CAP6307 - Natural Language Processing: Midterm Notes

Kobee Raveendran

Fall 2020

1 Basic Text Processing

1.1 General NLP Pipeline

1. Sentence splitting
2. Tokenization
3. Lemmatization
4. Part-of-speech tagging
5. Named-entity recognition
6. Constituency parsing
7. Dependency parsing
8. Coreference resolution

1.2 Regular Expressions

A formal language for specifying (matching) text strings.

1.2.1 Basic Regex patterns

- a, X, 9, < — explicit character matching. Does not apply to meta-characters: .^$*+?{[]\()
- `.` — period matches any character except newlines (\n)
- `\w` — matches "word" characters (a letter, digit, or underscore: [a-zA-Z0-9_]
- `\b` — boundary between word and non-word
- `\s` — matches any whitespace character – space, newline, return, tab, form – `[\n\r\t\f]`
- `\r` — return
- `\d` — decimal digit `[0-9]`
- `^, $` — start of string, end of string
- \ — escape character – inhibits the "specialness" of the above meta-characters
- `+` — 1 or more occurrences of the pattern to its left. Ex: '`i+`' matches 1 or more i's
- `*` — 0 or more occurrences of the pattern to its left
- `?` — match 0 or 1 occurrences of the pattern to its left

1.2.2 Python examples

```python
# NOTE: all of the pattern must match, but not all of the string needs to match
# (and the pattern can appear anywhere in the string)

# basic
match = re.search(r'iii', "piiig") # found, match.group() = "iii"
match = re.search(r'igs', "piiig") # not found, match = None

match = re.search(r'..g', "piiig") # found, match.group() = "iig"
match = re.search(r'\d\d\d', "p123g") # found, match.group() = "123"
match = re.search(r'\w\w\w', "@@abcd") # found, match.group() = "abc"

# repetition
# NOTE: search finds the leftmost match (first priority), then within that match
# goes as deep as possible (* and + are greedy specifiers)
match = re.search(r'pi+', "piiig") # found, match.group() = "piii"
match = re.search(r'i+', "piigiiii") # found, match.group() = "ii"
match = re.search(r'\d\s*\d\s*\d', "xx1 2   3xx") # found, match.group() = "1 2   3"
match = re.search(r'\d\s*\d\s*\d', "xx12  3xx") # found, match.group() = "12  3"
match = re.search(r'\d\s*\d\s*\d', "xx123xx") # found, match.group() = "123"

# ^ vs. $
match = re.search(r'^b\w+', "foobar") # not found (does start start with 'b'), match = None
match = re.search(r'b\w+', "foobar") # found, match.group() = "bar"

# applications - email validity verification
str = "purple alice-b@google.com monkey dishwasher"
match = re.search(r'\w+@\w+', str) # found, match.group() = "b@google"

# a better one
match = re.search(r'[\w+.-]+@[\w+.-]+', str) # found, match.group() = "alice-b@google.com"

# grouping
match = re.search(r'([\w+.-]+)@([\w+.-]+)', str) # found
                                                 # match.group() = "alice-b@google.com"
                                                 # match.group(1) = "alice-b"
                                                 # match.group(2) = "google.com"
```

1.3 Word Tokenization

Some tokenization examples in UNIX systems:

```
tr -sc 'A-Za-z' '\n' < shakes.txt   # change non-alpha to newlines
    | sort                          # sort in alphabetical order
    | uniq -c                       # merge and count each type
```

2 Text Classification and Naïve Bayes

2.1 Supervised learning

Input:

- a document d
- a fixed set of classes $C = \{c_1, c_2, ..., c_j\}$
- a training set of m hand-labeled documents $(d_1, c_1), ..., (d_m, c_m)$

Output:

- a learned classifier $\gamma : d \rightarrow c$

2.2 Naïve Bayes

- Use a bag of words representation: for a given document, keep a map between unique words and the number of times they appear in the document
- Bayes Rule applied to documents and classes: $P(c|d) = \frac{P(d|c)P(c)}{P(d)}$
- most likely class:
  - Maximum A Posteriori (MAP): $c_{MAP} = argmax P(c|d)$
  - Bayes Rule: $= argmax \frac{P(d|c)P(c)}{P(d)}$
  - Drop denominator: $= argmax P(d|c)P(c)$
  - Representation for a document (using features): $= argmax P(x_1, x_2, ..., x_n|c)P(c)$

2.3 Multinomial Naïve Bayes

2.3.1 Indepenence Assumptions

- **Bag of Words assumption:** Assume word position in a sentence doesn't matter
- **Conditional Indpendence:** Assume feature probabilities $P(x_i|c_j)$ are independent given the class c, i.e. $P(x_1, x_2, ..., x_n|c) = P(x_1|c) \cdot P(x_2|c) \cdot P(x_3|c) \cdot ... \cdot P(x_n|c)$

2.4 Training Phase

- Calculate class prior probabilities (priors): $P(c_j) \leftarrow \frac{doccount(C=c_j)}{N_{doc}}$
- Calculate vocabulary frequency (with smoothing to avoid 0 probabilities): $P(w_k|c_j) \leftarrow \frac{count(w_i,c_j)+\alpha}{\sum_{w \in V} count(w,c_j)+\alpha|V|}$
  - Create a "mega-document" of all documents belonging to class c_j by concatenating them
  - Use the frequency of w in this mega document

Broken down further:

1. Calculate $P(c_j)$ terms:

```
for each $c_j$ in C do:
    $docs_j \leftarrow$ all docs with class = $c_j$
    $P(c_j) \leftarrow \frac{|docs_j|}{|totaldocs|}$
```

2. Calculate $P(w_k|c_j)$ terms:

```
$text_j \leftarrow$ single doc containing all $docs_j$
for each word $w_k$ in $V$:
    $n_k \leftarrow num.occurrencesof$ w$_k$ in $text_j$
    $P(w_k|c_j) \leftarrow \frac{n_k+\alpha}{n+\alpha|V|}$
```

2.4.1 Multinomial Naïve Bayes Example

Steps:

- Calculate class prior probabilities (for each class i) by calculating number of documents belonging to class i in training set divided by total number of documents in training set
- Calculating word probabilities:
  - Construct bags of words for each class category, and count the number of occurrences of each unique word in that bag.
  - Compute conditional probabilities: for the numerator, add the number of occurrences of a word in this category to the smoothing parameter. For the denominator, add the total number of words belonging to this category to `smooth_param` $\cdot$ *total* training set vocabulary size.
  - Maintain a count of the total number of words belonging to each class category.
  - (For smoothing) determine the number of unique words in the vocabulary, and add to the word counts.

Given this train/test set:

Doc	Words	Class
1	Italy Florence Italy	c1
2	Italy Italy Venice	c1
3	Italy Rome	c1
4	Berlin Germany Italy	c2

Table 1: Training set

Doc	Words	Class
1	Italy Italy Italy Berlin Germany	?

Table 2: Test set

Class priors (what's the likelihood of a document belonging to this class?):
$P(c_1) = \frac{3}{4} \rightarrow$ (3 of the 4 training docs are of class c_1)
$P(c_2) = \frac{1}{4} \rightarrow$ (1 of the 4 training docs are of class c_2)

Conditional probabilities (likelihood of a word i occurring in class c?):
NOTE: We only need to compute these probabilities for words present in the test set (in this case, `Italy`, `Berlin`, and `Germany`.

$P(Italy|c_1) = \frac{5+1}{8+6} = \frac{6}{14}$

$P(Berlin|c_1) = \frac{0+1}{8+6} = \frac{1}{14}$

$P(Germany|c_1) = \frac{0+1}{8+6} = \frac{1}{14}$

$P(Italy|c_2) = \frac{1+1}{3+6} = \frac{2}{9}$

$P(Berlin|c_2) = \frac{1+1}{3+6} = \frac{2}{9}$

$P(Germany|c_2) = \frac{1+1}{3+6} = \frac{2}{9}$

Predicting the class of the test document (d_1):
$P(c_1|d_1) = P(c_1) \cdot P(Italy|c_1)^3 \cdot P(Berlin|c_1) \cdot P(Germany|c_1) = \frac{3}{4} \cdot \frac{6}{14}^3 \cdot \frac{1}{14} \cdot \frac{1}{14} \approx 0.0003$
$P(c_2|d_1) = P(c_2) \cdot P(Italy|c_2)^3 \cdot P(Berlin|c_2) \cdot P(Germany|c_2) = \frac{1}{4} \cdot \frac{2}{9}^3 \cdot \frac{2}{9} \cdot \frac{2}{9} \approx 0.0001$

$P(c_1|d_1)$ is higher, so the test document most likely belongs to class c_1.

2.5 Precision, Recall, and the F Measure

2.5.1 Contingency table

	correct	not correct
selected	TP	FP
not selected	FN	TN

Table 3: A 2×2 contingency table

2.5.2 Type I and II Errors

- **Type I:** A false positive. That is, a model predicts the positive class but the correct class was negative.
- **Type II:** A false negative. That is, a model predicts the negative class but the correct class was positive.

2.5.3 Precision vs. Recall

Scenario: Classifying a group of apples (positive) and oranges (negative).

- **Precision:** ratio of true positives to all positives, or percentage of selected items that are correct. In other words, how many of the examples you predicted as apples were actually apples? A high precision means that most of your predicted apples were actually apples (and very few were oranges in reality). TP will be large while FP will be small.
  $precision = \frac{TP}{TP+FP}$
- **Recall:** ratio of true positives to all selected (i.e. correctly classified) examples, or percentage of correct items that are selected. So, of the selected examples that were apples, how many did you correctly predict? In other words, did you minimize the number of cases where you said an example *was not* an apple when it really was? A high recall means that your model is not frequently predicting oranges given a group of apples. TP will be very large while FN will be small. $recall = \frac{TP}{TP+FN}$

2.5.4 F measure

Takes into account the trade-off between precision and recall.

$F = \frac{1}{\alpha\frac{1}{P}+(1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2P+R}$

Usually we use the F1 score ($\beta = 1$, or in the first equation, $\alpha = 0.5$):

$F1 = \frac{2PR}{P+R}$

In our scenario, the F1 score measures the amount of oranges mistakenly classified as apples (false positive) and the amount of apples that were not correctly classified as apples (false negative).

3 Language Modeling

3.1 Probabilistic Language Modeling

Used for:

- Machine translation
- Spell correction
- Speech recognition
- Text summarization, question-answering

Goal: compute the probability of a sentence or sequence of words occurring, or even the probability of the next word in a sequence

Computed using the chain rule of probabilities: $P(A, B, C, D) = P(A)P(B|A)P(C|A, B)P(D|A, B, C)$

3.1.1 N-gram models

- Condition the upcoming word on the N previous words.
- Usually insufficient for complex language due to **long-distance dependencies** in sentences.

3.2 Estimating N-gram Probabilities

Use maximum-likelihood estimation (MLE): $P(w_i|w_{i-1}) = \frac{count(w_{i-1},w_i)}{count(w_{i-1})}$. In plain English, this means to count the number of times where both words occur in order ($w_{i-1}w_i$), divided by the number of times the previous word appears (w_{i-1}).

Example:

- `<s> I am Sam </s>`
- `<s> Sam I am </s>`
- `<s> I do not like green eggs and ham </s>`

First, compute relevant probabilities (the exact ones needed will depend on the test sentence). Here are some extras for illustration purposes:

$$\begin{array}{lll} P(\texttt{I}|\texttt{<s>}) = \frac{2}{3} & P(\texttt{Sam}|\texttt{<s>}) = \frac{1}{3} & P(\texttt{am}|\texttt{I}) = \frac{2}{3} \\ P(\texttt{</s>}|\texttt{Sam}) = \frac{1}{2} & P(\texttt{Sam}|\texttt{am}) = \frac{1}{2} & P(\texttt{do}|\texttt{I}) = \frac{1}{3} \end{array}$$

Test string: "I want english food"

NOTE: In this case, some words were not present in the training set, but in reality they would be. The following computation is for illustration purposes (to show how it would be done if all words were present). Also, in reality, these probabilities would not be multiplied together, and would instead be converted into log-space and added (i.e. $\log p_1 + \log p_2 + ...$)

$P(\texttt{<s>I want english food</s>}) =$

$P(\texttt{I}|\texttt{<s>})$
$\times P(\texttt{want}|\texttt{I})$
$\times P(\texttt{english}|\texttt{want})$
$\times P(\texttt{food}|\texttt{english})$
$\times P(\texttt{</s>}|\texttt{food})$
$= 0.00031$

3.3 Evaluation and Perplexity

Perplexity is the inverse probability of the test set, normalized by the number of words.

$PP(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}}$

Example: Given a sentence consisting of random digits, what is the perplexity of the sentence according to a model that assigns uniform probability to each digit ($P = 1/10$)?
$PP(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}}$
$= (\frac{1}{10}^N)^{-\frac{1}{N}}$
$= \frac{1}{10}^{-1}$
$= 10$

4 Hidden Markov Models

4.1 Markov Chains, Viterbi Decoding, Expectation Maximization Algorithms

Hidden Markov model definition:

- $Q = q_1 q_2 ... q_N$: set of N states
- $A = a_{11} a_{12} ... a_{n1} a_{n2} ... a_{nn}$: transition probability matrix, which represents at each cell the probability of moving from hidden state i to state j. Each row's probabilities must sum up to 1 (that is, the outgoing edges of any state should sum to 1).
- $O = o_1 o_2 ... o_T$: Sequence of T observations
- $B = b_i(o_t)$: sequence of observation likelihoods (emission probabilities), which express the probability of an observation o_t generated from state i
- q_0, q_F: start and end states (not associated with observations)

Markov Assumption: The probability of a particular state only depend son the probability of the previous state.

Characterized by three fundamental problems:

- **Problem 1 (Likelihood):** Given an HMM $\lambda = (A, B)$ and an observation sequence O, determine the likelihood $P(O|\lambda)$
- **Problem 2 (Decoding):** Given an observation sequence O and an HMM $\lambda = (A, B)$, discover the best hidden state sequence Q.
- **Problem 3 (Learning):** Given an observation sequence O and the set of states in the HMM, learn the HMM parameters A and B.

4.1.1 The Forward and Viterbi Algorithms

More efficient than calculating all possible probabilities (i.e. $P(313) = P(3,1,3|cold, cold, cold)+P(3,1,3|cold, cold, hot)+P(3,1,3|hot, hot, cold) + ...$)

Instead use the Forward algorithm, which is $O(N^2T)$ — N is the number of hidden states and T is the length of the sequence.

The size of the α matrix (the forward algo's equivalent to Viterbi's V-matrix) is $N \times T$.

The forward algorithm is similar to the Viterbi algorithm in the way the α matrix is computed: at each column, you fill in a cell using the previous column's probability, the transition probability from the previous state to the current state, and the probability of observing the current time step given the current state:

$(\alpha_{i-1}) \cdot P(\texttt{curr\_state}|\texttt{prev\_state}) \cdot P(\texttt{observation}|\texttt{curr\_state})$

However, unlike in Viterbi where the max is taken among the hidden states, for each cell the sum over all options of previous hidden states is stored.

Additionally, it is important to note that Forward and Viterbi are for different HMM cases.

- Forward is for computing likelihood (Problem 1 above) of seeing a sequence. The in-class example was computing the likelihood of observing the sequence (3 ice creams, 1 ice cream, 3 ice creams). The final probability is computed by taking the sum of $\alpha_3(H) + \alpha_3(C)$ (the last columns).
- Viterbi, instead, is for decoding (problem 2 above) the hidden state sequence that yields a sequence of observations. In Viterbi, at each column we only record the state which yields the maximum probability, and also use a backtracking matrix to record the states that got us here so far. In short, the two algorithms both have the similar-sized probability matrices (α and $V - matrix$), but in the forward alg. you sum over all possible hidden states leading you to the current position, whereas in Viterbi you take the maximum only. (See HW2 part 1 for clarification).

4.1.2 Decoding

For Viterbi, when decoding, it's important to start at the cell in the last column with the maximum probability. Record that hidden state as the final state then recurse back to the front by consulting the backtracking matrix. So: write the state of the cell with max. probability, look at that state's cell in the last column, write its contents, and so on and so forth.

5 Statistical Natural Language Parsing

5.1 Two Views of Linguistic Structure

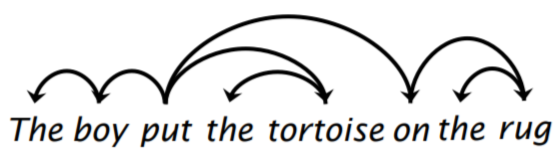
5.1.1 Constituency (Phrase) Structure

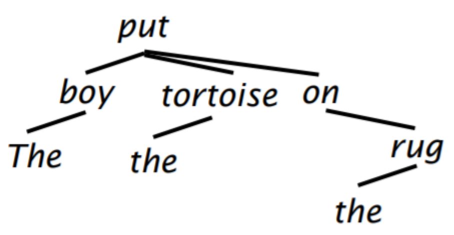
Phrase structure organizes words into nested constituents. A **constituent** behaves as a unit that can appear in different places:

- John talked [to the students] [about drugs].
- John talked [about drugs] [to the students].
- I sat [on the box/right on top of the box/there].

5.1.2 Dependency Structure

Dependency structure shows which words depend on which other words. That is, words that modify or are arguments of other words.



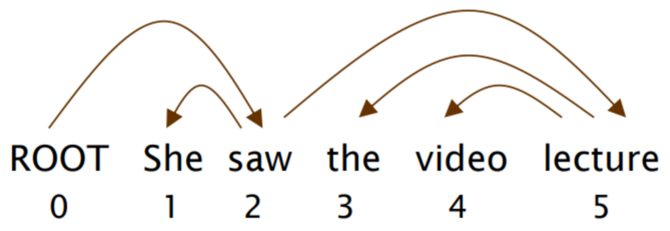


To calculate the probability of a specific dependency tree (from a probabilistic CFG [PCFG]), just multiply the probabilities of the rules that generate it, going level by level.

To calculate the probability of a string s generated from a set of dependency trees, just add the sums each tree's probability (from the method directly above).

5.1.3 Dependency Parsing Evaluation

UAS: Unlabeled accuracy score
LAS: Labeled accuracy score



$$\text{Acc} = \frac{\text{\# correct deps}}{\text{\# of deps}}$$

UAS = 4 / 5 = 80%
LAS = 2 / 5 = 40%

Gold

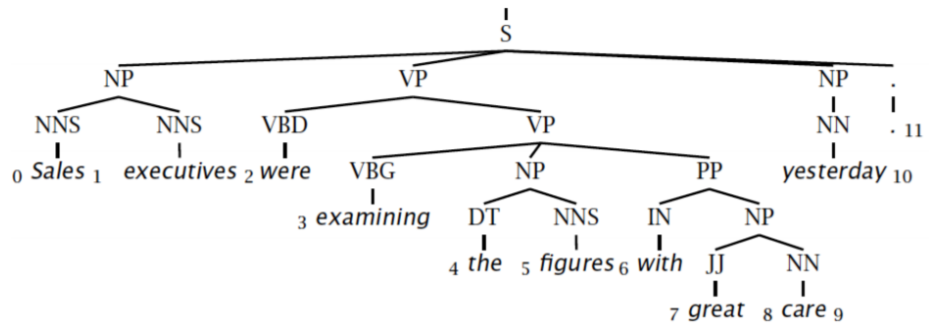
1	2	She	nsubj
2	0	saw	root
3	5	the	det
4	5	video	nn
5	2	lecture	dobj

Parsed

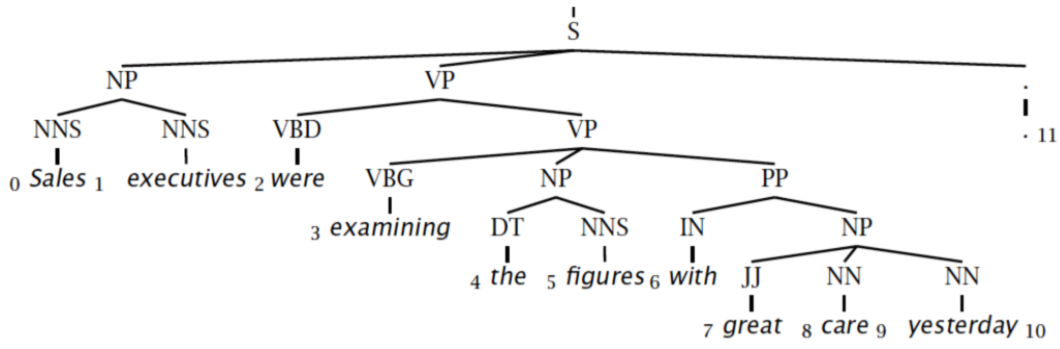
1	2	She	nsubj
2	0	saw	root
3	4	the	det
4	5	video	nsubj
5	2	lecture	ccomp

5.1.4 Constituency Parsing Evaluation

Gold standard brackets: **S-(0:11)**, **NP-(0:2)**, VP-(2:9), VP-(3:9), **NP-(4:6)**, PP-(6-9), NP-(7,9), NP-(9:10)



Candidate brackets: **S-(0:11)**, **NP-(0:2)**, VP-(2:10), VP-(3:10), **NP-(4:6)**, PP-(6-10), NP-(7,10)



Gold standard brackets:

S-(0:11), NP-(0:2), VP-(2:9), VP-(3:9), **NP-(4:6)**, PP-(6-9), NP-(7,9), NP-(9:10)

Candidate brackets:

S-(0:11), NP-(0:2), VP-(2:10), VP-(3:10), **NP-(4:6)**, PP-(6-10), NP-(7,10)

Labeled Precision	3/7 = 42.9%
Labeled Recall	3/8 = 37.5%
LP/LR F1	40.0%
Tagging Accuracy	11/11 = 100.0%

Notes:

- Tagging accuracy: is 100% in this example because the candidate part of speech tags (the tags just above the words in the leaf nodes [i.e. NNS, VBD, etc.]) completely match the gold standard POS tags.
- Labeled Precision: is 3/7 because, looking at the 7 candidate brackets, 3 of them are correct (exactly matching the gold standard in both label (i.e. NP) and range (i.e. [4:6]).

- Labeled Recall: is 3/8 because, looking at the 8 **gold standard** brackets, 3 of them exactly match the candidate predictions.
- LP/LR F1 score: is 40% by using the F1 formula: $\frac{2PR}{P+R}$

6 Deep Learning for NLP

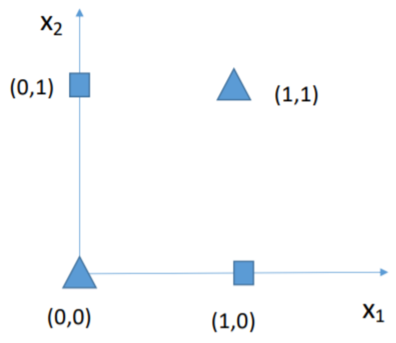
6.1 Neural Network Basics

6.1.1 Logistic Unit as a Neuron

- Input layer: an input vector $X = (X_1, X_2, ..., X_n)$
- Activation: weighted sum of input features $a = W_0 + \sum_{n=1}^{N} W_n X_n$
- Activation function: logistic function h applied to the weighted sum
- Output: $z = h(a)$

6.1.2 XOR Problem

Neural networks are needed because some tasks' data classes have non-linearly separable properties. For example, given the following "dataset," it is impossible to separate them with a straight line (which is what a linear classifier would produce).



6.1.3 Neural Network Training

Given a training set of M examples $(x^i, t^i)|i = 1, ..., M$ ...

Training a neural network is equivalent to minimizing the least squares error between the network output and the true value.

$min_w L(w) = \frac{1}{2} \sum_{i=1}^{M} (y^{(i)} - t^{(i)})^2$

Where $y^{(i)}$ is the output depending on the network parameters w.

6.1.4 Gradient Ascent/Descent

- Iterative algorithm for finding the "peak" or "valley" (ascent or descent, respectively) of a function.
- A gradient is a vector that points in the steepest direction (toward the peak for ascent, valley for descent).
- At each point, the weights w is updated so it moves a size of step λ in the gradient direction:
  - Gradient ascent: $w \leftarrow w + \lambda \Delta L(w)$
  - Gradient descent: $w \leftarrow w - \lambda \Delta L(w)$

6.2 Deep Neural Networks

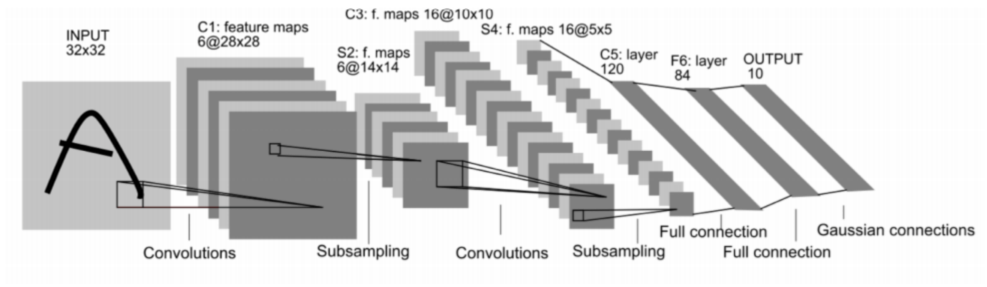
6.2.1 Deep Convolutional Networks

Comparison to standard feedforward fully-connected networks:

- CNNS have much fewer connections and parameters (parameters are shared)
- Thus, they are easier to train
- Do not sacrifice much performance to achieve this

6.2.2 LeNet-5

- Input: 32×32 pixel image
- Cx: conv layer
- Sx: subsample layer
- Fx: FC layer



Examples

C1 = conv. layer with 6 feature maps of size 28×28.
Each unit of C1 has a 5×5 receptive field in the input layer.

- Topological structure
- Sparse connections
- Shared weights

Number of learned parameters is receptive field of the layer (+1 for bias term) multiplied by the number of feature maps: $(5 \cdot 5 + 1) \cdot 6 = 156$

Number of connections is feature map size × receptive field size (+1 for bias term) × number of feature maps: $(28 \cdot 28) \cdot (5 \cdot 5 + 1) \cdot 6 = 122,304$

If this was instead a fully connected layer, we would have had image size (+1) × feature map size × number of feature maps: $(32 \cdot 32 + 1) \cdot (28 \cdot 28) \cdot 6 = 4,821,600$ parameters!

S2 = subsampling layer with 6 feature maps of size 14×14. Has 2×2 non-overlapping receptive fields in C1

Layer S2: num. feature maps × receptive field len.: $6 \cdot 2 = 12$ learnable parameters
Connections S2: feature map size × receptive field size (+1) × num. feature maps: $(14 \cdot 14) \cdot (2 \cdot 2 + 1) \cdot 6 = 5,880$

6.2.3 Dropout

- Combining models is useful (i.e. ensembling, boosting, mixture of experts)
- But training many models is more compute intensive
- Solution: use *dropout*

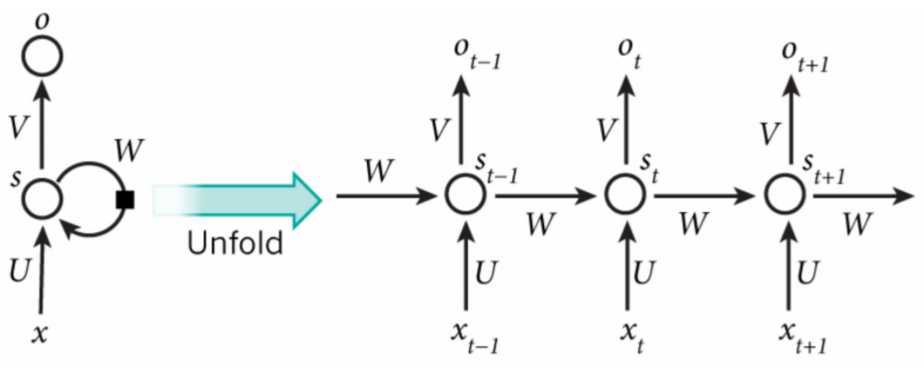
Dropout: Set the output of each hidden neuron to zero (i.e. nullify it) with probability 0.5 (other probabilities work, but 0.5 is the most commonly used in practice)

- The neurons that are "dropped out" do not contribute to the forward pass nor backpropagation
- So, in essence, every time an input is presented, the NN samples a different architecture, though all of these "pseudo-architectures" share weights
- Reduces the complex co-adaptations of neurons, as neurons can no longer depend on other neurons at the same level for identifying features and prediction
- Thus, each one is forced to learn more robust features that can be used in conjunction with different random subsets of other neurons
- Reduces overfitting
- But roughly doubles the number of training steps required for convergence

6.3 Recurrent Neural Networks and LSTM

6.3.1 RNNs

RNNs process input sequences one elemnt at a time, maintaining in their hidden states a "state vector" that contains implicit information about the history of past elements. This makes them especially good at predicting the next word in a sequence.



Like regular neural networks, we can apply backpropagation to train RNNs, and all layers share the same weights.

However, they have some difficulty storing information for very long sequences:

- During gradient backprop, the gradient is multiplied a large number of times by the weight matrix
- If the weights are small, this leads to the **vanishing gradient** problem
- If the weights are large, this leads to the **exploding gradient** problem

To remedy this, and to better capture long-distances dependencies in sentences, RNNs need more persistent memory. This is what the LSTM cell attempts to solve.

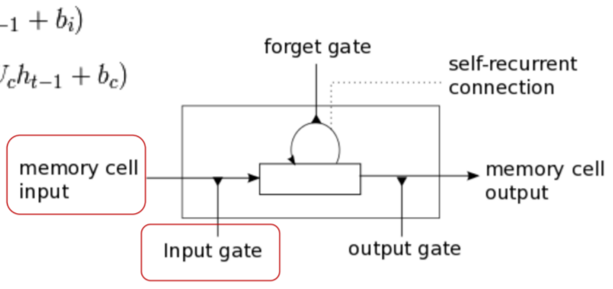
6.3.2 LSTM

LSTM cells have a connection to themselves at the next time-step, but this connection is also gated by another unit that learns to decide whether to clear the contents of the memory cell. Below are the steps and relevant LSTM cell formulas:

First, we compute the values for i_t, the input gate, and $\widetilde{C}_t$ the candidate value for the states of the memory cells at time t :

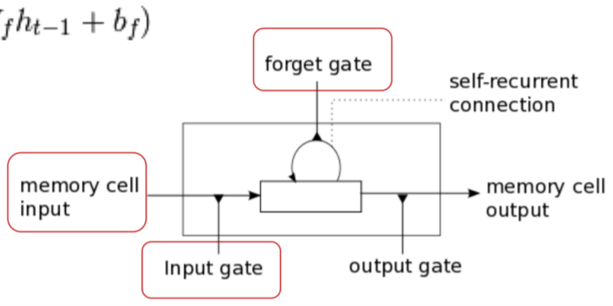
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

$$\widetilde{C}_t = tanh(W_c x_t + U_c h_{t-1} + b_c)$$



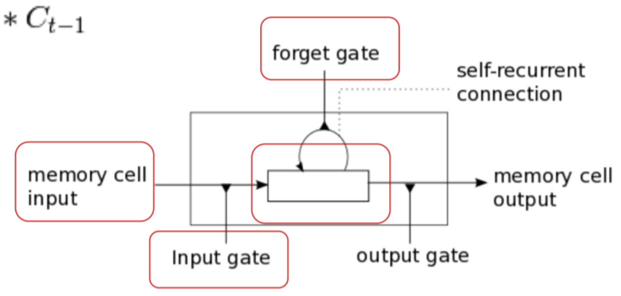
Second, we compute the value for f_t, the activation of the memory cells' forget gates at time t :

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$



Given the value of the input gate activation i_t, the forget gate activation f_t and the candidate state value $\widetilde{C}_t$, we can compute C_t the memory cells' new state at time t :

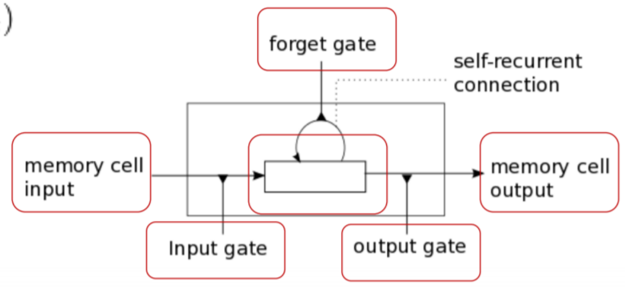
$$C_t = i_t * \widetilde{C}_t + f_t * C_{t-1}$$



With the new state of the memory cells, we can compute the value of their output gates and, subsequently, their outputs :

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$$

$$h_t = o_t * tanh(C_t)$$


forget gate
self-recurrent connection
memory cell input
memory cell output
Input gate
output gate