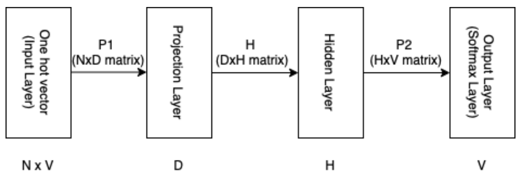
Efficient Estimation of Word Representations in Vector Space

- [[📚 1301.3781.pdf]]
- **Basic Idea**:
  - **Distributional hypothesis**: The meaning of a word is determined by the words that appear in the context of the given word.
  - Can we learn word embeddings/vectors efficiently on a large corpus using neural networks?
  - The learned word representations retain certain linear properties which helps them solve tasks of semantic and syntactic similarities.
  - The learned word embeddings are distributional representations (because of distributional hypothesis) as well as because the meaning of a word is spread across all the dimensions of the word representation. This is in contrast to the one-hot encoding where the meaning of the word is localised in a dimension (localised hypothesis).
- **Model Architecture**
  - The training complexity of an architecture is proportional to:

$$O = E \times T \times Q$$

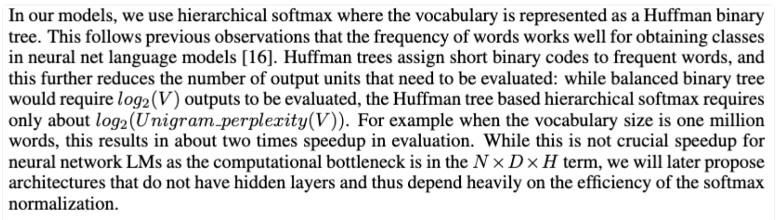
  - where
    - E is the number of training epochs
    - T is the number of words in the training set
    - Q is the defined as for each model architecture.
  - **Feedforward Neural Net Language Model (NNLM)**
    - 

      1
    - 🟡 **P3** At the input layer, N previous words are encoded using 1-of-V coding, where V is size of the vocabulary.
    - Typical values: $N = 10$, $D = 500 - 2000$ and $H = 500 - 1000$
    - The computational complexity per each training example (i.e. word) is

$$Q = N \times D + N \times D \times H + H \times V$$

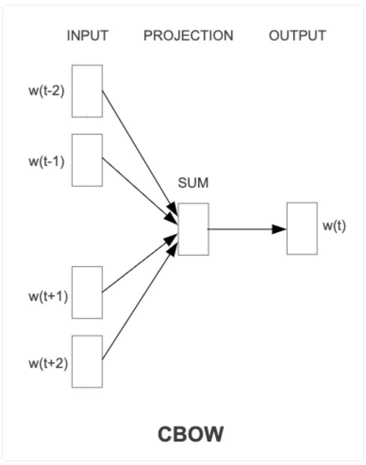
      where
      - the first term comes due to the projection matrix $P1$. Note that the complexity is not $N \times D \times V$ because each word is represented as one-hot vector and hence only 1 column of $P1$ survives. There are N words, so the complexity is $N \times D$.
      - the last term doesn't have N multiplier. Because after the hidden layer, the hidden representations of all N words may be reduced to a single vector through sum or average operation.
    - Clearly, the dominant term is $H \times V$ in the above equation. However, we can use hierarchical softmax for reducing the complexity of this.

      🟡 **P3**

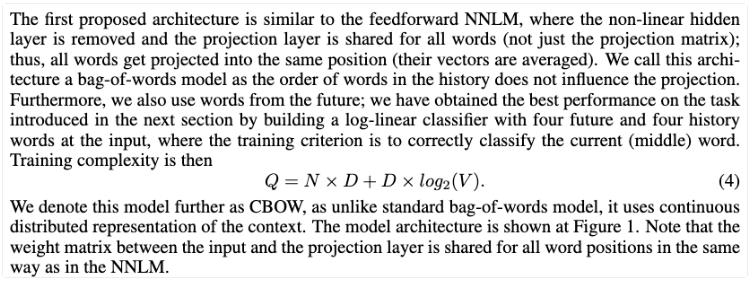
      

      In our models, we use hierarchical softmax where the vocabulary is represented as a Huffman binary tree. This follows previous observations that the frequency of words works well for obtaining classes in neural net language models [16]. Huffman trees assign short binary codes to frequent words, and this further reduces the number of output units that need to be evaluated: while balanced binary tree would require $log_2(V)$ outputs to be evaluated, the Huffman tree based hierarchical softmax requires only about $log_2(Unigram\_perplexity(V))$. For example when the vocabulary size is one million words, this results in about two times speedup in evaluation. While this is not crucial speedup for neural network LMs as the computational bottleneck is in the $N \times D \times H$ term, we will later propose architectures that do not have hidden layers and thus depend heavily on the efficiency of the softmax normalization.
    - Therefore, the next computationally expensive term in the above equation is $N \times D \times H$.
- **New Log-Linear Models**
  - **Continuous Bag-of-Words Model**
    -

🟡 **P5**



- To reduce the complexity of the feedforward model, they get rid of the non-linear hidden layer.
- 🟡 **P4**

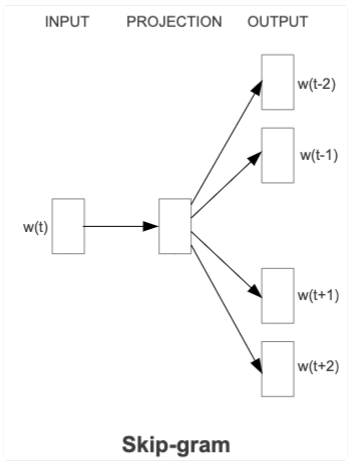


The first proposed architecture is similar to the feedforward NNLM, where the non-linear hidden layer is removed and the projection layer is shared for all words (not just the projection matrix); thus, all words get projected into the same position (their vectors are averaged). We call this architecture a bag-of-words model as the order of words in the history does not influence the projection. Furthermore, we also use words from the future; we have obtained the best performance on the task introduced in the next section by building a log-linear classifier with four future and four history words at the input, where the training criterion is to correctly classify the current (middle) word. Training complexity is then

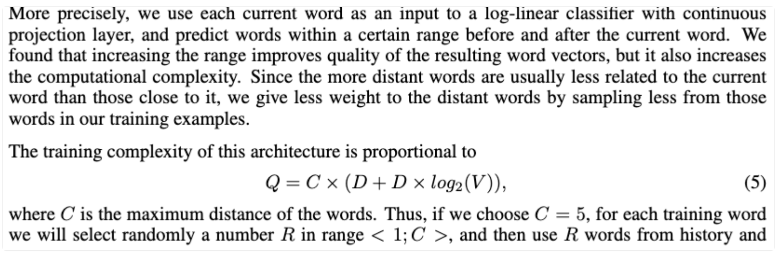
$$Q = N \times D + D \times log_2(V). \qquad (4)$$

We denote this model further as CBOW, as unlike standard bag-of-words model, it uses continuous distributed representation of the context. The model architecture is shown at Figure 1. Note that the weight matrix between the input and the projection layer is shared for all word positions in the same way as in the NNLM.

- **Continuous Skip-gram Model**
  - 🟡 **P5**



  - 🟡 **P4**



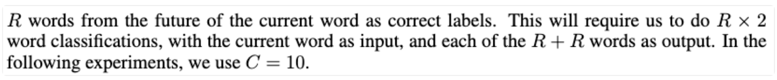
More precisely, we use each current word as an input to a log-linear classifier with continuous projection layer, and predict words within a certain range before and after the current word. We found that increasing the range improves quality of the resulting word vectors, but it also increases the computational complexity. Since the more distant words are usually less related to the current word than those close to it, we give less weight to the distant words by sampling less from those words in our training examples.

The training complexity of this architecture is proportional to

$$Q = C \times (D + D \times log_2(V)), \qquad (5)$$

where C is the maximum distance of the words. Thus, if we choose $C = 5$, for each training word we will select randomly a number R in range $< 1; C >$, and then use R words from history and

🟡 **P5**



R words from the future of the current word as correct labels. This will require us to do $R \times 2$ word classifications, with the current word as input, and each of the $R + R$ words as output. In the following experiments, we use $C = 10$.

  - Note that each pair `<middle word, randomly sampled word from 2R words>` is treated as a separate training example. Therefore, each middle word gives rise to $2R$ training examples.
- **Results**
  - Simple algebraic interpretation of word vectors:
    - 🟡 **P5** We further denote two pairs of words with the same relationship as a question, as we can ask: "What is the word that is similar to small in the same sense as biggest is similar to big?" Somewhat surprisingly, these questions can be answered by performing simple

> algebraic operations with the vector representation of words. To find a word that is similar to small in the same sense as biggest is similar to big, we can simply compute vector X = vector(”biggest”) − vector(”big”) + vector(”small”). Then, we search in the vector space for the word closest to X measured by cosine distance, and use it as the answer to the question (we discard the input question words during this search). When the word vectors are well trained, it is possible to find the correct answer (word smallest) using this method.

Unlinked References