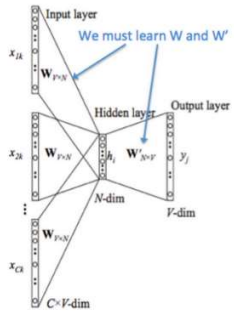
Notes: Natural Language Processing

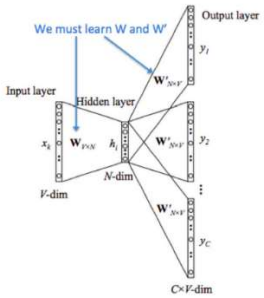
Wonderfully simple lecture on "ML 4 LM, n-gram smoothing"
http://www.marekrei.com/pub/Machine_Learning_for_Language_Modelling_-_lecture2.pdf

Embedding & Word Embedding: -

- Embedding is a very powerful technique to handle problem data sparsity
    - Matrix Factorization
    - Representation Learning: NOTE that layers of Representation Learning in Deep Learning is actually learning of hierarchy of features / semantics of subject matter. Must refer to https://arxiv.org/pdf/1206.5538.pdf
    - Word2vec & later GloVec are pre-trained Representation Learning or Word Embedding to identify
- In fact, any categorical variable can be represented by lower dimensional embedding vector. Fast.ai lecture #4 uses this funda. One simple example is representing days of week by either weekday or weekend but NOTE that this lower order embedding must be learned in the given context => it all boils down to throwing away redundant information => how to choose the order of lower dimension has to be guided by domain knowledge
- In word2vec, each word is represented as low dimensional vector instead of 1-hot vector
    - Low dimension vector captures Word SEMANTICS, i.e., meaning and this results in similar meaning words represented close to each other in cosine-distance sense
    - Predicting next word using word2vec https://stackoverflow.com/questions/36780491/predicting-next-word-with-text2vec-in-r?rq=1
    - https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md pre-trained word vectors by Facebook research
    - CBOW -vs- Skip-gram:
        - Refer to Iterative LM section in this concise notes on https://cs224d.stanford.edu/lecture_notes/notes1.pdf for below topics
        - CBOW: "predicting the word given its context"… wi−2,wi−1,wi+1,wi+2 => wi



        - Skip-gram: "predicting the context given a word"… wi => wi−1,wi−2,wi+1,wi+2



        - Negative Sampling: Refer to Andrew N G deeplearing.ai short lecture

Plain Vanilla RNN -vs- LSTM/GRU: The criteria is the performance on the validation set. Typically LSTM outperforms RNN, as it does a better job at avoiding the vanishing gradient problem, and can model longer dependences. Some other RNN variants sometimes outperform LSTM for some tasks, e.g. GRU.

RNN/LSTM/GRU Backpropagation

- RNN is in fact called Vanilla RNN and not really used in practice due to its vanishing gradient issue & hence not able to handle reasonable temporal dependency! LSTM was invented in 1997 specifically to avoid the vanishing gradient problem.
- Hidden State is the key of RNN*… aka Cell State in LSTM
- Backpropagation is temporal in RNN/LSTM/GRU & it is called BPTT (BP-through-time => actually NN network design itself is temporal
- Stacking RNN* with DNN as in CS231n lec#10 (VGGNet-output-layer feeding into RNN* for Image Captioning) is altogether a decoupled spatial-temporal problem, i.e., ignore stacking or spatial-backpropagation
- Truncated BPTT: In vanilla RNN, it is natural for gradient to vanish if we go too far back in time to remember long temporal dependency… In this context, I guess there is no concept of "real-time or online learning of time series with RNN* or some kind of moving-time-window"
- Crux of LSTM: Ref to skymind.ai wiki on LSTM… Instead of determining the subsequent cell state by multiplying its current state with new input, **they add the two (aka CEC Constant Error Carausel)… forget gate says to either forget or add error from previous time step as-is**, and that quite literally makes the difference.

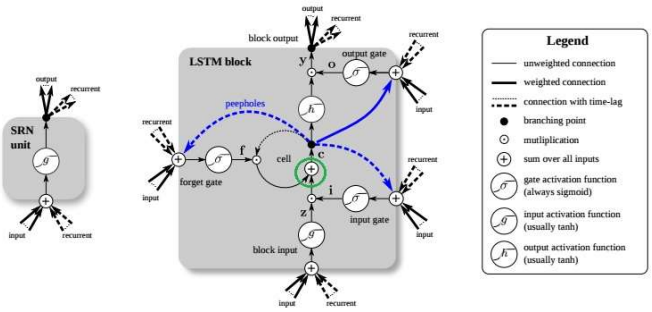


Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

- Why is LSTM called so? => Though nowhere mentioned, my guess is that it retention of temporal memory is adaptively long & short.
- [??] What is the mechanism of forget get deciding 0 or 1??
- Which parameter or hyperparameter of LSTM is for T-BPTT if at all it is applicable to LSTM too? => There is no such parameter for LSTM because truncation is adaptively controlled by 'forget' date and LSTM can remember long history per use case.
- [??] Any specific use case scenario where LSTM remembers long history??
- LSTM variants & their relative comparison using fANOVA framework (random Grid Search) - https://arxiv.org/pdf/1503.04069.pdf, Oct 2017

Stop words assessment

- Process blogs/news/twitters separately => to get helpful insight in deciding whether to remove Stop Words or not
- Rank words as per tf-idf (refer to https://cran.r-project.org/web/packages/tidytext/vignettes/tf_idf.html & http://www.tfidf.com/)
  - tf-idf is a weight often used in information retrieval and text mining
  - It measures how important a word is to a document in a collection or corpus
  - The importance increases proportionally to the number of times a word appears in the document (tf) but is offset by the frequency of the word in the corpus (idf – inverse log)
  - Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query, e.g., summing the tf-idf for each query term
  - Stop words filtering can be aided by tfidf

  TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).
  IDF(t) = log10(Total number of documents / Number of documents with term t in it).
  Tfidf = TF(t) * IDF(t)
- Identify reference list of Stop Words
- Check how many of Stop Words are in the top 80% content (it would mean how many rows to list in tf-idf ranking by-trial-error to get 80% words)
- Check the proportion of Stop Words in top 80% words
- Identify PREDICTION ACCURACY measurement approach => extract small population (get random rows indices & freeze) and extract those exact rows/sample WITH/WITHOUT Stop Words

An analysis of the Brown Corpus has shown that the hundred most frequent words account for 42% of the corpus, while only 0.1% in the vocabulary. On the other hand, words occurring only once account merely 5.7% in the corpus but 58% in the vocabulary (Bell et al. 1990).
=> **Zipf's Law**

Models that assign probabilities to sequences of words are called **language model or LMs**.

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{C(its\ water\ is\ so\ transparent\ that\ the)}{C(its\ water\ is\ so\ transparent\ that)}$$

The intuition of the N-gram model is that instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words.

'ngram' probability estimation: -

- Add one smoothing (for unseen ngram… nu is language model dictionary size… MOST NAÏVE as this change is applied to all SEEN/UNSEEN)

$$q(w_i|w_{i-2}, w_{i-1}) = \frac{1 + count(w_{i-2}, w_{i-1}, w_i)}{|\mathcal{V}| + count(w_{i-2}, w_{i-1})}$$

- Stupid Backoff: https://rpubs.com/pferriere/dscapreport
- **Katz Backoff** smoothing… it is a king of Discounting method https://www.youtube.com/watch?v=flZdbT73it8&t=626s
  - Re-distribute probability mass from SEEN n-1_grams (missing probability mass… it is 5/48 from SEEN instances in below pic) to UNSEEN n-1_grams

Let's define discounted counts: $count*(w) = count(w) - 0.5$

New estimates:

w	$count(w)$	$count*(w)$	$count*(w)/count(the$
the	48		
the,dog	15	14.5	$14.5/48$
the,woman	11	10.5	$10.5/48$
the,man	10	9.5	$9.5/48$
the,park	5	4.5	$4.5/48$
the,job	2	1.5	$1.5/48$
the,telescope	1	0.5	$0.5/48$
the,manual	1	0.5	$0.5/48$
the, afternoon	1	0.5	$0.5/48$
the, country	1	0.5	$0.5/48$
the,street	1	0.5	$0.5/48$

for a bigram model we define two sets

$\mathcal{A}(w_{i-1}) = \{w : count(w_{i-1}, w_i) > 0\}$

$\mathcal{B}(w_{i-1}) = \{w : count(w_{i-1}, w_i) = 0\}$

Then:

$$q_{BO}(w_i|w_{i-1}) = \begin{cases} \frac{count*(w_{i-1},w)}{count(w_{i-1})} & \text{if } w_i \in \mathcal{A}(w_{i-1}) \\ \alpha(w_{i-1}) \frac{q_{ML}(w_i)}{\sum_{w \in \mathcal{B}(w_{i-1})} q_{ML}(w)} & \text{if } w_i \in \mathcal{B}(w_{i-1}) \end{cases}$$

where

$$\alpha(w_{i-1}) = 1 - \sum_{w} \frac{count*(w_{i-1}, w)}{count(w_{i-1})}$$

for a trigram model we define two sets

$\mathcal{A}(w_{i-2}, w_{i-1}) = \{w : count(w_{i-2}, w_{i-1}, w_i) > 0\}$

$\mathcal{B}(w_{i-2}, w_{i-1}) = \{w : count(w_{i-2}, w_{i-1}, w_i) = 0\}$

The trigram model is defined in terms in the bigram.

$$q_{BO}(w_i|w_{i-2}, w_{i-1}) = \begin{cases} \frac{count*(w_{i-2},w_{i-1},w_i)}{count(w_{i-2},w_{i-1})} & \text{if } w_i \in \mathcal{A}(w_{i-2}, w_{i-1}) \\ \alpha(w_{i-2}, w_{i-1}) \frac{q_{ML}(w_i|w_{i-1})}{\sum_{w \in \mathcal{B}(w_{i-2},w_{i-1})} q_{ML}(w|w_{i-1})} & \text{if } w_i \in \mathcal{B}(w_{i-2}, w_{i-1}) \end{cases}$$

where

$$\alpha(w_{i-2}, w_{i-1}) = 1 - \sum_{w \in \mathcal{A}(w_{i-2},w_{i-1})} \frac{count*(w_{i-2}, w_{i-1}, w)}{count(w_{i-2}, w_{i-1})}$$

- NOTE that Katz Backoff is a RECURSIVE approach of finding discounted/adjusted probability of certain n-gram
  - If given n-gram is SEEN then p_kbo = Good Turing pseudo estimate based c'(given n-gram) / c(given n-1-gram)
  - If given n-gram is UNSEEN then p_kbo = alpha(i.e., missing probability mass of all n-grams) * p_kbo(given n-1-gram => if SEEN otherwise RECURSE till we identify SEEN smaller i-gram)

- It seems that the DISCOUNTED COUNT, i.e., pseudo count for each SEEN & UNSEEN n-gram in **Katz Backoff** is determined by **Good Turing** count estimation
  - Good-Turing estimation
    - derived from leave-one-out discounting
    - if a word occurred ***n*** times, its "adjusted" frequency is:
    - $n^* = (n+1)\, E\{\#_{n+1}\} / E\{\#_n\}$
    - probability of that word is: n^* / N^*
    - $E\{\#_n\}$ is the number of words with ***n*** occurrences
    - $E\{\#_n\}$ very unreliable for high values of ***n***

V = {a,b,c,d,e,f,g,h,i}
D = "a a a b d d e f g g"
|D| = 13
$n_{ML}(c)=0$ $n_{GT}(c) = (0+1) * 3/3 = 1$
$n_{ML}(b)=1$ $n_{GT}(b) = (1+1) * 2/3 = 1.3$
$n_{ML}(d)=2$ $n_{GT}(d) = (2+1) * 1/2 = 1.5$

  - Pseudo_count_of_c = [0(UNSEEN) + 1] * [3('b', 'e', 'f')] / [3(how many such 0-freq, i.e., UNSEEN words are there, i.e., 'c', 'h', 'I')
  - Pseudo_count_of_b = [1(SEEN once) + 1] * [2(no of words with +1, i.e., 2 occurances, i.e., 'd' & 'g')] / [3 (how many such 1-freq words are there, i.e., 'b', 'e', 'f')]
- Kneser-Ney Smoothing: It is a recursive process of finding probability of a word in a unfamiliar/unseen context https://github.com/rumbaugh/NextWordPredict/blob/master/R/calc_ngram_probs.R
  - Absolute discounting

$$P_{\text{AbsoluteDiscounting}}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i) - d}{c(w_{i-1})} + \lambda(w_{i-1}) P(w)$$

(discounted bigram; Interpolation weight; unigram)

  - Kneser-Ney approach helps estimate lower-order-probability P(w) in a better way than Absolute Discounting => proposed $P_{continuation}(w)$
  - $P_{continuation}(w)$ is "how likely is w to appear as a novel continuation"=> [??] useful for unobserved bi-grams [??] => count of bigrams completed by 'w' as last word => w_j means all words, i.e., all bi-grams

- How many times does w appear as a novel continuation:

$$P_{CONTINUATION}(w) \propto \left|\{w_{i-1} : c(w_{i-1}, w) > 0\}\right|$$

- Normalized by the total number of word bigram types

$$\left|\{(w_{j-1}, w_j) : c(w_{j-1}, w_j) > 0\}\right|$$

$$P_{CONTINUATION}(w) = \frac{\left|\{w_{i-1} : c(w_{i-1}, w) > 0\}\right|}{\left|\{(w_{j-1}, w_j) : c(w_{j-1}, w_j) > 0\}\right|}$$

- Kneser-Ney algo

$$P_{KN}(w_i \mid w_{i-1}) = \frac{\max(c(w_{i-1}, w_i) - d, 0)}{c(w_{i-1})} + \lambda(w_{i-1}) P_{CONTINUATION}(w_i)$$

$$\lambda(w_{i-1}) = \frac{d}{c(w_{i-1})} \left|\{w : c(w_{i-1}, w) > 0\}\right|$$

the normalized discount

The number of word types that can follow w_{i-1}
= # of word types we discounted
= # of times we applied normalized discount

Project references: -

- http://www.sthda.com/english/wiki/text-mining-and-word-cloud-fundamentals-in-r-5-simple-steps-you-should-know
- Read lines => create Corpus
  - [??] utf8 –vs- latin1, which one to use while reading lines from news/twitters/blogs??
  - [??] How handle Russian files??
  - [??] extract text only (r >> getText)
- Cleaning
  - Pruning based on min_docfreq => text_stat_frequency / topFeatures… all need DFM
  - [??] Tokenization, Lemmatization / Stemmer
    - Avoid costly LEMMATIZATION
    - Stemmer is expected to reduce the dimension of the tokens
  - [??] emoji & NLP
  - Replace special chars (/@\)
  - Remove white space
  - Tolower => Before building the word frequency tables (explained in the next section) all text is converted to lowercase. As a consequence the prediction model will predict lowercase words only. For English text this is less of an issue than for a language like German that uses capitalized nouns.
  - [??] Remove stop words
    - if we drop words like 'the', then how do we form n-gram (2-gram or 3-gram)?? Are n-gram not supposed to involve 'stop words'??
    - Find how many of the most frequent words are Stop Words
    - Find the % of Stop Words of total words
  - Remove numbers => Numeric characters that may appear in numbers and dates are also removed. The combination of certain numbers and words can have predictive value, but their frequency might be too low to be useful.
  - Remove punctuations => We will remove punctuation characters such as commas, parentheses and the like, under the assumption that they have little impact on word order and n-gram composition.
  - Remove URL => removeURL <- function(x) gsub("http:[[:alnum:]]*", "", x)
  - Remove profanity words => Initially we will not treat typos, garbage words and wrong language any differently from "proper" words, under the assumption that they represent a minor portion of the text. Profanity words will be removed.
- Term-document-matrix => word frequency, word cloud
- Language Coverage => ratio of unique words (or n-grams) and total number of occurrences of these words (or n-grams)

NLP pre-processing

- Tokenization
- Normalization
  - to lower case
  - remove periods
  - remove punctuations
  - remove numbers
  - remove STOP WORDS as they don't add SYMANTICS / predictive power
  - remove symbols (<@# etc – but social media text analytics may not remove @#)
  - Fix names (4m, 2nd, 1st, 8am, etc)
- Asymmetric Expansion (for searching 'window/windows/Windows' when user wants to search 'window')
- Lemmatization (reduce variant forms to base form => "reading" -> "read"; "am/are/is" -> "be"; "car/cars/car's/cars'" -> "car")
- Stem-ming
  - Difference b/w Lemmatization & Stemming? https://stackoverflow.com/questions/17317418/stemmers-vs-lemmatizers

- Stemmer: A function that reduces inflectional forms to stems or base forms, using rules and lists of known suffixes.
- Lemmatizer: A function that performs the same reduction, but using a comprehensive full-form dictionary to be able to deal with irregular forms.
- Lemmatizer is essentially a higher-quality (and more expensive) version of a stemmer. => One classical application of either stemming or lemmatization is the improvement of search engine results => it is not actually important whether the stem (or lemma) is meaningful ("have") or not ("hav"). It only needs to able to represent the word in question, and all its inflectional forms. In fact, some systems use numbers or other kinds of id-strings instead of either stem or lemma (or base form or whatever it may be called). => Ultimate requirement is to clusters inflectionally related words together, hence Stemmer –vs- Lemmatizer does not matter much!

Language Modelling

- Estimating probability of full-sentence or follow-up-word occurring
- Used in
  - Machine Translation => P(**high** winds tonite) > P(**large** winds tonite)
  - Spell Correction => P(about fifteen **minutes** from…) > P(about fifteen **minuets** from….)
  - Speech Recognition => P(**I saw** a van) > P(**eyes awe** a van)
- Chain rule of conditional probability for P(full-sentence)

  The Chain Rule applied to compute joint probability of words in sentence

  $$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i \mid w_1 w_2 \ldots w_{i-1})$$

  P("its water is so transparent") =

  P(its) × P(water | its) × P(is | its water)

  × P(so | its water is) × P(transparent | its water is so)
- Refer to above long-conditional-probability
  - Markov assumption – future behaviour of a dynamical system depends only on its **recent history**
  - K[th] order Markov model – next state depends on k most recent states

    Each term approximation using Markov assumption with k-order => $P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-k} \ldots w_{i-1})$

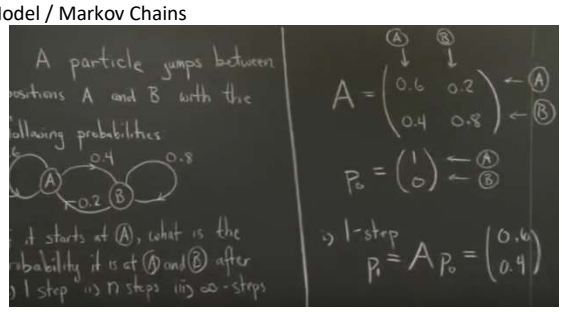
    $$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i \mid w_{i-k} \ldots w_{i-1})$$

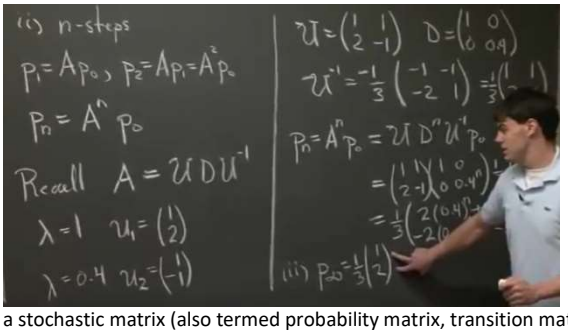
    Full sentence =>
    - Log Probability: We always represent and compute language model probabilities in log format as log probabilities. Since probabilities are (by definition) less than or equal to 1, the more probabilities we multiply together, the smaller the product becomes. Multiplying enough N-grams together would result in numerical underflow. By using log probabilities instead of raw probabilities, we get numbers that are not as small.
  - Unigram Markov model
  - N-gram model == (N-1) order Markov model

- Markov Chain / Markov Matrix / Markov Model
  - PageRank as 1 of the best 5 applications of Markov Chain http://langvillea.people.cofc.edu/MCapps7.pdf

n-gram model

- Markov Model / Markov Chains





  - a stochastic matrix (also termed probability matrix, transition matrix, substitution matrix, or Markov matrix)

- A Markov chain is "a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event".
- Store n-gram models as Markov Chains

- DFM (document frequency matrix)
  - Words as columns
  - Docs as rows
  - CAUTION! Word ordering is lost => bag-of-words

- Hidden Markov Model
- Recommendation Systems (user comments / ratings, books/movies/etc) => Trying to get into this to solve Movie-Recommendation-Matrix-Factorization Kaggle competition
  - Pivot 'ratings' columns around 'userID'
  - CENTERED COSINE SIMILARITY (center each user row to their mean)
    - Normalize ratings by subtracting row mean

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

10/3
14/3

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	2/3			5/3	−7/3		
B	1/3	1/3	−2/3				
C				−5/3	1/3	4/3	
D		0					0

    - sim(A,B) = cos(r_A, r_B) = 0.09; sim(A,C) = -0.56
      - sim(A,B) > sim(A,C)
    - Captures intuition better
      - Missing ratings treated as "average"
      - Handles "tough raters" and "easy raters"
- Recommendation Systems
  - Content based Filtering – Recommend based on movie/item content information (in practice, not useful as it seems users have evolved or conditioned to be less influenced by the content rather than user-user collaboration
  - Collaborative Filtering – Recommend based on user-user or item-item similarity but NOTE that what matters is user-user, i.e., item-item collaboration does not matter so much
- Recommender Systems >> Collaborative Filtering
  - memory based
    - nearest neighbors (user-2-user, item-2-item) => Centered Cosine Similarity (kNN should be used with COSINE SIMILARITY)
  - model based
    - latent factors
    - matrix factorization
- Overview Matrix Factorization
  - Background: Low rank Factorizations
  - Residual matrix
  - Unconstrained Matrix Factorization
    - Optimization problem
    - Gradient Descent, SGD, Alternating Least Squares
    - User/item bias terms (matrix trick)
  - Singular Value Decomposition (SVD)
  - Non-negative Matrix Factorization
- matrix factorization techniques (SVD) work with full matrix
  - ratings – sparse matrix
  - solutions:
    - value imputation – expensive, imprecise
    - alternative algorithms (greedy, heuristic):
      - gradient descent,
      - ALS (alternating least squares)

NLP algorithms: -

- Document Classification: Assigning a single category to the given text document –vs- Topic Modeling: Assigning multiple topics/tags to a text document!
- Topic modelling (common input parameters… n_topics; DWM or DTM documents term matrix => outputs… Word Topic Matrix WTM or Topic Document Matrix TDM such that their multiple results in close approximation to original DTM
  - LDA (Latent Dirichlet Allocation)
  - PCA
  - NMF (non-negative matrix factorization)

- corpus –vs- data.frame, DocumentTermMatrix, TermDocumentMatrix => DTM & TDM are simply transpose
- both are for text mining on bag-of-words (BOW)
- DTM retains all ‘D’ documents and throws away irrelevant ‘T’ terms and vica-versa

```
# Term-document matrices tend to get very big even for normal sized data sets & method to remove sparse terms
> inspect(removeSparseTerms(dtm, 0.4))
> inspect(DocumentTermMatrix(reuters, list(dictionary = c("prices", "crude", "oil"))))
```