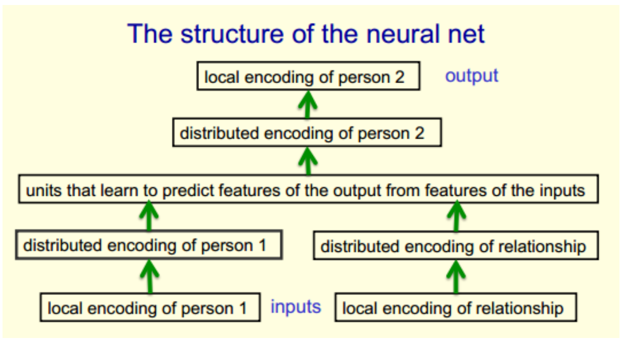
L4

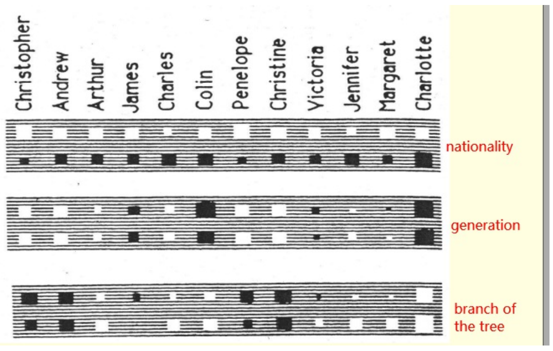
- 2017/11/04 11:36
- Learning to predict the next word
    - goal
        - **to use the backpropagation algorithm to learn the feature vector from relational information**
        - here, feature vector is the representation of the meaning of the word
    - A relational learning task
        - **is to figure out the regularities in a large set of triples that express information in those trees.**
        - ways
            - The obvious way to express the regularities is as symbolic rules
                - e.g.
                    - (x has-mother y) & (y has-husband z) => (x has-father z)
                - drawbacks
                    - it's very **difficult to search all these relationships** in a large database
                    - Finding the symbolic rules involves **a difficult search through a very large discrete space of possibilities.**
            - use a neural network to capture the same

knowledge by searching through a continuous space of weights ???

- How NN does this
  - The structure of the neural net
    - there are six units in the local encoding of person 1 stage.


The structure of the neural net
local encoding of person 2
output
distributed encoding of person 2
units that learn to predict features of the output from features of the inputs
distributed encoding of person 1
distributed encoding of relationship
local encoding of person 1
inputs
local encoding of relationship

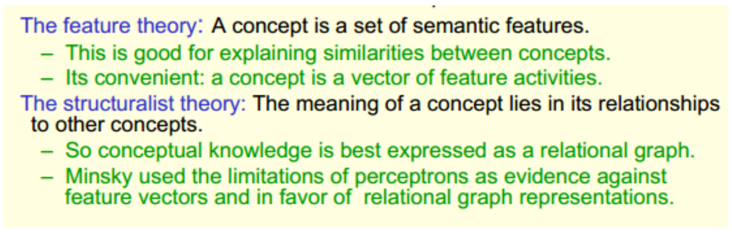
- three of units/ learned features
  - each units learned a typical feature that is useful to make the decision
  - ???These features are only useful if the other bottlenecks use similar representations and the central layer learns how features predict other features.



  - Train and test
    - train all but a few triples that can be used in the test.
    - larger datasets, better performance
  - A large-scale example - datasets cleaning/wrong cases detection
    - a database of millions of relational facts of the form (A R B)
      - predict the third term from the first two

        - predict the unlikely third term from the first two
        - use all three terms as input and predict the probability that the fact is correct

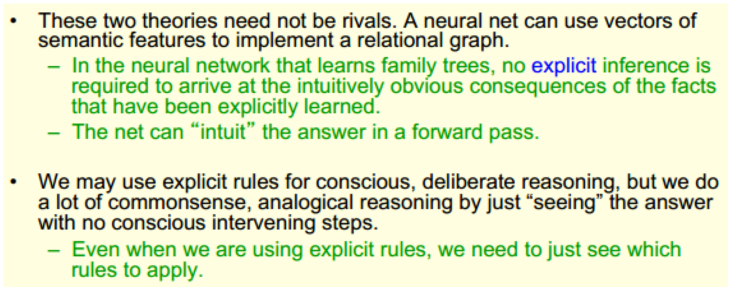
- A brief diversion into cognitive science
    - There has been a long debate in cognitive science between two rival theories of **what it means to have a concept:**



The feature theory: A concept is a set of semantic features.
- This is good for explaining similarities between concepts.
- Its convenient: a concept is a vector of feature activities.

The structuralist theory: The meaning of a concept lies in its relationships to other concepts.
- So conceptual knowledge is best expressed as a relational graph.
- Minsky used the limitations of perceptrons as evidence against feature vectors and in favor of relational graph representations.

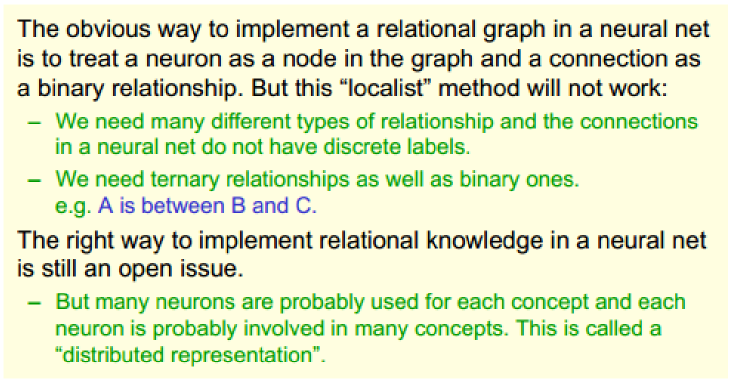
    - in Geoffrey Hinton's view, Both sides are wrong
        - **A neural net can use vectors of semantic features to implement a relational graph.**
            - that means NN combines both features and relationships.
            - and we don't have to know the explicit meaning of features, but know the features used are beneficial to get the right answers. This is just like our brains make decisions/predictions.



- These two theories need not be rivals. A neural net can use vectors of semantic features to implement a relational graph.
    - In the neural network that learns family trees, no explicit inference is required to arrive at the intuitively obvious consequences of the facts that have been explicitly learned.
    - The net can "intuit" the answer in a forward pass.
- We may use explicit rules for conscious, deliberate reasoning, but we do a lot of commonsense, analogical reasoning by just "seeing" the answer with no conscious intervening steps.
    - Even when we are using explicit rules, we need to just see which rules to apply.

    - Localist and distributed representations of concepts
        - We still don't know for sure the right way to implement relational knowledge in a neural net.
        - But it seems very probable that many neurons are used for representing each of the concepts we know, and each of those neurons is probably involved in dealing with many different concepts.

- This is called a **distributed representation**. It's a **many to many mapping** between concepts and neurons.



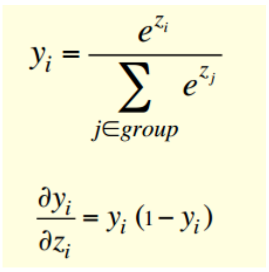
The obvious way to implement a relational graph in a neural net is to treat a neuron as a node in the graph and a connection as a binary relationship. But this "localist" method will not work:

- We need many different types of relationship and the connections in a neural net do not have discrete labels.
- We need ternary relationships as well as binary ones. e.g. A is between B and C.

The right way to implement relational knowledge in a neural net is still an open issue.

- But many neurons are probably used for each concept and each neuron is probably involved in many concepts. This is called a "distributed representation".

- Another diversion: The softmax output function
  - goal
    - Force the outputs to **represent a probability distribution across discrete alternatives.**
      - probabilities to mutually exclusive class should sum to 1,
  - drawbacks of **squared error measure**
    - **small gradient for big errors so that it takes long time to train the correct weights.**
      - ??? If the desired output is 1 and the actual output is 0.00000001there is almost no gradient for a logistic unit to fix up the error.
    - **the outputs can't be regards as strict probabities**
      - probabilities to mutually exclusive class should sum to 1, but we are depriving the network of this knowledge
  - new CF - Cross-entropy with Softmax
    - the output neuron - hypethesis
      - Softmax
        - The output units in a softmax groupuse a non-local non-linearity
        - It's a kind of soft



$$y_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$

$$\frac{\partial y_i}{\partial z_i} = y_i\,(1 - y_i)$$

continuous version of the maximum function

- Cross-entropy: the right cost function to use with softmax
    - the most appropriate cost function is **the negative log probability of the correct answer.**
    - goal
        - **to maximize the log probability of getting the answer right**

$$C = -\sum_j \underset{\nwarrow\,\text{target value}}{t_j} \log y_j$$

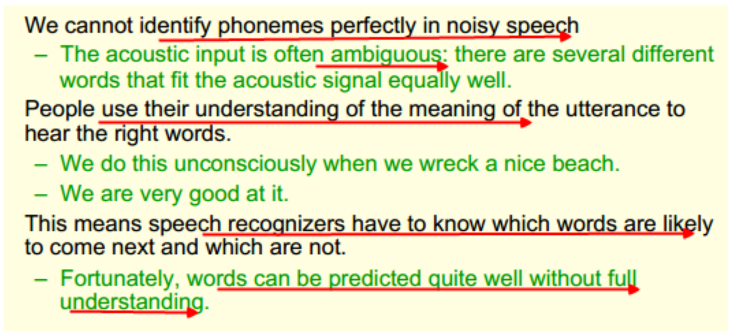
$$\frac{\partial C}{\partial z_i} = \sum_j \frac{\partial C}{\partial y_j}\frac{\partial y_j}{\partial z_i} = y_i - t_i$$

    - rationale
        - the output is a vector, e.g.[0, 0, 1, 0, 0....]
            - if one of the target values is a one and the remaining ones are zero, then we simply sum of all possible answers.
        - We put zeros in front of all the wrong answers. And we put one in front of the right answer and that gets us the negative log probability of the correct answer
    - property
        - **it has a very big gradient when the target value is one and the output is almost zero.**
            - it's very sensitive : A value of 0.000001 is much

better than 0.000000001

- dC/dy is very steep, while dy/dz is very flat. So the multiplication balances them well. If the answer is wrong, the slope is always 1 or -1, until getting the right answer.

- Neuro-probabilistic language models - a practical example
  - A basic problem in speech recognition

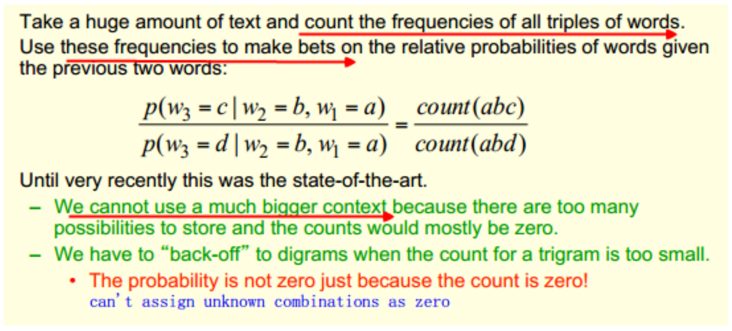


We cannot identify phonemes perfectly in noisy speech
- The acoustic input is often ambiguous: there are several different words that fit the acoustic signal equally well.

People use their understanding of the meaning of the utterance to hear the right words.
- We do this unconsciously when we wreck a nice beach.
- We are very good at it.

This means speech recognizers have to know which words are likely to come next and which are not.
- Fortunately, words can be predicted quite well without full understanding.

  - The standard “trigram” method

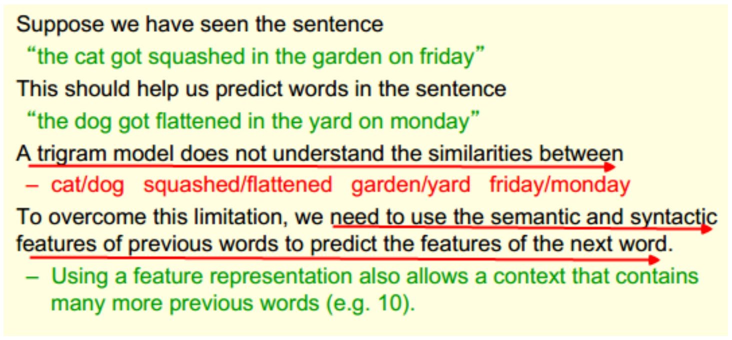


Take a huge amount of text and count the frequencies of all triples of words.
Use these frequencies to make bets on the relative probabilities of words given the previous two words:

$$\frac{p(w_3 = c \mid w_2 = b, w_1 = a)}{p(w_3 = d \mid w_2 = b, w_1 = a)} = \frac{count(abc)}{count(abd)}$$

Until very recently this was the state-of-the-art.
- We cannot use a much bigger context because there are too many possibilities to store and the counts would mostly be zero.
- We have to “back-off” to digrams when the count for a trigram is too small.
  - The probability is not zero just because the count is zero!
    can't assign unknown combinations as zero

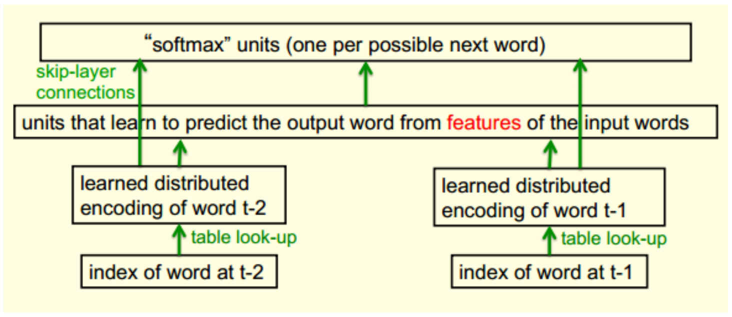
  - Drawbacks: Information that the trigram model fails to use



Suppose we have seen the sentence
“the cat got squashed in the garden on friday”
This should help us predict words in the sentence
“the dog got flattened in the yard on monday”
A trigram model does not understand the similarities between
- cat/dog squashed/flattened garden/yard friday/monday

To overcome this limitation, we need to use the semantic and syntactic features of previous words to predict the features of the next word.
- Using a feature representation also allows a context that contains many more previous words (e.g. 10).

- Bengio’ s neural net for predicting the next word - just as the model before


"softmax" units (one per possible next word)
skip-layer connections
units that learn to predict the output word from features of the input words
learned distributed encoding of word t-2
learned distributed encoding of word t-1
table look-up
table look-up
index of word at t-2
index of word at t-1

- A problem with having 100,000 output words
    - Each unit in the last hidden layer has 100,000 outgoing weights
    - why so many
        - Because typically in these language models, the plural of a word is a different word from the singular.
        - And the various different tenses of a verb are different words from other tenses.
    - problem with this
        - And that means we can only afford to have a few units there before we start over-fitting.
            - but we can get plenty of train example to overcome overfitting
        - A very large number of words will have small probabilities. So it is hard to make decisions
        - We could make the last hidden layer small, but then its hard to get the 100,000 probabilities right.
    - solutions see next part.

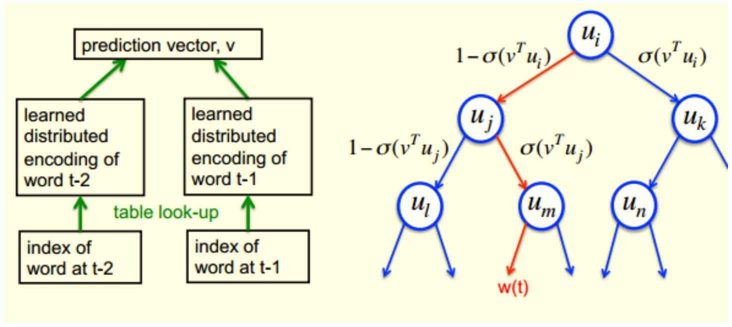
- Ways to deal with the large number of possible outputs in neuro-probabilistic language models, such as 100,000
    - one way is to use a serial architecture.
        - procedure
            - computing the logit score for each candidate word

at one time,

- then use all of the logits in a softmax to get word probabilities.

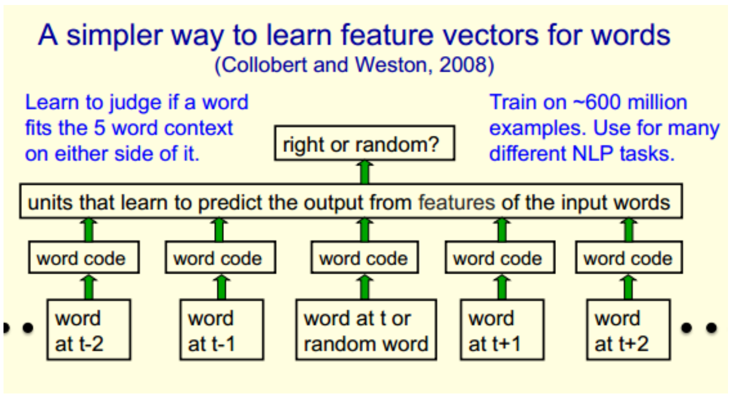
- analysis
    - The difference between the word probabilities and their target probabilities gives cross-entropy error derivatives. yi - ti
        - The derivatives try to raise the score of the correct candidate and lower the scores of its high-scoring rivals
    - We can save a lot of time if we only use a small set of candidates suggested by some other kind of predictor.
    - time complexity is O(n)

- one way is by predicting a path through a tree
    - procedure
        - Arrange all the words in a binary tree with words as the leaves
        - Use the previous context to generate a “prediction vector”, v
        - a learned vector, u, at each node of the tree
        - Apply the logistic function to the scalar product of u and v to predict the probabilities of taking the two branches of the tree.
        - take the branch with the highest probability, which is the highest product of the probability of the nodes in that branch



- analysis
    - Maximizing the log probability of picking the target word is equivalent to maximizing the sum of the log probabilities of taking all the branches on the path that leads to the target word
    - For each of these nodes, we know the correct branch and we know the current probability of taking it so we can get derivatives for learning both the prediction vector v and that node vector u.
    - log(N) instead of N. but it is still slow at test time.

- A simpler way to learn feature vectors for words
    - 11 words, 10 as context, the middlest is used for training
    - learn a feature vector for each word


A simpler way to learn feature vectors for words
(Collobert and Weston, 2008)
Learn to judge if a word fits the 5 word context on either side of it.
right or random?
Train on ~600 million examples. Use for many different NLP tasks.
units that learn to predict the output from features of the input words
word code
word code
word code
word code
word code
word at t-2
word at t-1
word at t or random word
word at t+1
word at t+2

    - We can get an idea of the quality of the learned feature vectors by displaying them in a 2-D map
    - The learned feature vectors capture lots of subtle semantic distinctions, just by looking at strings of words
        - No extra supervision is required.
        - The information is all in the contexts that the

word is used in.

- 非常棒的一个东西，描述相似性。