Chapter 4

Prediction-based Distributional Models

Keywords: prediction-based models, neural-based models, neural networks, feed-forward network, hierarchical softmax, negative sampling, continuous bag-of-words, skip-gram, Word2Vec, dense vectors, word embeddings.

4.1 All the Buzz about Neural

So, as we have mentioned before, another big class of distributional semantic models are called **prediction-based models**. They are also called **neural-based**. This simply means that such models are implying on neural networks that predict word co-occurrence. What are neural networks and how do they predict? In this paragraph we will try to briefly explain this (and if you are familiar with this topic, feel free to skip it)!

You must live on another planet, if you have never heard about neural networks. Neural networks (or, more precisely, artificial neural networks) currently used almost in every aspect of software products: video recommendation, spam filtering, loan risk prediction, and so on. And distributional semantics is not an exception! But how do these networks work?

As the word "neural" suggests, these systems are inspired by our brain,—or, more precisely, biological systems that responsible of the way that we humans learn. Such systems are called **"neural networks"**, and computational models that replicate such systems are called **"artificial neural networks"**.

The very basic idea of these biological systems is that we try to process unstructured information, find patterns in this information and receive feedback of how successful was this finding. If we have found pattern correctly, we proceed. If not, we try to change something in our behavior, and then try again. And by changing behavior we mean tuning the parameters of our biological system. This process is called learning: we try to adapt our neurons to the situation, trying to get best feedback.

The same situation is in artificial neural networks. Such model consists of several layers, like a cake. Every layer has a list of neurons every of which is presented by some mathematical function (the same for each neuron in the layer) and a some scalar parameter (weight of neuron). Every network has an input layer (from which information comes to network) and an output layer (through which we receive feedback). Between input and output layer there could be a numerous amount of other layers. Their purpose is to receive information, put it into function, add their weight and receive modified piece of information. By doing

such operations the neural network tries to find patterns in the information by extracting different features until it can recognize what type of data it is processing.

Then, after iterations through all the layers, we receive some output. And we try to compare this output with desired. By comparing we mean finding degree of similarity of received output and desired one (in other word, we find a function of similarity, it is called **loss function**). If received output differs very much, we propagate the information back through the neural network (such process is called back-propagation), using the coefficient of similarity and updating weights of neurons on each layer. This allows networks to adjust their hidden layers of neurons in situations where the outcome doesn't match what the creator is hoping for—like a network designed to recognize dogs, which misidentifies a cat, for example.

For a basic idea of how a deep learning neural network learns, imagine a factory line. After the raw materials (the data set) are input, they are then passed down the conveyer belt, with each subsequent stop or layer extracting a different set of high-level features. If the network is intended to recognize an object, the first layer might analyze the brightness of its pixels. The next layer could then identify any edges in the image, based on lines of similar pixels. After this, another layer may recognize textures and shapes, and so on. By the time the fourth or fifth layer is reached, the deep learning net will have created complex feature detectors. It can figure out that certain image elements (such as a pair of eyes, a nose, and a mouth) are commonly found together. Once this is done, the researchers who have trained the network can give labels to the output, and then use backpropagation to correct any mistakes which have been made. After a while, the network can carry out its own classification tasks without needing humans to help every time.

So, this is simply how neural networks work. There is a big variety of different networks architectures – convolutional neural networks, recurrent neural networks, etc. All of them have more complicated features and tricks, but the basic idea is the same. The good thing is that all you need to know to understand how neural-based distributional semantic models work: the models considered in this paragraph are based on very simple three-layer neural networks. Despite this they are very effective, though! Let's continue and understand how can we connect neural networks and distributional semantics.

4.2 Doing the Prediction

So, imagine a neural network that does certain task. For example, predicts part-of-speech tags. The idea is that we have a set of possible tags (like "noun", "adjective", etc) and we need to associate tag to each word. We can try to put each word one-by-one in the network and predict tag for it. It will looks like we are putting the word, receive tag on the output, compare it with desired tag and update weights if it is not the same.

Of course, words and tags are unstuctured information, so we need to somehow present them in a computational form. In case with tags we can use numerical features. What should we do with words?

We can also try to use numerical features: representation of each word would be its index in vocabulary. Such approach also could take place, but indices are usually replaced with vectors where all components are zeros and one component is one. Such representations are called **one-hot encodings (OHEs)**.

Thus, we can represent words with one-hot encodings, and predict POS-tags, and update weights of neurons on hidden layers. Conventionally, supervised lexicalized NLP approaches take a word and convert it to a symbolic ID, which is then transformed into a feature vector

using a one-hot representation: the feature vector has the same length as the size of the vocabulary, and only one dimension is on. However, the **one-hot representation** of a word suffers from data sparsity: namely, for words that are rare in the labeled training data, their corresponding model parameters will be poorly estimated. Moreover, at test time, the model cannot handle words that do not appear in the labeled training data. These limitations of one-hot word representations have prompted researchers to investigate unsupervised methods for inducing word representations over large unlabeled corpora. Word features can be hand-designed, but our goal is to learn them.

One common approach to inducing unsupervised word representation is to use clustering, perhaps hierarchical. This leads to a one-hot representation over a smaller vocabulary size. Neural language models, on the other hand, induce dense real-valued low-dimensional word embeddings using unsupervised approaches.

We can try to use only one, input layer! We will be updating weights only on one layer. You remember, on the start of initialization we have network with random weights. The same would be with vocabulary. We can represent each word with a random vector and then tune its component to make it possible to predict POS-tag. And this is how neural network works! We have network that is doing some tasks, and word representation obtained by learning it are called **word embeddings**!

However, task of POS-tagging is not very suitable. Therefore, many models rely on word prediction task.

4.3 History

To understand how much today prediction-based model evolved we will briefly overview historical approaches.

4.3.1 Statistical Lanuage Modeling

The idea of prediction-based models is to predict how it is likely to encounter word in a given context. This process is called **language modeling**. Thus, a **statistical model of language** can be represented by the conditional probability of the next word given all the previous ones, since

$$\hat{P}(w_1^T) = \prod_{t=1}^{T} \hat{P}(w_t \mid w_1^{t-1})$$

where w_t is the t-th word, and writing sub-sequence $w_i^j = (w_i, w_{i+1}, \ldots, w_{j-1}, w_j)$.

Such statistical language models have been found useful in many technological applications involving natural language, such as speech recognition, machine translation, and information retrieval.

When building statistical models of natural language, one considerably reduces the difficulty of this modeling problem by taking advantage of word order, and the fact that temporally closer words in the word sequence are statistically more dependent. Thus, **n-gram models** construct (n-dimensional) tables of conditional probabilities for the next word, for each one of a large number of *contexts*, i. e. combinations of the last $n-1$ words:

$$\hat{P}(w_t \mid w_1^{t-1}) \approx \hat{P}(w_t \mid w_{t-n+1}^{t-1}).$$

However, such tables have $O\left(|\mathcal{W}|^n\right)$ elements, which is almost impossible to store, and, moreover, to obtain satisfying statistical estimations of trainable parameters (which are

tables in this case), we need a corpora of unrealistic size. This phenomenon is called **curse of dimensionality**. That's why in practice it is common to choice $n = 1, 2, 3$.

The training objective tries to maximize the log-likelihood function:

$$L = \frac{1}{T} \log \hat{P}(w_1^T) \approx \frac{1}{T} \log \left(\prod_{t=1}^{T} \hat{P}(w_t \mid w_{t-n+1}^{t-1}) \right) = \frac{1}{T} \sum_{t=1}^{T} \log \hat{P}(w_t \mid w_{t-n+1}^{t-1}).$$

Usually, the term word embeddings in previous literature was also used interchangeably with term "distributed representations" (not to be confused with distributional representations). A **distributed representation** is dense, low-dimensional, and real-valued. Distributed word representations are called **word embeddings**. Each dimension of the embedding represents a latent feature of the word, hopefully capturing useful syntactic and semantic properties. A distributed representation is compact, in the sense that it can represent an exponential number of clusters in the number of dimensions.

4.3.2 A Neural Probabilistic Language Model

In order to fight the curse of dimensionality, a **neural probabilistic language model (NPLM)** was proposed in Bengio et al. (2003). In a nutshell, the idea of the proposed approach can be summarized as follows:

1. associate with each word in the vocabulary an embedding (a real-valued vector in $\mathbb{R}^d$);
2. express the joint probability function of word sequences in terms of the embeddings of these words in the sequence,
3. learn simultaneously the word embeddings and the parameters of that probability function.

The feature vector represents different aspects of the word: each word is associated with a point in a vector space. The number of features d is much smaller than the size of the vocabulary. The probability function is expressed as a product of conditional probabilities of the next word given the previous ones, (e. g. using a multi-layer neural network to predict the next word given the previous ones). This function has parameters that can be iteratively tuned in order to maximize the log-likelihood of the training data or a regularized criterion, e. g. by adding a weight decay penalty. The feature vectors associated with each word are learned, but they could be initialized using prior knowledge of semantic features.

The training set is a sequence $w_1, \ldots, w_T$ of words $w_t \in \mathcal{W}$, where the vocabulary $\mathcal{W}$ is a large but finite set. The objective is to learn a good model $f(w_t, \ldots, w_{t-n+1}) = \hat{P}(w_t \mid w_1^{t-1})$, in the sense that it gives high out-of-sample likelihood. Below, we report the geometric average of $1/\hat{P}(w_t \mid w_1^{t-1})$, also known as perplexity, which is also the exponential of the average negative log-likelihood. The only constraint on the model is that for any choice of w_1^{t-1}, $\sum_{w \in \mathcal{W}} f(w, w_{t-1}, \ldots, w_{t-n+1}) = 1$, with $f > 0$. By the product of these conditional probabilities, one obtains a model of the joint probability of sequences of words.

We decompose the function $f(w_t, w_{t-1}, \ldots, w_{t-n+1}) = \hat{P}(w_t \mid w_1^{t-1})$ in two parts:

1. A mapping C from any element w of $\mathcal{W}$ to a real vector $C(w) \in \mathbb{R}^d$. It represents the distributed feature vectors associated with each word in the vocabulary. In practice, C is represented by a $|\mathcal{W}| \times d$ matrix of free parameters.

2. The probability function over words, expressed with C: a function g maps an input sequence of feature vectors for words in context, $(C(w_{t-n+1}), \ldots, C(w_{t-1}))$, to a conditional probability distribution over words in $\mathcal{W}$ for the next word w_t. The output of g is a vector whose i-th element estimates the probability $\hat{P}(w_t = w^{(i)} \mid w_1^{t-1})$:

$$f(w^{(i)}, w_{t-1}, \ldots, w_{t-n+1}) = g(i, C(w_{t-1}), \ldots, C(w_{t-n+1})).$$

The function f is a composition of these two mappings (C and g), with C being shared across all the words in the context. With each of these two parts are associated some parameters. The parameters of the mapping C are simply the feature vectors themselves, represented by a $|\mathcal{W}| \times d$ matrix $\boldsymbol{C}$ whose row i is the feature vector $C(w^{(i)})$ for word $w^{(i)}$. The function g may be implemented by a feed-forward or recurrent neural network or another parametrized function, with parameters $\boldsymbol{\omega}$. The overall parameter set is $\boldsymbol{\theta} = (\boldsymbol{C}, \boldsymbol{\omega})$.

Training is achieved by looking for $\boldsymbol{\theta}$ that maximizes the training corpus penalized log-likelihood:

$$L = \frac{1}{T} \sum_t \log f(w_t, w_{t-1}, \ldots, w_{t-n+1} \mid \boldsymbol{\theta}) + R(\boldsymbol{\theta}),$$

where $R(\boldsymbol{\theta})$ is a regularization term.

The neural network computes the following function, with a softmax output layer, which guarantees positive probabilities summing to 1:

$$\hat{P}(w_t \mid w_1^{t-1}) = \frac{\exp y_{w_t}}{\sum_{w \in \mathcal{W}} \exp y_w}$$

The y_w are the unnormalized log-probabilities for each output word w, computed as follows, with parameters $\boldsymbol{b}$, $\boldsymbol{W}$, $\boldsymbol{U}$, $\boldsymbol{d}$ and $\boldsymbol{H}$:

$$\boldsymbol{y} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x} + \boldsymbol{U} \tanh(\boldsymbol{d} + \boldsymbol{H}\boldsymbol{x})$$

where the hyperbolic tangent tanh is applied element by element, $\boldsymbol{W}$ is optionally zero (no direct connections), and $\boldsymbol{x}$ is the word features layer activation vector, which is the concatenation of the input word features from the matrix $\boldsymbol{C}$:

$$\boldsymbol{x} = [C(w_{t-1}), C(w_{t-2}), \ldots, C(w_{t-n+1})].$$

In this model, $\boldsymbol{\theta} = (\boldsymbol{C}, \boldsymbol{b}, \boldsymbol{W}, \boldsymbol{U}, \boldsymbol{d}, \boldsymbol{H})$.

How many trainable parameters are in $\boldsymbol{\theta}$?

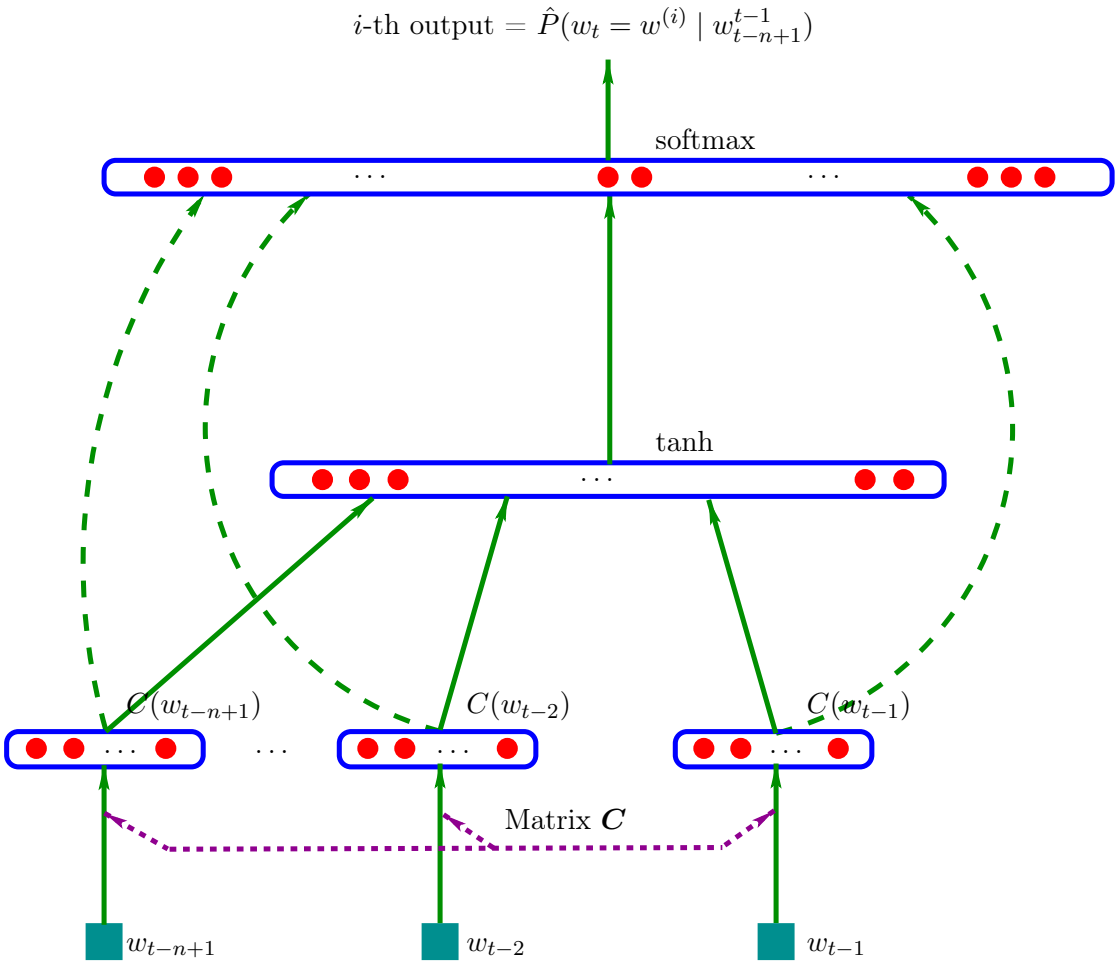

Figure 4.1: Architecture of Bengio NPLM

4.3.3 SENNA

Collobert and Weston (2008) presented a neural language model that could be trained over billions of words, because the gradient of the loss was computed stochastically over a small sample of possible outputs, in a spirit similar to Bengio.

The architecture is as follows. For each training update, we read an n-gram $w_{t-n+1}, \ldots, w_t$ from the corpus and predict w_t using previous $n-1$ words. The embedding matrix here is $\boldsymbol{W}$. We use a look-up table to find the indices of the given words. After that, the input sequence is transformed into emdeddings $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n-1} := \boldsymbol{W}_{w_{t-n+1},:}, \ldots, \boldsymbol{W}_{w_{t-1},:}$. On the next step, we apply a convolution layer:

$$\boldsymbol{o}_s = \sum_{j=1-s}^{n-1-s} \boldsymbol{L}_j \boldsymbol{x}_{s+j}$$

where $\boldsymbol{L}_j \in \mathbb{R}^{h \times d}$, $-n+1 \leq j \leq n-1$ (h is a size of hidden units). One usually constrains this convolution by defining a kernel width, k, which enforces $|j| > (k-1)/2 \Rightarrow \boldsymbol{L}_j = 0$. After convolution, we use max-pooling over time, namely

$$\boldsymbol{m} = \max_{s=1}^{n-1} \boldsymbol{o}_s$$

(remind that max operation is applied by components). Finally, we apply a layer that gives the output of size $|\mathcal{W}|$ (e. g. linear layer):

$$\boldsymbol{y} = \boldsymbol{b} + \boldsymbol{H}\boldsymbol{m}$$

and after that to obtain the probability distribution we transform the output $\boldsymbol{y}$ using softmax.

In this model, $\boldsymbol{\theta} = (\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{H}, \mathsf{L}, \boldsymbol{W})$.

How many trainable parameters are in $\boldsymbol{\theta}$?

Note that instead of predicting the last word in a sequence, architecture allows modification that predict a central word.

4.3.4 HLBL

The **log-bilinear model (LBL)** is a probabilistic and linear neural model. Like virtually all neural language models, the LBL model represents each word with a real-valued feature vector. We will denote the feature vector for word w by $\boldsymbol{u}_w$.To predict the next word w_t given the context $w_{t-n+1}, \ldots, w_{t-1}$, the model computes the predicted feature vector $\hat{u}_t$ for the next word by linearly combining the context word feature vectors:

$$\hat{\boldsymbol{u}}_t = \sum_{j=t-n+1}^{t-1} c_{j-t+n} \boldsymbol{u}_{w_j},$$

where c_i is the weight associated with the context position i. Then the similarity between the predicted feature vector and the feature vector for each word in the vocabulary is computed using the inner product. The similarities are then exponentiated and normalized to obtain the distribution over the next word:

$$\hat{P}(w_t = w \mid w_{t-n+1}^{t-1}) = \frac{\exp\left(\hat{\boldsymbol{u}}_t^\top \boldsymbol{u}_w + b_w\right)}{\sum_{w' \in \mathcal{W}} \exp\left(\hat{\boldsymbol{u}}_t^\top \boldsymbol{u}_{w'} + b_{w'}\right)}.$$

Here b_w is the bias for word w, which is used to capture the context-independent word frequency.

Mnih and Hinton (2009) speed up model evaluation during training and testing by using a hierarchy to exponentially filter down the number of computations that are performed. This hierarchical evaluation technique was first proposed by Morin and Bengio (2005). The model, combined with this optimization, is called the **hierarchical log-bilinear (HLBL)** model.

The first component of the HLBL is a binary tree with words at its leaves. For now, we will assume that each word in the vocabulary is at exactly one leaf. Then each word can be uniquely specified by a path from the root of the tree to the leaf node the word is at. The path itself can be encoded as a binary string d of decisions made at each node, so that $d_i = 1$ corresponds to the decision to visit the left child of the current node. For example, the string $d = 10$ corresponds to a path that starts at the root, visits its left child, and then visits the right child of that child. This allows each word to be represented by a binary string which we will call a code.

The second component of the HLBL model is the probabilistic model for making the decisions at each node, which in our case is a modified version of the LBL model. In the HLBL model, just like in its non-hierarchical counterpart, context words are represented

using real-valued feature vectors. Each of the non-leaf nodes in the tree also has a feature vector associated with it that is used for discriminating the words in the left subtree form the words in the right subtree of the node. Unlike the context words, the words being predicted are represented using their binary codes that are determined by the word tree. However, this representation is still quite flexible, since each binary digit in the code encodes a decision made at a node, which depends on that node's feature vector.

In the HLBL model, the probability of the next word being w is the probability of making the sequences of binary decisions specified by the word's code, given the context. Since the probability of making a decision at a node depends only on the predicted feature vector, determined by the context, and the feature vector for that node, we can express the probability of the next word as a product of probabilities of the binary decisions:

$$\hat{P}(w_t = w \mid w_{t-n+1}^{t-1}) = \prod_i \hat{P}(d_i \mid \boldsymbol{q}_i, w_{t-n+1}^{t-1})$$

where d_i is i-th digit in the code for word w, and $\boldsymbol{q}_i$ is the feature vector for the i-th node in the path corresponding to that code. The probability of each decision is given by

$$\hat{P}(d_i = 1 \mid \boldsymbol{q}_i, w_{t-n+1}^{t-1}) = \sigma(\hat{\boldsymbol{u}}_t^\top \boldsymbol{q}_i + b_i)$$

where $\sigma(x)$ is the logistic function, and b_i is the node's bias that captures the context-independent tendency to visit the left child when leaving this node.

The definition of $\hat{P}(w_t = w \mid w_{t-n+1}^{t-1})$ can be extended to multiple codes per word by including a summation over all codes for w as follows:

$$\hat{P}(w_t = w \mid w_{t-n+1}^{t-1}) = \sum_{d \in \mathbb{D}(w)} \prod_i \hat{P}(d_i \mid \boldsymbol{q}_i, w_{t-n+1}^{t-1})$$

where $\mathbb{D}(w)$ is a set of codes corresponding to word w. Allowing multiple codes per word can allow better prediction of words that have multiple senses or multiple usage patterns. Using multiple codes per word also makes it easy to combine several separate words hierarchies to into a single one to to reflect the fact that no single hierarchy can express all the relationships between words.

HLBL models can be trained by maximizing the (penalized) log-likelihood. Since the probability of the next word depends only on the context weights, the feature vectors of the context words, and the feature vectors of the nodes on the paths from the root to the leaves containing the word in question, only a (logarithmically) small fraction of the parameters need to be updated for each training case.

4.4 Word2Vec

Two particular models for learning word representations that can be efficiently trained on large amounts of text data are Skip-gram and CBOW models introduced in Mikolov et al. (2013a).

4.4.1 Continuous Bag-of-Words Model

The first proposed architecture is similar to the NPLM, 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.

We denote this model further as **continuous bag-of-words (CBOW)**, as unlike standard bag-of-words model, it uses continuous distributed representation of the context. The model architecture is shown at 4.2. 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 NPLM.

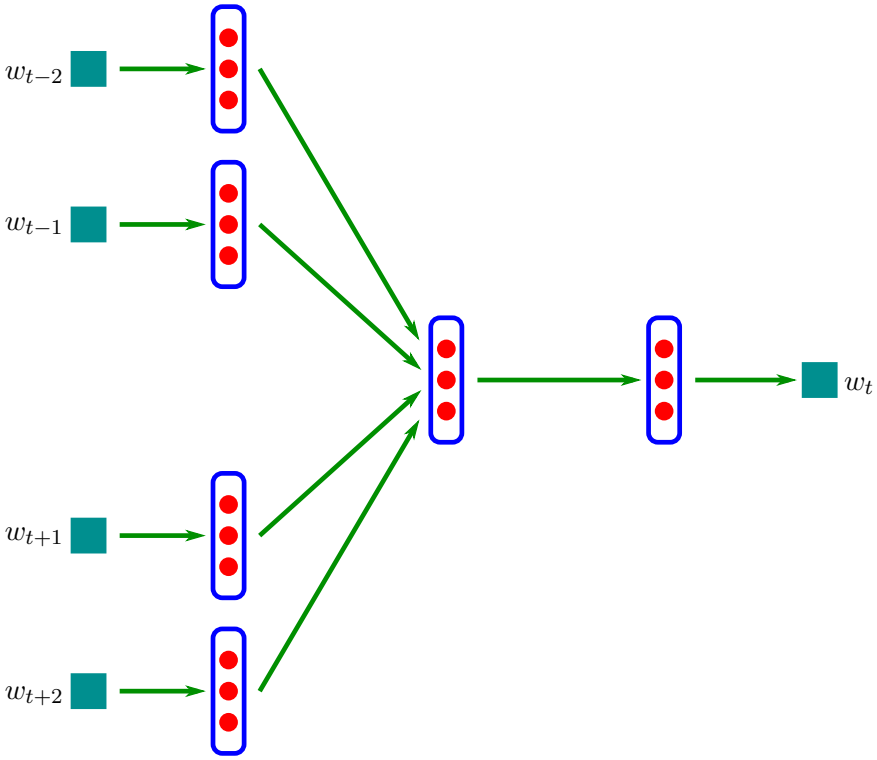

Figure 4.2: Architecture of the CBOW model

Similarly, in the **Skip-gram** model, the training objective is to learn word vector representations that are good at predicting its context in the same sentence. It is unlike traditional neural network based language models, where the objective is to predict the next word given the context of several preceding words. Due to their low computational complexity, the Skip-gram and CBOW models can be trained on a large corpus in a short time (billions of words in hours). In practice, Skip-gram gives better word representations when the monolingual data is small. CBOW however is faster and more suitable for larger datasets. They also tend to learn very similar representations for languages. Due to their similarity in terms of model architecture, the rest of the section will focus on describing the Skip-gram model.

4.4.2 Continuous Skip-gram Model

Given a sequence of training words $w_1, w_2, w_3, \ldots, w_T$, the objective of the Skip-gram model is to maximize the average log probability

$$L = \frac{1}{T} \sum_{t=1}^{T} \sum_{c \in \mathcal{C}_t} \log P(c \mid w_t),$$

where $\mathcal{C}_t = \cup_{j=-win, j \neq 0}^{win} w_{t+j}$ and win is the size of the training window (which can be a function of the center word w_t). The inner summation goes from $-win$ to win to compute the log probability of correctly predicting the word w_{t+j} given the word in the middle w_t. The outer summation goes over all words in the training corpus.

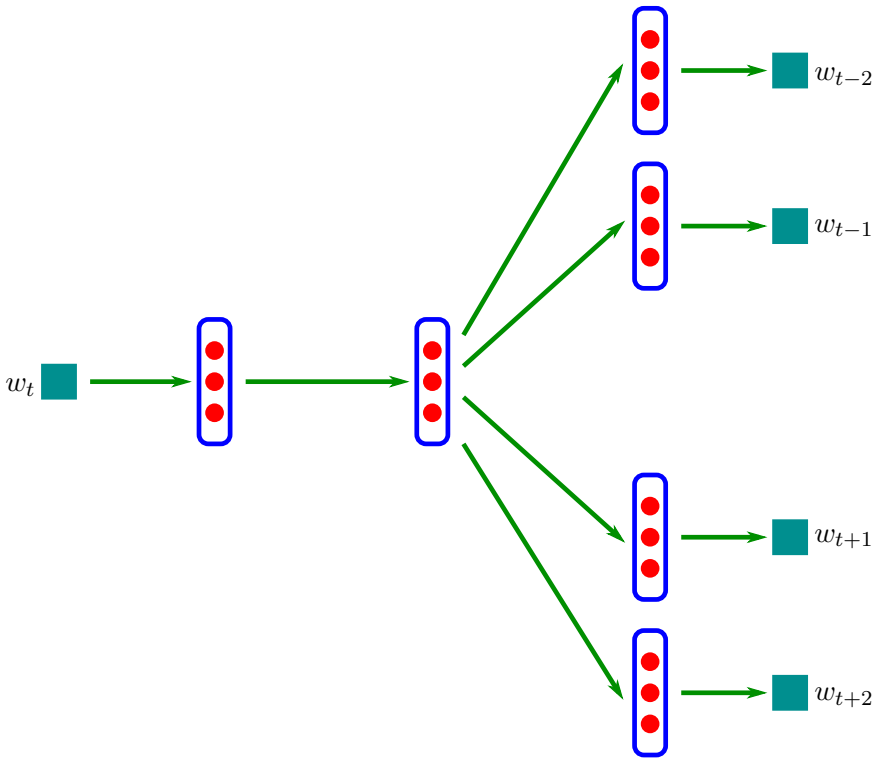

Figure 4.3: Architecture of the Skip-gram model

In the Skip-gram model, every word w is associated with two learnable parameter vectors, $\boldsymbol{u}_w$ and $\boldsymbol{v}_w$. They are the "input" and "output" vectors of the w respectively. The probability of correctly predicting the context word c given the central word w_t is defined as

$$P(c \mid w_t) = \frac{\exp\left(\boldsymbol{u}_{w_t}^\top \boldsymbol{v}_c\right)}{\sum_{c' \in \mathcal{W}} \exp\left(\boldsymbol{u}_{w_t}^\top \boldsymbol{v}_{c'}\right)}.$$

This formulation is impractical because the cost of computing $\nabla \log P(c \mid w_t)$ is proportional to the number of words in the vocabulary $|\mathcal{W}|$ (which can be easily in order of millions).

4.4.3 Hierarchical Softmax

A computationally efficient approximation of the full softmax is the **hierarchical softmax**. In the context of neural network language models, it was first introduced in Morin and

Bengio (2005). The main advantage is that instead of evaluating $|\mathcal{W}|$ output nodes in the neural network to obtain the probability distribution, it is needed to evaluate only about $\log_2 |\mathcal{W}|$ nodes.

The hierarchical softmax uses a binary tree representation of the output layer with the $|\mathcal{W}|$ words as its leaves and, for each node, explicitly represents the relative probabilities of its child nodes. These define a random walk that assigns probabilities to words.

More precisely, each word w can be reached by an appropriate path from the root of the tree. Let $n(w, j)$ be the j-th node on the path from the root to w, and let $L(w)$ be the length of this path, so $n(w, 1) = \text{root}$ and $n(w, L(w)) = w$. In addition, for any inner node n, let $\text{ch}(n)$ be an arbitrary fixed child of n and let $[\![x]\!]$ be 1 if x is true and -1 otherwise. Then the hierarchical softmax defines $P(c \mid w_t)$ as follows:

$$P(c \mid w_t) = \prod_{j=1}^{L(c)-1} \sigma\left([\![n(c, j+1) = \text{ch}(n(c, j))]\!] \boldsymbol{u}_{w_t}^\top \boldsymbol{v}_{n(c,j)}\right)$$

where $\sigma(x) = 1/(1 + \exp(-x))$. It can be verified that $\sum_{c=1}^{|} \mathcal{W}| P(c \mid w_t) = 1$. This implies that the cost of computing $\log P(c \mid w_t)$ and $\nabla \log P(c \mid w_t)$ is proportional to $L(c)$, which on average is no greater than $\log_2 |\mathcal{W}|$. Also, unlike the standard softmax formulation of the Skip-gram which assigns two representations $\boldsymbol{u}_w$ and $\boldsymbol{v}_w$ to each word w, the hierarchical softmax formulation has one representation $\boldsymbol{v}_w$ for each word w and one representation $\boldsymbol{v}_n$ for every inner node n of the binary tree.

The structure of the tree used by the hierarchical softmax has a considerable effect on the performance.

4.4.4 Negative Sampling

An alternative to the hierarchical softmax **is noise contrastive estimation (NCE)**, which was introduced by Gutmann and Hyvarinen **?** and applied to language modeling by Mnih and Teh **?**. NCE posits that a good model should be able to differentiate data from noise by means of logistic regression. This is similar to hinge loss used by Collobert and Weston **?** who trained the models by ranking the data above noise.

While NCE can be shown to approximately maximize the log probability of the softmax, the Skip-gram model is only concerned with learning high-quality vector representations, so we are free to simplify NCE as long as the vector representations retain their quality. We define **negative sampling (NEG)** by the objective

$$\log \sigma(\boldsymbol{u}_w^\top \boldsymbol{v}_c) + \sum_{j=1}^{k} \mathbb{E}_{c_{n,j} \sim P_n} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_{c_{n,j}})$$

which is used to replace every $\log P(c \mid w_t)$ term in the Skip-gram objective. Thus the task is to distinguish the target word c from draws c_n from the noise distribution $P_n(w)$ using logistic regression, where there are k negative samples for each data sample. The experiments indicate that values of k in the range 5–20 are useful for small training datasets, while for large datasets the k can be as small as 2–5. The main difference between the Negative sampling and NCE is that NCE needs both samples and the numerical probabilities of the noise distribution, while Negative sampling uses only samples. And while NCE approximately maximizes the log probability of the softmax, this property is not important for our application.

4.4.5 Subsampling of Frequent Words

In very large corpora, the most frequent words can easily occur hundreds of millions of times (e.g., "in", "the", and "a"). Such words usually provide less information value than the rare words. For example, while the Skip-gram model benefits from observing the co-occurrences of "France" and "Paris", it benefits much less from observing the frequent co-occurrences of "France" and "the", as nearly every word co-occurs frequently within a sentence with "the". This idea can also be applied in the opposite direction; the vector representations of frequent words do not change significantly after training on several million examples.

To counter the imbalance between the rare and frequent words, the authors used a simple subsampling approach: each word w_t in the training set is discarded with probability computed by the formula

$$P(w_t) = 1 - \sqrt{\frac{r}{\#(w_t)}}$$

where r is a chosen threshold, typically around 10^{-5}. This subsampling formula aggressively subsamples words whose frequency is greater than r while preserving the ranking of the frequencies. Although this subsampling formula was chosen heuristically, it works well in practice. It accelerates learning and even significantly improves the accuracy of the learned vectors of the rare words.

4.5 Connection to Count-based Models

Levy casted SGNS's training method as weighted matrix factorization, and shown that its objective is implicitly factorizing a shifted PMI matrix—the well-known word-context PMI matrix from the word-similarity literature, shifted by a constant offset. In this subsection, we will prove that fact.

4.5.1 SGNS's Objective

Consider a word-context pair (w, c). Did this pair come from the observed data? Let $P(D = 1 \mid w, c)$ be the probability that (w, c) came from the data, and $P(D = 0 \mid w, c) = 1 - P(D = 1 \mid w, c)$ the probability that (w, c) did not. The distribution is modeled as:

$$P(D = 1 \mid w, c) = \sigma(\boldsymbol{u}_w^\top \boldsymbol{v}_c) = \frac{1}{1 + \exp\left(-\boldsymbol{u}_w^\top \boldsymbol{v}_c\right)}$$

where $\boldsymbol{u}_w$ and $\boldsymbol{v}_c$ (d-dimensional vectors) are the model parameters to be learned.

The negative sampling objective tries to maximize $P(D = 1 \mid w, c)$ for observed (w, c) pairs while maximizing $P(D = 0 \mid w, c_n)$ for randomly sampled "negative" examples (hence the name "negative sampling"), under the assumption that randomly selecting a context for a given word is likely to result in an unobserved (w, c_n) pair. SGNS's objective for a single (w, c) observation is then:

$$\log \sigma(\boldsymbol{u}_w^\top \boldsymbol{v}_c) + k \cdot \mathbb{E}_{c_n \sim P_n} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_{c_n})$$

Derive this from the expression $P(D = 1 \mid w, c) \prod_{j=1}^{k} P(D = 0 \mid w, c_{n,j})$.

where k is the number of "negative" samples and c_n is the sampled context, drawn according to the empirical unigram distribution

$$P_n(c) = \frac{\#(c)}{T} = \frac{\#(c)}{\sum_{c' \in \mathcal{W}} \#(c')}.$$

The objective is trained in an online fashion using stochastic gradient updates over the observed pairs in the corpus. The global objective then sums over the observed (w, c) pairs in the corpus:

$$L = \sum_{w \in \mathcal{W}} \sum_{c \in \mathcal{W}} \#(w, c)(\log \sigma(\boldsymbol{u}_w^\top \boldsymbol{v}_c) + k \cdot \mathbb{E}_{c_n \sim P_n} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_{c_n})).$$

4.5.2 SGNS as Implicit Matrix Factorization

SGNS embeds both words and their contexts into a low-dimensional space $\mathbb{R}^d$, resulting in the word and context matrices $\boldsymbol{U}$ and $\boldsymbol{V}$. It is instructive to consider the product $\boldsymbol{U}\boldsymbol{V}^\top = \boldsymbol{M}$. Viewed this way, SGNS can be described as factorizing an implicit matrix $\boldsymbol{M}$ of dimensions $|\mathcal{W}| \times |\mathcal{W}|$ into two smaller matrices, and $M_{w,c} = \boldsymbol{u}_w^\top \boldsymbol{v}_c$. So, each cell of matrix $\boldsymbol{M}$ contains a quantity $f(w, c)$ reflecting the strength of association between that particular word-context pair. What can we say about the association function $f(w, c)$? In other words, which matrix is SGNS factorizing?

Consider the global objective (equation 2) above. For sufficiently large dimensionality d (i.e. allowing for a perfect reconstruction of $\boldsymbol{M}$), each product $\boldsymbol{u}_w^\top \boldsymbol{v}_c$ can assume a value independently of the others. Under these conditions, we can treat the objective L as a function of independent $\boldsymbol{u}_w^\top \boldsymbol{v}_c$ terms, and find the values of these terms that maximize it.

We begin by rewriting equation 2:

$$\begin{aligned} L &= \sum_{w \in \mathcal{W}} \sum_{c \in \mathcal{W}} \#(w, c) \log \sigma(\boldsymbol{u}_w^\top \boldsymbol{v}_c) + \sum_{w \in \mathcal{W}} \sum_{c \in \mathcal{W}} \#(w, c)(k \cdot \mathbb{E}_{c_n \sim P_n} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_{c_n})) = \\ &= \sum_{w \in \mathcal{W}} \sum_{c \in \mathcal{W}} \#(w, c) \log \sigma(\boldsymbol{u}_w^\top \boldsymbol{v}_c) + \sum_{w \in \mathcal{W}} \#(w)(k \cdot \mathbb{E}_{c_n \sim P_n} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_{c_n})) \end{aligned}$$

and explicitly expressing the expectation term:

$$\begin{aligned} \mathbb{E}_{c_n \sim P_n} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_{c_n}) &= \sum_{c_n \in \mathcal{W}} \frac{\#(c_n)}{T} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_{c_n}) = \\ &= \frac{\#(c)}{T} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_c) + \sum_{c_n \in \mathcal{W} \setminus \{c\}} \frac{\#(c_n)}{T} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_{c_n}). \end{aligned}$$

Combining equations 3 and 4 reveals the local objective for a *specific* (w, c) pair:

$$L(w, c) = \#(w, c) \log \sigma(\boldsymbol{u}_w^\top \boldsymbol{v}_c) + k \cdot \#(w) \cdot \frac{\#(c)}{T} \log \sigma(-\boldsymbol{u}_w^\top \boldsymbol{v}_c).$$

To optimize the objective, we define $x = \boldsymbol{u}_w^\top \boldsymbol{v}_c$ and find its partial derivative with respect to x:

$$\frac{\partial L}{\partial x} = \#(w, c) \cdot \sigma(-x) - k \cdot \#(w) \cdot \frac{\#(c)}{T} \cdot \sigma(x).$$

We compare the derivative to zero, and after some simplification, arrive at:

$$y^2 - (a - 1)y - a = 0$$

where

$$y = e^x,\ a = \frac{\#(w, c) \cdot T}{\#(w)\#(c)} \cdot \frac{1}{k}.$$

This quadratic equation has two solutions: $y = -1$ (which is invalid given the definition of y) and $y = a$. Substituting y with e^x and x with $\boldsymbol{u}_w^\top \boldsymbol{v}_c$ reveals:

$$\boldsymbol{u}_w^\top \boldsymbol{v}_c = \log a = \log \left(\frac{\#(w,c) \cdot T}{\#(w)\#(c)} \right) - \log k.$$

Thus, $M_{w,c} = f(w,c) = \log 2 \cdot PMI_{w,c} - \log k$.

In the original paper, PMI is defined as a natural logarithm, so a multiplier $\log 2$ does not appear there. What is the geometrical difference between matrices PMI and $const \cdot PMI$?

For a negative-sampling value of $k = 1$, the SGNS objective is factorizing a word-context matrix in which the association between a word and its context is measured by $f(w,c) = \log 2 \cdot PMI_{w,c}$.

4.6 Seminar

- Hands-on tutorial on Word2Vec (gensim).
- Arithmetic operations with word vectors.
- **Homework:** to implement skip-gram or CBOW.

Bibliography

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA. ACM.

Levy, O. and Goldberg, Y. (2014). Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185.

Manning, C. D. and Schütze, H. (1999). *Foundations of statistical natural language processing*. MIT press.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Mnih, A. and Hinton, G. E. (2009). A scalable hierarchical distributed language model. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems 21*, pages 1081–1088. Curran Associates, Inc.

Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In Cowell, R. G. and Ghahramani, Z., editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*, pages 246–252. Society for Artificial Intelligence and Statistics.