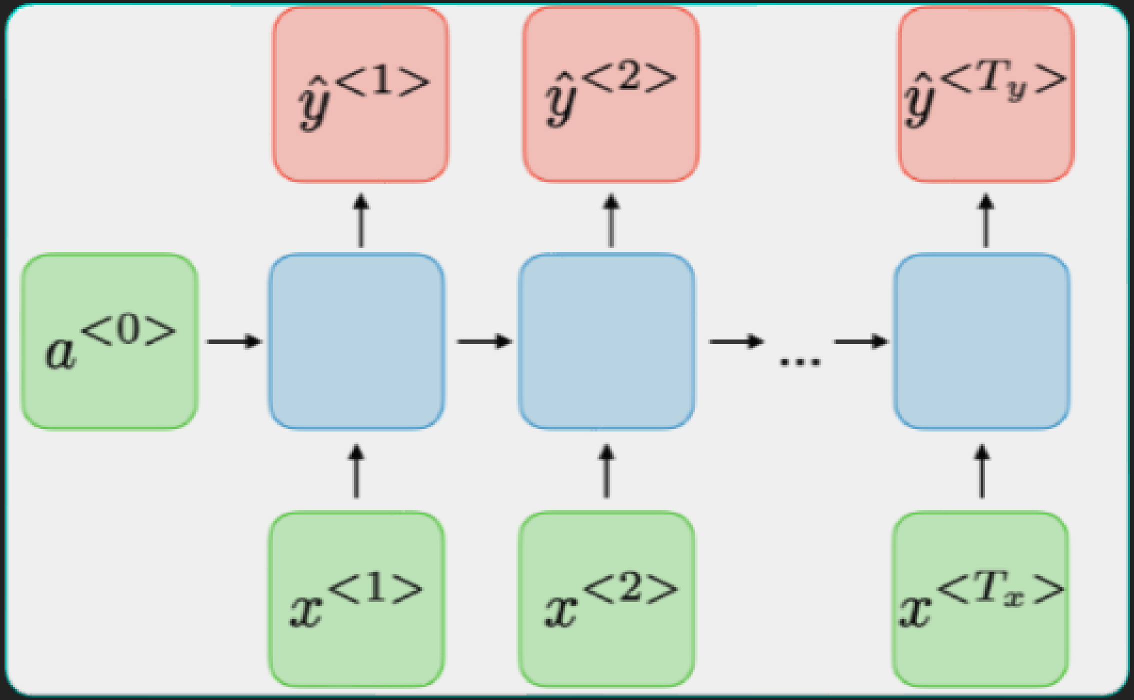
The standard FNNs do not share the learned features across different positions of the network, i.e., each input and its output is assumed to be independent. Another downside to using FNNs is the fact that FNNs require input of fixed length, which is not the case in NLP.

Sequence Modelling deals with this issue by learning the dependency of the current output on the previous inputs.

Recurrent Neural Networks

RNNs are a type of Neural Network architecture, where the output from the previous step is fed additionally as input the current step.

We denote the input at the i^{th} index by $x^{\langle i \rangle}$ and the output by $y^{\langle i \rangle} = \hat{f}(x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \dots, x^{\langle i \rangle})$.



Check out the paper for more details.

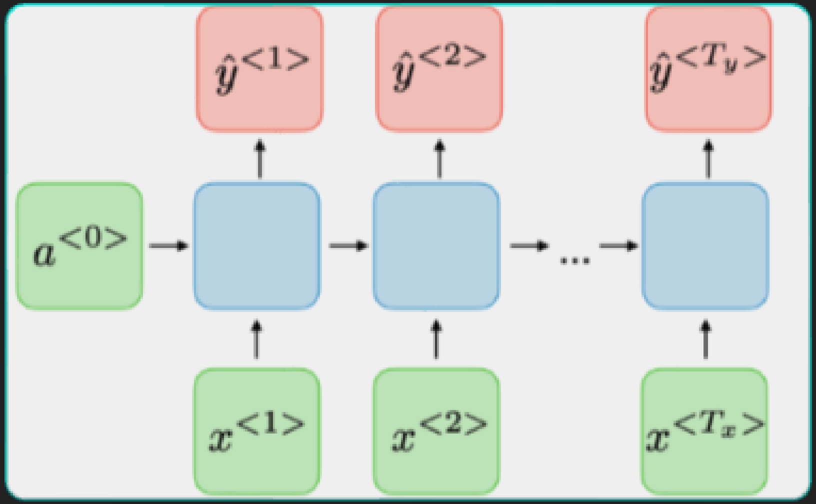
Recurrent Neural Networks

At time t, nodes with recurrent edges receive input from the current data point $x^{\langle t\rangle}$ and from hidden node values $a^{\langle t-1\rangle}$ in the network's previous state. The output $\hat{y}^{\langle t\rangle}$ at each time t is calculated given the hidden node values $a^{\langle t\rangle}$ at time t.

$$a^{\langle t\rangle} = g\left(W_{aa}a^{\langle t-1\rangle} + W_{ax}x^{\langle t\rangle} + b_a\right)$$
$$\hat{y}^{\langle t\rangle} = g'\left(W_{ya}a^{\langle t\rangle} + b_y\right)$$

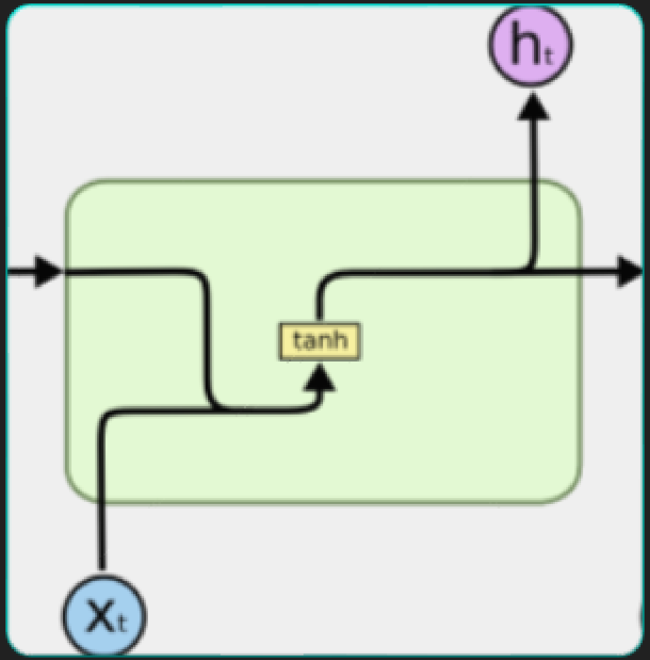
Here W represents the weight matrix and b represents the bias vector.

The RNN network can be trained across many time steps using backpropagation.



Recurrent Neural Networks

Learning with recurrent networks can be especially challenging due to the difficulty of learning long-range dependencies. The problems of vanishing and exploding gradients occur when backpropagating errors across many time steps.



Recurrent Neural Networks

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.tanh = nn.Tanh()
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input_tensor, hidden_tensor):
        combined = torch.cat((input_tensor, hidden_tensor), 1)
        hidden = self.i2h(combined)
        hidden = self.tanh(hidden)
        output = self.i2o(combined)
        output = self.tanh(output)
        output = self.softmax(output)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, self.hidden_size)
```

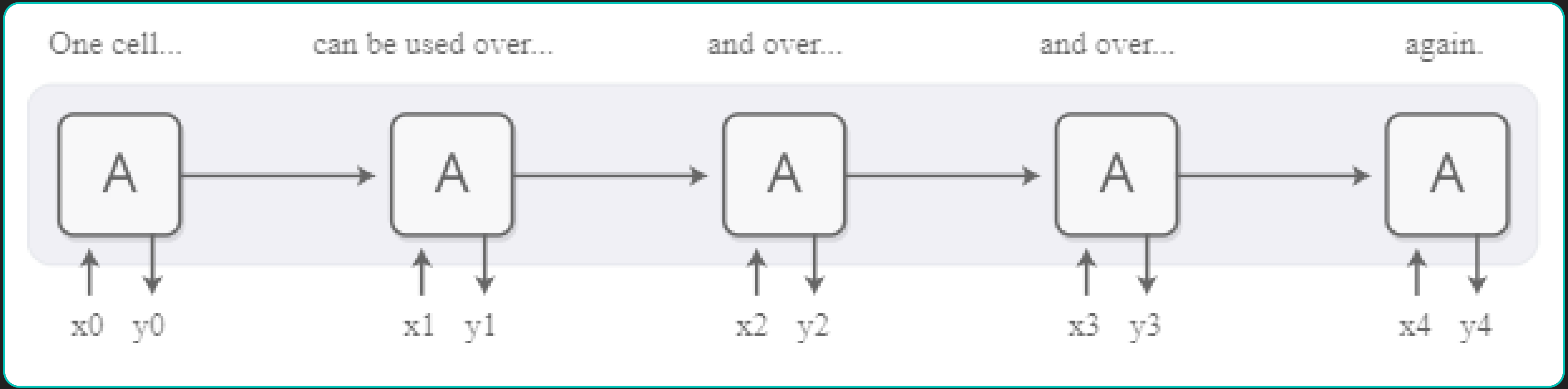
Question

Suppose the length of the input sentence is 100. How many RNN cells would you build and why?

Answer

We need to build only 1 cell.

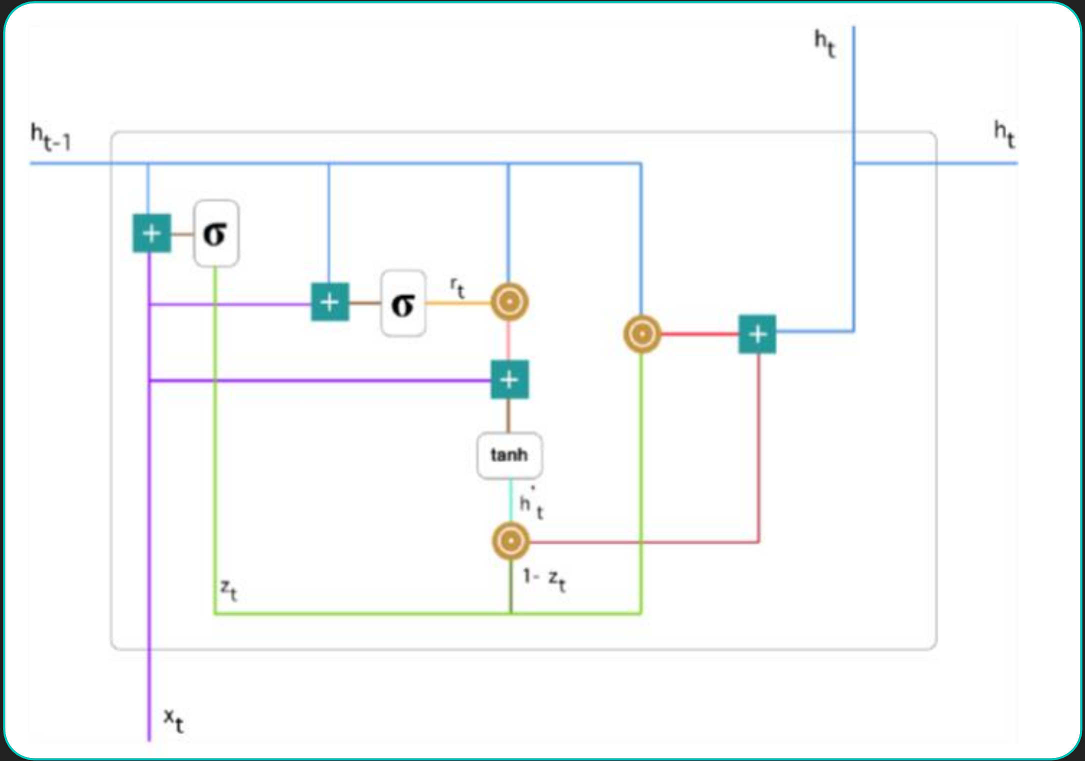
The same cell is reused to process every token of the input sequence.



Gated Recurrent Unit

To solve the vanishing gradient problem of a standard RNN, GRU uses, update gate and reset gate. Basically, these are two vectors which decide what information should be passed to the output.

The gates are different neural networks that decide which information is allowed on the cell state.



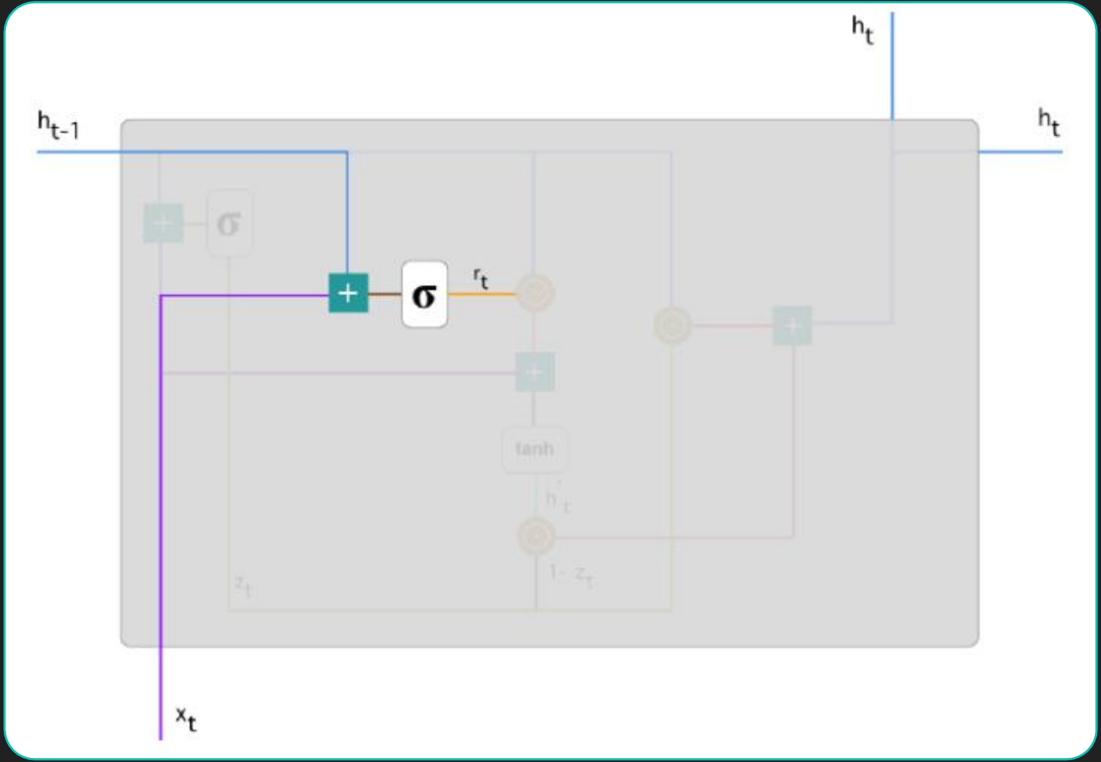
Gated Recurrent Unit

- Reset Gate

$$r_t = \sigma_g\left(W_{rx}x^{\langle t\rangle} + U_r a^{\langle t-1\rangle} + b_r\right)$$

Decides the amount of information to be forgotten.

It looks at the $a^{\langle t-1\rangle}$and $x^{\langle t\rangle}$ and then outputs a number between 0 and 1 for each entry in the cell state $c^{\langle t-1\rangle}$.



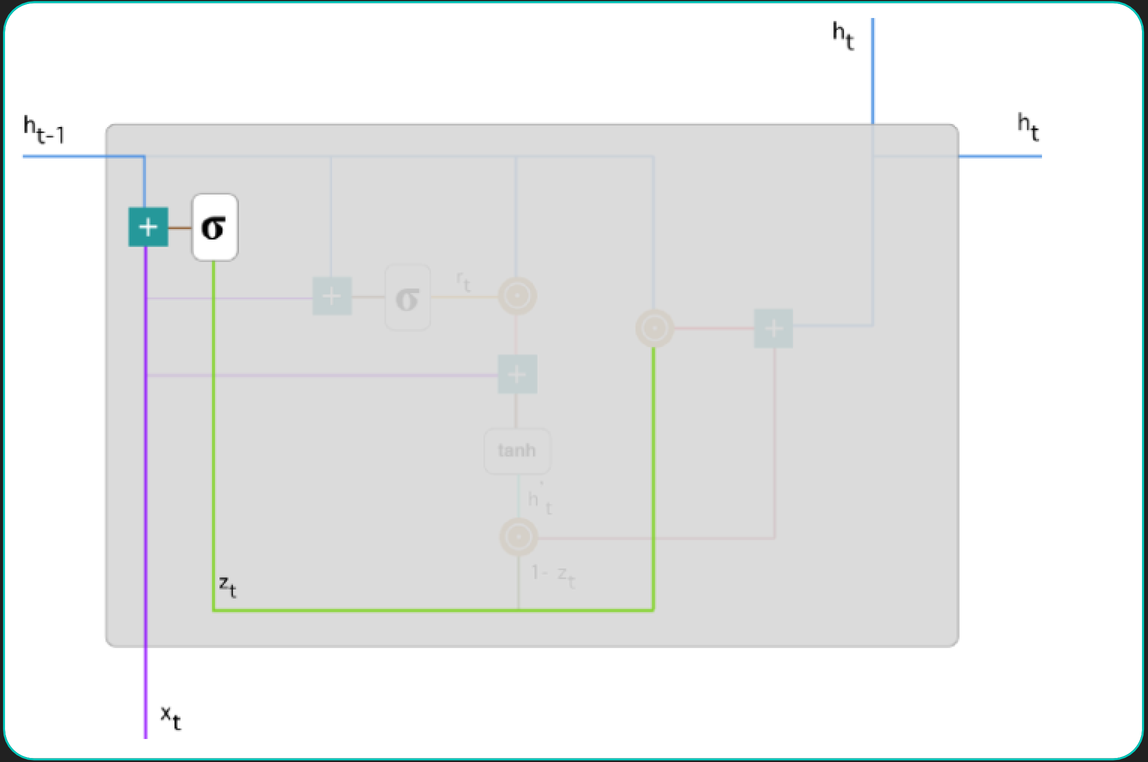
Gated Recurrent Unit

- Update Gate

$$z_t = \sigma_g\left(W_{zx}x^{\langle t\rangle} + U_z a^{\langle t-1\rangle} + b_z\right)$$

Decides the amount of past information that is to be passed along.

It looks at the $a^{\langle t-1\rangle}$and $x^{\langle t\rangle}$ and then outputs a number between 0 and 1 for each entry in the cell activation $\tilde{c}^{\langle t\rangle}$.

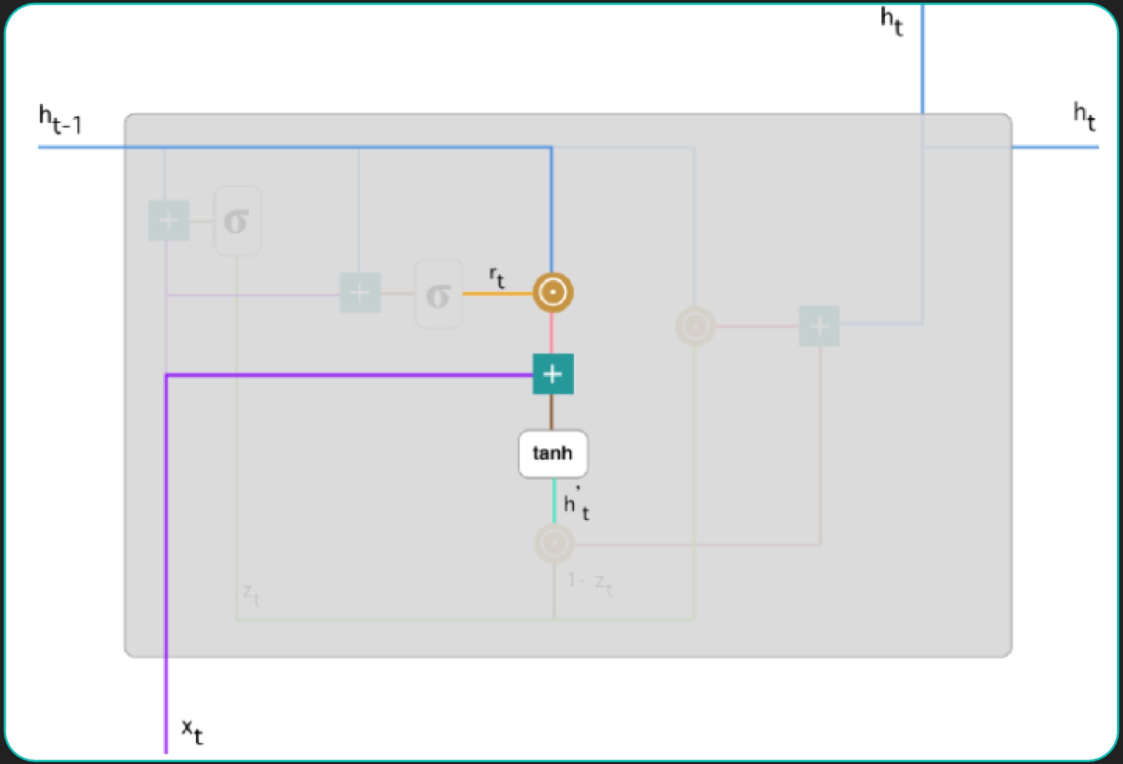


Gated Recurrent Unit

- Current Memory

$$\tilde{c}^{\langle t \rangle} = \phi_h\left(W_c x^{\langle t \rangle} + U_c\left(r_t \odot c^{\langle t-1 \rangle}\right) + b_c\right)$$

Computes the information that is to be extracted from the previous cell state and the information that could be added to the state.

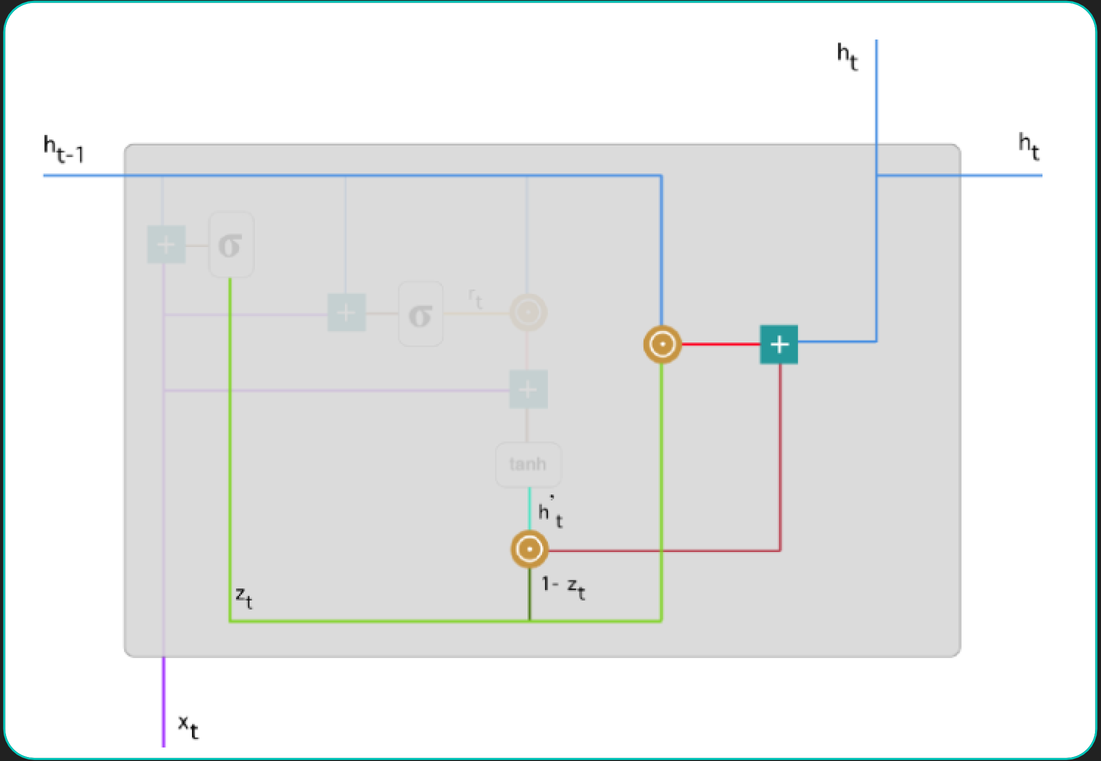


Gated Recurrent Unit

- Hidden State

$$a^{\langle t \rangle} = z_t \odot \tilde{c}^{\langle t \rangle} + (1 - z_t) \odot c^{\langle t-1 \rangle}$$

Contains information on previous inputs. The hidden state is also used for predictions.



Gated Recurrent Unit

GRUs can store and filter the information using their update and reset gates, thus eliminating the vanishing gradient problem since the model is not washing out the new input every single time but keeps the relevant information and passes it down to the next time steps of the network.

Gated Recurrent Unit

```
class GRU(nn.Module):
    def __init__(self,input_size, hidden_size, output_size):
        super(GRU, self).__init__()

        self.hidden_size = hidden_size
        self.sigmoid = nn.Sigmoid()
        self.i2u = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2r = nn.Linear(input_size + hidden_size, hidden_size)
        self.tanh = nn.Tanh()
        self.hidden_next = nn.Linear(input_size + hidden_size,
                                     hidden_size)
        self.yhat = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)
```

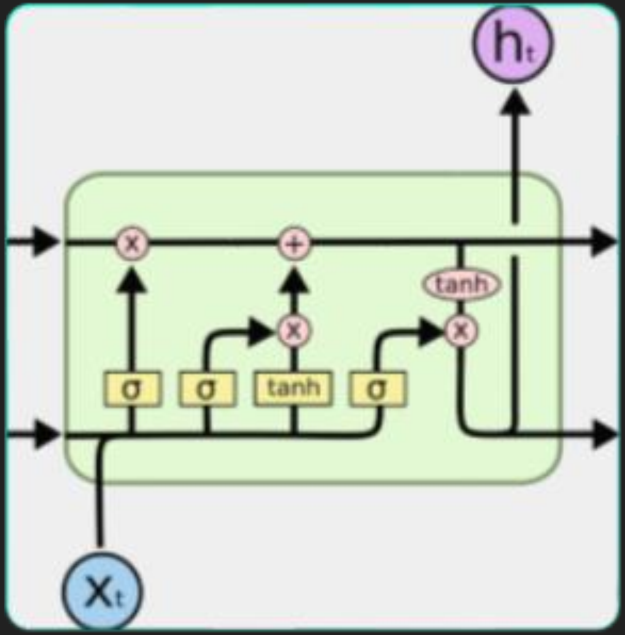
```
    def forward(self, input_tensor, hidden_tensor):
        combined = torch.cat((input_tensor, hidden_tensor), 1)
        z = self.i2u(combined)
        z = self.sigmoid(z)
        r = self.i2r(combined)
        r = self.sigmoid(z)
        reset = r*hidden_tensor
        combined = torch.cat((input_tensor, reset), 1)
        hidden_bar = self.hidden_next(combined)
        hidden_bar = self.tanh(hidden_bar)
        hidden = (1-z)*hidden_tensor + z*hidden_bar
        output = self.yhat(hidden)
        output = self.softmax(output)
        return output, hidden

      def init_hidden(self):
        return torch.zeros(1, self.hidden_size)
```

Long Short-Term Memory

Simple recurrent neural networks have long-term memory in the form of weights. The core concept of LSTM's are the cell state, and its various gates.

The cell state act as a transport highway that transfers relative information all the way down the sequence chain.

The gates are different neural networks that decide which information is allowed on the cell state.



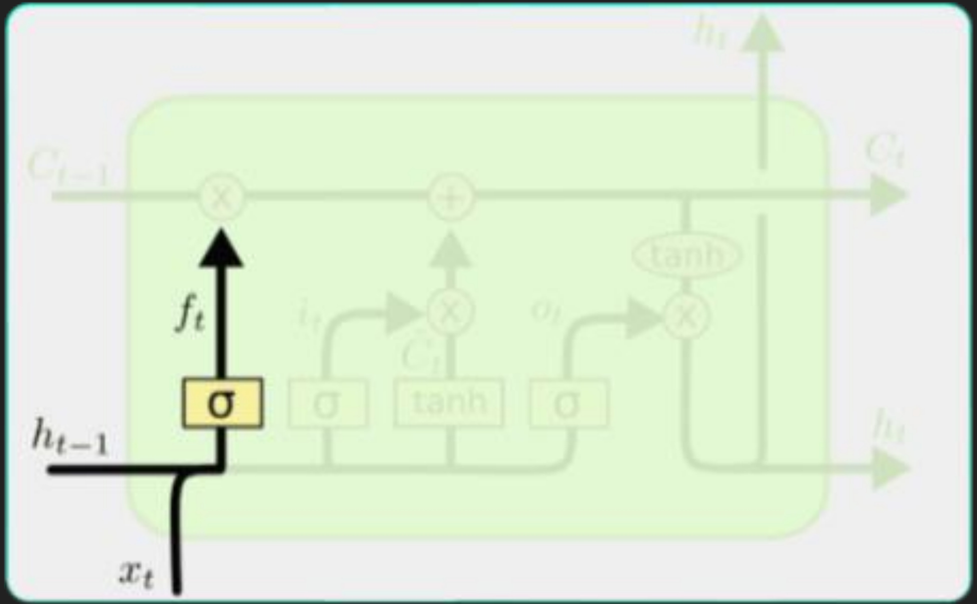
Long Short-Term Memory

- Forget Gate

$$f_t = \sigma_g\left(W_{fx}x^{\langle t\rangle} + U_f h^{\langle t-1\rangle} + b_f\right)$$

Decides what information should be thrown away or kept.

It looks at the $h^{\langle t-1\rangle}$ and $x^{\langle t\rangle}$ and then outputs a number between 0 (throw away) and 1 (preserve) for each entry in the cell state $c^{\langle t-1\rangle}$.



Long Short-Term Memory

- Input Gate

$$i_t = \sigma_g\left(W_{ix}x^{\langle t\rangle} + U_i h^{\langle t-1\rangle} + b_i\right)$$

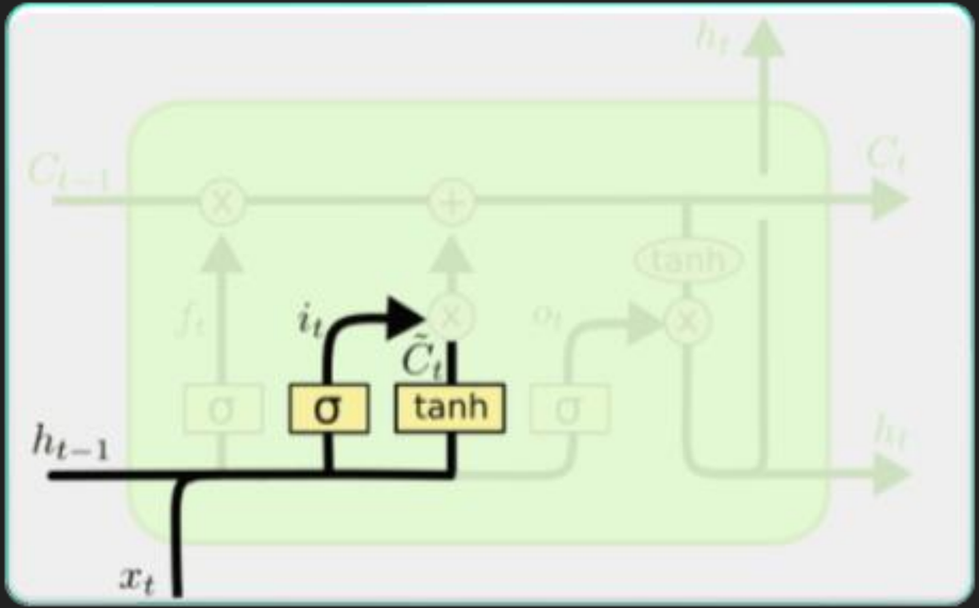
Decides what information to be stored in the cell state.

It looks at the $h^{\langle t-1\rangle}$ and $x^{\langle t\rangle}$ and then outputs a number between 0 and 1 for each entry in the cell activation $\tilde{c}^{\langle t\rangle}$.

- Cell Activation

$$\tilde{c}^{\langle t\rangle} = \phi_h\left(W_c x^{\langle t\rangle} + U_c h^{\langle t-1\rangle} + b_c\right)$$

Computes the information that could be added to the state.

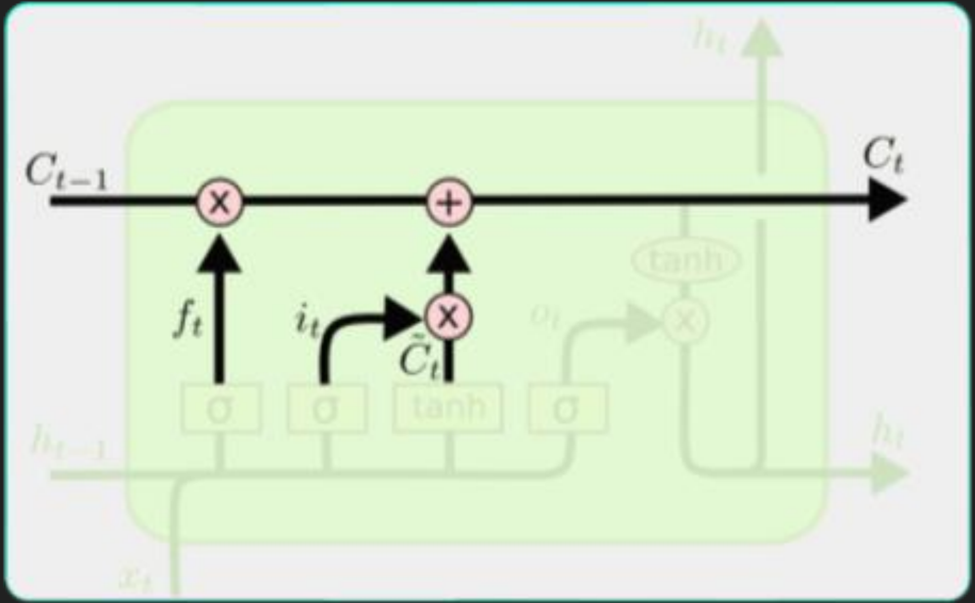


Long Short-Term Memory

- Cell State

$$c^{\langle t \rangle} = i_t \odot \tilde{c}^{\langle t \rangle} + f_t \odot c^{\langle t-1 \rangle}$$

The cell state is updated by using the altered previous cell state $f_t \odot c^{\langle t-1 \rangle}$, forgetting the information which is irreverent and the cell activation $i_t \odot \tilde{c}^{\langle t \rangle}$.



Long Short-Term Memory

- Output Gate

$$o_t = \sigma_g\left(W_{ox}x^{\langle t\rangle} + U_o h^{\langle t-1\rangle} + b_o\right)$$

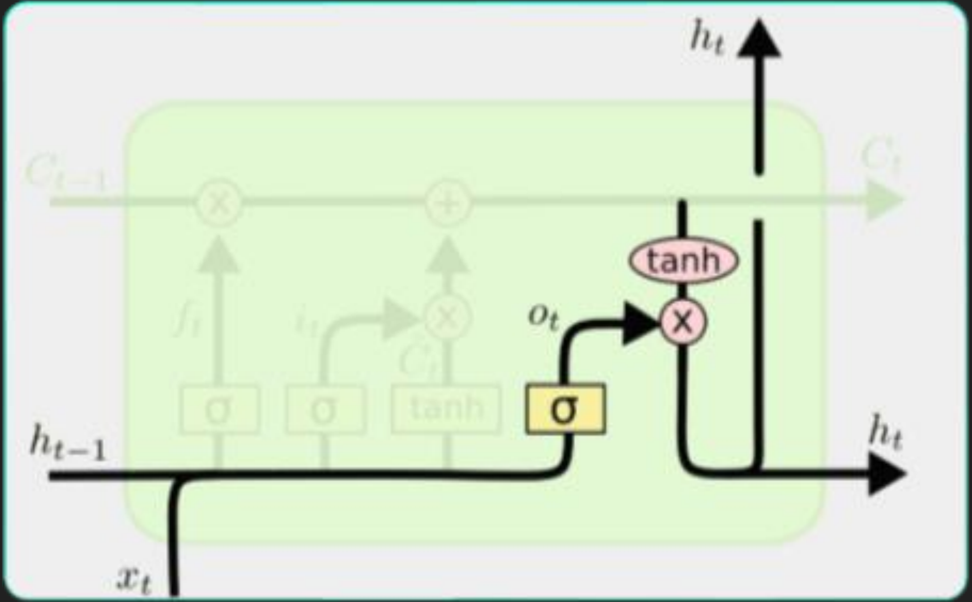
Decides what the next hidden state should be.

It looks at the $h^{\langle t-1\rangle}$ and $x^{\langle t\rangle}$ and then outputs a number between 0 and 1.

- Hidden State

$$h^{\langle t\rangle} = o_t \odot \phi_h\left(c^{\langle t\rangle}\right)$$

Contains information on previous inputs. The hidden state is also used for predictions.



Long Short-Term Memory

```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM, self).__init__()

        self.hidden_size = hidden_size
        self.input_size = input_size
        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()
        self.i2f = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2i = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2c = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, hidden_size)
        self.yhat = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)
```

```
def forward(self, input_tensor, hidden_tensor, long_term_tensor):
    combined = torch.cat((input_tensor, hidden_tensor), 1)
    f = self.i2f(combined)
    f = self.sigmoid(f)
    i = self.i2i(combined)
    i = self.sigmoid(i)
    c = self.i2c(combined)
    c = self.tanh(c)
    Ct = f*long_term_tensor + i*c
    Ct = self.tanh(Ct)
    o = self.i2o(combined)
    o = self.sigmoid(o)
    hidden = o*Ct
    output = self.yhat(hidden)
    output = self.softmax(output)
    return output, hidden

def init_hidden(self):
    return torch.zeros(1, self.hidden_size)

def init_long_term(self):
    return torch.zeros(1, self.hidden_size)
```

Question

Suppose the length of the input sentence is 10 and we use the 50-dimensional GloVe embedding.

What is a rough estimate of number of parameters in a simple RNN with hidden dimension of 10 to produce an output of dimension 5?

What if we use an LSTM instead?

Answer

The number of parameters of the RNN is roughly 650.

$$\text{Number of parameters } \approx hidden * (input(embedding) + hidden + output)$$

The number of parameters of an LSTM would be **four** times that of an RNN of similar dimensions.

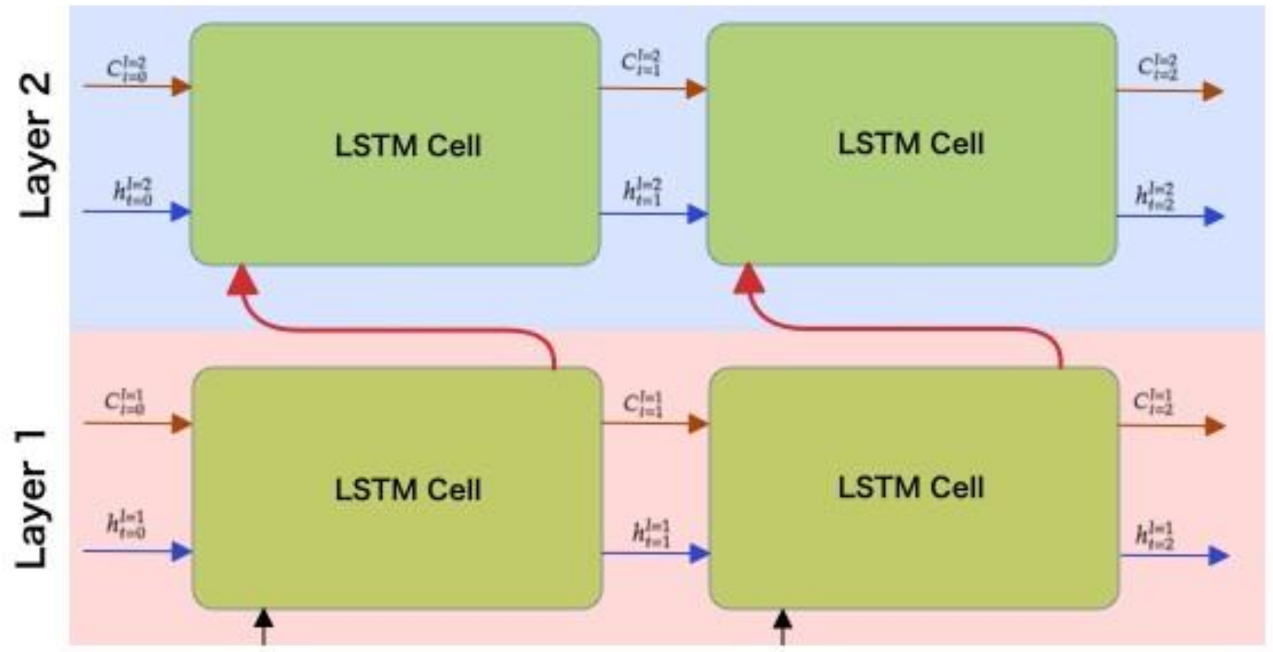
Question

Suppose we have a 2-layer RNN, what information is passed onto the second layer?

RNNs with Layers

Multiple RNNs one after another will result in the networks to learn to compute more complex representations.

The n^{th} cell state from Layer 1 is concatenated with $(n-1)^{\mathrm{th}}$ hidden state of Layer 2.



Question

Suppose in a paragraph, the author forgets to use compound sentences and instead breaks the sentence into two, i.e., “Mr. Simmons owns a dog. Its name is Buddy.” instead of “Mr. Simmons owns a dog named Buddy.”.

Will this affect our RNN? And what can you do to improve the scope?

Bidirectional RNNs

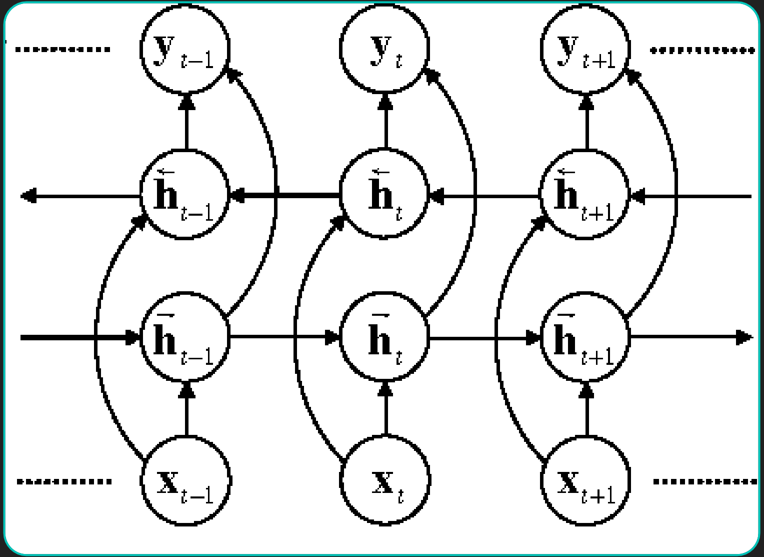
In this architecture, there are two layers of hidden nodes. Both hidden layers are connected to input and output.

The first recurrent connection learns from the past time steps while the second recurrent connection is flipped, passing activation from the back.

$$a_f^{\langle t\rangle} = g\left(W_{a_f a_f} a_f^{\langle t-1\rangle} + W_{a_f x} x^{\langle t\rangle} + b_{a_f}\right)$$

$$a_r^{\langle t\rangle} = g\left(W_{a_r a_r} a_r^{\langle t+1\rangle} + W_{a_r x} x^{\langle t\rangle} + b_{a_r}\right)$$

$$\hat{y}^{\langle t\rangle} = g'\left(W_{y a_f} a_f^{\langle t\rangle} + W_{y a_r} a_r^{\langle t\rangle} + b_y\right)$$



$\mathbf{y}_{t-1}$
$\mathbf{y}_t$
$\mathbf{y}_{t+1}$
$\overleftarrow{\mathbf{h}}_{t-1}$
$\overleftarrow{\mathbf{h}}_t$
$\overleftarrow{\mathbf{h}}_{t+1}$
$\overrightarrow{\mathbf{h}}_{t-1}$
$\overrightarrow{\mathbf{h}}_t$
$\overrightarrow{\mathbf{h}}_{t+1}$
$\mathbf{x}_{t-1}$
$\mathbf{x}_t$
$\mathbf{x}_{t+1}$

Bidirectional RNNs

The limitation of the BRNN is that cannot run continuously, as it requires a fixed endpoint in both the future and in the past. Further, it is not an appropriate machine learning algorithm for the online setting, as it is implausible to receive information from the future.

But, In fact, Bidirectional Long Short-Term Memory has been used to achieve state of the art results on handwriting recognition.

Details

- Implementation on Collab
- Quiz

Thank You!