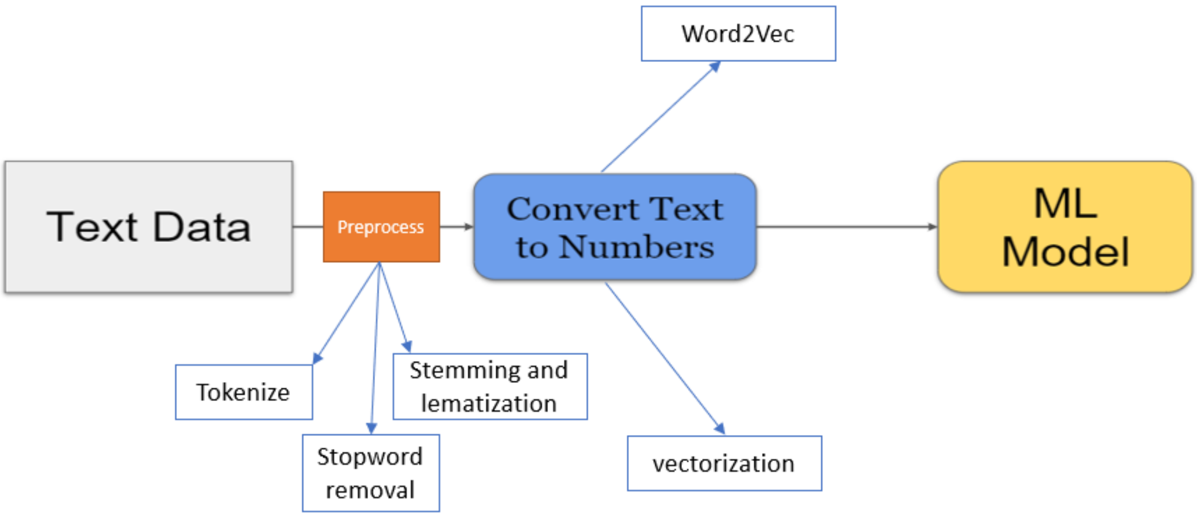
Notes by: Sujan Shirol

Lecture by: Kathirmani Sukumar

NLP Rough Notes

- Corpus: A body of text samples
- Document: A text sample
- Vocabulary: A list of words used in the corpus
- Language model: How the words are supposed to be organized



Tokenization

Chopping up text into pieces called tokens. Each word in a sentence is token.

Method 1

```
docs = tweets['text'].str.lower().str.replace('[^a-z\s#@]', '') # remove every
thing other than alphabets, spaces, # , @
docs_tokens = docs.str.split(' ')

tokens_all = []
for tokens in docs_tokens:
    tokens_all.extend(tokens)
print('No. of tokens in entire corpus:', len(tokens_all))
```

Method 2

```
from keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer()
tokenizer.fit_on_texts(train_x)
```

Stop words

- Articles: a, an, the
- Common verbs: is, was, are
- Pronouns: he, she, it
- Conjunctions: for, and
- Prepositions: at, on, with

Method 1

```
import nltk # natural language tool kit
nltk.download('stopwords')
common_stopwords = nltk.corpus.stopwords.words('english')
custom_stopwords = ['amp', 'rt']
all_stopwords = np.hstack([common_stopwords, custom_stopwords])
df_tokens = pd.DataFrame(tokens_freq).reset_index().rename(columns={'index': 'token', 0: 'frequency'})
df_tokens = df_tokens[~df_tokens['token'].isin(all_stopwords)]
```

Method 2

```
from gensim.parsing.preprocessing import remove_stopwords
```

```
remove_stopwords('this movie is really pathetic')
```

```
'movie pathetic'
```

Word Cloud

```
docs = tweets['text']
docs_strings = ' '.join(docs)
wc = WordCloud(background_color='white', stopwords=all_stopwords).generate(docs_strings)
plt.figure(figsize=(20,5))
plt.imshow(wc)
plt.axis('off');
```

Stemming

Stemming –chopping off the end of words

Nannies become nanni

Caresses become caress

PorterStemmer

```
from gensim.parsing.preprocessing import PorterStemmer
docs = imdb['review'].str.lower().str.replace('[^a-z\s]', '')
docs = docs.apply(remove_stopwords)
docs = stemmer.stem_documents(docs)
```

Lemmatization

Lemma of a word is a more exact task than stemming. Perform lemma rather than stemming.

Method 1

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = imdb['review'].iloc[0]

proc_doc = nlp(doc)
for token in proc_doc:
    print(token, '|', token.lemma_, '|', token.pos_)
```

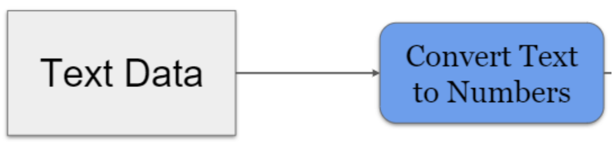
Method 2

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

lemmatizer.lemmatize("rocks")

Vectorization



Document #1

He is a good boy. She is also good.

Document #2

Radhika is a good person.

Vocabulary

a, also, boy, good, He, is, person, She, Radhika

	a	also	boy	good	He	Is	person	She	Radhika
Index	0	1	2	3	4	5	6	7	8
Document #1	1	1	1	2	1	2	0	1	0
Document #2	1	0	0	1	0	1	1	0	1

CountVectorizer

```
from sklearn.feature_extraction.text import CountVectorizer
docs = imdb['review'].str.lower().str.replace('[^a-z\s]', '')
train_docs, test_docs = train_test_split(docs, test_size=0.2, random_state=1)
stopwords = nltk.corpus.stopwords.words('english')
vectorizer = CountVectorizer(stop_words=stopwords, min_df=10).fit(train_docs)
train_dtm = vectorizer.transform(train_docs)
test_dtm = vectorizer.transform(test_docs)
```

TfidfVectorizer

Advantage: Gives less weightage to most occurring words in the corpus. Why?

Frequently occurring words are not significant for differentiating docs.

For an example and formula check GL 'Session 1.pdf' page 50.

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(min_df=5).fit(train_docs)
train_dtm_tfidf = vectorizer.transform(train_docs)
test_dtm_tfidf = vectorizer.transform(test_docs)
```

Sentiment analysis

ML Model

```
naive_bayes_model = MultinomialNB().fit(train_dtm, train_y)
test_y_pred = naive_bayes_model.predict(test_dtm)
print('Accuracy score: ', accuracy_score(test_y, test_y_pred))
print('F1 score: ', f1_score(test_y, test_y_pred, pos_label='negative'))
```

Rule Based Algorithm

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()
```

```
review = 'i hate tea and i love cofee'
analyzer.polarity_scores(review)
```

```
{'neg': 0.339, 'neu': 0.275, 'pos': 0.385, 'compound': 0.128}
```

Calculation of neg, pos, neu and compound scores

Take the above sentence as an example ‘I hate tea and I love coffee’.

Step 1:

Each word is given a score/weight according to VEDAR sentiment. Check the full list here.

I -> Ignored (stopword)

Hate -> -2.7

Tea, and, coffee ->0

Love -> 3.2

Step 2:

Increment weights by 1

I -> Ignored (stopword)

Hate -> -3.7

Tea, and, coffee ->1

Love -> 4.2

Step 3:

Total = 3.7 + 1 + 1 + 1 + 4.2 = 10.9

Pos = % of positive score = 4.2/10.9 = 0.385

Neg = % of negative score = 3.7/10.9 = 0.339

Neu = % of neutral score = 3/10.9 = 0.275

compound scores = $\dfrac{\text{Total score}}{\sqrt{\text{Total score}^2 + \underbrace{\text{alpha}}_{15}}}$

= total score before step 2 (increment) is 0.5

= 0.5 / np.sqrt(np.square(0.5) + 15)

= 0.128

Word2Vec

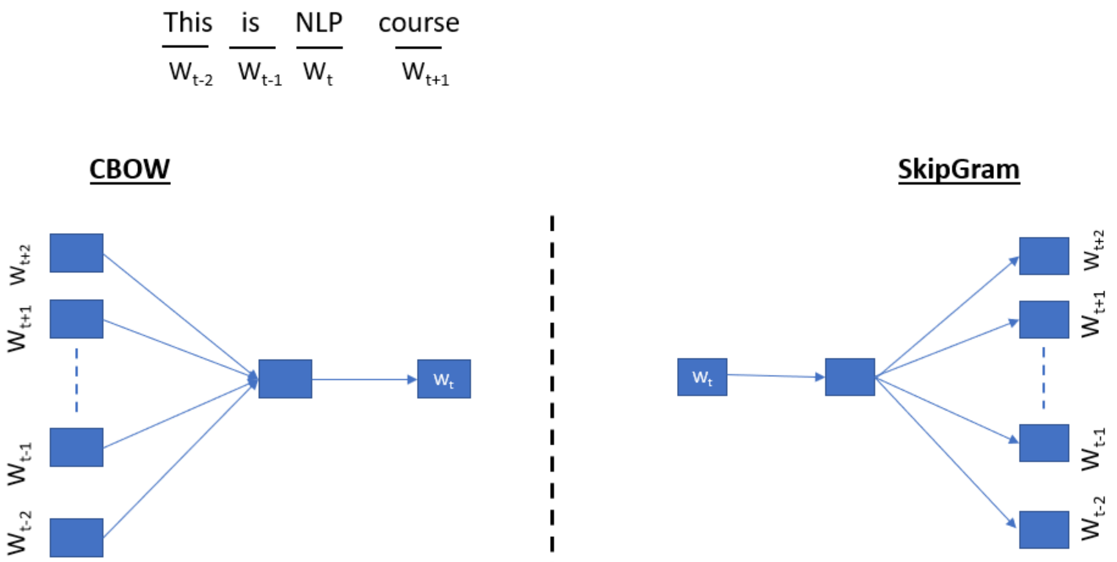
Overcome disadvantages of vectorization:

1. The order of the words in a sentence is not considered
2. Context of a sentence gets missed out

Objective:

1. Should be dense, not sparse like vectorization methods
2. Lower dimension, typically 300 vocabs
3. Represent meaning of the word
4. Should be comparable with each other

Two methods: **C**ount **B**ag **o**f **W**ords and SkipGram



CBOW

- Predict centre word given context words
- Faster than SkipGram

SkipGram

1. Predict context word given centre words
2. Slower than CBOW

SkipGram

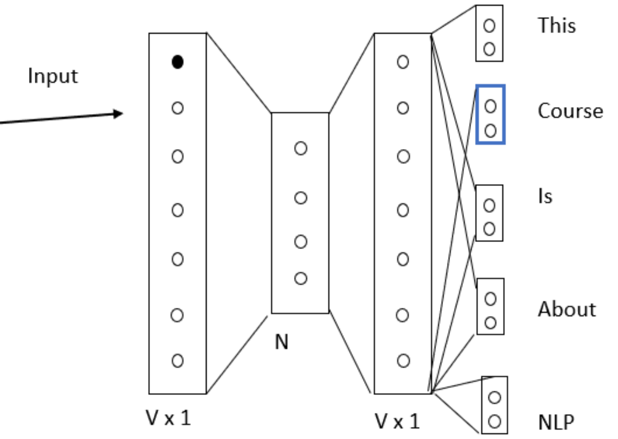
Sentence:

This	Course	is	about	NLP

With sliding window = 1

Centre word	Context words
This	Course
Course	(this, is)
Is	(course, about)
About	(is, nlp)
NLP	about

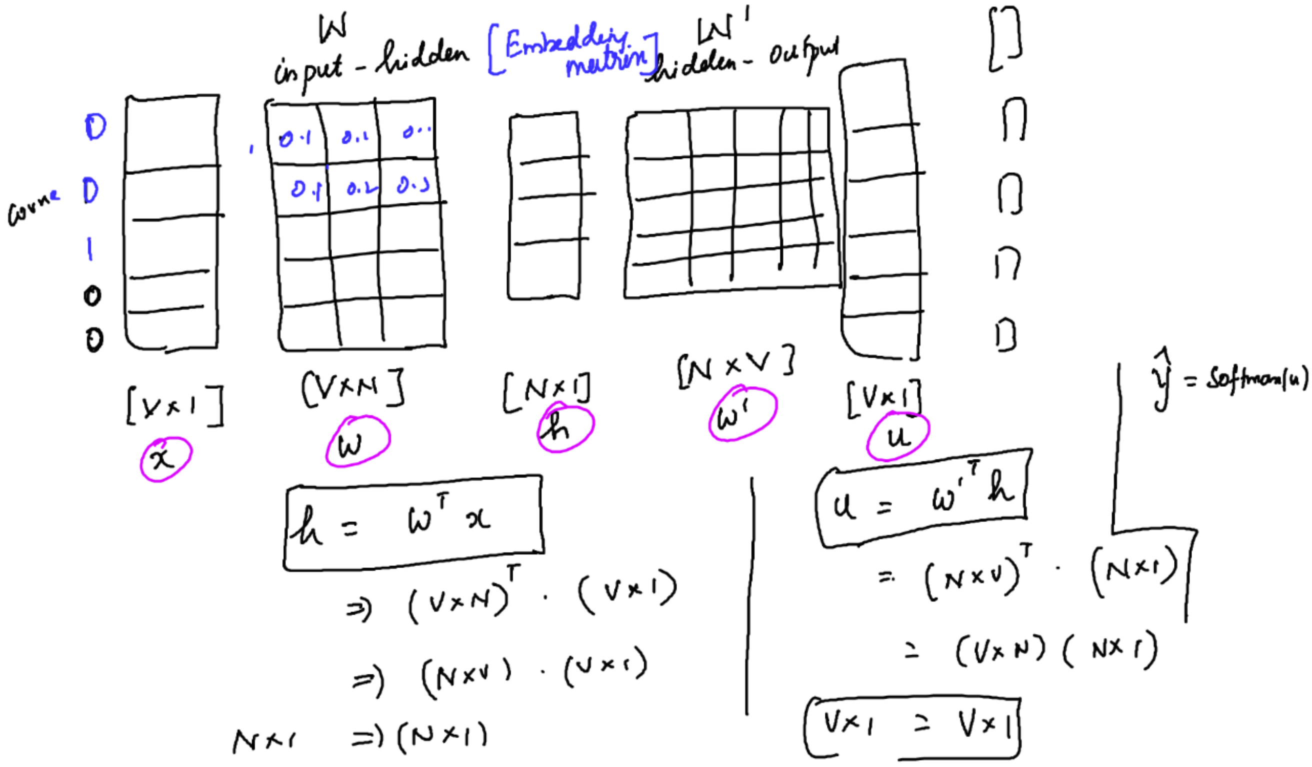
	This	Is	course	About	NLP
This	1	0	0	0	0
Is	0	1	0	0	0
Course	0	0	1	0	0
About	0	0	0	1	0
NLP	0	0	0	0	1



V -> vocab size

N -> hidden layer nodes

Row-wise input to the neural network. When row 1 is fed, ‘course’ dense layer is activated. Above this is initial architecture. Below is the current architecture used.



- Output will be word embedding matrix.
- This method is computationally challenging due to large dense layers at output end.

Negative sampling to overcome the challenge

Centre word	Context words
This	Course
Course	(this, is)
Is	(course, about)
About	(is, nlp)
NLP	about

Add noise →

1 context pair to 1 context word

Centre word	Context words	
This	Course	+ve
this	math	-ve/noise
course	this	+ve
Course	is	-ve/noise
course	science	-ve/noise

- Convert context pair into single context word
- Introduce noise, meaning, context word that does not exist in the document and label it as -ve or 0.
- Context word that appears in the doc will be labeled +ve or 1
- Now we just have to build a binary classifier neural network that predicts 1 or 0. This significantly reduced complexity at the output layer.
- A lot of pre-trained models are available to use.

Pre-trained Glove Word2Vec embedding

```
glove_path = 'glove.840B.300d/glove.840B.300d.txt'

with zf.open(glove_path) as file:
    embeddings = {}
    for line in file:
        line = line.decode('utf-8').replace('\n', '').split(' ')
        word = line[0]
        if word in vocab:
            vector = line[1:]
            vector = [float(x) for x in vector]
            embeddings[word] = vector
embedding_dim = len(vector) embedding_matrix = np.zeros((vocab_size, em
bedding_dim)) for word, index in tokenizer.word_index.items(): if word in
embeddings: embedding_matrix[index] = embeddings[word]

model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, weights=[embedding_matri
x])
.
.
.
```

CBOW

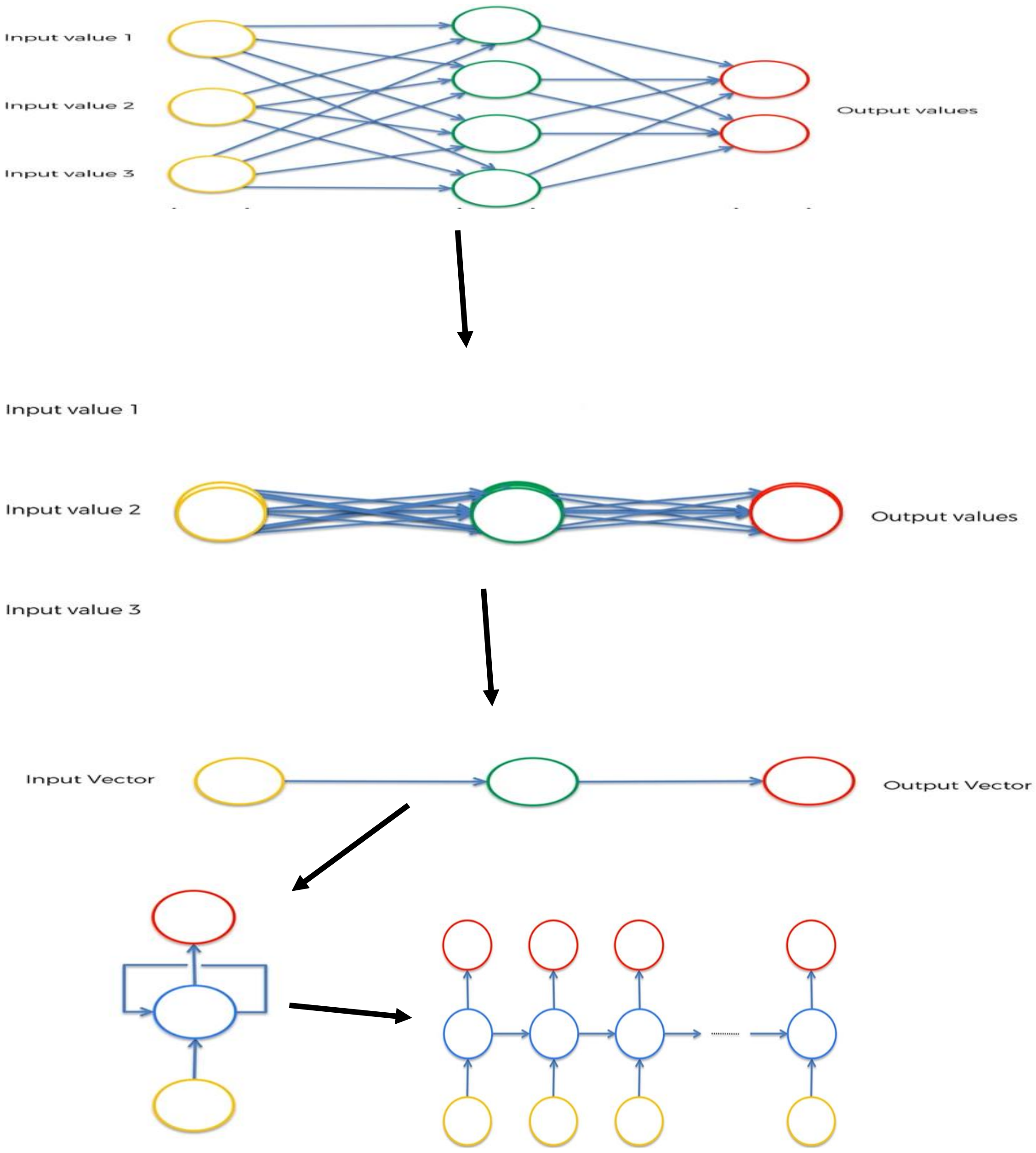
Exactly opposite of SkipGram

Self-trained SkipGram and CBOW example

```
# Create CBOW model

model1 = gensim.models.Word2Vec(data, min_count = 1, size = 100, window = 5)

model1.similarity('alice', 'wonderland')


# Create Skip Gram model

model2 = gensim.models.Word2Vec(data, min_count = 1, size = 100, window = 5, sg = 1)

model2.similarity('alice', 'wonderland')
```

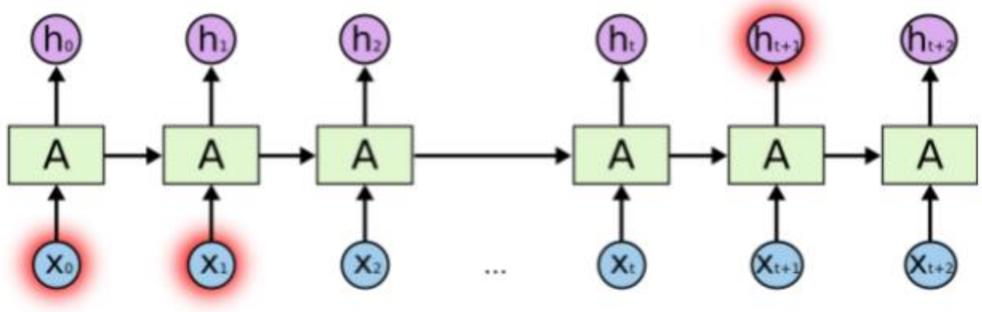
RNN

ANN Model



- RNN is a compressed version of ANN.
- It has a temporal loop in between.
- Common way to represent is to unwind the loop (last fig).
- This architecture will allow the network to have memory of previous input.

Disadvantages:



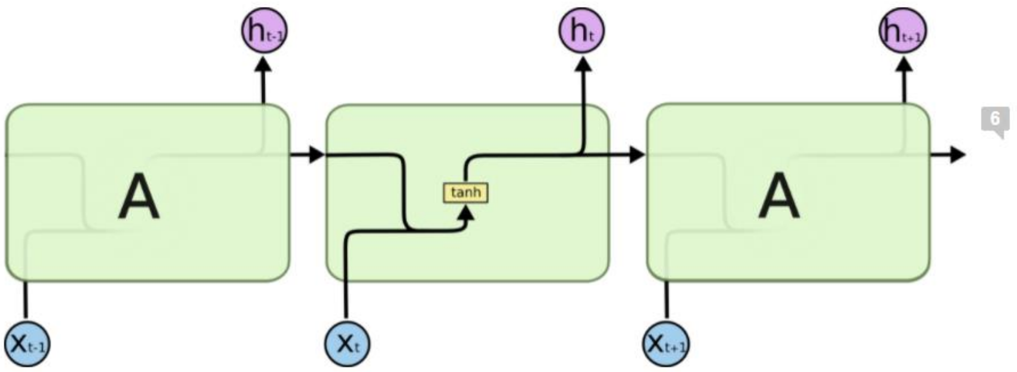
1. Vanishing gradient
2. **Long-Term Dependency Problem**: Information of x_0 will get lost by the time it reaches output node
3. Middle data (x_1 or x_3 or ...) might not be helpful

LSTM overcomes these disadvantages.

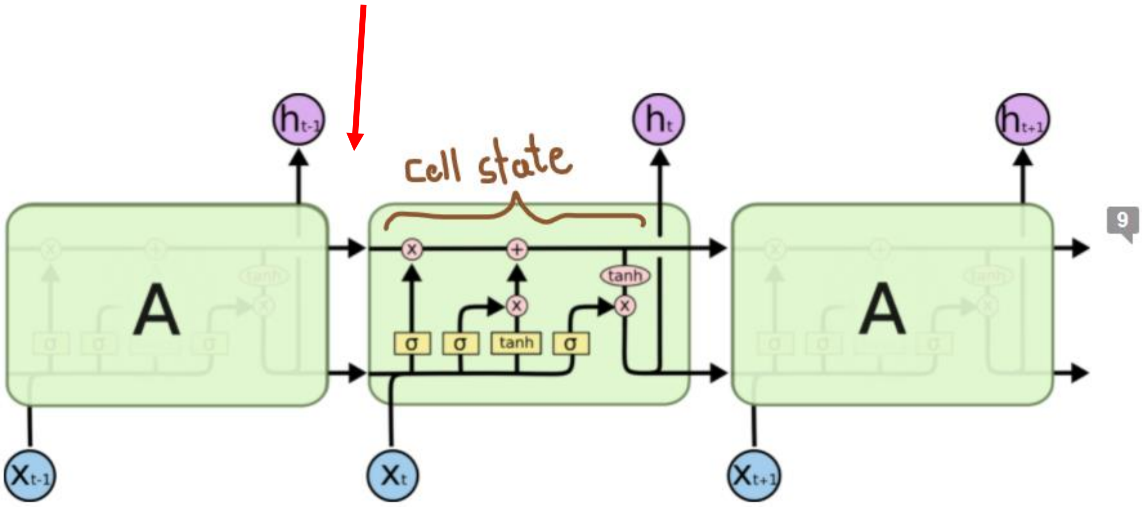
LSTM

LSTMs are explicitly designed to **avoid the long-term dependency problem**. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

- They are carefully regulated by structures called **gates** (yellow boxes in below diagram)
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.
- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”
- The key to LSTMs is the **cell state**, the horizontal line running through the top of the diagram.

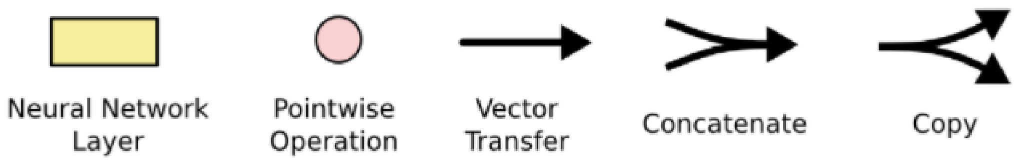


The repeating module in a standard RNN contains a single layer.

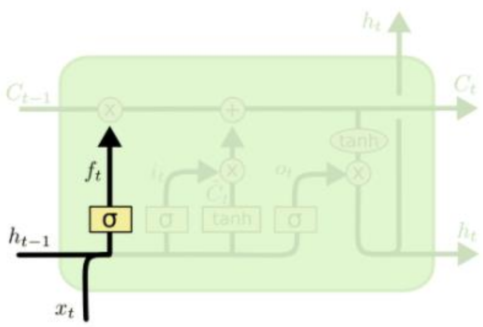


The repeating module in an LSTM contains four interacting layers.

Notations:



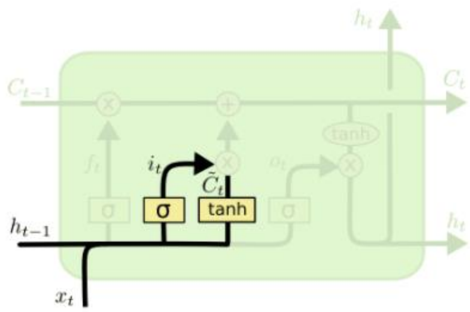
Steps 1:



- The first step in our LSTM is to decide what information we're going to throw away from the cell state.
- This decision is made by a sigmoid layer called the "**forget gate layer**." It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.

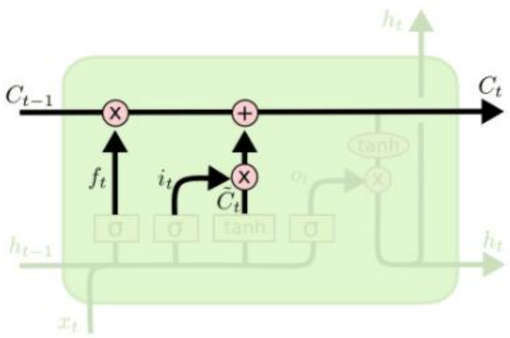
Courtesy: [Colah's blog]

Step 2:



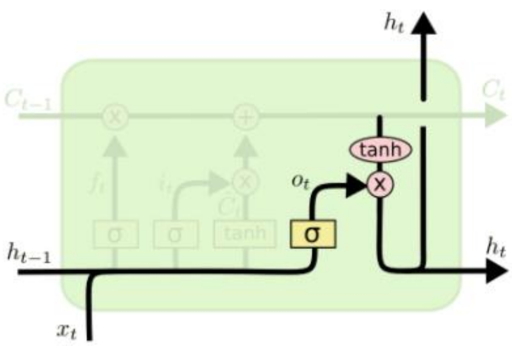
- The next step is to decide what new information we're going to store in the cell state.
- First, a sigmoid layer called the "**input gate layer**" decides which values we'll update.
- tanh layer creates a vector of new candidate values, $\tilde{C}_t$

Step 3:



- Update the old cell state, C_{t-1}, into the new cell state C_t.
- We multiply the old state by f_t, forgetting the things we decided to forget earlier.
- Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

Step 4:



- Decide what we're going to output
- sigmoid layer which decides what parts of the cell state we're going to output.
- Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we

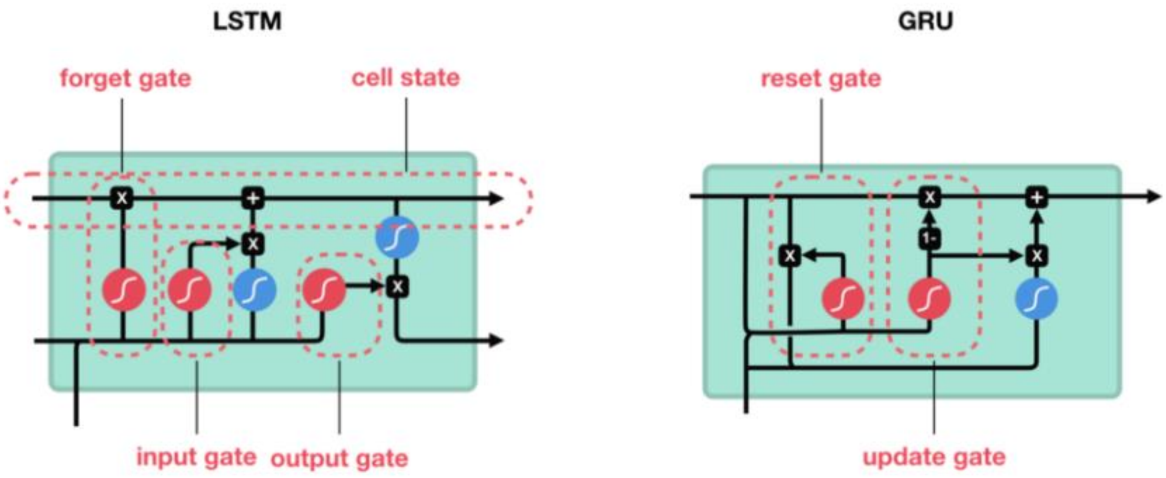
```
# units: no.of LSTM memory cells(neurons)

# return_sequences: True if more than 1 LSTM layes

model.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1], 1)))

model.add(LSTM(units = 50, return_sequences = False))

#output layer

model.add(Dense(units = 1))
```

Courtesy: [Colah's blog]

- Opening and closing of gates is not controlled manually. The neural network will become smart enough to control it over time.
- Hence, each gate is a separate neural network.
- Each input(X_{t-1}, X_t, X_{t+1}) and output(h_{t-1}, h_t, h_{t+1}) nodes are not just a single node. They are series of nodes, one behind the other, not visible in 2D diagrams.

GRU

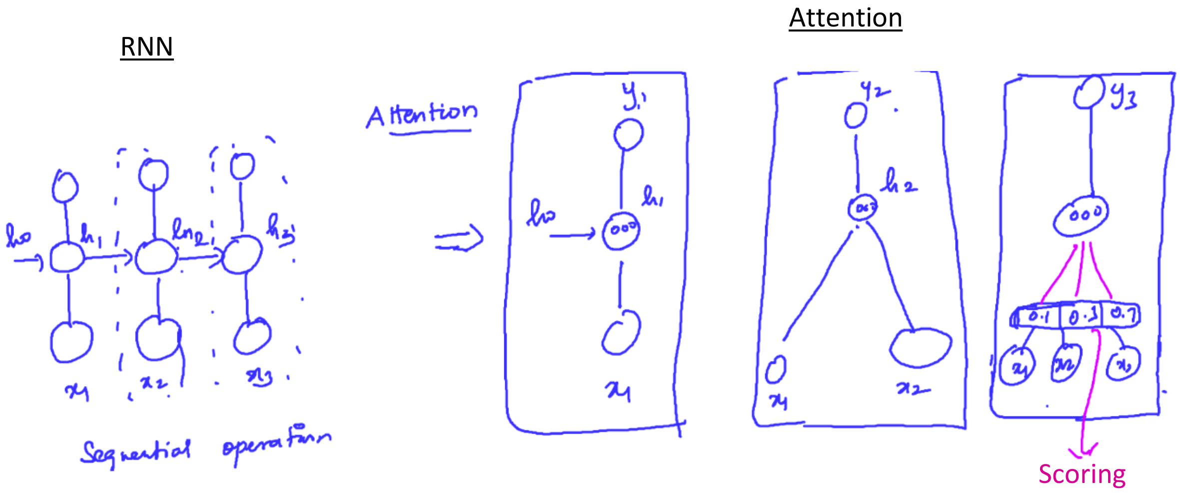
LSTM has many gates, which results in overfitting most of the time. GRU is a simplified version of LSTM with a fewer number of gates.



Amazing intuition video of RNN, LSTM and GRU. Animated explanation of working of gates: https://www.youtube.com/watch?v=8HyCNIVRbSU

Attention method

Alternate structure of RNN. What attention is given to each input and process the model in parallel. This attention method will be used in Encoder-Decoder.



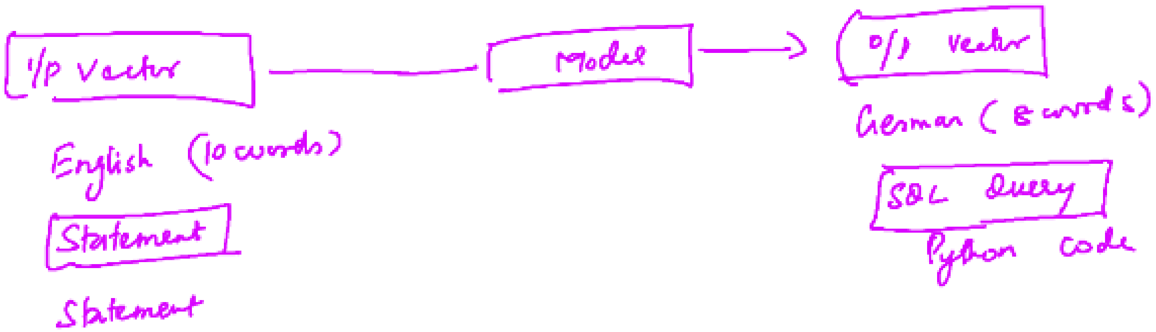
- Sequential operation. Without comouting h_1 you can not compute h_2. Interdependent
- Hence, time to compute RNN is very high

- Separate parallel units for each h_i.
- Every input is subsequently passed.
- At every instance h_i, we add new input x_i.
- Execute each unite in parallel.
- Scoring mechanism to each input x_i. Weights of each input x_i is computed by the model.
- The weigh represents how much of information from heach x_i is passed to the network.

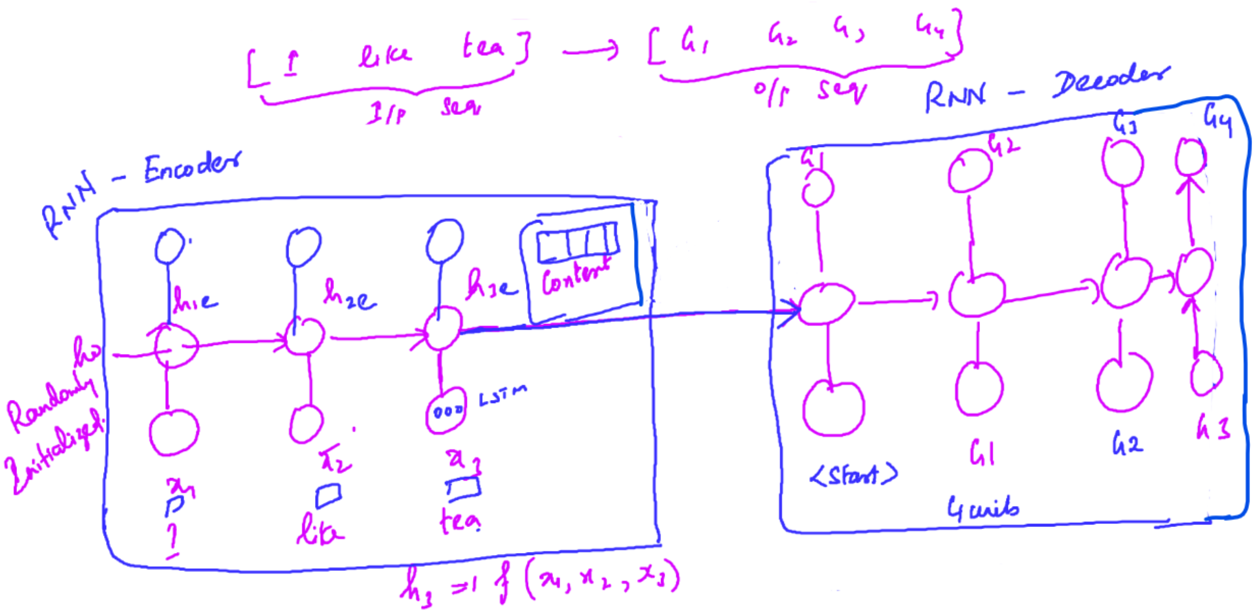
Encoder-Decoder

Another variant of RNN. Vector to a vector model. Applications:

1. Machine translation (one language to another)
2. sequence-to-sequence modeling
3. Natural language query -> SQL query
4. Natural language -> program

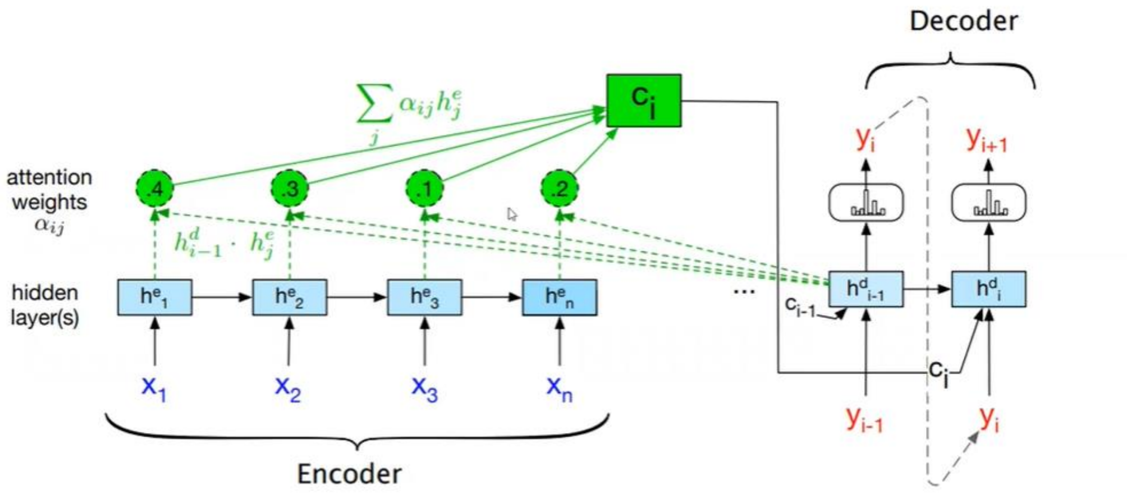


- Consider an example for translating english sentence “I like tea” into German words “G_1 G_2 G_3 G_4”.
- Notice, number of words in output sentence might to be equal to number of words in input sentence.
- Which is why, words by words prediction is not perfomed, that is, I->G_1, like->G_2. Instead process the entire sentence and then start prediction.
- What we get at the end of processing entire sentence is called a context.
- This approach has two RNN’s one for encoder and one for decoder.



- Encoder is to process input sentence and output a context.
- Think of encoder like a PCA, consolidate all the feature values and the context vector as different PC’s
- H_0 to encoder is randomly initializer or set to zero.
- Input for decoder will be the condext which is used to predict the German words.
- Each predicted German word is inputed to the next unit in the network.
- Decoder is where we have attention machanism.

Attention mechanism in the decoder



- We learned that we need the final output context as input for the decoder.
- In the improved version, we should also pass each intermediate hidden state of the encoder as decoder input.
- There intermediate hidden state should be weighted.This is where the attention mechanism kicks in.

Encoder-Decoder code with comments [link]