1 Word Representations

You shall know a word by the company it keeps.

— J.R. Firth, 1957

The epigraph at the beginning of this section is overused. However, it is obligatory to include it in a work such as this, as it so perfectly sums up why representations useful for language modelling are representations that capture semantics (as well as syntax).

Word embeddings are the core innovation that has brought machine learning to the forefront of natural language processing. This chapter discusses how one can create a numerical vector that captures the salient features (e.g. semantic meaning) of a word. Discussion begins with the classic language modelling problem. By solving this, using a neural network-based approach, word-embeddings are created. Techniques such as CBOW and skip-gram models (`word2vec`), and more recent advances in relating this to common linear algebraic reductions on co-locations as discussed. The chapter also includes a detailed discussion of the often confusing hierarchical softmax, and negative sampling techniques. It concludes with a brief look at some other applications and related techniques.

Word Vector or Word Embedding?

Some literature uses the term *word vector*, or *vector-space model* to refer to representations from LDA and LSA etc. Other works use the terms are used synonymously with *word embedding*. Word embeddings are vectors, in any case.

We begin the consideration of the representation of words using neural networks with the work on language modeling. This is not the only place one could begin the consideration: the information retrieval models, such as LSI (**dumais1988using**) and LDA (**blei2003latent**), based on word co-location with documents would be the other obvious starting point. However, these models are closer to the end point, than they are to the beginning, both chronologically, and in this chapter's layout. From the language modeling work, comes the contextual (or acausal) language model works such as skip-gram, which in turn lead to the post-neural network co-occurrence based works. These co-occurrence works are more similar to the information retrieval co-location based methods than the probabilistic language

dumais1988using, **dumais1988using**

blei2003latent, **blei2003latent**

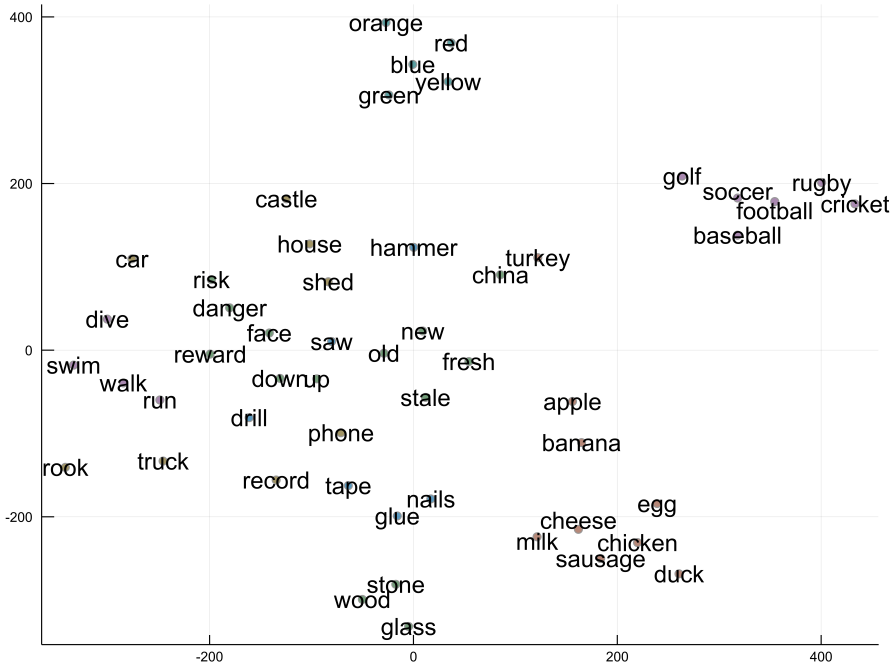

Figure 1.1: Some word embeddings from the FastText project (**bojanowski2016enriching**). They were originally 300 dimensions but have been reduced to 2 using t-SNE (**maaten2008tsne**) algorithm. The colors are from 5 manually annotated categories done before this visualisation was produced: `foods`, `sports`, `colors`, `tools`, `other objects`, `other`. Note that many of these words have multiple meanings (see **??**), and could fit into multiple categories. Also notice that the information captioned by the unsupervised word embeddings is far finer grained than the manual categorisation. Notice, for example, the separation of ball-sports, from words like `run` and `walk`. Not also that `china` and `turkey` are together; this no doubt represents that they are both also countries.

Probability writing convention
We follow convention that capitalised W^i is a random variable, and w^i is a particular value which W^i may take. The probability of it taking that value would normally be written $P(W^i{=}w^i)$. We simply write $P(w^i)$ to mean the same thing. This is a common abridged (abuse-of) notation. The random variable in question is implicitly given by the name of its value.

rosenfeld2000two, **rosenfeld2000two**

modeling methods for word embeddings from which we begin this discussion.

Word embeddings are vector representations of words. An dimensionality reduced scatter plot example of some word embeddings is shown in Figure 1.1.

1.1 Representations for Language Modeling

The language modeling task is to predict the next word given the prior words (**rosenfeld2000two**). For example, if a sentence begins `For lunch I will have a hot`, then there is a high probability that the next word will be `dog` or `meal`, and lower probabilities of words such as `day` or `are`. Mathematically it is formulated as:

$$P(W^i{=}w^i \mid W^{i-1}{=}w^{i-1}, \ldots, W^1{=}w^1) \qquad (1.1)$$

or to use the compact notation

$$P(w^i \mid w^{i-1}, \ldots, w^1) \qquad (1.2)$$

where W^i is a random variable for the ith word, and w^i is a value (a word) it could, (or does) take. For exam-

ple:

$$P(\texttt{dog} \mid \texttt{hot}, \texttt{a}, \texttt{want}, \texttt{I}, \texttt{lunch}, \texttt{For})$$

The task is to find the probabilities for the various words that w^i could represent.

The classical approach is trigram statistical language modeling. In this, the number of occurrences of word triples in a corpus is counted. From this joint probability of triples, one can condition upon the first two words, to get a conditional probability of the third. This makes the Markov assumption that the next state depends only on the current state, and that that state can be described by the previous two words. Under this assumption Equation (1.2) becomes:

$$P(w^i \mid w^{i-1}, \ldots, w^1) = P(w^i \mid w^{i-1}, w^{i-2}) \tag{1.3}$$

More generally, one can use an n-gram language model where for any value of n, this is simply a matter of defining the Markov state to contain different numbers of words.

This Markov assumption is, of-course, an approximation. In the previous example, a trigram language model finds $P(w^i \mid \texttt{hot}, \texttt{a})$. It can be seen that the approximation has lost key information. Based only on the previous 2 words the next word w^i could now reasonably be `day`, but the sentence: `For lunch I will have a hot day` makes no sense. However, the Markov assumption in using n-grams is required in order to make the problem tractable – otherwise an unbounded amount of information would need to be stored.

A key issue with n-gram language models is that there exists a data-sparsity problem which causes issues in training them. Particularly for larger values of n. Most combinations of words occur very rarely (**ha2009extending**). It is thus hard to estimate their occurrence probability. Combinations of words that do not occur in the corpus are naturally given a probability of zero. This is unlikely to be true though – it is simply a matter of rare phrases never occurring in a finite corpus. Several

maaten2008tsne, **maaten2008tsne**

Google n-gram corpora
Google has created a very large scale corpora of 1,2,3,4, and 5-grams from over 10^{12} words from the Google Books project. It has been made freely available at `https://books.google.com/ngrams/datasets` (**lin2012syntactic**). Large scale n-gram corpora are also used outside of statistical language modeling by corpus linguists investigating the use of language.

ha2009extending, **ha2009extending**

approaches have been taken to handle this. The simplest is add-one smoothing which adds an extra "fake" observation to every combination of terms. In common use are various back-off methods (**katz1987estimation**; **kneser1995improved**) which use the bigram probabilities to estimate the probabilities of unseen trigrams (and so forth for other n-grams.). However, these methods are merely clever statistical tricks – ways to reassign probability mass to leave some left-over for unseen cases. Back-off is smarter than add-one smoothing, as it portions the probability fairly based on the $(n-1)$-gram probability. Better still would be a method which can learn to see the common-role of words (**brown1992class**). By looking at the fragment: `For lunch I want a hot`, any reader knows that the next word is most likely going to be a food. We know this for the same reason we know the next word in `For elevenses I had a cold . . .` is also going to be a food. Even though `elevenses` is a vary rare word, we know from the context that it is a meal (more on this later), and we know it shares other traits with meals, and similarly `have / had`, and `hot / cold`. These traits influence the words that can occur after them. Hard-clustering words into groups is nontrivial, particularly given words having multiple meanings, and subtle differences in use. Thus the motivation is for a language modeling method which makes use of these shared properties of the words, but considers them in a flexible soft way. This motivates the need for representations which hold such linguistic information. Such representations must be discoverable from the corpus, as it is beyond reasonable to effectively hard-code suitable feature extractors. This is exactly the kind of task which a neural network achieves implicitly in its internal representations.

katz1987estimation; **kneser1995improved**, **katz1987estimation**; **kneser1995improved**

An extended look at classical techniques in statistical language modelling can be found in **DBLP:journals/corr/cs-CL-0108005**

brown1992class, **brown1992class**

1.1.1 The Neural Probabilistic Language Model

NPLM, **NPLM**

NPLM present a method that uses a neural network to create a language model. In doing so it implicitly learns

the crucial traits of words, during training. The core mechanism that allowed this was using an embedding or loop-up layer for the input.

1.1.1.1 Simplified Model considered with Input Embeddings

To understand the neural probabilistic language model, let's first consider a simplified neural trigram language model. This model is a simplification of the model introduced by **NPLM**. It follows the same principles, and highlights the most important idea in neural language representations. This is that of training a vector representation of a word using a lookup table to map a discrete scalar word to a continuous-space vector which becomes the first layer of the network.

The neural trigram probabilistic network is defined by:

$$P(w^i \mid w^{i-1}, w^{i-2}) = \operatorname{smax}\left(V\varphi\left(U\left[C_{:,w^{i-1}}; C_{:,w^{i-2}}\right] + \tilde{b}\right) + \tilde{k}\right) \quad (1.4)$$

where U, V, $\tilde{b}$, $\tilde{k}$ are the weight matrices and biases of the network. The matrix C defines the embedding table, from which the word embeddings, $C_{:,w^{i-1}}$ and $C_{:,w^{i-2}}$, representing the previous two words (w^{i-1} and w^{i-2}) are retrieved. The network is shown in Figure 1.2

In the neural trigram language model, each of the previous two words is used to look-up a vector from the embedding matrix. These are then concatenated to give a dense, continuous-space input to the above hidden layer. The output layer is a softmax layer, it gives the probabilities for each word in the vocabulary, such that $\hat{y}_{w^i} = P(w^i \mid w^{i-1}, w^{i-2})$. Thus producing a useful language model.

The word embeddings are trained, just like any other parameter of the network (i.e. the other weights and

Lookup word embeddings: Hashmap or Array?
The question is purely one of implementation. For purposes of the theory, it does not matter if the implementation is using a String to Vector dictionary (e.g. a hashmap), or a 2D array from which a column is indexed-out (sliced-from) via an integer index representing the word. In the tokenization of the source text, it is common to transform all the words into integers, so as to save memory, especially if string interning is not in use. At that point it makes sense to work with an array. For our notational purposes in this book, we will treat the word w^i as if it were an integer index, though thinking of it as a string index into a hashmap changes little in the logic.

$C_{:,w^i}$ **not** $C_{:,i}$
Note that here we use the word w^i as the index to lookup the word embeddings. i is the index of the word index in the corpus. That is to say that if the ith word, and the jth word are the same: i.e $w^i = w^j$, then they will index out the same vector from C. $w^i = w^j \implies C_{:,w^i} = C_{:,w^j}$.

One-hot product or Indexed-lookup

In some works you may see the process of retrieving the word vector from a matrix of word vectors described as a one-hot multiplication. For a word represented by the index w, where $\tilde{e}^w$ the one-hot vector with a 1 in the wth position, and for C, the table of word embeddings, one can write $C\,\tilde{e}^w$ to find the embedding for w. We will write $C_{:,w}$ and refer to this as looking up the word vector from the wth column. Of-course $C_{:,w} = C\,\tilde{e}^w$, however in practical implementation the performance ramifications are huge. Matrix column indexing is effectively an $O(1)$ operation (in a column major languages), whereas a dense matrix-vector product is $O(n^2)$. The one-hot product can be used in a pinch to support using embeddings in neural network toolkits that do not support lookup/embedding layers. However, we strongly suggest that if your toolkit does not support lookup/embedding layers then it is unsuitable for use in NLP applications. Some tool-kits, e.g. Flux.jl (`https://github.com/FluxML/Flux.jl`), explicitly handle sparse one-hot types, and automatically make this transformation. In that case, it is outright equivalent.

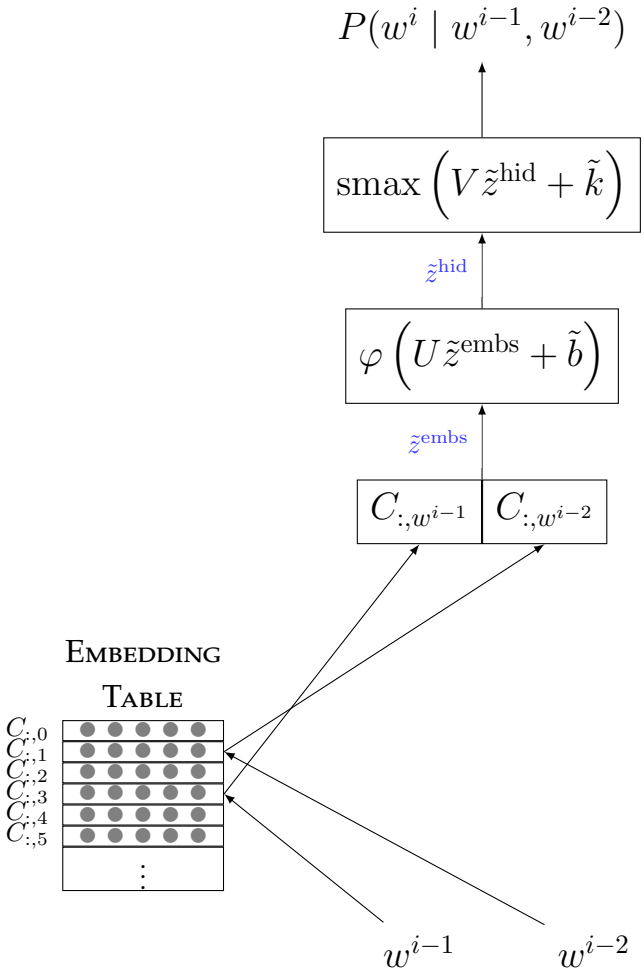

Figure 1.2: The Neural Trigram Language Model

biases) via gradient descent. An effect of this is that the embeddings of words which predict the same future word will be adjusted to be nearer to each other in the vector space. The hidden layer learns to associate information with regions of the embedding space, as the whole network (and every layer) is a continuous function. This effectively allows for information sharing between words. If two word's vectors are close together because they mostly predict the same future words, then that area of the embedding space is associated with predicting those words. If words a and b often occur as the word prior to some similar set of words $(w, x, y, \ldots)$ in the training set and word b also often occurs in the training set before word z, but (by chance) a never does, then this neural language model will predict that z is likely to occur after a. Where-as an n-gram language model would not. This is because a and b have similar embeddings, due to predicting a similar set of words. The model has learnt common features about these words implicitly from how they are used, and can use those to make better predictions. These features are stored in the embeddings which are looked up during the input.

1.1.1.2 Simplified Model considered with input and output embeddings

We can actually reinterpret the softmax output layer as also having embeddings. An alternative but equivalent diagram is shown in Figure 1.3.

The final layer of the neural trigram language model can be rewritten per each index corresponding to a possible next word (w^i):

$$\text{smax}(V\tilde{z}^{\text{hid}} + \tilde{k})_{w^i} = \frac{\exp\left(V_{w^i,:}\tilde{z}^{\text{hid}} + \tilde{k}_{w^i}\right)}{\sum_{\forall j}\exp\left(V_{j,:}\tilde{z}^{\text{hid}} + \tilde{k}_j\right)} \tag{1.5}$$

Consider, that the matrix product of a row vector with a column vector is the dot product $V_{w_i,:}\tilde{z}^{\text{hid}}$ can be seen as computing the dot product between the output embedding for w_i and the hidden layer representation of the prior words/context (w^{i-1} and w^{i-2} in this case)

in the form of $\tilde{z}^{\text{hid}}$. This leads to an alternate interpretation of the whole process as minimising the dot-product distance between the output embedding and the context representation This is particularly relevant for the skip-gram model discussed in Section 1.2.2 (with just one input word and no hidden layer).

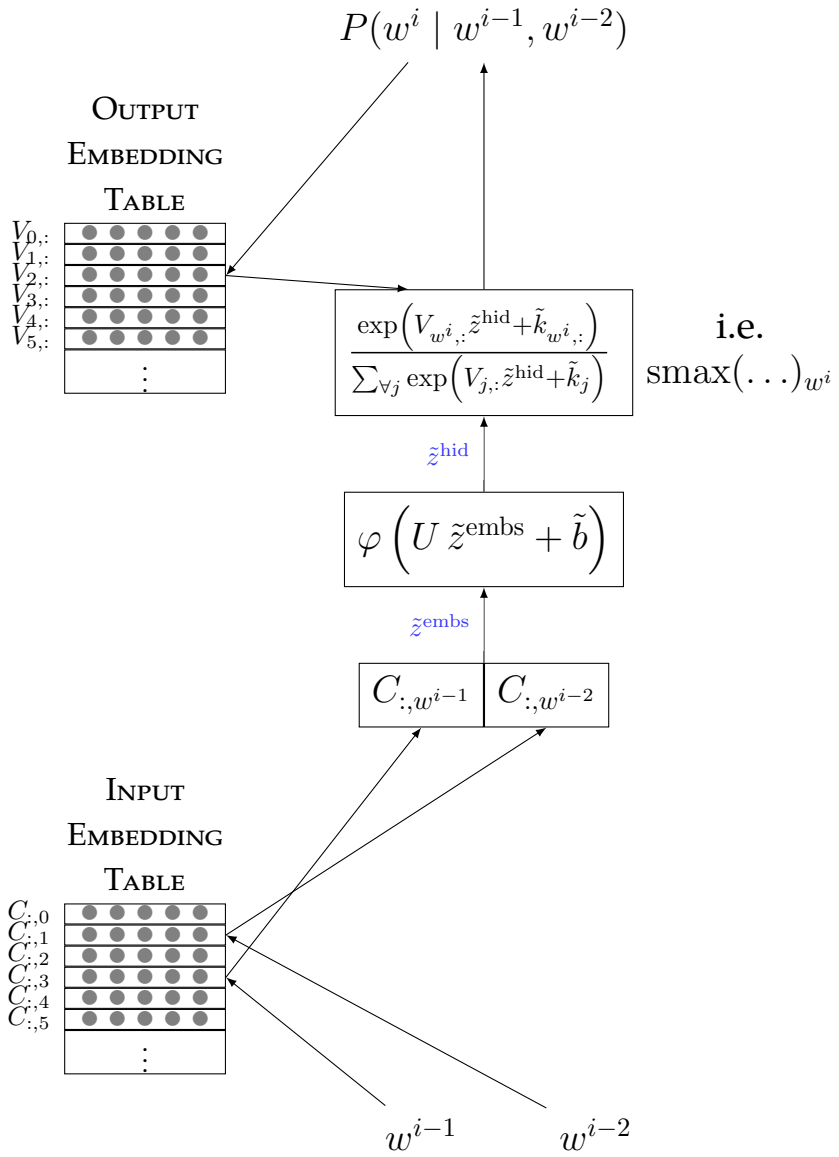

Figure 1.3: Neural Trigram Language Model as considered with output embeddings. This is mathematically identical to Figure 1.2

In this formulation, we have $V_{w_i,:}$ as the output embedding for w^i. As we considered $C_{:,w_i}$ as its input embedding.

1.1.1.3 Bayes-like Reformulation

When we consider the model with output embeddings, it is natural to also consider it under the light of the Bayes-like reformulation from **??**:

$$P(Y{=}i \mid Z{=}\tilde{z}) = \frac{R(Z{=}\tilde{z} \mid Y{=}i)\, R(Y{=}i)}{\sum_{\forall j} R(Z{=}\tilde{z} \mid Y{=}j)\, R(Y{=}j)} \quad (1.6)$$

which in this case is:

$$P(w^i \mid w^{i-1}, w^{i-2}) = \frac{R(Z{=}\tilde{z}^{\text{hid}} \mid W^i{=}w^i)\, R(W^i{=}w^i)}{\sum_{\forall v \in \mathbb{V}} R(Z = \tilde{z}^{\text{hid}} \mid W^i{=}v)\, R(W^i{=}v)} \quad (1.7)$$

where $\sum_{\forall v \in \mathbb{V}}$ is summing over every possible word v from the vocabulary $\mathbb{V}$, which does include the case $v = w^i$.

Notice the term:

$$\frac{R(W^i{=}w^i)}{\sum_{\forall v \in \mathbb{V}} R(W^i{=}v)} = \frac{\exp\left(\tilde{k}_{w^i}\right)}{\sum_{\forall v \in \mathbb{V}} \exp\left(\tilde{k}_v\right)} \quad (1.8)$$

$$= \frac{1}{\sum_{\forall v \in \mathbb{V}} \exp\left(\tilde{k}_v - \tilde{k}_{w^i}\right)} \quad (1.9)$$

The term $R(W^i{=}w^i) = \exp\left(\tilde{k}_{w^i}\right)$ is linked to the unigram word probabilities: $P(Y = y)$. If $\mathbb{E}(R(Z \mid W_i) = 1$ then the optimal value for $\tilde{k}$ would be given by the log unigram probabilities: $k_{w^i} = \log P(W^i{=}w^i)$. This condition is equivalent to if $\mathbb{E}(V\tilde{z}^{\text{hid}}) = 0$. Given that V is normally[1] initialized as a zero mean Gaussian, this

[1] no pun intended

condition is at least initially true. This suggests, interestingly, that we can predetermine good initial values for the output bias $\tilde{k}$ using the log of the unigram probabilities. In practice this is not required, as it is learnt rapidly by the network during training.

1.1.1.4 The Neural Probabilistic Language Model

NPLM, NPLM

NPLM derived a more advanced version of the neural language model discussed above. Rather than being a trigram language model, it is an n-gram language model, where n is a hyper-parameter of the model. The knowledge sharing allows the data-sparsity issues to be ameliorated, thus allowing for a larger n than in traditional n-gram language models. **NPLM** investigated values for 2, 4 and 5 prior words (i.e. a trigram, 5-gram and 6-gram model). The network used in their work was marginally more complex than the trigram neural language model. As shown in Figure 1.4, it features a layer-bypass connection. For n prior words, the model is described by:

$$\begin{aligned} P(w^i \mid w^{i-1}, \ldots, w^{i-n}) = \operatorname{smax}(& \\ & + V \varphi \left(U^{\text{hid}} \left[C_{:,w^{i-1}}; \ldots; C_{:,w^{i-n}} \right] + \tilde{b} \right) \\ & + U^{\text{bypass}} \left[C_{:,w^{i-1}}; \ldots; C_{:,w^{i-n}} \right] \\ & + \tilde{k})_{w^i} \end{aligned} \tag{1.10}$$

Input vocabulary and output vocabulary do not have to be the same

schwenk2004efficient suggests using only a subset of the vocabulary as options for the output, while allowing the full vocabulary in the input space – with a fall-back to classical language models for the missed words. This decreases the size of the softmax output layer, which substantially decreases the time taken to train or evaluate the network. As a speed-up technique this is now eclipsed by hierarchical softmax and negative sampling discussed in Section 1.4. The notion of a different input and output vocabulary though remains important for word-sense embeddings as will be discussed in **??**.

The layer-bypass is a connivance to aid in the learning. It allows the input to directly affect the output without being mediated by the shared hidden layer. This layer-bypass is an unusual feature, not present in future works deriving from this, such as **schwenk2004efficient**. Though in general it is not an unheard of technique in neural network machine learning.

schwenk2004efficient, **schwenk2004efficient**

This is the network which begins the notions of using neural networks with vector representations of words. Bengio et al. focused on the use of the of sliding window of previous words – much like the traditional

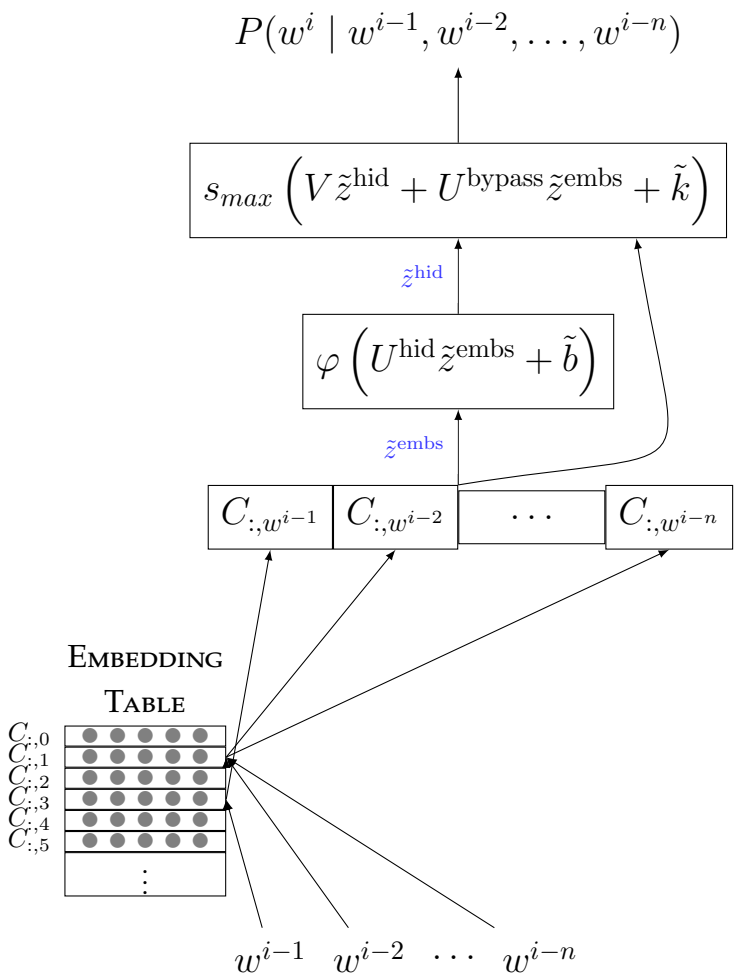

Figure 1.4: Neural Probabilistic Language Model

n-grams. At each time-step the window is advanced forward and the next is word predicted based on the shifted context of prior words. This is of-course exactly identical to extracting all n-grams from the corpus and using those as the training data. They very briefly mention that an RNN could be used in its place.

1.1.2 RNN Language Models

In **mikolov2010recurrent** an RNN is used for language modelling, as shown in Figure 1.5. Using the terminology of **??**, this is an encoder RNN, made using Basic Recurrent Units. Using an RNN eliminates the Markov assumption of a finite window of prior words forming the state. Instead, the state is learned, and stored in the state component of the RUs.

mikolov2010recurrent, **mikolov2010recurrent**

No Bias?

It should be noticed that Equations (1.11) and (1.12) are missing the bias terms. This is not commented on in **mikolov2010recurrent**. But in the corresponding chapter of Mikolov's thesis (**mikolov2012thesis**), it is explicitly noted that biases were not used in the network as it was not found that they gave a significant improvement to the result. This is perhaps surprising, particularly in the output softmax layer given the very unbalanced class (unigram) frequencies.
In the papers for several of Mikolov's other works, including those for skip-gram and CBOW discussed in Section 1.2, the bias terms are also excluded. We have matched those equations here. We do note though, that it is likely that many publicly available implementations of these algorithms would include the bias term: due either to a less close reading of the papers, or to the assumption that the equations are given in *design matrix* form: where the bias is not treated as a separate term to the weights, and the input is padded with an extra 1. We do not think this is at all problematic.
We discuss this further for the case of hierarchical softmax in Section 1.4.1, where the level is a proxy for the unigram frequency – and thus for the bias.

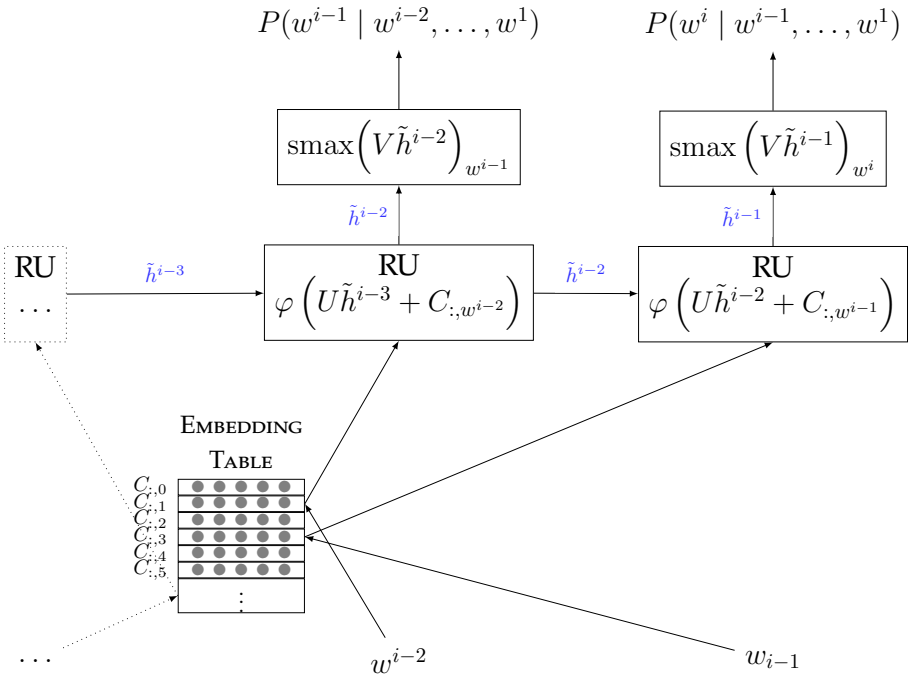

Figure 1.5: RNN Language Model. The RU equation

This state $\tilde{h}_i$ being the hidden state (and output as this is a basic RU) from the i time-step. The ith time-step takes as its input the ith word. As usual this hidden layer was an input to the hidden-layer at the next time-step, as well as to the output softmax.

$$\tilde{h}^i = \varphi\left(U\tilde{h}^{i-1} + C_{:,w_{i-1}}\right) \tag{1.11}$$

$$P(w^i \mid w^{i-1}, \ldots w^1) = \text{smax}\left(V\tilde{h}^{i-1}\right)_{w^i} \tag{1.12}$$

Rather than using a basic RU, a more advanced RNN such as a LSTM or GRU-based network can be used. This was done by **sundermeyer2012lstm** and **jozefowicz2015e** both of whom found that the more advanced networks gave significantly better results.

1.2 Acausal Language Modeling

The step beyond a normal language model, which uses the prior words to predict the next word, is what we will term acausal language modelling. Here we use

the word acausal in the signal processing sense. It is also sometimes called contextual language modelling, as the whole context is used, not just the prior context. The task here is to predict a missing word, using the words that precede it, as well as the words that come after it.

As it is acausal it cannot be implemented in a real-time system, and for many tasks this renders it less, directly, useful than a normal language model. However, it is very useful as a task to learn a good representation for words.

The several of the works discussed in this section also feature hierarchical softmax and negative sampling methods as alternative output methods. As these are complicated and easily misunderstood topics they are discussed in a more tutorial fashion in Section 1.4. This section will focus just on the language model logic; and assume the output is a normal softmax layer.

shown is the basic RU used in **mikolov2010recurrent**. It can be substituted for a LSTM RU or an GRU as was done in **sundermeyer2012lstm**; **jozefowicz2015empirical**, with appropriate changes.

sundermeyer2012lstm, **sundermeyer2012lstm**

jozefowicz2015empirical, **jozefowicz2015empirical**

Are CBOW & Skip-Gram Neural Networks?
It is sometimes asserted that these models are not in-fact neural networks at all. This assertion is often based on their lack of a traditional hidden-layer, and similarities in form to several other mathematical models (discussed in Section 1.3). This distinction is purely academic though. Any toolkit that can handle the prior discussed neural network models can be used to implement CBOW and Skip-Gram, more simply than using a non-neural network focused optimiser.
It also should be noted that embedding lookup is functionally an unusual hidden layer – this becomes obvious when considering the lookup as an one-hot product. Though it does lack a non-linear activation function.

mikolov2013efficient, **mikolov2013efficient**

1.2.1 Continuous Bag of Words

The continuous bag of words (CBOW) method was introduced by **mikolov2013efficient**. In truth, this is not particularly similar to bag of words at all. No more so than any other word representation that does not have regard for order of the context words (e.g. skip-gram, and GloVe).

The CBOW model takes as its input a context window surrounding a central skipped word, and tries to predict the word that it skipped over. It is very similar to earlier discussed neural language models, except that the window is on both sides. It also does not have any non-linearities; and the only hidden layer is the embedding layer.

For a context window of width n words – i.e. $\frac{n}{2}$ words to either side, of the target word w^i, the CBOW model

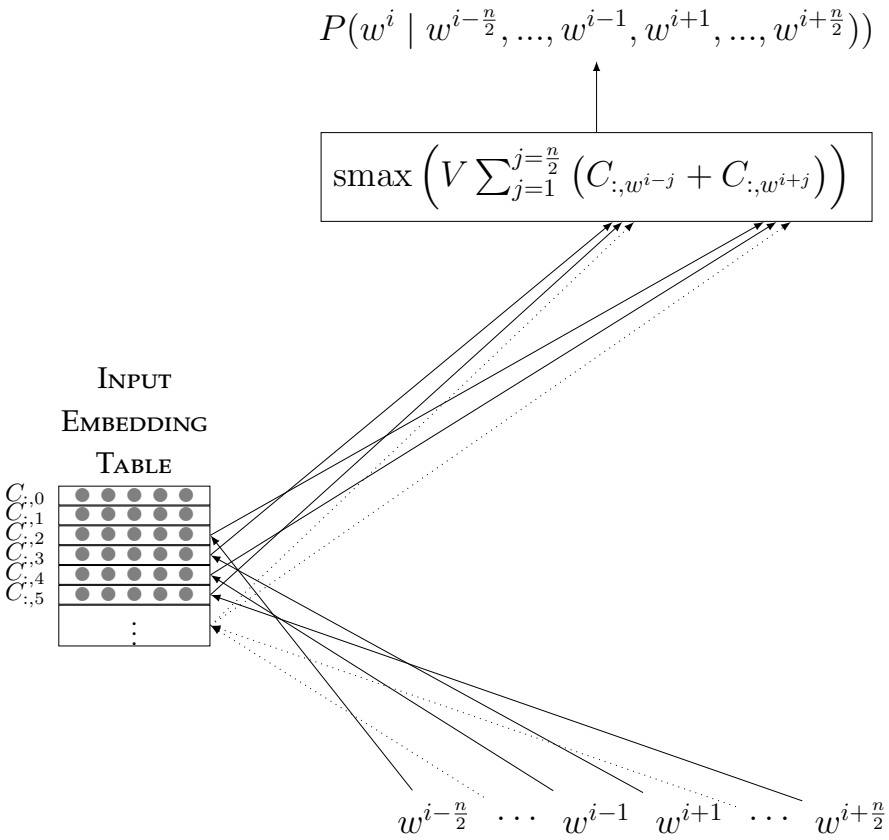

Figure 1.6: CBOW Language Model

is defined by:

$$P(w^i \mid w^{i-\frac{n}{2}}, \ldots, w^{i-1}, w^{i+1}, \ldots, w^{i+\frac{n}{2}}) = \text{smax}\left(V \sum_{j=i+1}^{j=\frac{n}{2}} \left(C_{:,w^{i-j}} + C_{:,w^{i+j}}\right)\right)_{w^i} \quad (1.13)$$

This is shown in diagrammatic form in Figure 1.6. By optimising across a training dataset, useful word embeddings are found, just like in the normal language model approaches.

1.2.2 Skip-gram

Skip-gram naming
In different publications this model may be called skipgram, skip-gram, skip-ngram, skip gram etc. Further, it may be called `word2vec` after the publicly released implementation of the algorithm. Though the `word2vec` software can

The converse of CBOW is the skip-grams model **mikolov2013e**. In this model, the central word is used to predict the words in the context.

The model itself is single word input, and its output is a softmax for the probability of each word in the vocabulary occurring in the context of the input word.

also be used for CBOW, so sometimes it can refer to CBOW.

mikolov2013efficient, **mikolov2013efficient**

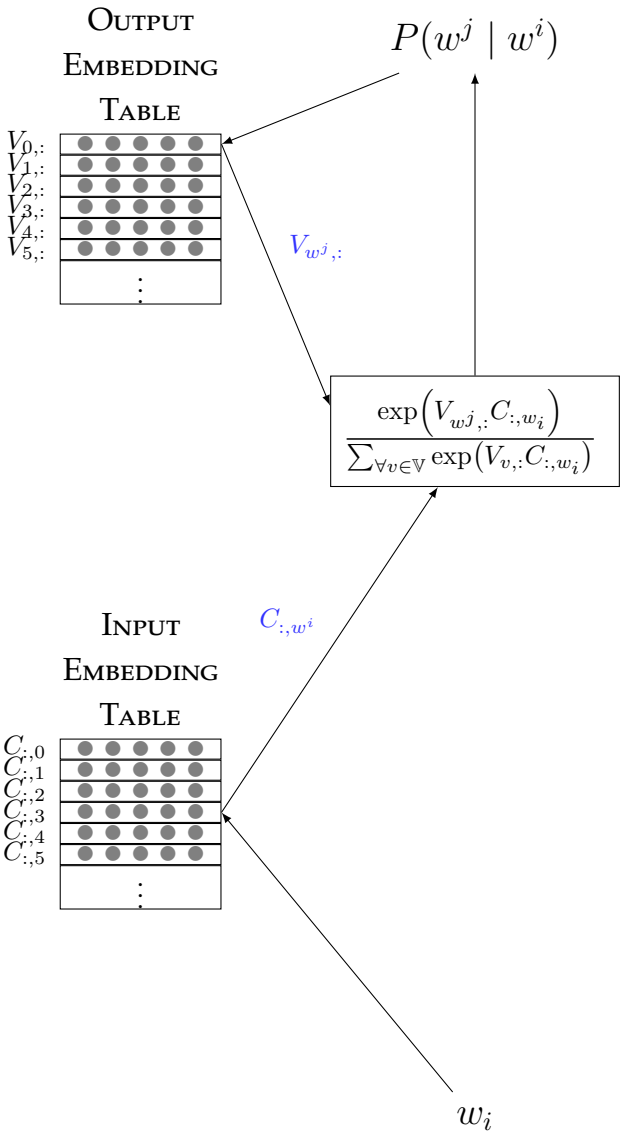

Figure 1.7: Skip-gram language Language Model. Note that the probability $P(w^j \mid w^i)$ is optimised during training for every w^j in a window around the central word w^i. Note that the final layer in this diagram is just a softmax layer, written in in output embedding form.

This can be indexed to get the individual probability of a given word occurring as usual for a language model. So for input word w^i the probability of w^j occurring in its context is given by:

$$P(w^j \mid w^i) = \text{smax}\left(V\, C_{:,w^i}\right)_{w^j} \tag{1.14}$$

The goal, is to maximise the probabilities of all the observed outputs that actually *do* occur in its context. This is done, as in CBOW by defining a window for the context of a word in the training corpus, $(i - \frac{n}{2}, \ldots, i - 1, i+i, \ldots, i+\frac{n}{2})$. It should be understood that while this is presented similarly to a classification task, there is no expectation that the model will actually predict the correct result, given that even during training there are multiple correct results. It is a regression to an accurate estimate of the probabilities of co-occurrence (this is true for probabilistic language models more generally, but is particularly obvious in the skip-gram case).

Note that in skip-gram, like CBOW, the only hidden layer is the embedding layer. Rewriting Equation (1.14) in output embedding form:

$$P(w^j \mid w^i) = \text{smax}\left(V\, C_{:,w^i}\right)_{w^j} \tag{1.15}$$

$$P(w^j \mid w^i) = \frac{\exp\left(V_{w^j,:} C_{:,w^i}\right)}{\sum_{\forall v \in \mathbb{V}} \exp\left(V_{v,:} C_{:,v}\right)} \tag{1.16}$$

The key term here is the product $V_{w^j,:}\, C_{:,w^i}$. The remainder of Equation (1.16) is to normalise this into a probability. Maximising the probability $P(w^j \mid w^i)$ is equivalent to maximising the dot produce between $V_{w^j,:}$, the output embedding for w^j and $C_{:,w^i}$ the input embedding for w^i. This is to say that the skip-gram probability is maximised when the angular difference between the input embedding for a word, and the output embeddings for its co-occurring words is minimised. The dot-product is a measure of vector similarity – closely related ot the cosine similarity.

Skip-gram is much more commonly used than CBOW.

1.2.3 Analogy Tasks

One of the most notable features of word embeddings is their ability to be used to express analogies using linear algebra. These tasks are keyed around answering the question: b is to a, as what is to c? For example, a semantic analogy would be answering that `Aunt` is to `Uncle` as `King` is to `Queen`. A syntactic analogy would be answering that `King` is to `Kings` as `Queen` is to `Queens`. The latest and largest analogy test set is presented by **gladkova2016analogy**, which evaluates embeddings on 40 subcategories of knowledge. Analogy completion is not a practical task, but rather serves to illustrate the kinds of information being captured, and the way in which it is represented (in this case linearly).

Analogy Tasks uncover prejudice in corpora

bolukbasi2016man and **Caliskan183** use analogy tasks, and related variant formulations to find troubling associations between words, such as Bolukbasi et. al's titular `Man` is to `Computer Programmer`, as `Woman` is to `Homemaker`. Finding these relationships in the embedding space, indicated that they are present in the training corpus, which in turn shows their prevalence in society at large. It has been observed that machine learning can be a very good mirror upon society.

The analogies work by relating similarities of differences between the word vectors. When evaluating word similarity using using word embeddings a number of measures can be employed. By far the cosine similarity is the most common. This is given by

$$\text{sim}(\tilde{u}, \tilde{v}) = \frac{\tilde{u} \cdot \tilde{v}}{\|\tilde{u}\| \, \|\tilde{v}\|} \tag{1.17}$$

This value becomes higher the closer the word embedding $\tilde{u}$ and $\tilde{v}$ are to each other, ignoring vector magnitude. For word embeddings that are working well, then words with closer embeddings should have correspondingly greater similarity. This similarity could be syntactic, semantic or other. The analogy tasks can help identify what kinds of similarities the embeddings are capturing.

Using the similarity scores, a ranking of words to complete the analogy is found. To find the correct word for d in: d is to c as b is to a the following is computed using

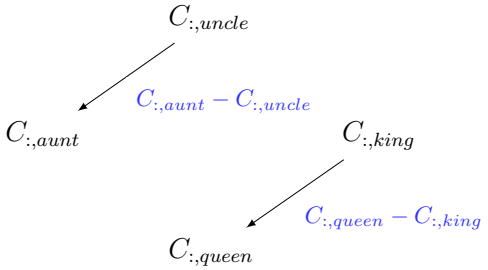

Figure 1.8: Example of analogy algebra

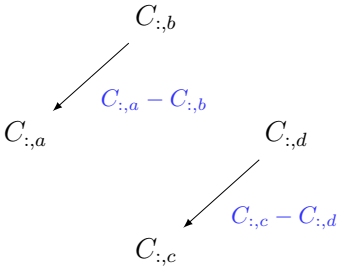

Figure 1.9: Vectors involved in analogy ranking tasks, this may help to understand the math in Equation (1.19)

gladkova2016analogy, gladkova2016analogy

the table of embeddings C over the vocabulary $\mathbb{V}$:

$$\underset{\forall d \in \mathbb{V}}{\text{argmax}}\, \text{sim}(C_{:,d} - C_{:,c}, C_{:,a} - C_{:,b}) \tag{1.18}$$

$$\text{i.e}\, \underset{\forall d \in \mathbb{V}}{\text{argmax}}\, \text{sim}(C_{:,d},\ C_{:,a} - C_{:,b} + C_{:,c}) \tag{1.19}$$

This is shown diagrammaticality in Figures 1.8 and 1.9. Sets of embeddings where the vector displacement between analogy terms are more consistent score better.

Initial results in **mikolov2013linguisticsubstructures** were relatively poor, but the surprising finding was that this worked at all. **mikolov2013efficient** found that CBOW performed poorly for semantic tasks, but comparatively well for syntactic tasks; skip-gram performed comparatively well for both, though not quite as good in the syntactic tasks as CBOW. Subsequent results found in **pennington2014glove** were significantly better again for both.

pennington2014glove, pennington2014glove

1.3 Co-location Factorisation

1.3.1 GloVe

Distance weighted co-occurrence and dynamic window sizing
When training skip-gram and CBOW, Mikolov et al. used dynamic window sizing. This meant that if the specified window size was n, in any given training case being considered the actual window size was determined as a random number between 0 and n. Pennington et al. achieve a similar effect by weighting co-occurrences within a window with inverse proportion to the distance between the word. That is to say if w^i and w^j occur in the

Skip-gram, like all probabilistic language models, is a intrinsically prediction-based method. It is effectively optimising a neutral network to predict which words will co-occur in the with in the range of given by the context window width. That optimisation is carried out per-context window, that is to say the network is updated based on the local co-occurrences. In **pennington2014glove** the authors show that if one were to change that optimisation to be global over all co-occurrences, then the optimisation criteria becomes minimising the cross-entropy between the true co-occurrence probabilities, and the value of the embedding product, with the cross entropy measure being weighted by the frequency of the occurrence of the word. That is to say if skip-gram were optimised globally it would be

equivalent to minimising:

$$Loss = -\sum_{\forall w^i \in \mathbb{V}} \sum_{\forall w^j \in \mathbb{V}} X_{w^i,w^j} P(w^j \mid w^i) \log(V_{w^j,:} C_{:,w^i}) \tag{1.20}$$

for $\mathbb{V}$ being the vocabulary and for X being the a matrix of the true co-occurrence counts, (such that X_{w^i,w^j} is the number of times words w^i and w^j co-occur), and for P being the predicted probability output by the skip-gram.

Minimising this cross-entropy efficiently means factorising the true co-occurrence matrix X, into the input and output embedding matrices C and V, under a particular set of weightings given by the cross entropy measure.

pennington2014glove propose an approach based on this idea. For each word co-occurrence of w^i and w^j in the vocabulary: they attempt to find optimal values for the embedding tables C, V and the per word biases $\tilde{b}$, $\tilde{k}$ such that the function $s(w^i, w^j)$ (below) expresses an approximate log-likelihood of w^i and w^j.

$$\text{optimise} \quad s(w^i, w^j) = V_{w^j,:} C_{:,w^i} + \tilde{b}_{w^i} + \tilde{k}_{w^j} \tag{1.21}$$

$$\text{such that} \quad s(w^i, w^j) \approx \log(X_{w^i,w^j}) \tag{1.22}$$

This is done via the minimisation of

$$Loss = -\sum_{\forall w^i} \sum_{\forall w^j} f(X_{w^i,w^j}) \left(s(w^i, w^j) - \log(X_{w^i,w^j})\right) \tag{1.23}$$

Where $f(x)$ is a weighing between 0 and 1 given by:

$$f(x) = \begin{cases} \left(\frac{x}{100}\right)^{0.75} & x < 100 \\ 1 & \text{otherwise} \end{cases} \tag{1.24}$$

This can be considered as a saturating variant of the effective weighing of skip-gram being X_{w^i,w^j}.

While GloVe out-performed skip-gram in initial tests subsequent more extensive testing in **levy2015lsaisbasicallyskipgramswithexperiments** with more tuned parameters, found that skip-gram marginally out-performed GloVe on all tasks.

same window (i.e. $|i-j|<n$), then rather than contributing 1 to the entry in the co-occurrence count X_{w^i,w^j}, they contribute $\frac{1}{|i-j|}$.

Subsampling, and weight saturation

Skip-gram and CBOW models use a method called subsampling to decrease the effect of common words. The subsampling method is to randomly discard words from training windows based on their unigram frequency. This is closely related to the saturation of the co-occurrence weights as calculated by $f(X)$ used by GloVe. Averaged over all training cases the effect is nearly the same.

Key Factors Mentioned as Asides

There is an interesting pattern of factors being considered as not part of the core algorithm. We have continued this in the side-notes of this section; with the preceding notes on Distance weighting and subsampling. While the original papers consider these as unimportant to the main thrust of the algorithms **levy2015lsaisbasicallyskipgramswith** found them to be crucial hyper-parameters.

pennington2014glove, **pennington2014glove**

levy2015lsaisbasicallyskipgramswith
levy2015lsaisbasicallyskipgramswith

1.3.2 Further equivalence of Co-location Prediction to Factorisation

Implementing GloVe
To implement GloVe in any technical programming language with good support for optimisation is quiet easy, as it is formed into a pure optimization problem. It is also easy to do in a neural network framework, as these always include an optimiser. Though unlike in normal neural network training there are no discrete training cases, just the global co-occurrence statistics.

levy2014neural, **levy2014neural**

li2015wordemedingasEMF, **li2015wordemedingasEMF**

cotterell2017SkipgramisEPCA, **cotterell2017SkipgramisEPCA**

wordvecispca, **wordvecispca**

GloVe highlights the relationship between the co-located word prediction neural network models, and the more traditional non-negative matrix factorization of co-location counts used in topic modeling. Very similar properties were also explored for skip-grams with negative sampling in **levy2014neural** and in **li2015wordemedingasEMF** with more direct mathematical equivalence to weighed co-occurrence matrix factorisation; Later, **cotterell2017Skipgra** showed the equivalence to exponential principal component analysis (PCA). **wordvecispca** goes on to extend this to show that it is a weighted logistic PCA, which is a special case of the exponential PCA. Many works exist in this area now.

1.3.3 Conclusion

We have now concluded that neural predictive co-location models are functionally very similar to matrix factorisation of co-location counts with suitable weightings, and suitable similarity metrics. One might now suggest a variety of word embeddings to be created from a variety of different matrix factorisations with different weightings and constraints. Traditionally large matrix factorisations have significant problems in terms of computational time and memory usage. A common solution to this, in applied mathematics, is to handle the factorisation using an iterative optimisation procedure. Training a neural network, such as skip-gram, is indeed just such an iterative optimisation procedure.

1.4 Hierarchical Softmax and Negative Sampling

Hierarchical softmax, and negative sampling are effectively alternative output layers which are computationally cheaper to evaluate than regular softmax. They are powerful methods which pragmatically allow for large speed-up in any task which involves outputting very large classification probabilities – such as language modelling.

1.4.1 Hierarchical Softmax

Hierarchical softmax was first presented in **morin2005hierarchical**. Its recent use was popularised by **mikolov2013efficient**, where words are placed as leaves in a Huffman tree, with their depth determined by their frequency.

morin2005hierarchical, **morin2005hierarchical**

mikolov2013efficient, **mikolov2013efficient**

One of the most expensive parts of training and using a neural language model is to calculate the final softmax layer output. This is because the softmax denominator includes terms for each word in the vocabulary. Even if only one word's probability is to be calculated, one denominator term per word in the vocabulary must be evaluated. In hierarchical softmax, each word (output choice), is considered as a leaf on a binary tree. Each level of the tree roughly halves the space of the output words to be considered. The final level to be evaluated for a given word contains the word's leaf-node and another branch, which may be a leaf-node for another word, or a deeper sub-tree

The tree is normally a Huffman tree (**huffman1952method**) as was found to be effective by **mikolov2013efficient**. This means that for each word w^i, the word's depth (i.e its code's length) $l(w^i)$ is such that over all words: $\sum_{\forall w^j \in \mathbb{V}} P(w^j) \times l(w^j)$ is minimised. Where $P(w^i)$ is word w^i's unigram probability, and $\mathbb{V}$ is the vocabulary. The approximate solution to this is that $l(w^i) \approx$

SemHuff
It can be noted that the Huffman encoding scheme specifies only the depth of a given word in the tree. It does not specify the order. **SemHuff** make use of the BlossomV algorithm (**Kolmogorov2009**) to pair the nodes on each layer according to their similarity. They found that on the language modelling task this improved performance, in the way one would expect. They used a lexical resource to determine similarity, however noted that a prior trained word-embedding model could be used to define similarity instead – the new encoding can then be used to define a new model which will find new (hopefully better) embeddings. This is similar

to the original method used by (**morin2005hierarchical**), but only using the similarity measure for reordering nodes at the same depth, after the depth is decided by Huffman encoding. In our own experimentation, when applying it to other tasks, we did not see large improvements. It is nevertheless a very interesting idea, and quite fun to implement and observe the results.

huffman1952method, **huffman1952method**

$-\log_2(P(w^i))$. From the tree, each word can be assign a code in the usual way, with 0 for example representing taking one branch, and 1 representing the other. Each point in the code corresponds to a node in the binary tree, which has decision tied to it. This code is used to transform the large multinomial softmax classification into a series of binary logistic classifications. It is important to understand that the layers in the tree are not layers of the neural network in the normal sense – the layers of the tree do not have an output that is used as the input to another. The layers of the tree are rather subsets of the neurons on the output layer, with a relationship imparted on them.

It was noted by **mikolov2013efficient**, that for vocabulary $\mathbb{V}$:

- Using normal softmax would require each evaluation to perform $|\mathbb{V}|$ operations.
- Using hierarchical softmax with a balanced tree, would mean the expected number of operations across all words would be $\log_2(|\mathbb{V}|)$.
- Using a Huffman tree gives the expected number of operations $\sum_{\forall w^j \in \mathbb{V}} -P(w^j)\log_2(P(w^i)) = H(\mathbb{V})$, where $H(\mathbb{V})$ is the unigram entropy of words in the training corpus.

The worse case value for the entropy is $\log_2(|\mathbb{V}|)$. In-fact Huffman encoding is provably optimal in this way. As such this is the minimal number of operations required in the average case.

1.4.1.1 An incredibly gentle introduction to hierarchical softmax

In this section, for brevity, we will ignore the bias component of each decision at each node. It can either be handled nearly identically to the weight; or the matrix can be written in *design matrix form* with an implicitly ap-

pended column of ones; or it can even be ignored in the implementation (as was done in **mikolov2013efficient**). The reasoning for being able to ignore it is that the bias in normal softmax encodes unigram probability information; in hierarchical softmax, when used with the common Huffman encoding, its the tree's depth in tree encodes its unigram probability. In this case, not using a bias would at most cause an error proportionate to 2^{-k}, where k is the smallest integer such that $2^{-k} > P(w^i)$.

1.4.1.1.1 First consider a binary tree with just 1 layer and 2 leaves The leaves are n^{00} and n^{01}, each of these leaf nodes corresponds to a word from the vocabulary, which has size two, for this toy example.

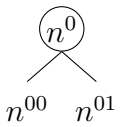

Figure 1.10: Tree for 2 words

From the initial root which we call n^0, we can go to either node n^{00} or node n^{01}, based on the input from the layer below which we will call $\tilde{z}$.

Here we write n^{01} to represent the event of the first non-root node being the branch given by following left branch, while n^{01} being to follow the right branch. (The order within the same level is arbitrary in any-case, but for our visualisation purposes we'll used this convention.)

We are naming the root node as a notation convenience so we can talk about the decision made at n^0. Note that $P(n^0) = 1$, as all words include the root-node on their path.

We wish to know the probability of the next node being the left node (i.e. $P(n^{00} \mid \tilde{z})$) or the right-node (i.e. $P(n^{01} \mid \tilde{z})$). As these are leaf nodes, the prediction either equivalent to the prediction of one or the other of the two words in our vocabulary.

We could represent the decision with a softmax with two outputs. However, since it is a binary decision, we do not need a softmax, we can just use a sigmoid.

$$P(n^{01} \mid \tilde{z}) = 1 - P(n^{00} \mid \tilde{z}) \tag{1.25}$$

The weight matrix for a sigmoid layer has a number of columns governed by the number of outputs. As there is only one output, it is just a row vector. We are going to index it out of a matrix V. For the notation, we will use index 0 as it is associated with the decision at node n^0. Thus we call it $V_{0,:}$.

$V_{0,:}\tilde{z}$ is a dot product We mentioned in the marginalia earlier, but just as an extra reminder: the matrix product of a row vector like $V_{0,:}$ with a (column) vector like $\tilde{z}$ is their vector dot product.

$$P(n^{00} \mid \tilde{z}) = \sigma(V_{0,:}\tilde{z}) \tag{1.26}$$
$$P(n^{01} \mid \tilde{z}) = 1 - \sigma(V_{0,:}\tilde{z}) \tag{1.27}$$

Note that for the sigmoid function: $1 - \sigma(x) = \sigma(-x)$. Allowing the formulation to be written:

$$P(n^{01} \mid \tilde{z}) = \sigma(-V_{0,:}\tilde{z}) \tag{1.28}$$

thus

$$P(n^{0i} \mid \tilde{z}) = \sigma((-1)^i V_{0,:}\tilde{z}) \tag{1.29}$$

Noting that in Equation (1.29), i is either 0 (with $-1^0 = 1$) or 1 (with $-1^1 = -1$)).

1.4.1.1.2 Now consider 2 layers with 3 leaves

Consider a tree with nodes: n^0, n^{00},n^{000}, n^{001}, n^{01}. The leaves are n^{000}, n^{001}, and n^{01}, each of which represents one of the 3 words from the vocabulary.

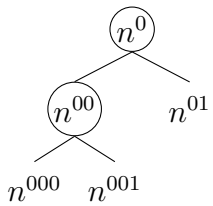

Figure 1.11: Tree for 3 words

From earlier we still have:

$$P(n^{00} \mid \tilde{z}) = \sigma(V_{0,:}\tilde{z}) \tag{1.30}$$
$$P(n^{01} \mid \tilde{z}) = \sigma(-V_{0,:}\tilde{z}) \tag{1.31}$$

We must now to calculate $P(n^{000} \mid \tilde{z})$. Another binary decision must be made at node n^{00}. The decision at n^{00} is to find out if the predicted next node is n^{000} or n^{001}. This decision is made, with the assumption that we have reached n^{00} already.

So the decision is defined by $P(n^{000} \mid z,\ n^{00})$ is given by:

$$P(n^{000} \mid \tilde{z}) = P(n^{000} \mid \tilde{z},\ n^{00})\, P(n^{00} \mid \tilde{z}) \tag{1.32}$$
$$P(n^{000} \mid \tilde{z}, n^{00}) = \sigma(V_{00,:}\tilde{z}) \tag{1.33}$$
$$P(n^{001} \mid \tilde{z}, n^{00}) = \sigma(-V_{00,:}\tilde{z}) \tag{1.34}$$

We can use the conditional probability chain rule to recombine to compute the three leaf nodes final probabilities.

$$P(n^{01} \mid \tilde{z}) = \sigma(-V_{0,:}\tilde{z}) \tag{1.35}$$
$$P(n^{000} \mid \tilde{z}) = \sigma(V_{00,:}\tilde{z})\sigma(V_{0,:}\tilde{z}) \tag{1.36}$$
$$P(n^{001} \mid \tilde{z}) = \sigma(-V_{00,:}\tilde{z})\sigma(V_{0,:}\tilde{z}) \tag{1.37}$$

1.4.1.1.3 Continuing this logic Using this system, we know that for a node encoded at position $[0t^1t^2t^3 \dots t^L]$, e.g. $[010 \dots 1]$, its probability can be found recursively as:

$$P(n^{0t^1 \dots t^L} \mid \tilde{z}) = P(n^{0t^1 \dots t^L} \mid \tilde{z},\ n^{0t^1 \dots t^{L-1}})\, P(n^{0t^1 \dots t^{L-1}} \mid \tilde{z}) \tag{1.38}$$

Thus:

$$P(n^{0t^1} \mid \tilde{z}) = \sigma\left((-1)^{t^1} V_{0,:}\tilde{z}\right) \tag{1.39}$$
$$P(n^{0t^1,t^2} \mid \tilde{z},\ n^{0t^1}) = \sigma\left((-1)^{t^2} V_{0t^1,:}\tilde{z}\right) \tag{1.40}$$
$$P(n^{0t^1 \dots t^i} \mid \tilde{z},\ n^{0t^1 \dots t^{i-1}}) = \sigma\left((-1)^{t^i} V_{0t^1 \dots t^{i-1},:}\tilde{z}\right) \tag{1.41}$$

The conditional probability chain rule, is applied to get:

$$P(n^{0t^1 \dots t^L} \mid \tilde{z}) = \prod_{i=1}^{i=L} \sigma\left((-1)^{t^i} V_{0t^1 \dots t^{i-1},:}\tilde{z}\right) \tag{1.42}$$

1.4.1.2 Formulation

The formulation above is not the same as in other works. This subsection shows the final steps to reach the conventional form used in **mikolovSkip**.

Here we have determined that the 0th/left branch represents the positive choice, and the other probability is defined in terms of this. It is equivalent to have the 1th/right branch representing the positive choice:

$$P(n^{0t^1\ldots t^L} \mid \tilde{z}) = \prod_{i=1}^{i=L} \sigma\left((-1)^{t^i+1} V_{0t^1\ldots t^{i-1},:}\tilde{z}\right) \tag{1.43}$$

or to allow it to vary per node: as in the formulation of **mikolovSkip**. In that work they use $ch(n)$ to represent an arbitrary child node of the node n and use an indicator function $[\![a=b]\!] = \begin{cases} 1 & a=b \\ -1 & a\neq b \end{cases}$ such that they can write $\left[\!\left[n^b = ch(n^a)\right]\!\right]$ which will be 1 if n^a is an arbitrary (but consistent) child of n^b, and 0 otherwise.

$$P(n^{0t^1\ldots t^L} \mid \tilde{z}) = \prod_{i=1}^{i=L} \sigma\left(\left[\!\left[n^{0t^1\ldots t^i} = ch(n^{0t^1\ldots t^{i-1}})\right]\!\right] V_{0t^1\ldots t^{i-1},:}\tilde{z}\right) \tag{1.44}$$

There is no functional difference between the three formulations. Though the final one is perhaps a key reason for the difficulties in understanding the hierarchical softmax algorithm.

1.4.1.3 Loss Function

Using normal softmax, during the training, the cross-entropy between the model's predictions and the ground

Combining multiplications
If one wants to find both $V_{00,:}\tilde{z}$ and $V_{0,:}\tilde{z}$, then this can be done using matrices simultaneously, thus potentially taking advantage of optimized matrix multiplication routines.
$\begin{bmatrix} V_{0,:} \\ V_{00,:} \end{bmatrix} z = \begin{bmatrix} V_{0,:}\tilde{z} \\ V_{00,:}\tilde{z} \end{bmatrix}$
Thus the whole product for all of the decisions can be written as $V\tilde{z}$. The problem then becomes indexing the relevant node rows.
However computing every single decision is beyond what is required for most uses: hierarchical softmax lets us only compute the decisions that are on the path to the word-leaf we which we wish to query. Computing all of them is beyond what is required.
Packing tree node elements into a matrix with fast indexing is a non-trivial problem. The details on optimising such multiplications and tree packing are beyond the scope of this book.
In general there may be very little scope here for optimisation, as on most hardware (and BLAS systems) matrix products with n columns, takes a similar amount of time to n vector dot products. As such storing the row vectors of V in a hashmap indexed by node-path, and looping over them as required may be more practical.
In languages/libraries with slow looping constructs (numpy, R, octave), where calling into suitable library routines is much faster, this may give some speed-up;

truth as given in the training set is minimised. Cross entropy is given by

$$CE(P^\star, P) = \sum_{\forall w^i \in \mathbb{V}} \sum_{\forall z^j \in \mathbb{Z}} -P^\star(w^i \mid z^j) \log P(w^i \mid z^j) \tag{1.45}$$

Where $P^\star$ is the true distribution, and P is the approximate distribution given by our model (in other sections we have abused notation to use P for both). $\mathbb{Z}$ is the set of values that are input into the model, (or equivalently the values derived from them from lower layers) – Ithe context words in language modelling. $\mathbb{V}$ is the set of outputs, the vocabulary in language modeling. The training dataset $\mathcal{X}$ consists of pairs from $\mathbb{V} \times \mathbb{Z}$.

The true probabilities (from $P^\star$) are implicitly given by the frequency of the training pairs in the training dataset $\mathcal{X}$.

$$Loss = CE(P^\star, P) = \frac{1}{|\mathcal{X}|} \sum_{\forall (w^i, z^i) \in \mathcal{X}} -\log P(w^i \mid z^i) \tag{1.46}$$

The intuitive understanding of this, is that we are maximising the probability estimate of all pairings which actually occur in the training set, proportionate to how often the occur. Note that the $\mathbb{Z}$ can be non-discrete values, as was the whole benefit of using embeddings, as discussed in Section 1.1.1.

This works identically for hierarchical softmax as for normal softmax. It is simply a matter of substituting in the (different) equations for P. Then applying back-propagation as usual.

1.4.2 Negative Sampling

Negative sampling was introduced in **mikolovSkip** as another method to speed up this problem. Much like

but even here it is likely to be minor. The time may be better spent writing a C extension library to do this part of the program. Or learning to use a language with fast for loops (e.g. Julia (**Julia**)).

mikolovSkip, **mikolovSkip**

How does this relate to word vectors?
After the length of this section, one may have forgotten why we are doing this in the first place. Recall that CBOW, skip-gram and all other language modelling based word embedding methods are based around predicting $P(w^o \mid w^i, \ldots, w^j)$ for some words. For skip-gram that is just $P(w^o \mid w^i)$. The term $n^{0t^1 \ldots t^L}$ in $P(n^{0t^1 \ldots t^L} \mid z)$, just represents as a path through the tree to the leaf node which represents the word w^o. i.e $P(n^{0t^1 \ldots t^L} \mid z) = P(w^o \mid z)$. The output of the final hidden layer is z (i.e. the z is the input to the output layer) In normal language models z encodes all the information about what the model knows of predictions based on $w^i \ldots, w^j$. z is thus a proxy term in the conditional probability for those words. In skip-gram there is no hidden layer, and it is just $z = C_{:,w^i}$ proxying only for w_i, and the model defines its probability output by $P(w^o \mid w^i) = P(w^o \mid C_{:,w^i})$.

The gradient calculations
They are not fun. They never are for back-propagation. We recommend using a framework with automated

differentiation, and/or performing gradient checks against a numerical differentiation tool (simple finite-differencing will do in a pinch).

mikolovSkip, **mikolovSkip**

gutmann2012noise, **gutmann2012noise**

hierarchical softmax in its purpose. However, negative sampling does not modify the network's output, but rather the loss function.

Negative Sampling is a simplification of Noise Contrast Estimation (**gutmann2012noise**). Unlike Noise Contrast Estimation (and unlike softmax), it does not in fact result in the model converging to the same output as if it were trained with softmax and cross-entropy loss. However the goal with these word embeddings is not to actually perform the language modelling task, but only to capture a high-quality vector representation of the words involved.

1.4.2.1 A Motivation of Negative Sampling

Recall from Section 1.2.2 that the (supposed) goal, is to estimate $P(w^j \mid w^i)$. In truth, the goal is just to get a good representation, but that is achieved via optimising the model to predict the words. In Section 1.2.2 we considered the representation of $P(w^j \mid w^i)$ as the w^jth element of the softmax output.

$$P(w^j \mid w^i) = \text{smax}(V\, C_{:,w^i})_{w^j} \tag{1.47}$$

$$P(w^j \mid w^i) = \frac{\exp\left(V_{w^j,:} C_{:,w^i}\right)}{\sum_{k=1}^{k=N} \exp\left(V_{k,:} C_{:,k}\right)} \tag{1.48}$$

Why is not using softmax wrong?
The notation abuse may be hiding just how bad it is to not use softmax. Recall that the true meaning of $P(w^j \mid w^i)$ is actually $P(W^j{=}w^j \mid W^i{=}w^i)$. By not using softmax, with its normalising denominator this means that: $\sum_{\forall w^j \in \mathbb{V}} P(w^j \mid w^i) \neq 1$ (except by coincidence).

This is not the only valid representation. One could use a sigmoid neuron for a direct answer to the co-location probability of w^j occurring near w^i. Though this would throw away the promise of the probability distribution to sum to one across all possible words that could be co-located with w^i. That promise could be enforced by other constraints during training, but in this case it will not be. It is a valid probability if one does not consider it as a single categorical prediction, but rather as independent predictions.

$$P(w^j \mid w^i) = \sigma(V\, C_{:,w^i})_{w^j} \tag{1.49}$$

$$\text{i.e.} \quad P(w^j \mid w^i) = \sigma(V_{w^j,:} C_{:,w^i}) \tag{1.50}$$

Lets start from the cross-entropy loss. In training word w^j does occur near w^i, we know this because they are a training pair presented from the training dataset $\mathcal{X}$. Therefore, since it occurs, we could make a loss function based on minimising the negative log-likelihood of all observations.

$$Loss = \sum_{\forall(w^i,w^j)\in\mathcal{X}} -\log P(w^j \mid w^i) \tag{1.51}$$

Loss Function
Readers may want to reread Section 1.4.1.3 to brush up on how we use the training dataset as a ground truth probability estimate implicitly when using cross-entropy loss. When doing so one should remember that that the conditioning term, z, for skip-grams is the co-located words as there is no hidden layer.

This is the cross-entropy loss, excluding the scaling factor for how often it occurs.

However, we are not using softmax in the model output, which means that there is no trade off for increasing (for example) $P(w^1 \mid w^i)$ vs $P(w^2 \mid w^i)$. This thus admits the trivially optimal solution $\forall w^j \in \mathbb{V}\ P(w^j \mid w^i) = 1$. This is obviously wrong – even beyond not being a proper distribution – some words are more commonly co-occurring than others.

So from this we can improve the statement. What is desired from the loss function is to reward models that predict the probability of words that *do* co-occur as being higher, than the probability of words that *do not*. We know that w^j does occur near w^i as it is in the training set. Now, let us select via some arbitrary means a w^k that does not – a negative sample. We want the loss function to be such that $P(w^k \mid w^i) < P(w^j \mid w^i)$. So for this single term in the loss we would have:

$$loss(w^j, w^i) = \log P(w^k \mid w^i) - \log P(w^j \mid w^i) \tag{1.52}$$

The question is then: how is the negative sample w^k to be found? One option would be to deterministically search the corpus for these negative samples, making sure to never select words that actually do co-occur. However that would require enumerating the entire

Most words do not co-occur
Some simple reasoning can account for this as a reasonable consequence of Zipf's law (**zipf1949human**) and a

prior of the principle of indifference, but there is a further depth to it as explained by **ha2009extending**.

corpus. We can instead just pick them randomly, we can sample from the unigram distribution. As statistically, in any given corpus most words do not co-occur, a randomly selected word in all likelihood will not be one that truly does co-occur – and if it is, then that small mistake will vanish as noise in the training, overcome by all the correct truly negative samples.

Is Equation (1.53) a function?
No, at the point at which the Loss started including randomly selected samples, it ceased to be a function in the usual mathematical sense. It is still a function in the common computer programming sense though – it is just not deterministic.

At this point, we can question, why limit ourselves to one negative sample? We could take many, and do several at a time, and get more confidence that $P(w^j \mid w^i)$ is indeed greater than other (non-existent) co-occurrence probabilities. This gives the improved loss function of

$$loss(w^j, w^i) = \left(\sum_{\forall w^k \in \text{samples}(D^{1g})} \log P(w^k \mid w^i) \right) - \log P(w^j \mid w^i) \tag{1.53}$$

where D^{1g} stands for the unigram distribution of the vocabulary and $\text{samples}(D^{1g})$ is a function that returns some number of samples from it.

Consider, though is this fair to the samples? We are taking them as representatives of all words that do not co-occur. Should a word that is unlikely to occur at all, *but was unlucky enough to be sampled*, contribute the same to the loss as a word that was very likely to occur? More reasonable is that the loss contribution should be in proportion to how likely the samples were to occur. Otherwise it will add unexpected changes and result in noisy training. Adding a weighting based on the unigram probability ($P^{1g}(w^k)$) gives:

$$loss(w^j, w^i) = \left(\sum_{\forall w^k \in \text{samples}(D^{1g})} P^{1g}(w^k) \log P(w^k \mid w^i) \right) - \log P(w^j \mid w^i) \tag{1.54}$$

The expected value is defined by

$$\mathop{\mathbb{E}}_{X\sim D}[f(x)] = \sum_{\forall x \text{ values for } X} P^{\mathrm{d}} f(x) \tag{1.55}$$

In an abuse of notation, we apply this to the samples, as a sample expected value and write:

$$\sum_{k=1}^{k=n} \mathop{\mathbb{E}}_{w^k\sim D^{\mathrm{1g}}}[\log P(w^k \mid w^i)] \tag{1.56}$$

to be the sum of the n samples expected values. This notation (abuse) is as used in **mikolovSkip**. It gives the form:

mikolovSkip, mikolovSkip

$$loss(w^j, w^i) = \left(\sum_{k=1}^{k=n} \mathop{\mathbb{E}}_{w^k\sim D^{\mathrm{1g}}}[\log P(w^k \mid w^i)]\right) - \log P(w^j \mid w^i) \tag{1.57}$$

Consider that the choice of unigram distribution for the negative samples is not the only choice. For example, we might wish to increase the relative occurrence of rare words in the negative samples, to help them fit better from limited training data. This is commonly done via subsampling in the positive samples (i.e. the training cases)). So we replace D^{1g} with D^{ns} being the distribution of negative samples from the vocabulary, to be specified as a hyper-parameter of training.

mikolovSkip uses a distribution such that

$$P^{\mathrm{D^{ns}}}(w^k) = \frac{P^{\mathrm{D^{1g}}}(w^k)^{\frac{2}{3}}}{\sum_{\forall w^o\in\mathbb{V}} P^{\mathrm{D^{1g}}}(w^o)^{\frac{2}{3}}} \tag{1.58}$$

which they find to give better performance than the unigram or uniform distributions.

Using this, and substituting in the sigmoid for the probabilities, this becomes:

$$loss(w^j, w^i) = \left(\sum_{k=1}^{k=n} \mathop{\mathbb{E}}_{w^k\sim D^{\mathrm{ns}}}[\log \sigma(V_{w^k,:} C_{:,w^i})\right) - \log \sigma(V_{w^j,:} C_{:,w^i}) \tag{1.59}$$

By adding a constant we do not change the optimal value. If we add the constant $-K$, we can subtract 1 in each sample term.

$$loss(w^j, w^i) = \left(\sum_{k=1}^{k=n} \mathop{\mathbb{E}}_{w^k \sim D^{\text{ns}}} [-1 + \log \sigma(V_{w^k,:} C_{:,w^i}) \right) - \log \sigma(V_{w^j,:} C_{:,w^i}) \tag{1.60}$$

Finally we make use of the identity $1 - \sigma(\tilde{z}) = \sigma(-\tilde{z})$ giving:

$$loss(w^j, w^i) = - \log \sigma(V_{w^j,:} C_{:,w^i}) - \sum_{k=1}^{k=n} \mathop{\mathbb{E}}_{w^k \sim D^{\text{ns}}} [\log \sigma(-V_{w^k,:} C_{:,w^i})] \tag{1.61}$$

Calculating the total loss over the training set $\mathcal{X}$:

$$Loss = - \sum_{\forall (w^i, w^j) \in \mathcal{X}} \left(\log \sigma(V_{w^j,:} C_{:,w^i}) + \sum_{k=1}^{k=n} \mathop{\mathbb{E}}_{w^k \sim D^{\text{ns}}} [\log \sigma(-V_{w^k,:} C_{:,w^i})] \right) \tag{1.62}$$

This is the negative sampling loss function used in **mikolovSkip**. Perhaps the most confusing part of this is the notation. Without the abuses around expected value, this is written:

$$Loss = - \sum_{\forall (w^i, w^j) \in \mathcal{X}} \left(\log \sigma(V_{w^j,:} C_{:,w^i}) + \sum_{\forall w^k \in \text{samples}(D^{\text{ns}})} P^{\text{D}^{\text{ns}}}(w^k) \log \sigma(-V_{w^k,:} C_{:,w^i}) \right) \tag{1.63}$$

1.5 Natural Language Applications – beyond language modeling

While statistical language models are useful, they are of-course in no way the be-all and end-all of natural language processing. Simultaneously with the developments around representations for the language modelling tasks, work was being done on solving other NLP problems using similar techniques (**collobert2008unified**).

collobert2008unified, collobert2008unified

1.5.1 Using Word Embeddings as Features

turian2010word discuss what is now perhaps the most important use of word embeddings. The use of the embeddings as features, in unrelated feature driven models. One can find word embeddings using any of the methods discussed above. These embeddings can be then used as features instead of, for example bag of words or hand-crafted feature sets. **turian2010word** found improvements on the state of the art for chunking and Named Entity Recognition (NER), using the word embedding methods of that time. Since then, these results have been superseded again using newer methods.

Pretrained Word-Embeddings
Pretrained Word Embeddings are available for most models discussed here. They are trained on a lot more data than most people have reasonable access to. It can be useful to substitute word embeddings as a representation in most systems, or to use them as initial value for neural network systems that will learn them as they train the system as a whole. There are many many online pretrained word embeddings. One of the more recent and comprehensive set is that of **bojanowski2016enriching** (based on a skip-gram extension), `https://fasttext.cc/docs/en/pretrained-vectors.html` They provide embeddings for 294 languages, trained on Wikipedia based on the work of which is an extension to skip-grams.

1.6 Aligning Vector Spaces Across Languages

Given two vocabulary vector spaces, for example one for German and one for English, a natural and common question is if they can be aligned such that one has a single vector space for both. Using canonical correlation analysis (CCA) one can do exactly that. There

turian2010word, turian2010word

bojanowski2016enriching, bojanowski2016enriching

gcca, gcca

also exists generalised CCA for any number of vector spaces (**gcca**), as well as kernel CCA for a non-linear alignment.

The inputs to CCA, are two sets of vectors, normally expressed as matrices. We will call these: $C \in \mathbb{R}^{n^{\mathrm{C}} \times m^{\mathrm{C}}}$ and $V \in \mathbb{R}^{n^{\mathrm{V}} \times m^{\mathrm{V}}}$. They are both sets of vector representations, not necessarily of the same dimensionality. They could be the output of any of the embedding models discussed earlier, or even a sparse (non-embedding) representations such as the point-wise mutual information of the co-occurrence counts. The other input is series pairs of elements from within those those sets that are to be aligned. We will call the elements from that series of pairs from the original sets $C^\star$ and $V^\star$ respectively. $C^\star$ and $V^\star$ are subsets of the original sets, with the same number of representations. In the example of applying this to translation, if each vector was a word embedding: $C^\star$ and $V^\star$ would contains only words with a single known best translation, and this does not have to be the whole vocabulary of either language.

By performing CCA one solves to find a series of vectors (also expressed as a matrix), $S = \left[\tilde{s}^1 \dots \tilde{s}^{\mathrm{d}}\right]$ and $T = \left[\tilde{t}^1 \dots \tilde{t}^{\mathrm{d}}\right]$, such that the correlation between $C^\star \tilde{s}^{\mathrm{i}}$ and $V^\star \tilde{t}^{\mathrm{i}}$ is maximised, with the constraint that for all $j < i$ that $C^\star \tilde{s}^{\mathrm{i}}$ is uncorrelated with $C^\star \tilde{s}^{\mathrm{j}}$ and that $V^\star \tilde{t}^{\mathrm{i}}$ is uncorrelated with $V^\star \tilde{t}^{\mathrm{j}}$. This is very similar to principal component analysis (PCA), and like PCA the number of components to use (d) is a variable which can be decreased to achieve dimensionality reduction. When complete, taking S and T as matrices gives projection matrices which project C and V to a space where aligned elements are as correlated as possible. The new common vector space embeddings are given by: CS and VT. Even for sparse inputs the outputs will be dense embeddings.

faruqui2014improving investigated this primarily as a means to use additional data to improve performance on monolingual tasks. In this, they found a small and

inconsistent improvement. However, we suggest it is much more interesting as a multi-lingual tool. It allows similarity measures to be made between words of different languages. **translating-unknown-words-2016** use this as part of a hybrid system to translate out of vocabulary words. **klein2015associating** use it to link word-embeddings with image embeddings.

translating-unknown-words-2016, translating-unknown-words-2016

klein2015associating, klein2015associating

dhillon2011multi investigated using this to create word-embeddings. We noted in Equation (1.16), that skip-gram maximise the similarity of the output and input embeddings according to the dot-product. CCA also maximises similarity (according the correlation), between the vectors from one set, and the vectors for another. As such given representations for two words from the same context, initialised randomly, CCA could be used repeatedly to optimise towards good word embedding capturing shared meaning from contexts. This principle was used by **dhillon2011multi**, though their final process more complex than described here. It is perhaps one of the more unusual ways to create word embeddings as compared to any of the methods discussed earlier.

dhillon2011multi, dhillon2011multi

dhillon2011multi, dhillon2011multi

Aligning embeddings using linear algebra after they are fully trained is not the only means to end up with a common vector space. One can also directly train embeddings on multiple languages concurrently as was done in **shi2015learningbiligualcofactorisation**, amongst others. Similarly, on the sentence embedding side **zou2013bilingual**, and **socherDTRNN** train embeddings from different languages and modalities (respectively) directly to be near to their partners (these are discussed in **??**). A survey paper on such methods was recently published by **Ruder17crosslingreview**.

shi2015learningbiligualcofactorisatio shi2015learningbiligualcofactorisatio

socherDTRNN, socherDTRNN

Ruder17crosslingreview, Ruder17crosslingreview