NLP PART2

Word2vec is a two-layer neural net that processes text.
Its input is a text corpus and its output is a set of vectors: feature vectors for words in that corpus.

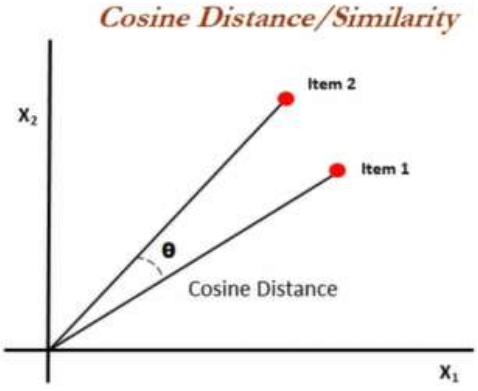
The purpose and usefulness of Word2vec is to group the vectors of similar words together in vectorspace.
That is, it detects similarities mathematically.

Word2vec creates vectors that are distributed numerical representations of word features, features such as the context of individual words.
It does so without human intervention.

- Given enough data, usage and contexts, Word2vec can make highly accurate guesses about a word's meaning based on past appearances.
- Those guesses can be used to establish a word's association with other words (e.g. "man" is to "boy" what "woman" is to "girl")

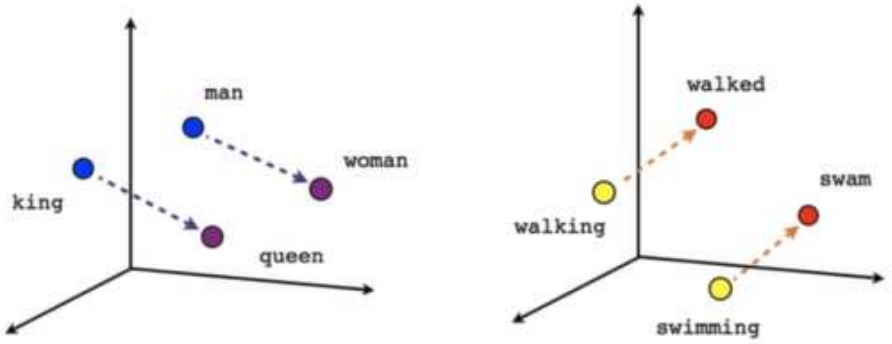
- Recall that each word is now represented by a **vector.**
- In spacy each of these vectors has 300 dimensions.

- This means we can use Cosine Similarity to measure how similar word vectors are to each other.



- This means we can also perform vector arithmetic with the word vectors.
  - **new_vector = king - man + woman**
- This creates new vectors (not directly associated with a word) that we can then attempt to find most similar vectors to.
  - **new_vector closest to vector for queen**

Interesting relationships can also be established between the word vectors



- import spacy
- nlp = spacy.load('en_core_web_md')
- nlp(u'lion').vector

The results are the vector components for the string lion.

Also DOC and span object have vectors, the vectors are derive from the average of the individual token vectors.

Word 2 vec but also doc 2 vec.

- nlp(u'the fox is on the table').vector.shape (document) 300
- nlp(u'fox').vector.shape (word) 300

- token = nlp(u'lion cat pet')
- for token1 in tokens:
  for token2 in tokens:
  print(token1.text, token2.text, token1.similarity(token2))

- token = nlp(u'like love hate')
- for token1 in tokens:
  for token2 in tokens:
  print(token1.text, token2.text, token1.similarity(token2))

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

We've already explored text classification and using it to predict sentiment labels on pre-labeled movie reviews.
But what if we don't already have those labels?
Are there methods of attempting to discern sentiment on raw unlabeled text?

- VADER (Valence Aware Dictionary for sEntiment Reasoning) is a model used for text sentiment analysis that is sensitive to both polarity (positive/negative) and intensity (strength) of emotion.
- It is available in the NLTK package and can be applied directly to unlabeled text data.

The sentiment score of a text can be obtained by summing up the intensity of each word in the text.

For example, words like “love”, “like”, “enjoy”, “happy” all convey a **positive** sentiment.
VADER is intelligent enough to understand basic context of these words, such as “**did not love**” as a negative sentiment.
It also understands capitalization and punctuation, such as “**LOVE!!!!**”

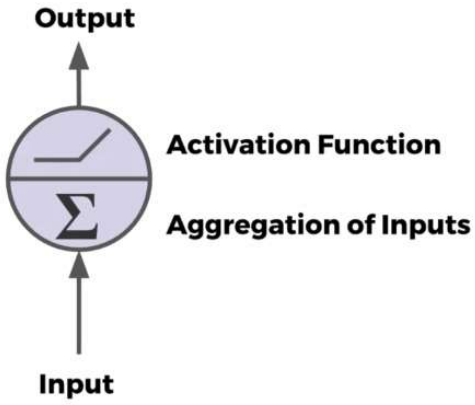
Sentiment Analysis on raw text is always challenging however, due to a variety of possible factors:

- Positive and Negative sentiment in the same text data.
- Sarcasm using positive words in a negative way.

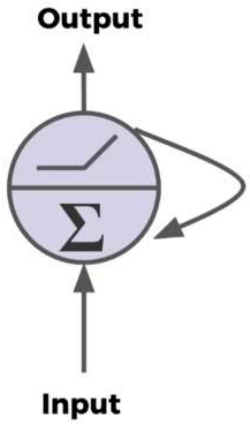
Recurrent Neural Networks Theory

- Examples of Sequences
  - Time Series Data (Sales)
  - Sentences
  - Audio
  - Car Trajectories
  - Music

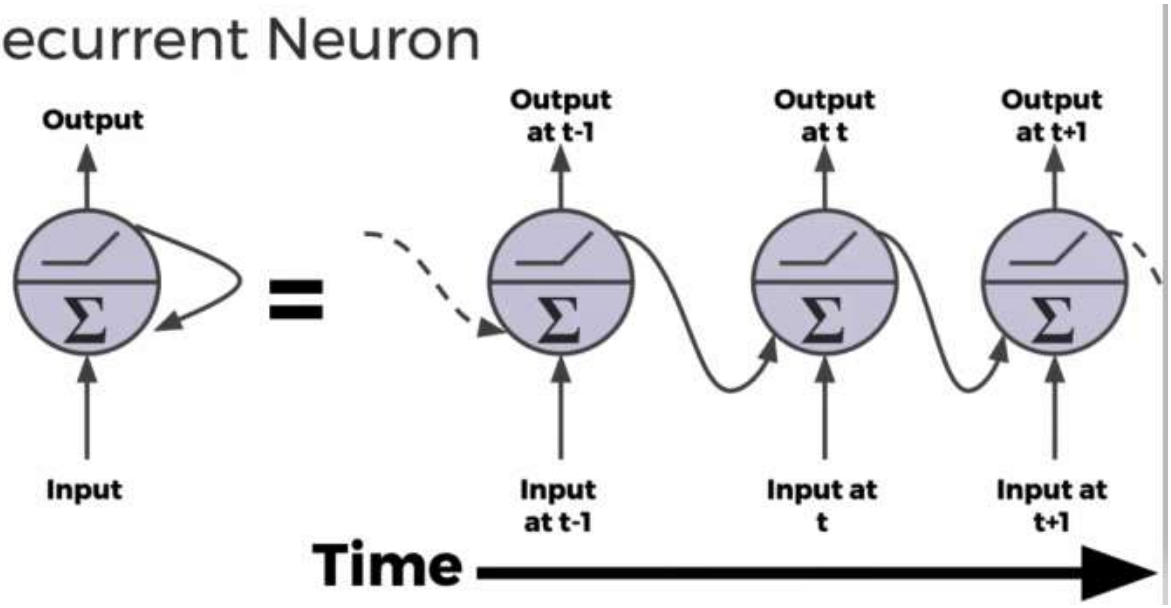
- Normal Neuron in Feed Forward Network



- Recurrent Neuron - Sends output back to itself!
  - Let's see what this looks like over time!



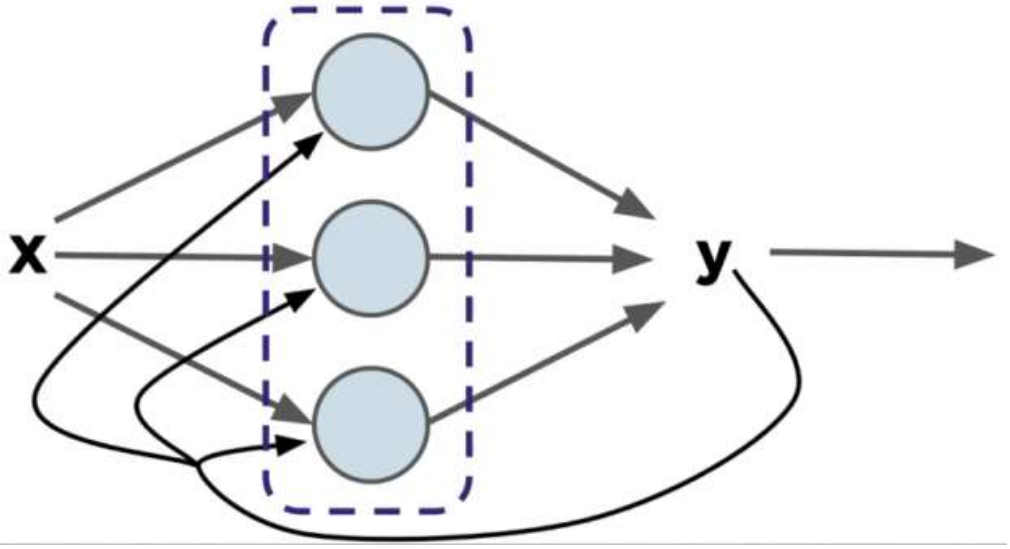
- Recurrent Neuron



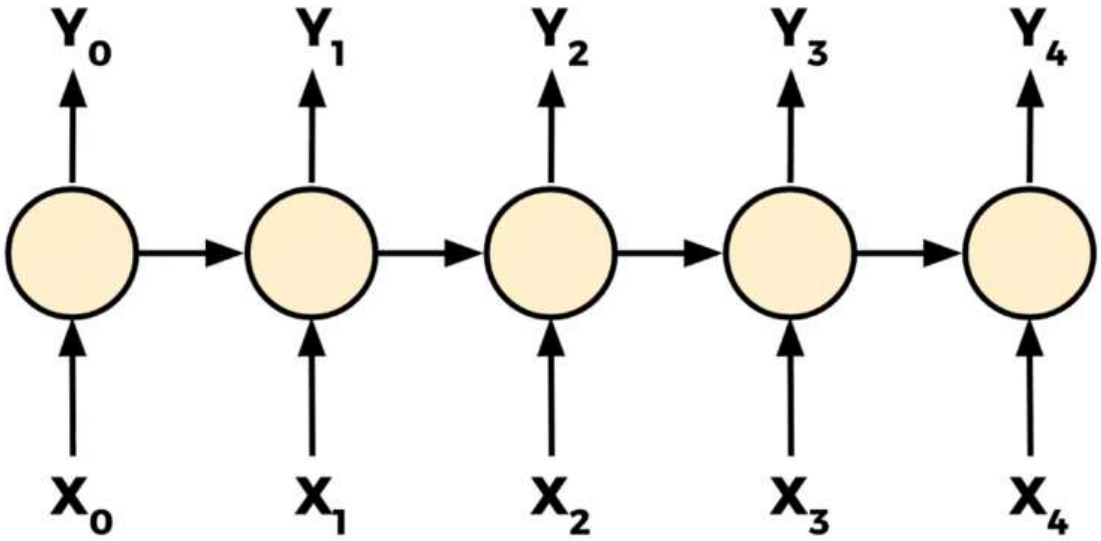
Neuron is receiving both input from a previous time step as well as input from the current time step

- Cells that are a function of inputs from previous time steps are also known as *memory cells*.
- RNN are also flexible in their inputs and outputs, for both sequences and single vector values.

- RNN Layer with 3 Neurons:



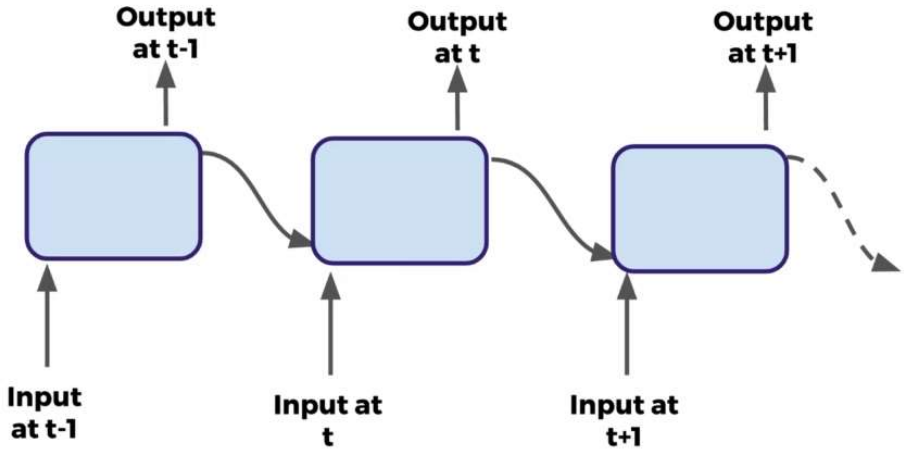
- Sequence to Sequence



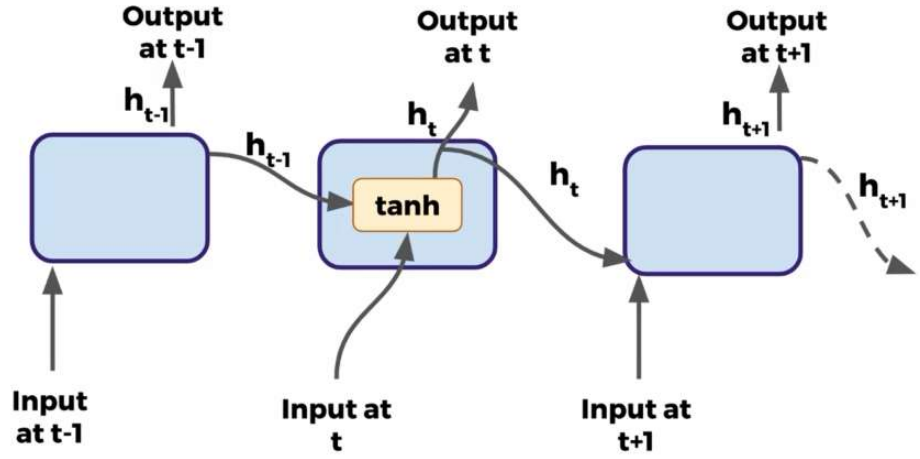
- An issue RNN face is that after awhile the network will begin to “forget” the first inputs, as information is lost at each step going through the RNN.
- We need some sort of “long-term memory” for our networks.

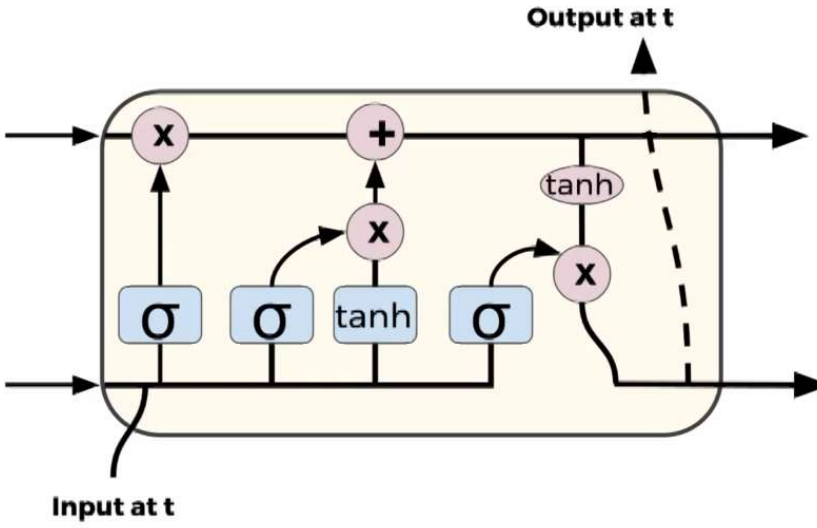
- The LSTM (Long Short-Term Memory) cell was created to help address these RNN issues.
- Let’s go through how an LSTM cell works!

- A typical RNN



- A typical RNN cell

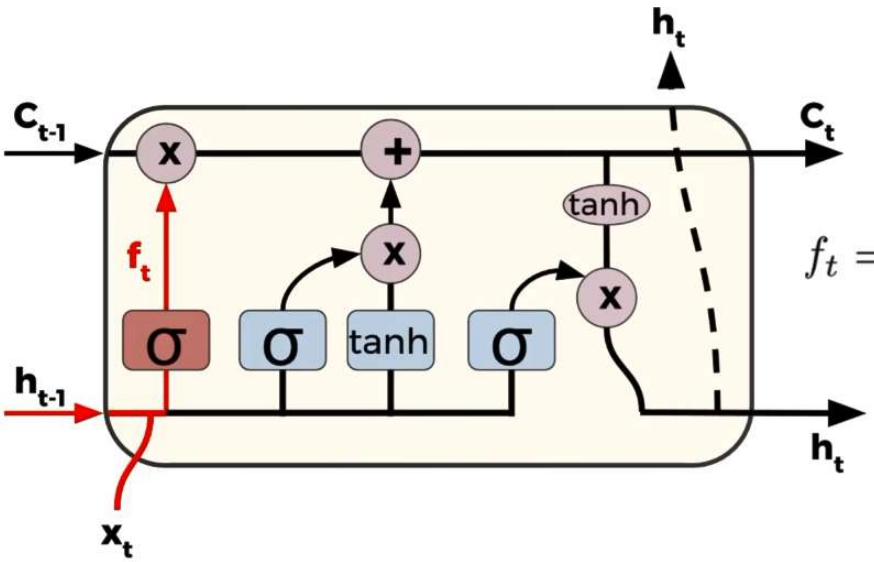




Here we can see the entire LSTM cell.

Let's go through the process!

- 1 Part: FORGET GATE LAYER . What kind of information we are going to throw away?

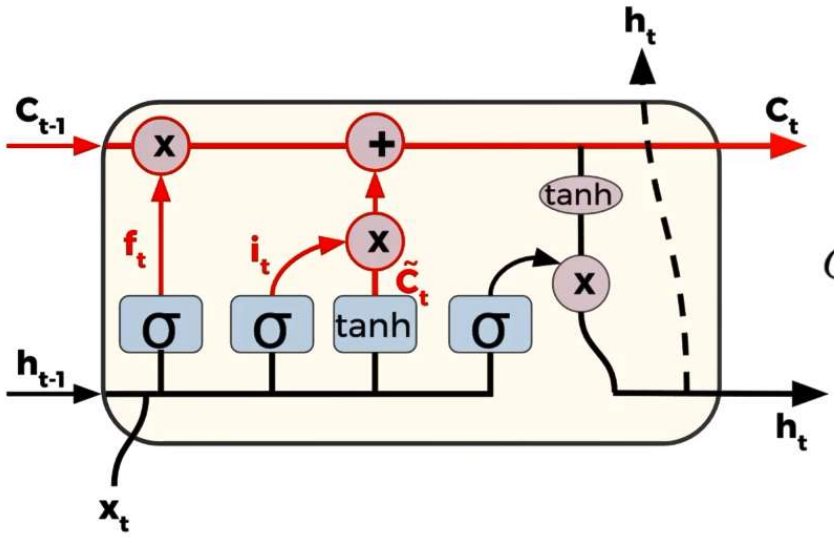


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- 2 Part: STORE GATE LAYER . What kind of information we are going to store in the cell state(Ct)?
  The sigmoid layer : input gate layer
  The tanh layer: new candidate value
  → combine both!

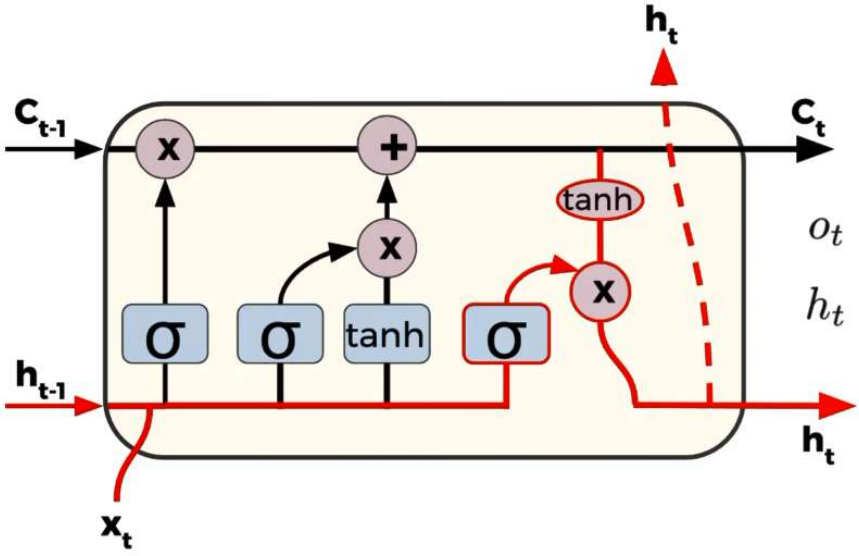
Now it's time to update the new cell state!

(Ct-1= old cell state)



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

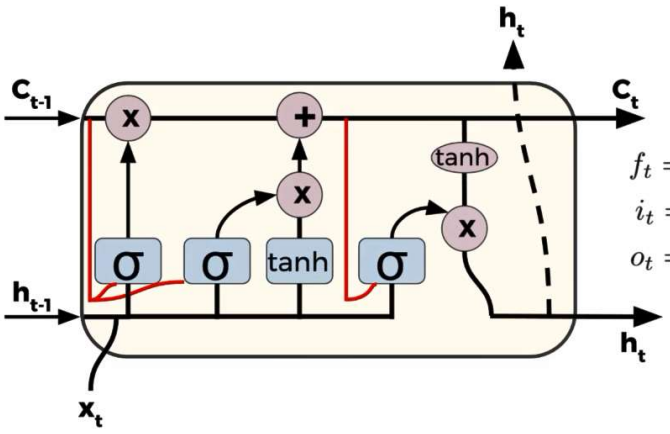
- Final Decision: What do we output for ht?



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

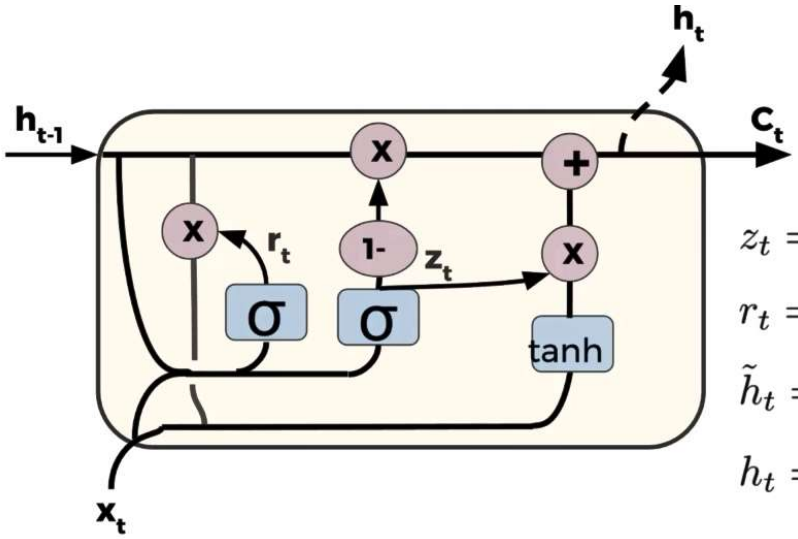
$$h_t = o_t * \tanh\left(C_t\right)$$

An LSTM Cell with “peepholes”



$$f_t = \sigma\left(W_f \cdot [C_{t-1}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [C_{t-1}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [C_t, h_{t-1}, x_t] \; + \; b_o\right)$$

Gated Recurrent Unit (GRU)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- Fortunately Keras has a really nice API that makes LSTM and RNN easy to work with.

LSTM

Text Generator

- Create the LSTM Based Model
- Split the Data into Features and Labels
  - X Features (First n words of Sequence)
  - Y Label (Next Word after the sequence)
- Fit the Model