Part I

Literature Review

Chapter 1

Word Representations

You shall know a word by the company it keeps.

– J.R. Firth, 1957

Abstract

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.

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 (Dumais et al. 1988) and LDA (Blei, Ng, and Jordan 2003), 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 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 (Rosenfeld 2000). 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:

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

or to use the compact notation

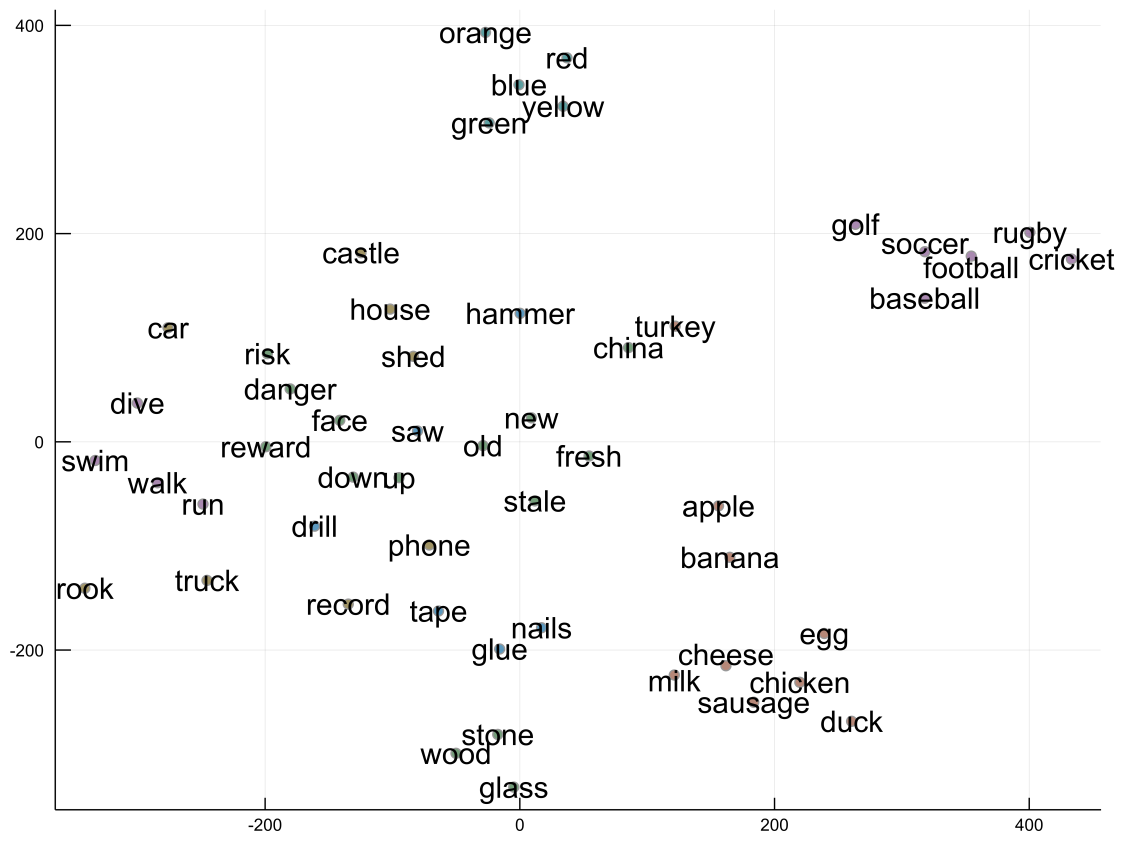
$$P(w^i \mid w^{i-1}, \ldots, w^1) \tag{1.2}$$

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 example:

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

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

Figure 1.1: Some word embeddings from the FastText project (Bojanowski et al. 2017). They were originally 300 dimensions but have been reduced to 2 using t-SNE (Maaten and Hinton 2008) 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 Chapter 2), 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.



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 (Ha et al. 2009). 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 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 (Katz 1987; Kneser and Ney 1995) 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 (Brown et al. 1992). 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.

1.1.1 The Neural Probabilistic Language Model

Bengio et al. (2003) 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.

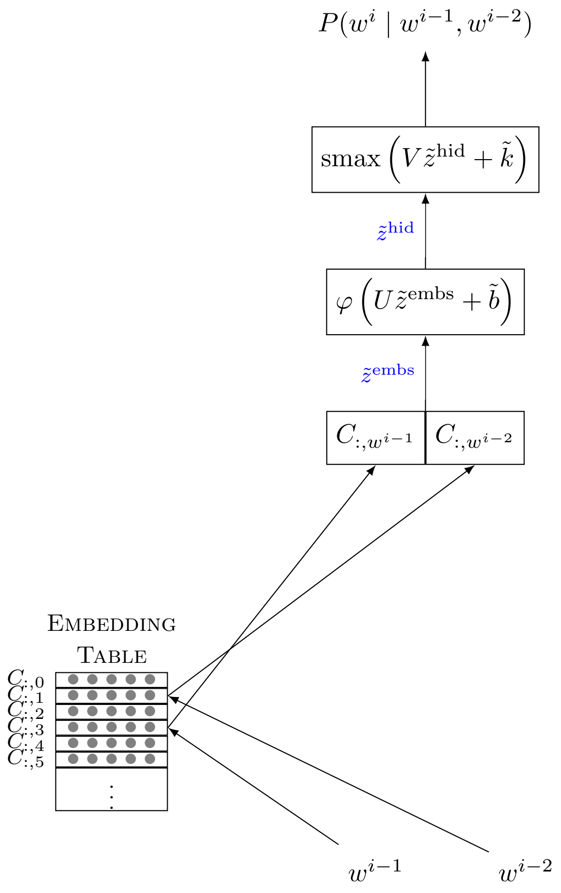
Simplified Model considered with Input Embeddings

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 Bengio et al. (2003). 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:

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

Figure 1.2: The Neural Trigram Language Model



$P(w^i \mid w^{i-1}, w^{i-2})$
$\mathrm{smax}\left(V\tilde{z}^{\mathrm{hid}} + \tilde{k}\right)$
$\tilde{z}^{\mathrm{hid}}$
$\varphi\left(U\tilde{z}^{\mathrm{embs}} + \tilde{b}\right)$
$\tilde{z}^{\mathrm{embs}}$
$C_{:,w^{i-1}}$
$C_{:,w^{i-2}}$
Embedding Table
$C_{:,0}$
$C_{:,1}$
$C_{:,2}$
$C_{:,3}$
$C_{:,4}$
$C_{:,5}$
w^{i-1}
w^{i-2}

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.

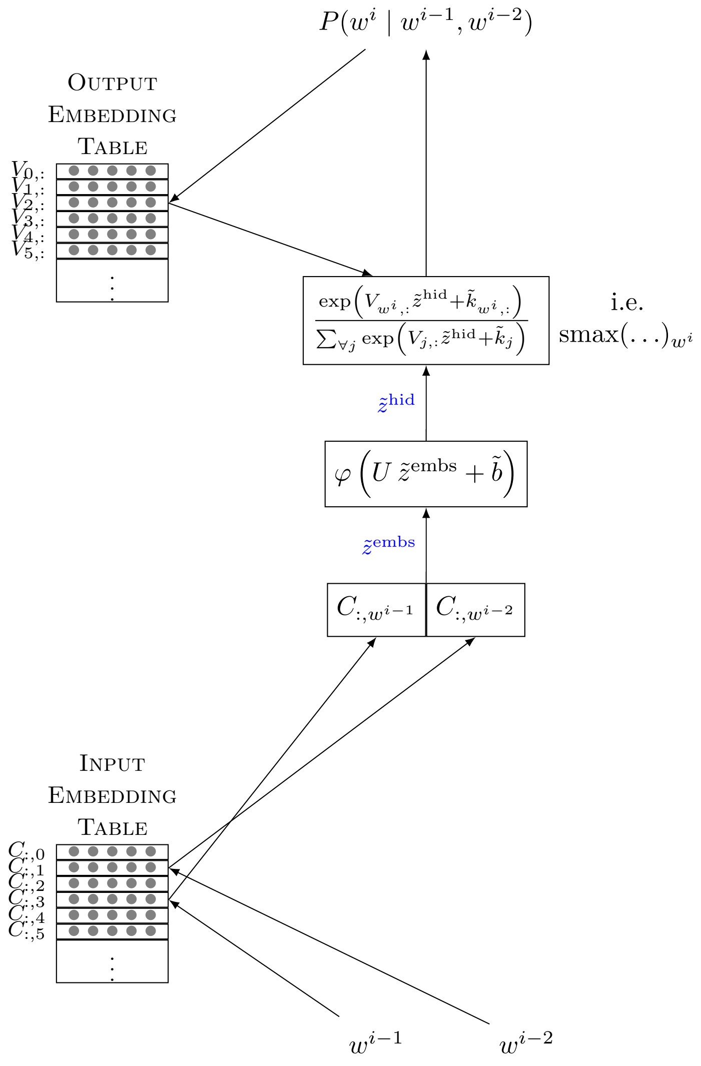
The word embeddings are trained, just like any other parameter of the network (i.e. the other weights and biases) via gradient descent. An effect of this is that the embeddings of words which predict the same future word will be adjusted to be nearer to each other in the vector space. The hidden layer 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.

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 corre-

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



$P(w^i \mid w^{i-1}, w^{i-2})$
OUTPUT EMBEDDING TABLE
$V_{0,:}$
$V_{1,:}$
$V_{2,:}$
$V_{3,:}$
$V_{4,:}$
$V_{5,:}$
$\frac{\exp\left(V_{w^i,:}\tilde{z}^{\text{hid}}+\tilde{k}_{w^i,:}\right)}{\sum_{\forall j}\exp\left(V_{j,:}\tilde{z}^{\text{hid}}+\tilde{k}_j\right)}$
i.e. $\text{smax}(\ldots)_{w^i}$
$\tilde{z}^{\text{hid}}$
$\varphi\left(U\,\tilde{z}^{\text{embs}}+\tilde{b}\right)$
$\tilde{z}^{\text{embs}}$
$C_{:,w^{i-1}}$
$C_{:,w^{i-2}}$
INPUT EMBEDDING TABLE
$C_{:,0}$
$C_{:,1}$
$C_{:,2}$
$C_{:,3}$
$C_{:,4}$
$C_{:,5}$
w^{i-1}
w^{i-2}

sponding to a possible next word (w^i):

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

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.

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 Chapter 1 of **??**:

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

which in this case is:

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

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:

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

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

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 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.

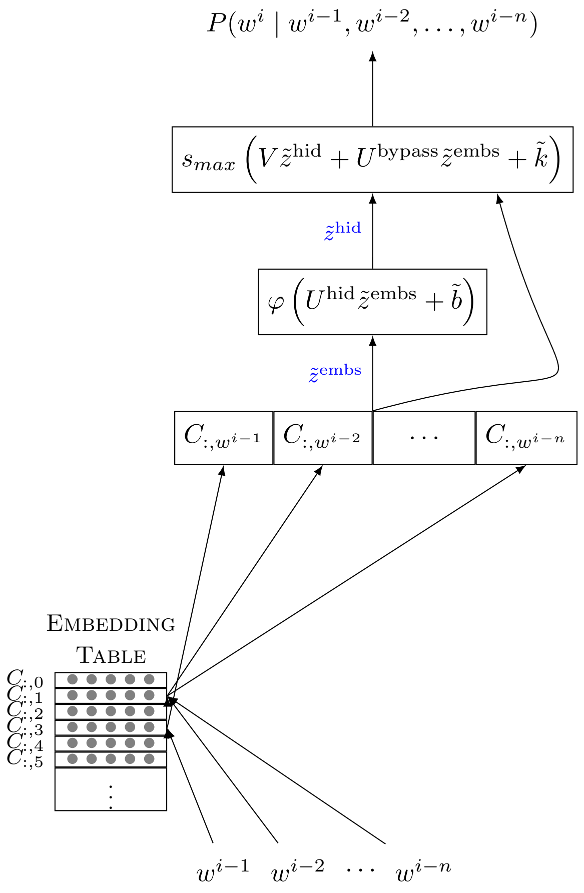
The Neural Probabilistic Language Model

Bengio et al. (2003) 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 hyperparameter 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. Bengio et al. (2003) 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:

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

[1]no pun intended

Figure 1.4: Neural Probabilistic Language Model



$P(w^i \mid w^{i-1}, w^{i-2}, \dots, w^{i-n})$
$s_{max}\left(V\tilde{z}^{\text{hid}} + U^{\text{bypass}}\tilde{z}^{\text{embs}} + \tilde{k}\right)$
$\tilde{z}^{\text{hid}}$
$\varphi\left(U^{\text{hid}}\tilde{z}^{\text{embs}} + \tilde{b}\right)$
$\tilde{z}^{\text{embs}}$
$C_{:,w^{i-1}}$
$C_{:,w^{i-2}}$
$\dots$
$C_{:,w^{i-n}}$
Embedding Table
$C_{:,0}$
$C_{:,1}$
$C_{:,2}$
$C_{:,3}$
$C_{:,4}$
$C_{:,5}$
w^{i-1}
w^{i-2}
$\dots$
w^{i-n}

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 Schwenk (2004). Though in general it is not an unheard of technique in neural network machine learning.

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 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 Mikolov et al. (2010) 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.

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.

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

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

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 Sundermeyer, Schlüter, and Ney (2012) and Jozefowicz, Zaremba, and Sutskever (2015), both of whom found that the more advanced networks gave significantly better results.

Figure 1.5: RNN Language Model. The RU equation shown is the basic RU used in Mikolov et al. (2010). It can be substituted for a LSTM RU or an GRU as was done in Sundermeyer, Schlüter, and Ney (2012) and Jozefowicz, Zaremba, and Sutskever (2015), with appropriate changes.

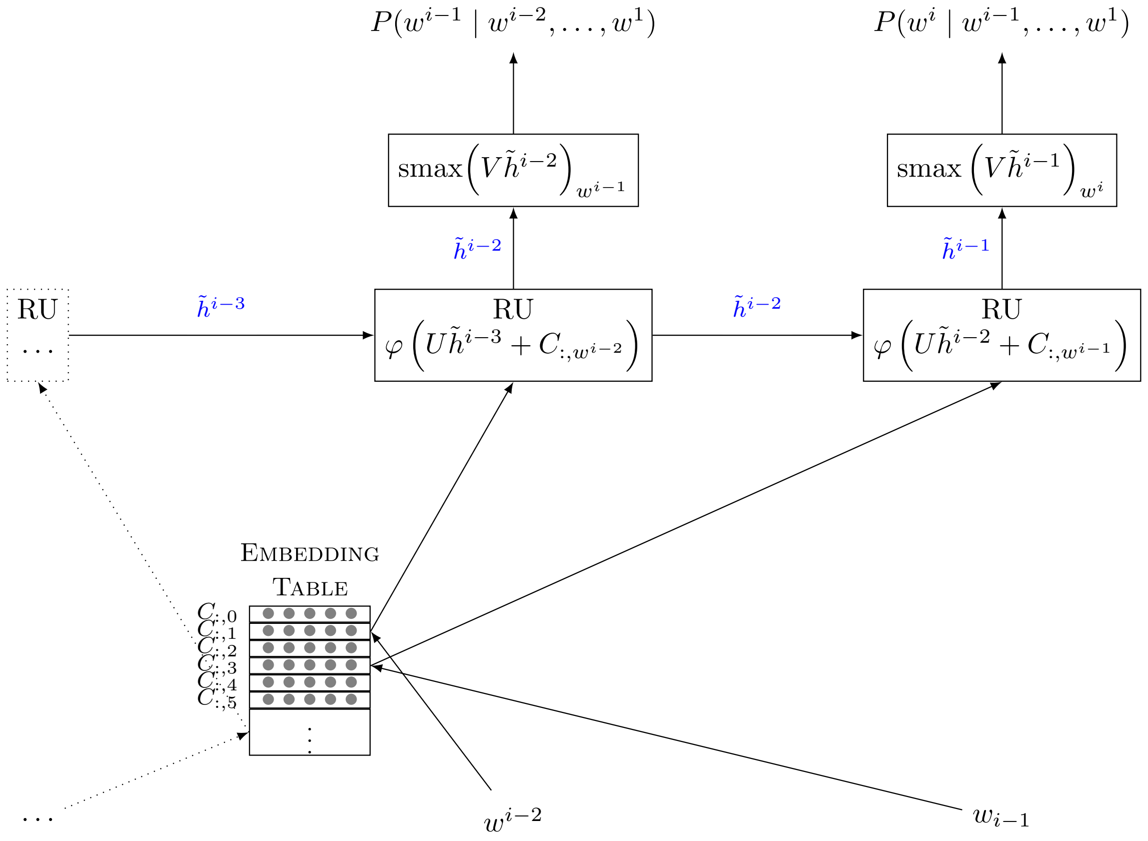
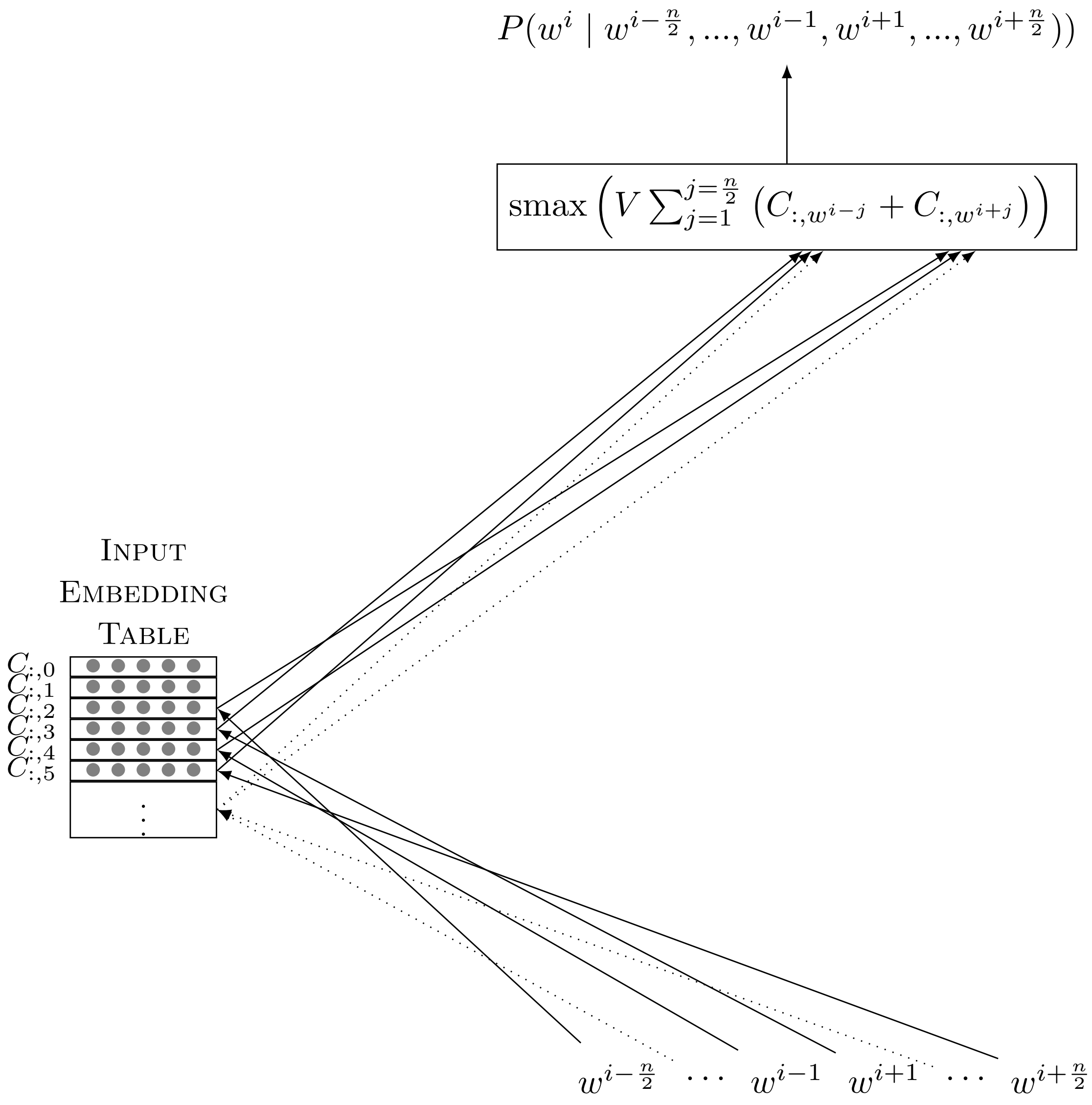

$P(w^{i-1} \mid w^{i-2}, \dots, w^1)$
$P(w^{i} \mid w^{i-1}, \dots, w^1)$
$\text{smax}\left(V\tilde{h}^{i-2}\right)_{w^{i-1}}$
$\text{smax}\left(V\tilde{h}^{i-1}\right)_{w^{i}}$
$\tilde{h}^{i-2}$
$\tilde{h}^{i-1}$
RU
…
$\tilde{h}^{i-3}$
RU
$\varphi\left(U\tilde{h}^{i-3} + C_{:,w^{i-2}}\right)$
$\tilde{h}^{i-2}$
RU
$\varphi\left(U\tilde{h}^{i-2} + C_{:,w^{i-1}}\right)$
Embedding Table
$C_{:,0}$
$C_{:,1}$
$C_{:,2}$
$C_{:,3}$
$C_{:,4}$
$C_{:,5}$
…
w^{i-2}
w_{i-1}

Figure 1.6: CBOW Language Model



$P(w^i \mid w^{i-\frac{n}{2}}, ..., w^{i-1}, w^{i+1}, ..., w^{i+\frac{n}{2}}))$
$\text{smax}\left(V \sum_{j=1}^{j=\frac{n}{2}} \left(C_{:,w^{i-j}} + C_{:,w^{i+j}}\right)\right)$
Input Embedding Table
$C_{:,0}$
$C_{:,1}$
$C_{:,2}$
$C_{:,3}$
$C_{:,4}$
$C_{:,5}$
$w^{i-\frac{n}{2}}$ ... w^{i-1} w^{i+1} ... $w^{i+\frac{n}{2}}$

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.

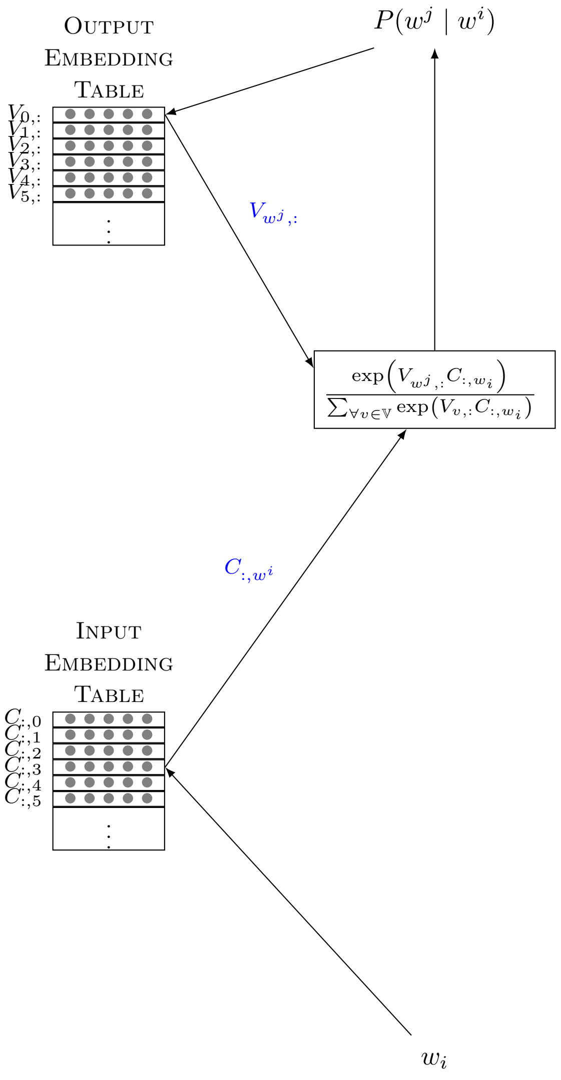
The several of the works discussed in this section also feature hierarchical softmax and negative sampling methods as alternative output methods. As these are complicated and easily misunderstood topics they are discussed in a more tutorial fashion in Section 1.4. This section will focus just on the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

1.2.1 Continuous Bag of Words

The continuous bag of words (CBOW) method was introduced by Mikolov et al. (2013b). 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

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.



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 is defined by:

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

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

The converse of CBOW is the skip-grams model Mikolov et al. (2013b). 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. This can be indexed to get the individual probability of a given word occurring as usual for a language model. So for input word

Figure 1.8: Example of analogy algebra

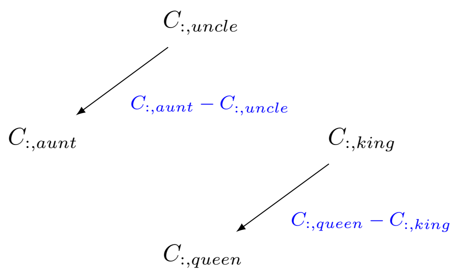
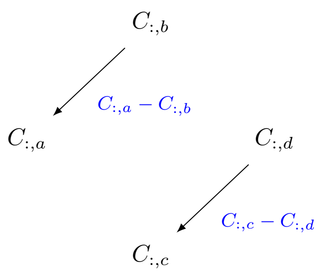


Figure 1.9: Vectors involved in analogy ranking tasks, this may help to understand the math in Equation (1.19)



w^i the probability of w^j occurring in its context is given by:

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

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 Gladkova, Drozd, and Matsuoka (2016), 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).

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

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

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

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

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

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 Mikolov, Yih, and Zweig (2013) were relatively poor, but the surprising finding was that this worked at all. Mikolov et al. (2013b) 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 Pennington, Socher, and Manning (2014) were significantly better again for both.

1.3 Co-location Factorisation

1.3.1 GloVe

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 Pennington, Socher, and Manning (2014) 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.

Pennington, Socher, and Manning (2014) 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.

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

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

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 Levy, Goldberg, and Dagan (2015) with more tuned parameters, found that skip-gram marginally out-performed GloVe on all tasks.

1.3.2 Further equivalence of Co-location Prediction to Factorisation

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 Levy and Goldberg (2014) and in Li et al. (2015) with more direct mathematical equivalence to weighed co-occurrence matrix factorisation; Later, Cotterell et al. (2017) showed the equivalence to exponential principal component analysis (PCA). Landgraf and Bellay (2017) 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.

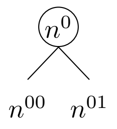
1.4.1 Hierarchical Softmax

Hierarchical softmax was first presented in Morin and Bengio (2005). Its recent use was popularised by Mikolov et al. (2013b), where words are placed as leaves in a Huffman tree, with their depth determined by their frequency.

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 (Huffman 1952), as was found to be effective by Mikolov et al. (2013b). 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 $l(w^i) \approx -\log_2(P(w^i))$. 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 –

Figure 1.10: Tree for 2 words



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 Mikolov et al. (2013b), 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.

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 appended column of ones; or it can even be ignored in the implementation (as was done in Mikolov et al. (2013b)). 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 $2^{-k} > P(w^i)$.

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.

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 $V_{0,:}$.

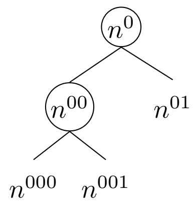
$$P(n^{00} \mid \tilde{z}) = \sigma(V_{0,:}\tilde{z}) \tag{1.26}$$

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

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

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

Figure 1.11: Tree for 3 words



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 $-1^1 = -1$)).

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.

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.

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

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

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

Thus:

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

The conditional probability chain rule, is applied to get:

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

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 Mikolov et al. (2013a).

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 Mikolov et al. (2013a). 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 $[\![n^b = ch(n^a)]\!]$ which will be 1 if n^a is an arbitrary (but consistent) child of n^b, and 0 otherwise.

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

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.

Loss Function

Using normal softmax, during the training, the cross-entropy between the model's predictions and the ground 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 Mikolov et al. (2013a) as another method to speed up this problem. Much like 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 (Gutmann and Hyvärinen 2012). 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.

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.

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

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

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.

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

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 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.

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 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 ($P^{1\mathrm{g}}(w^k)$) gives:

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

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:

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

to be the sum of the n samples expected values. This notation (abuse) is as used in Mikolov et al. (2013a). It gives the form:

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

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 $D^{1\mathrm{g}}$ with D^{ns} being the distribution of negative samples from the vocabulary, to be specified as a hyper-parameter of training.

Mikolov et al. (2013a) uses a distribution such that

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

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.

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

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

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

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

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

This is the negative sampling loss function used in Mikolov et al. (2013a). Perhaps the most confusing part of this is the notation. Without the abuses around expected value, this is written:

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

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 (Collobert and Weston 2008).

1.5.1 Using Word Embeddings as Features

Turian, Ratinov, and Bengio (2010) 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. Turian, Ratinov, and Bengio (2010) 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.

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 also exists generalised CCA for any number of vector spaces (Fu et al. 2016), 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), $S = \left[\tilde{s}^1 \ldots \tilde{s}^{\mathrm{d}}\right]$ and $T = \left[\tilde{t}^1 \ldots \tilde{t}^{\mathrm{d}}\right]$, 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.

Faruqui and Dyer (2014) 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. Gujral, Khayrallah, and Koehn (2016) use this as part of a hybrid system to translate out of vocabulary words. Klein et al. (2015) use it to link word-embeddings with image embeddings.

Dhillon, Foster, and Ungar (2011) 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 Dhillon, Foster, and Ungar (2011), 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.

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 Shi et al. (2015), amongst others. Similarly, on the sentence embedding side Zou et al. (2013), and Socher et al. (2014) train embeddings from different languages and modalities (respectively) directly to be near to their partners (these are discussed in Chapter 3). A survey paper on such methods was recently published by Ruder (2017).

Chapter 2

Word Sense Representations

1a. *In a literal, exact, or actual sense; not figuratively, allegorically, etc.*

1b. *Used to indicate that the following word or phrase must be taken in its literal sense, usually to add emphasis.*

1c. *colloq. Used to indicate that some (frequently conventional) metaphorical or hyperbolical expression is to be taken in the strongest admissible sense: 'virtually, as good as'; (also) 'completely, utterly, absolutely' ...*

2a *With reference to a version of something, as a transcription, translation, etc.: in the very words, word for word.*

2b. *In extended use. With exact fidelity of representation; faithfully.*

3a. *With or by the letters (of a word). Obs. rare.*

3b. *In or with regard to letters or literature. Obs. rare.*

– the seven senses of `literally`, *Oxford English Dictionary*, 3rd ed., 2011

Abstract

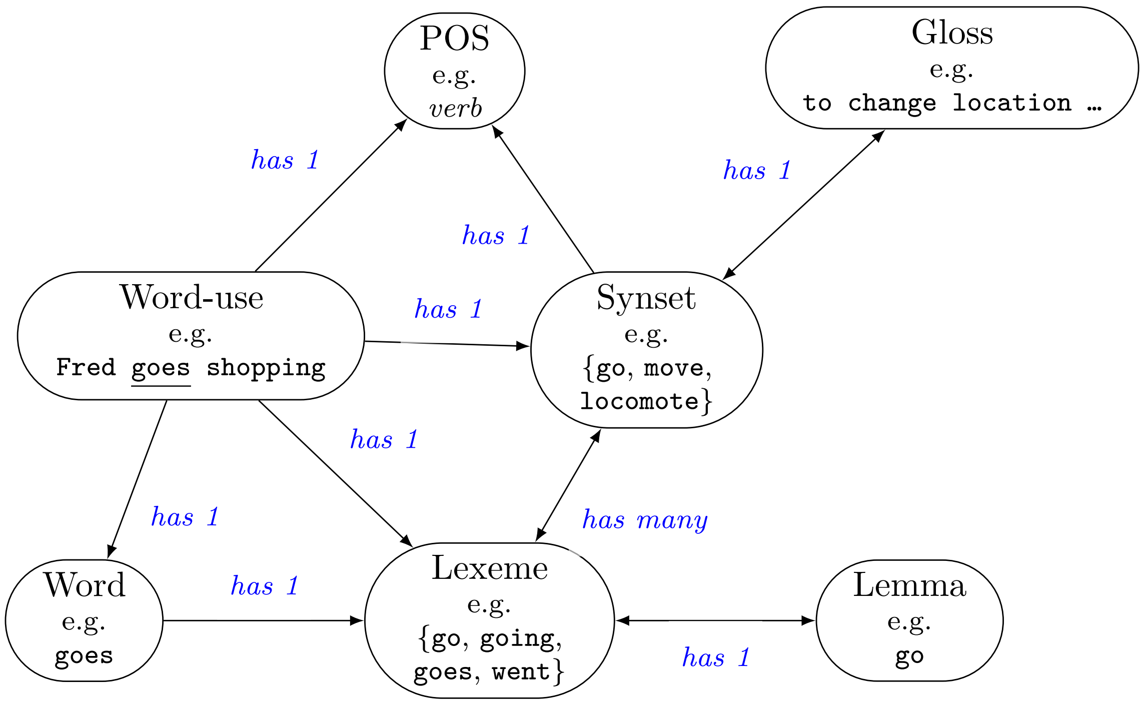
In this chapter, techniques for representing the multiple meanings of a single word are discussed. This is a growing area, and is particularly important in languages where polysemous and homonymous words are common. This includes English, but it is even more prevalent in Mandarin for example. The techniques discussed can broadly be classified as lexical word sense representation, and as word sense induction. The inductive techniques can be sub-classified as clustering-based or as prediction-based.

2.1 Word Senses

Words have multiple meanings. A single representation for a word cannot truly describe the correct meaning in all contexts. It may have some features that are applicable to some uses but not to others, it may be an average of all features for all uses, or it may only represent the most common sense. For most word-embeddings it will be an unclear combination of all of the above. Word sense embeddings attempt to find representations not of words, but of particular senses of words.

The standard way to assign word senses is via some lexicographical resource, such as a dictionary, or a thesaurus. There is not a canonical list of word senses that are consistently defined in English. Every dictionary is unique, with different definitions and numbers of word senses. The most commonly used lexicographical resource is WordNet (**miller1995wordnet**), and the multi-lingual BabelNet (**navigli2010babelnet**). The relationship between the terminology used in word sense problems is shown in Figure 2.1

Figure 2.1: The relationship between terms used to discuss various word sense problems. The lemma is used as the representation for the lexeme, for WordNet's purposes when indexing. For many tasks each the word-use is pre-tagged with its lemma and POS tag, as these can be found with high reliability using standard tools. Note that the arrows in this diagram are *directional*. That is to say, for example, each Synset *has 1* POS, but each POS *has many* Synsets.



2.1.1 Word Sense Disambiguation

Word sense disambiguation is one of the hardest problems in NLP. Very few systems significantly out perform the baseline, i.e. the most frequent sense (MFS) technique.

Progress on the problem is made difficult by several factors.

The sense is hard to identify from the context. Determining the sense may require very long range information: for example the information on context may not even be in the same sentence. It may require knowing the domain of the text, because word sense uses vary between domains. Such information is external to the text itself. It may in-fact be intentionally unclear, with multiple correct interpretations, as in a pun. It maybe unintentionally unknowable, due to a poor writing style, such that it would confuse any human reader. These difficulties are compounded by the limited amount of data available.

There is only a relatively small amount of labelled data for word sense problems. It is the general virtue of machine learning that given enough data, almost any input-output mapping problem (i.e. function approximation) can be solved. Such an amount of word sense annotated data is not available. This is in contrast to finding unsupervised word embeddings, which can be trained on any text that has ever been written. The lack of very large scale training corpora renders fully supervised methods difficult. It also results in small sized testing corpora; which leads to systems that may appear to perform well (on those small test corpora), but do not generalise to real world uses. In addition, the lack of human agreement on the correct sense, resulting in weak ground truth, further makes creating new resources harder. This limited amount of data compounds the problem's inherent difficulties.

It can also be said that word senses are highly artificial and do not adequately represent meaning. However, WSD is required to interface with lexicographical resources, such as translation dictionaries (e.g. BabelNet), ontologies (e.g. OpenCyc), and other datasets (e.g. ImageNet (**imagenet_cvpr09**)).

It may be interesting to note, that the number of meanings that a word has is approximately inversely proportional related to its frequency of use rank (**zipf1945meaning**). That is to say the most common words have far more meanings than rarer words. It is related to (and compounds with) the more well-known Zipf's Law on word use (**zipf1949human**), and can similarly be explained-based on Zipf's core premise of the principle of least effort. This aligns well with our notion that precise (e.g. technical) words exist but are used only infrequently – since they are only explaining a single situation. This also means that by most word-uses are potentially very ambiguous.

The most commonly used word sense (for a given word) is also overwhelmingly more frequent than its less common brethren – word sense usage also being roughly Zipfian distributed (**Kilgarriff2004**). For this reason the Most Frequent Sense (MFS) is a surprisingly hard baseline to beat in any WSD task.

Most Frequent Sense

Given a sense annotated corpus, it is easy to count how often each sense of a word occurs. Due to the over-whelming frequency of the most frequent sense, it is unlikely for even a small training corpus to have the most frequent sense differing from the use in the language as a whole.

The Most Frequent Sense (MFS) method of word sense disambiguation is defined by counting the frequency of a particular word sense for a particular POS tagged word. For the ith word use being the word w^i, having some sense s^j then without any further context the probability of that sense being the correct sense is $P(s^j \mid w^i)$. One can use the part of speech tag p_i (for the ith word use) as an additional condition, and thus find $P(s^j \mid w^i, p_i)$. WordNet encodes this information for each lemma-synset pair (i.e. each word sense) using the SemCor corpus counts. This is also used for sense ordering, which is why most frequent sense is sometimes called first sense. This is a readily available and practical method for getting a baseline probability of each sense. Most frequent sense can be applied for word sense disambiguation using this frequency-based probability estimate: $\operatorname{argmax}_{\forall s^j} P(s^j \mid w^i, p_i)$.

In the most recent SemEval WSD task (**moro2015semeval**), MFS beat all submitted entries for English, both overall, and on almost all cuts of the data. The results for other languages were not as good, however in other languages the true corpus-derived sense counts were not used.

2.2 Word Sense Representation

It is desirable to create a vector representation of a word sense much like in Chapter 1 representations were created for words. We desire to an embedding to represent each word sense, as normally represented by a word-synset pair. This section considers the representations for the lexical word senses as given from a dictionary. We consider a direct method of using a labelled corpus, and an indirect method makes use of simpler sense-embeddings to partially label a corpus before retraining. These methods create representations corresponding to senses from WordNet. Section 2.3 considers the case when the senses are to also be discovered, as well as represented.

2.2.1 Directly supervised method

The simple and direct method is to take a dataset that is annotated with word senses, and then treat each sense-word pair as if it were a single word, then apply any of the methods for word representation discussed in Chapter 1. **iacobacci2015sensembed** use a CBOW language model (Mikolov et al. 2013b) to do this. This does, however, run into the aforementioned problem, that there is relatively little training data that has been manually sense annotated. **iacobacci2015sensembed** use a third-party WSD tool, namely BabelFly (**Moro2014**), to annotate the corpus with senses. This allows for existing word representation techniques to be applied.

Chen2014 applies a similar technique, but using a word-embedding-based partial WSD system of their own devising, rather than an external WSD tool.

2.2.2 Word embedding-based disambiguation method

Chen2014 uses an almost semi-supervised approach to train sense vectors. They partially disambiguate their training corpus, using initial word sense vectors and WordNet. They then completely replace these original (phase one) sense-vectors, by using the partially disambiguated corpus to train new (phase two) sense-vectors via a skip-gram variant. This process is shown in Figure 2.2.

The **first phase** of this method is in essence a word-embedding-based WSD system. When assessed as such, they report that it only marginally exceeds the MFS baseline, though that is not at all unusual for WSD algorithms as discussed above.

They assign a sense vector to every word sense in WordNet. This sense vector is the average of word-embeddings of a subset of words in the gloss, as determined using pretrained skip-grams (Mikolov et al. 2013b). For the word w with word sense $w^{\mathrm{s}^{\mathrm{i}}}$, a set of candidate words, $cands(w^{\mathrm{s}^{\mathrm{i}}})$, is selected from the gloss based on the following set of requirements. First, the word must be a content word: that is a verb, noun, adverb or adjective; secondly, its cosine distance to w must be below some threshold δ; finally, it must not be the word itself. When these requirements are followed $cands(w^{s^i})$ is a set of significant closely related words from the gloss.

The phase one sense vector for w^{s^i} is the mean of the word vectors for all the words in $cands(w^{s^i})$. The effect of this is that we expect that the phase one sense vectors for most words in the same synset will be similar but not necessarily identical. This expectation is not guaranteed however. As an example, consider the use of the word `china` as a synonym for `porcelain`: the single sense vector for `china` will likely be dominated by its more significant use referring to the country, which would cause very few words in the gloss for the `porcelain` synset to be included in *cands*. Resulting in the phase one sense vectors for the synonymous senses of `porcelain` and `china` actually being very different.

The phase one sense vectors are used to disambiguate the words in their unlabelled training corpus. For each sentence in the corpus, an initial *content vector* is defined by taking the mean of the skip-gram word embedding (not word sense) for all content words in the sentence. For each word in the sentence, each possible sense-embedding is compared to the context vector. If one or more senses vectors are found to be closer than a given threshold, then that word is tagged with the closest of those senses, and the context vector is updated to use the sense-vector instead of the word vector. Words that do not come within the threshold are not tagged, and the context vector is not updated. This is an important part of their algorithm, as it ensures that words without clear senses do not get a sense ascribed to them. This thus avoids any dubious sense tags for the next training step.

In **phase two** of training **Chen2014** employ the skip-gram word-embedding method, with a variation, to predict the word senses. They train it on the partially disambiguated corpus produced in phase one. The original sense vectors are discarded. Rather than the model being tasked only to predict the surrounding words, it is tasked to predict surrounding words and their sense-tags (where present). In the loss function the prediction of tags and words is weighted equally.

Note that the input of the skip-gram is the just central word, not the pair of central word with sense-tag. In this method, the word sense embeddings are output embeddings; though it would not be unreasonable to reverse it to use input embeddings with sense tags, or even to do both. The option to have input embeddings and output embeddings be from different sets, is reminiscent of Schwenk (2004) for word embeddings.

The phase one sense vectors have not been assessed on their representational quality. It could be assumed that because the results for these were not reported, they were worse than those found in phase two. The phase two sense vectors were not assessed for their capacity to be used for word sense disambiguation. It would be desirable to extend the method of **Chen2014**, to use the phase two vectors for WSD. This would allow this method to be used to disambiguate its own training data, thus enabling the method to become self-supervised.

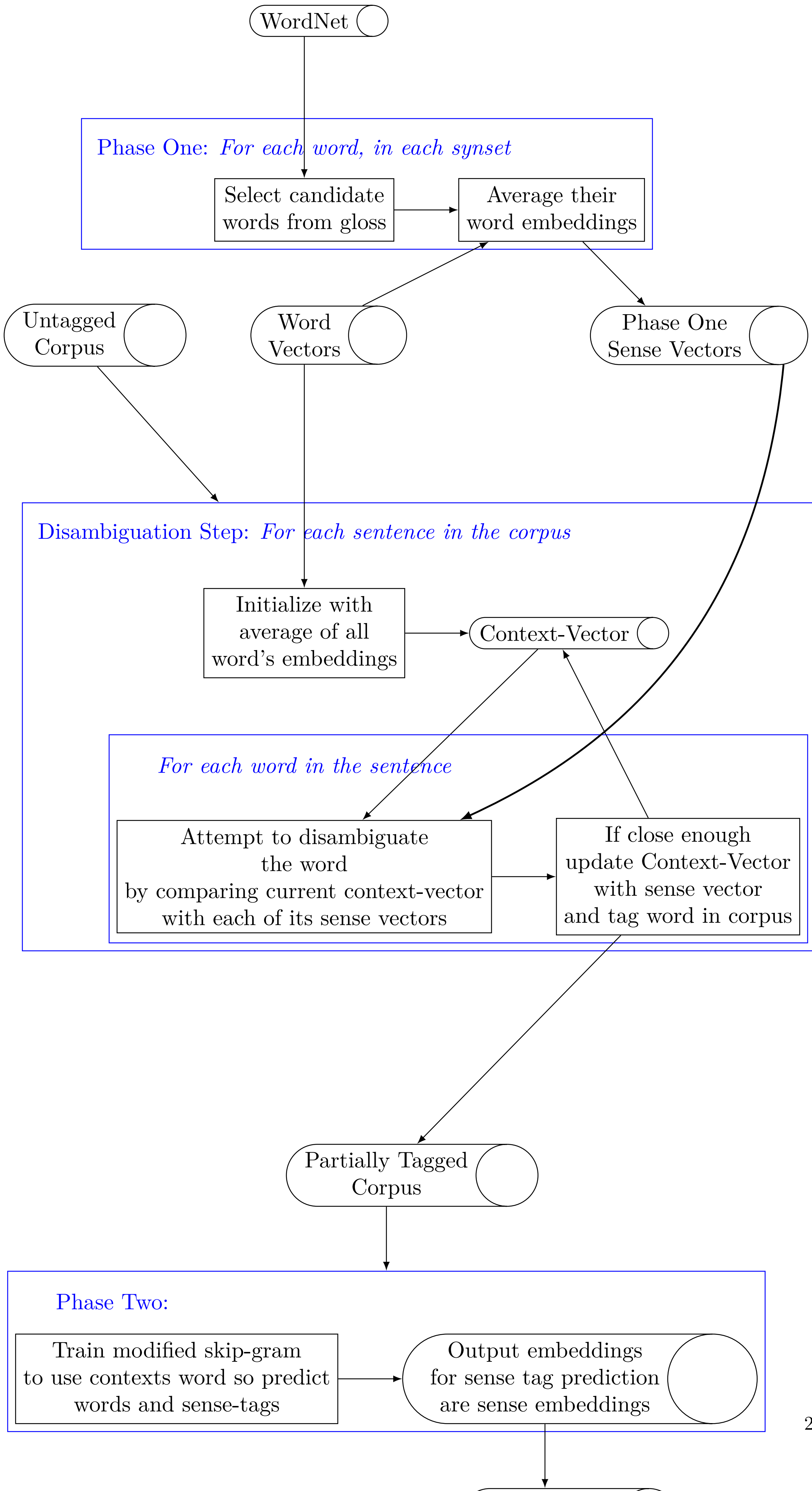
2.3 Word Sense Induction (WSI)

In this section we will discuss methods for finding a word sense without reference to a standard set of senses. Such systems must discover the word senses at the same time as they find their representations. One strong advantage of these methods is that they do not require a labelled dataset. As discussed there are relatively few high-quality word sense labelled datasets. The other key advantage of these systems is that they do not rely on fixed senses determined by a lexicographer. This is particularly useful if the word senses are highly domain specific; or in a language without strong lexicographical resources. This allows the data to inform on what word senses exist.

Most vector word sense induction and representation approaches are evaluated on similarity tests. Such tests include WordSim-353 (**WordSim353**) for context-less, or Stanford's Contextual Word Similarities (SCWS) for similarity with context information (**Huang2012**). This evaluation is also suitable for evaluating single sense word-embeddings, e.g. skip-grams.

We can divide the WSI systems into context clustering-based approaches, and co-location prediction-based approaches. This division is similar to the separation of co-location matrix factorisation, and co-location prediction-based approaches discussed in Chapter 1. It can be assumed thus that at the core, like for word embeddings, they are fundamentally very similar. One could think of prediction of collocated words as a soft indirect clustering of contexts that can have those words.

Figure 2.2: The process used by **Chen2014** to create word sense embeddings.



WordNet
Phase One: *For each word, in each synset*
Select candidate words from gloss
Average their word embeddings
Untagged Corpus
Word Vectors
Phase One Sense Vectors
Disambiguation Step: *For each sentence in the corpus*
Initialize with average of all word's embeddings
Context-Vector
For each word in the sentence
Attempt to disambiguate the word by comparing current context-vector with each of its sense vectors
If close enough update Context-Vector with sense vector and tag word in corpus
Partially Tagged Corpus
Phase Two:
Train modified skip-gram to use contexts word so predict words and sense-tags
Output embeddings for sense tag prediction are sense embeddings

2.3.1 Context Clustering-based Approaches

As the meaning of a word, according to word embedding principles, is determined by the contexts in which it occurs, we expect that different meanings (senses) of the same words should occur in different contexts. If we cluster the contexts that a word occurs in, one would expect to find distinct clusters for each sense of the word. It is on this principle that the context clustering-based approaches function.

Offline clustering

The fundamental method for most clustering-based approaches is as per **Schutze:1998wordsenseclustering**. That original work is not a neural word sense embedding, however the approach remains the same. **pantel2002WSI** and **Reisinger2010** are also not strictly neural word embedding approaches (being more classical vector representations), however the overall method is also very similar.

The clustering process is done by considering all word uses, with their contexts. The contexts can be a fixed-sized window of words (as is done with many word-embedding models), the sentence, or defined using some other rule. Given a pair of contexts, some method of measuring their similarity must be defined. In vector representational works, this is ubiquitously done by assigning each context a vector, and then using the cosine similarity between those vectors.

The **first step** in all the offline clustering methods is thus to define the representations of the contexts. Different methods define the context vectors differently:

- **Schutze:1998wordsenseclustering** uses variations of inverse-document-frequency (idf) weighted bags of words, including applying dimensionality reduction to find a dense representation.
- **pantel2002WSI** use the mutual information vectors between words and their contexts.
- **Reisinger2010**, use td-idf or χ^2 weighted bag of words.
- **Huang2012** uses td-idf weighted averages of (their own) single sense word embeddings for all words in the context.
- **kaageback2015neural** also uses a weighted average of single sense word skip-gram embeddings, with the weighting based on two factors. One based on how close the words were, and the other on how likely the co-occurrence was according to the skip-gram model.

It is interesting to note that idf, td-idf, mutual information, skip-gram co-occurrence probabilities (being a proxy for point-wise mutual information (Levy and Goldberg 2014)), are all closely related measures.

The **second step** in off-line clustering is to apply a clustering method to cluster the word-uses. This clustering is done based on the calculated similarity of the context representation where the words are used. Again, different WSI methods use different clustering algorithms.

- **Schutze:1998wordsenseclustering** uses a group average agglomerative clustering method.
- **pantel2002WSI** use a custom hierarchical clustering method.
- **Reisinger2010** use mixtures of von-Mises-Fisher distributions.
- **Huang2012** use spherical k-means.
- **kaageback2015neural** use k-means.

The **final step** is to find a vector representation of each cluster. For non-neural embedding methods this step is not always done, as defining a representation is not the goal, though in general it can be derived from most clustering techniques. **Schutze:1998wordsenseclustering** and **kaageback2015neural** use the centroids of their clusters. **Huang2012** use a method of relabelling the word uses with a cluster identifier, then train a (single-sense) word embedding method on cluster identifiers rather than words. This relabelling technique is similar to the method later used by **Chen2014** for learning lexical sense representations, as discussed in Section 2.2.2. As each cluster of contexts represents a sense, those cluster embeddings are thus also considered as suitable word sense embeddings.

To summarize, all the methods for inducing word sense embeddings via off-line clustering follow the same process. **First**: represent the contexts of word use, so as to be able to measure their similarity. **Second**: use the context's similarity to cluster them. **Finally**: find a vector representation of each cluster. This cluster representation is the induced sense embedding.

Online clustering

The methods discussed above all use off-line clustering. That is to say the clustering is performed after the embedding is trained. **neelakantan2015efficient** perform the clustering during training. To do this they use a modified skip-gram-based method. They start with a fixed number of randomly initialised sense vectors for each context. These sense vectors are used as input embeddings for the skip-gram context prediction task, over single sense output embeddings. Each sense also has, linked to it, a context cluster centroid, which is the average of all output embeddings for the contexts that the sense is assigned to. Each time a training instance is presented, the average of the context output embeddings is compared to each sense's context cluster centroid. The context is assigned to the cluster with the closest centroid, updating the centroid value. This can be seen as similar to performing a single k-means update step for each training instance. Optionally, if the closest centroid is further from the context vector than some threshold, a new sense can be created using that context vector as the initial centroid. After the assignment of the context to a cluster, the corresponding sense vector is selected for use as the input vector in the skip-gram context prediction task.

kaageback2015neural investigated using their weighting function (as discussed in Section 2.3.1) with the online clustering used by **neelakantan2015efficient**. They found that this improved the quality of the representations. More generally any such weighting function could be used. This online clustering approach is loosely similar to the co-location prediction-based approaches.

2.3.2 Co-location Prediction-based Approaches

Rather than clustering the contexts, and using those clusters to determine embeddings for different senses, one could consider the sense as a latent variable in the task used to find word embeddings – normally a language modelling task. The principle is that it is not the word that determines its collocated context words, but rather the word sense. So the word sense can be modelled as a hidden variable, where the word, and the context words are being observed.

tian2014probabilistic used this to define a skip-gram-based method for word sense embeddings. For input word w^i with senses $\mathcal{S}(w^i)$, the probability of output word w^o occurring near w^i can be given as:

$$P(w^o \mid w^i) = \sum_{\forall s^k \in \mathcal{S}(w^i)} P(w^o \mid s^k, w^i) P(s^k \mid w^i) \tag{2.1}$$

Given that a sense s^k only belongs to one word w^i, we know that kth sense of the ith word only occurs when the ith word occurs. We have that the join probability $P(w^i, s^k) = P(s^k)$.

We can thus rewrite Equation (2.1) as:

$$P(w^o \mid w^i) = \sum_{\forall s^k \in \mathcal{S}(w^i)} P(w^o \mid s^k) \, P(s^k \mid w^i) \tag{2.2}$$

A softmax classifier can be used to define $P(w^o \mid s^k)$, just like in normal language modelling. With output embeddings for the words w^o, and input embeddings for the word senses s^k. This softmax can be sped-up using negative sampling or hierarchical softmax. The later was done by **tian2014probabilistic**.

Equation (2.2) is in the form of a mixture model with a latent variable. Such a class of problems are often solved using the Expectation Maximisation (EM) method. In short, the EM procedure functions by performing two alternating steps. The **E-step** calculates the expected chance of assigning word sense for each training case ($\hat{P}(s^l \mid w^o)$) in the training set $\mathcal{X}$. Where a training case is a pairing of a word use w^i, and context word w^o, with $s^l \in \mathcal{S}(w^i)$, formally we have:

$$\hat{P}(s^l \mid w^o) = \frac{\hat{P}(s^l \mid w^i) P(w^o \mid s^l)}{\sum_{\forall s^k \in \mathcal{S}(w^i)} \hat{P}(w^o \mid s^k) P(s^k \mid w^i)} \tag{2.3}$$

The **M-step** updates the prior likelihood of each sense (that is without context) using the expected assignments from the E-step.

$$\hat{P}(s^l \mid w^i) = \frac{1}{|\mathcal{X}|} \sum_{\forall (w^o, w^i) \in \mathcal{X}} \hat{P}(s^l \mid w^o) \tag{2.4}$$

During this step the likelihood of the $P(w^o \mid w^i)$ can be optimised to maximise the likelihood of the observations. This is done via gradient descent on the neural network parameters of the softmax component: $P(w^o \mid s^k)$. By using this EM optimisation the network can fit values for the embeddings in that softmax component.

A limitation of the method used by **tian2014probabilistic**, is that the number of each sense must be known in advance. One could attempt to solve this by using, for example, the number of senses assigned by a lexicographical resource (e.g. WordNet). However, situations where such resources are not available or not suitable are one of the main circumstances in which WSI is desirable (for example in work using domain specific terminology, or under-resourced languages). In these cases one could apply a heuristic-based on the distribution of senses-based on the distribution of words (**zipf1945meaning**). An attractive alternative would be to allow senses to be determined-based on how the words are used. If they are used in two different ways, then they should have two different senses. How a word is being used can be determined by the contexts in which it appears.

AdaGrams extend on this work by making the number of senses for each word itself a fit-able parameter of the model. This is a rather Bayesian modelling approach, where one considers the distribution of the prior.

Considering again the form of Equation (2.2)

$$P(w^o \mid w^i) = \sum_{\forall s^k \in \mathcal{S}(w^i)} P(w^o \mid s^k) P(s^k \mid w^i) \tag{2.5}$$

The prior probability of a sense given a word, but no context, is $P(s^k \mid w^i)$. This is Dirichlet distributed. This comes from the definition of the Dirichlet distribution as the the prior probability of any categorical classification task. When considering that the sense my be one from an unlimited collection of possible senses, then that prior becomes a Dirichlet process.

In essence, this prior over a potentially unlimited number of possible senses becomes another parameter of the model (along with the input sense embeddings and output word embeddings). The fitting of the parameters of such a model is beyond the scope of this book; it is not entirely dissimilar to the fitting via expectation maximisation incorporating gradient descent used by **tian2014probabilistic**. The final output of **AdaGrams** is as desired: a set of induced sense embeddings, and a language model that is able to predict how likely a word is to occur near that word sense ($P(w^o \mid s^k)$).

By application of Bayes' theorem, the sense language model can be inverted to take a word's context, and predict the probability of each word sense.

$$P(s^l \mid w^o) = \frac{P(w^o \mid s^l) P(s^l \mid w^i)}{\sum_{\forall s^k \in \mathcal{S}(w^i)} P(w^o \mid s^k) P(s^k \mid w^i)} \tag{2.6}$$

with the common (but technically incorrect) assumption that all words in the context are independent.

Given a context window:
$\mathcal{W}^i = \left(w^{i-\frac{n}{2}}, \ldots, w^{i-1}, w^{i+1}, \ldots, w^{i+\frac{n}{2}}\right)$, we have:

$$P(s^l \mid \mathcal{W}^i) = \frac{\prod_{\forall w^j \in \mathcal{W}^i} P(w^j \mid s^l) P(s^l | w^i)}{\sum_{s^k \in \mathcal{S}(w^i)} \prod_{\forall w^j \in \mathcal{W}^i} P(w^j \mid s^k) P(s^k | w^i)} \tag{2.7}$$

2.4 Conclusion

Word sense representations allow the representations of the senses of words when one word has multiple meanings. This increases the expressiveness of the representation. These representations can in general be applied anywhere word embeddings can. They are particularly useful for translation, and in languages with large numbers of homonyms.

The word representation discussions in this chapter naturally lead to the next section on phrase representation. Rather than a single word having many meanings, the next chapter will discuss how a single meaning may take multiple words to express. In such longer structure's representations, the sense embeddings discussed here are often unnecessary, as the ambiguity may be resolved by the longer structure. Indeed, the methods discussed in this chapter have relied on that fact to distinguish the senses using the contexts.

Chapter 3

Sentence Representations and Beyond

A sentence is a group of words expressing a complete thought.
– *English Composition and Literature*, Webster, 1923

Abstract

This chapter discusses representations for larger structures in natural language. The primary focus is on the sentence level. However, many of the techniques also apply to sub-sentence structures (phrases), and super-sentence structures (documents). The three main types of representations discussed here are: unordered models, such as sum of word embeddings; sequential models, such as recurrent neural networks; and structured models, such as recursive autoencoders.

It can be argued that the core of true AI, is in capturing and manipulating the representation of an idea. In natural language a sentence (as defined by Webster in the quote above), is such a representation of an idea, but it is not machine manipulatable. As such the conversion of sentences to a machine manipulatable representation is an important task in AI research.

All techniques which can represent documents (or paragraphs) by necessity represent sentences as well. A document (or a paragraph), can consist only of a single sentence. Many of these models always work for sub-sentence structures also, like key-phrases. When considering representing larger documents, neural network embedding models directly compete with vector information retrieval models, such as LSI (Dumais et al. 1988), probabilistic LSI (**hofmann2000learning**) and LDA (Blei, Ng, and Jordan 2003).

3.1 Unordered and Weakly Ordered Representations

A model that does not take into account word order cannot perfectly capture the meaning of a sentence. **Mitchell2008** give the poignant examples of:

- `It was not the sales manager who hit the bottle that day, but the office worker with the serious drinking problem.`
- `That day the office manager, who was drinking, hit the problem sales worker with a bottle, but it was not serious.`

These two sentences have the same words, but in a different structure, resulting in their very different meanings. In practice, however, representations which discard word order can be quite effective.

3.1.1 Sums of Word Embeddings

Classically, in information retrieval, documents have been represented as bags of words (BOW). That is to say a vector with length equal to the size of the vocabulary, with each position representing the count of the number of occurrences of a single word. This is much the same as a *one-hot* *vector* representing a word, but with every word in the sentence/document counted. The word embedding equivalent is sums of word embeddings (SOWE), and mean of word embeddings (MOWE). These methods, like BOW, lose all order information in the representation. In many cases it is possible to recover a BOW from a much lower dimensional SOWE (**White2015BOWgen**).

Surprisingly, these unordered methods have been found on many tasks to be extremely well performing, better than several of the more advanced techniques discussed later in this chapter. This has been noted in several works including: **White2015SentVecMeaning**, **RitterPosition** and **rui2017mvrusemantic**. It has been suggested that this is because in English there are only a few likely ways to order any given bag of words. It has been noted that given a simple n-gram language model the original sentences can often be recovered from BOW (**Horvat2014**) and thus also from a SOWE (**White2016a**). Thus word-order may not in-fact be as important as one would expect in many natural language tasks, as it is in practice more proscribed than one would expect. That is to say very few sentences with the same word content, will in-practice be able to have it rearranged for a very different meaning. However, this is unsatisfying, and certainly cannot capture fine grained meaning.

The step beyond this is to encode the n-grams into a bag of words like structure. This is a bag of n-grams (BON), e.g. bag of trigrams. Each index in the vector thus represents the occurrence of an n-gram in the text. So `It is a good day today`, has the trigrams: `(It is a)`,`(is a good)`,`(a good day)`, `(good day today)`. As is obvious for all but the most pathological sentences, recovering the full sentence order from a bag of n-grams is possible even without a language model.

The natural analogy to this with word embeddings might seem to be to find n-gram embeddings by the concatenation of n word embeddings; and then to sum these. However, such a sum is less informative than it might seem. As the sum in each concatenated section is equal to the others, minus the edge words.

Instead one should train an n-gram embedding model directly. The method discussed in Chapter 1, can be adapted to use n-grams rather than words as the basic token. This was explored in detail by (**li2017neural**). Their model is based on the skip-gram word embedding method. They take as input an n-gram embedding, and attempt to predict the surrounding n-grams. This reduces to the original skip-gram method for the case of unigrams. Note that the surrounding n-grams will overlap in words (for $n > 1$) with the input n-gram. As the overlap is not complete, this task remains difficult enough to encourage useful information to be represented in the embeddings. **li2017neural** also consider training n-gram embeddings as a bi-product of text classification tasks.

3.1.2 Paragraph Vector Models (Defunct)

le2014distributed introduced two models for representing documents of any length by using augmented word-embedding models. The models are called Paragraph Vector Distributed Memory (PV-DM) model, and the Paragraph Vector Distributed Bag of Words model (PV-DBOW). The name Paragraph Vector is a misnomer, it function on texts of any length and has most often (to our knowledge) been applied to documents and sentences rather than any in-between structures. The CBOW and skip-gram models are are extended with an additional context vector that represents the current document (or other super-word structure, such as sentence or paragraph). This, like the word embeddings, is initialised randomly, then trained during the task. **le2014distributed** considered that the context vector itself must contain useful information about the context. The effect in both cases of adding a context vector is to allow the network to learn a mildly different accusal language model depending on the context. To do this, the context vector would have to learn a representation for the context.

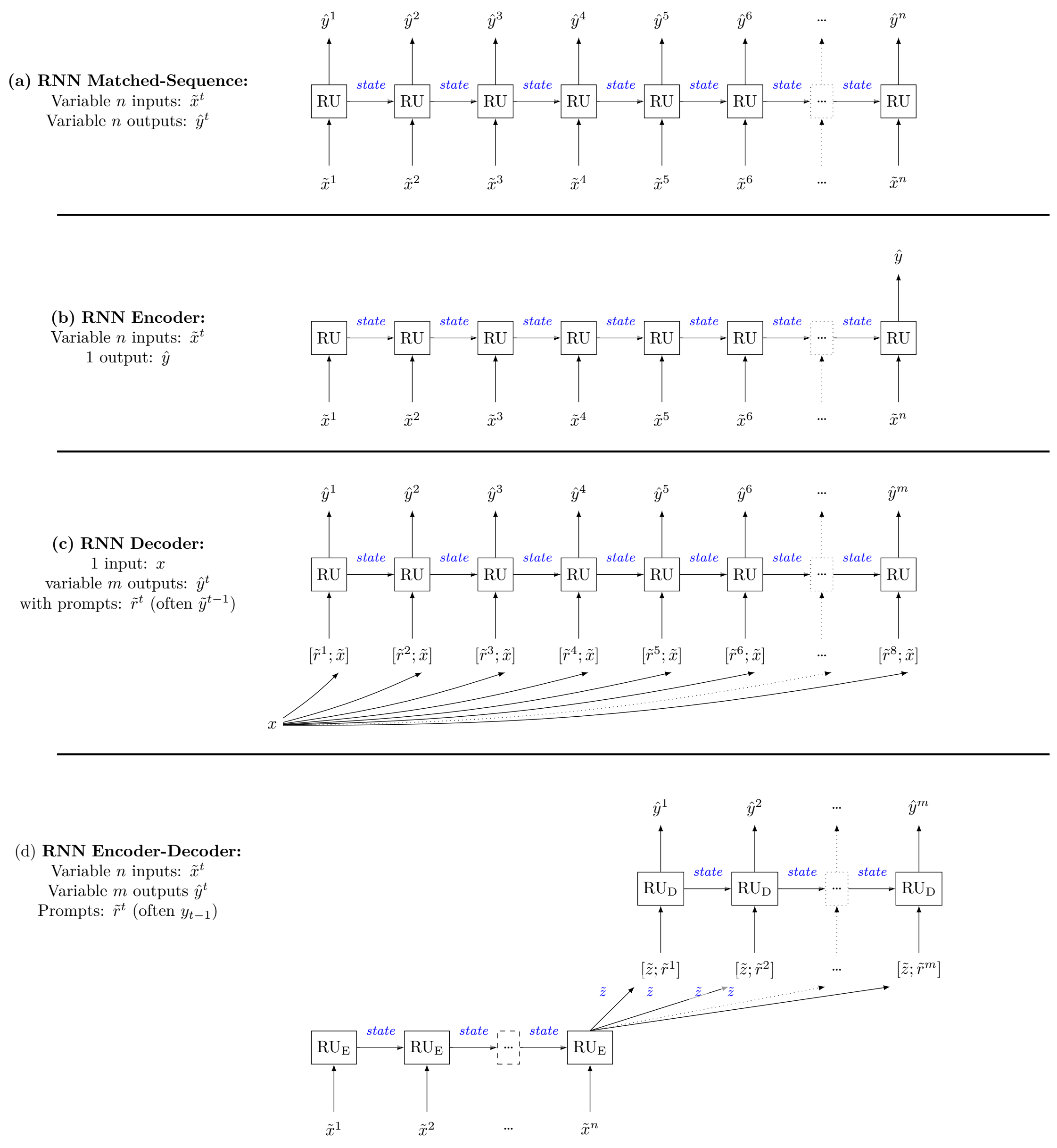
PV-DBOW is an extension of CBOW. The inputs to the model are not only the word-embedding $C_{:,w_j}$ for the words w^j from the window, but also a context-embedding $D_{:,d^k}$ for its current context (sentence, paragraph or document) d^k. The task remains to predict which word was the missing word from the center of the context w^i.

$$\begin{aligned}
&P(w^i \mid d^k,\, w^{i-\frac{n}{2}}, ..., w^{i-1}, w^{i+1}, ..., w^{i+\frac{n}{2}}) \\
&\qquad = \mathrm{smax}(W D_{:,d^k} + U \sum_{j=i+1}^{j=\frac{n}{2}} \left(C_{:,w^{i-j}} + C_{:,w^{i+j}}\right)) \qquad (3.1)
\end{aligned}$$

PV-DM is the equivalent extension for skip-grams. Here the input to the model is not only the central word, but also the context vector. Again, the task remains to predict the other words from the window.

$$P(w^j \mid d^k,\, w^i) = \left[\mathrm{smax}(W D_{:,d^k} + V\, C_{:,w^i})\right]_{w_j} \qquad (3.2)$$

Figure 3.1: The unrolled structure of an RNN for use in (a) Matched-sequence (b) Encoding, (c) Decoding and (d) Encoding-Decoding (sequence-to-sequence) problems. RU is the recurrent unit – the neural network which reoccurs at each time step.



(a) RNN Matched-Sequence:
Variable n inputs: $\tilde{x}^t$
Variable n outputs: $\hat{y}^t$
$\hat{y}^1$ $\hat{y}^2$ $\hat{y}^3$ $\hat{y}^4$ $\hat{y}^5$ $\hat{y}^6$ ... $\hat{y}^n$
RU
state
...
$\tilde{x}^1$ $\tilde{x}^2$ $\tilde{x}^3$ $\tilde{x}^4$ $\tilde{x}^5$ $\tilde{x}^6$ ... $\tilde{x}^n$
(b) RNN Encoder:
Variable n inputs: $\tilde{x}^t$
1 output: $\hat{y}$
$\hat{y}$
(c) RNN Decoder:
1 input: x
variable m outputs: $\hat{y}^t$
with prompts: $\tilde{r}^t$ (often $\tilde{y}^{t-1}$)
$\hat{y}^1$ $\hat{y}^2$ $\hat{y}^3$ $\hat{y}^4$ $\hat{y}^5$ $\hat{y}^6$ ... $\hat{y}^m$
$[\tilde{r}^1;\tilde{x}]$ $[\tilde{r}^2;\tilde{x}]$ $[\tilde{r}^3;\tilde{x}]$ $[\tilde{r}^4;\tilde{x}]$ $[\tilde{r}^5;\tilde{x}]$ $[\tilde{r}^6;\tilde{x}]$ ... $[\tilde{r}^8;\tilde{x}]$
x
(d) RNN Encoder-Decoder:
Variable n inputs: $\tilde{x}^t$
Variable m outputs $\hat{y}^t$
Prompts: $\tilde{r}^t$ (often y_{t-1})
$\hat{y}^1$ $\hat{y}^2$ ... $\hat{y}^m$
$\mathrm{RU_D}$
$[\tilde{z};\tilde{r}^1]$ $[\tilde{z};\tilde{r}^2]$ ... $[\tilde{z};\tilde{r}^m]$
$\tilde{z}$
$\mathrm{RU_E}$
$\tilde{x}^1$ $\tilde{x}^2$ ... $\tilde{x}^n$

The results of this work are now considered of limited validity. There were failures to reproduce the reported results in the original evaluations which were on sentiment analysis tasks. These were documented online by several users, including by the second author.[1] A follow up paper, **mesnil2014ensemble** found that reweighed bags of n-grams (**wang2012baselines**) out performed the paragraph vector models. Conversely, **lau2016doc2vecissues** found that on short text-similarity problems, with the right tuning, the paragraph vector models could perform well; however they did not consider the reweighed n-grams of (**wang2012baselines**). On a different short text task, **White2015SentVecMeaning** found the paragraph vector models to significantly be out-performed by SOWE, MOWE, BOW, and BOW with dimensionality reduction. This highlights the importance of rigorous testing against a suitable baseline, on the task in question.

3.2 Sequential Models

The majority of this section draws on the recurrent neural networks (RNN) as discussed in Chapter 2 of **??**. Every RNN learns a representation of all its input and output in its state. We can use

[1] `https://groups.google.com/forum/#!msg/word2vec-toolkit/Q49FIrNOQRo/DoRuBoVNFb0J`

RNN encoders and decoders (as shown in Figure 3.1) to generate representations of sequences by extracting a coding layer. One can take any RNN encoder, and select one of the hidden state layers after the final recurrent unit (RU) that has processed the last word in the sentence. Similarly for any RNN decoder, one can select any hidden state layer before the first recurrent unit that begins to produce words. For an RNN encoder-decoder, this means selecting the hidden layer from between. This was originally considered in **cho-EtAl:2014:EMNLP2014**, when using a machine translation RNN, to create embeddings for the translated phrases. Several other RNNs have been used in this way since.

3.2.1 VAE and encoder-decoder

Bowman2015SmoothGeneration presents an extension on this notion, where in-between the encode and the decode stages there is a variational autoencoder (VAE). This is shown in Figure 3.2. The variational autoencoder (**2014VAE**) has been demonstrated to have very good properties in a number of machine learning applications: they are able to work to find continuous latent variable distributions over arbitrary likelihood functions (such as in the neural network); and are very fast to train. Using the VAE, it is hoped that a better representation can be found for the sequence of words in the input and output.

Bowman2015SmoothGeneration trained the network as encoder-decoder reproducing its exact input. They found that short syntactically similar sentences were located near to each other according to this space, further to that, because it has a decoder, it can generate these nearby sentences, which is not possible for most sentence embedding methods.

Interestingly, they use the VAE output, i.e. the *code*, only as the state input to the decoder. This is in-contrast to the encoder-decoders of **cho-EtAl:2014:EMNLP2014**, where the *code* was concatenated to the input at every timestep of the decoder. **Bowman2015SmoothGeneration** investigated such a configuration, and found that it did not yield an improvement in performance.

3.2.2 Skip-thought

DBLP:journals/corr/KirosZSZTUF15 draws inspiration from the works on acausal language modelling, to attempt to predict the previous and next sentence. Like in the acausal language modelling methods, this task is not the true goal. Their true goal is to capture a good representation of the current sentence. As shown in Figure 3.3 they use an encoder-decoder RNN, with two decoder parts. One decoder is to produce the previous sentence. The encoder part takes as it's input is the current sentence, and produces as its output the code, which is input to the decoders. The other decoder is to produce the next sentence. As described in **??**, the prompt used for the decoders includes the previous word, concatenated to the code (from the encoder output).

That output code is the representation of the sentence.

3.3 Structured Models

The sequential models are limited to processed the information as a series of time-steps one after the other. They processes sentences as ordered lists of words. However, the actual structure of a natural language is not so simple. Linguists tend to break sentences down into a tree structure. This is referred to as parsing. The two most common forms are constituency parse trees, and dependency parse trees. Examples of each are shown in Figures 3.4 and 3.5. It is beyond the scope of this book to explain the precise meaning of these trees, and how to find them. The essence is that these trees represent the structure of the sentence, according to how linguists believe sentences are processed by humans.

The constituency parse breaks the sentence down into parts such as noun phrase (NP) and verb phrase (VP), which are in turn broken down into phrases, or (POS tagged) words. The constituency parse is well thought-of as a hierarchical breakdown of a sentence into its parts. Conversely, a dependency parse is better thought of as a set of binary relations between head-terms and their dependent terms. These structures are well linguistically motivated, so it makes sense to use them in the processing of natural language.

We refer here to models incorporating tree (or graph) structures as structural models. Particular variations have their own names, such as recursive neural networks (RvNN), and recursive autoencoders (RAE). We use the term structural model as an all encompassing term, and minimise the use of the easily misread terms: recursive vs recurrent neural networks. A sequential model

Figure 3.2: The VAE plus encoder-decoder of **Bowman2015SmoothGeneration**. Note that during training, $\hat{y}^i = w^i$, as it is an autoencoder model. As is normal for encoder-decoders the prompts are the previous output (target during training, predicted during testing): $r^i = y^{i-1}$, with $r^1 = y^0 =$ `<EOS>` being a pseudo-token marker for the end of the string. The Emb. step represents the embedding table lookup. In the diagrams for Chapter 1 we showed this as as a table but just as a block here for conciseness.

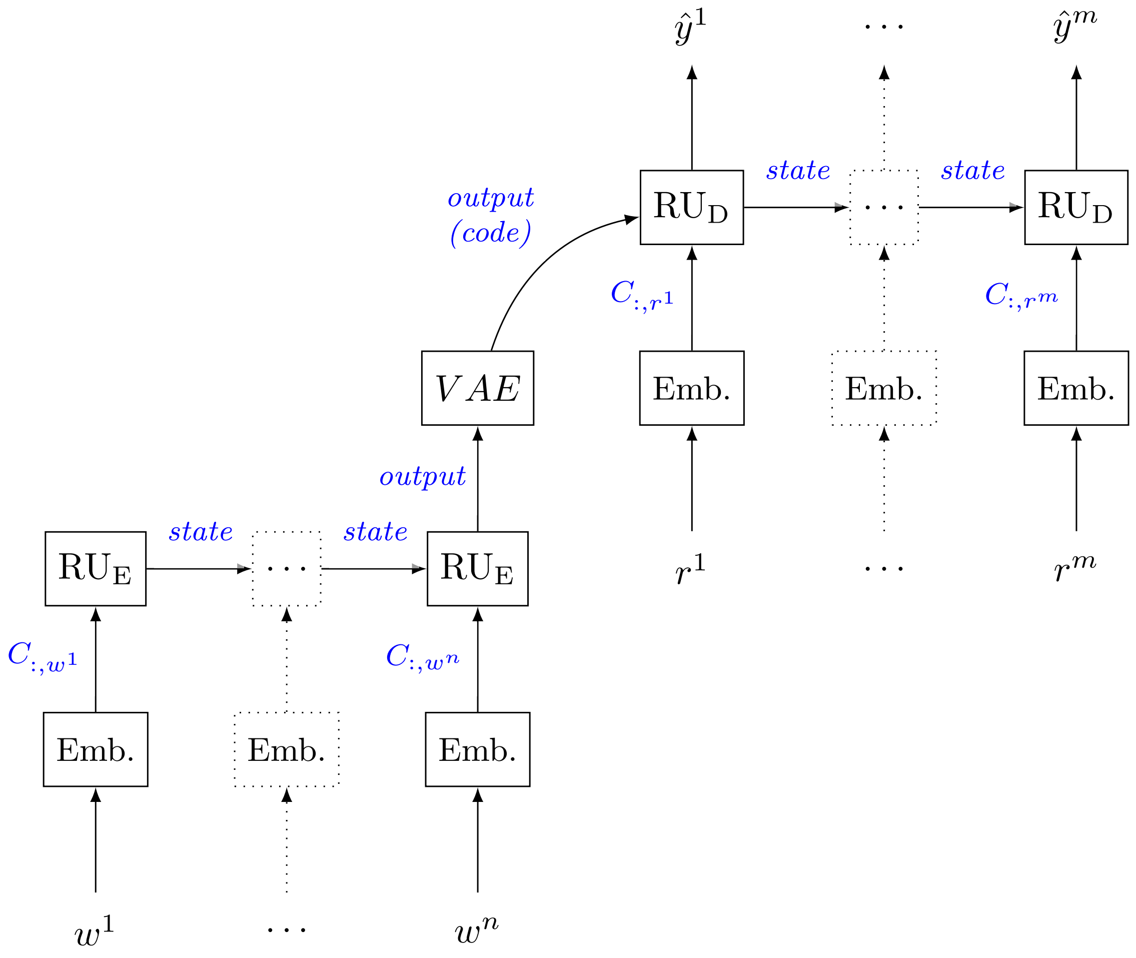


Figure 3.3: The skip-thought model (**DBLP:journals/corr/KirosZSZTUF15**). Note that for the next and previous sentences respectively the outputs are $\hat{q}^i$ and $\hat{p}^i$, and the prompts are q^{i-1} and p^{i-1}. As there is no intent to use the decoders after training, there is no need to worry about providing an evaluation-time prompt, so the prompt is always the previous word. $p^0 = p^{m^{\mathrm{P}}} = q^0 = q^{m^{\mathrm{q}}} =$ `<EOS>` being a pseudo-token marker for the end of the string. The input words are w^i, which come from the current sentence. the Emb. steps represents the look-up of the embedding for the word.

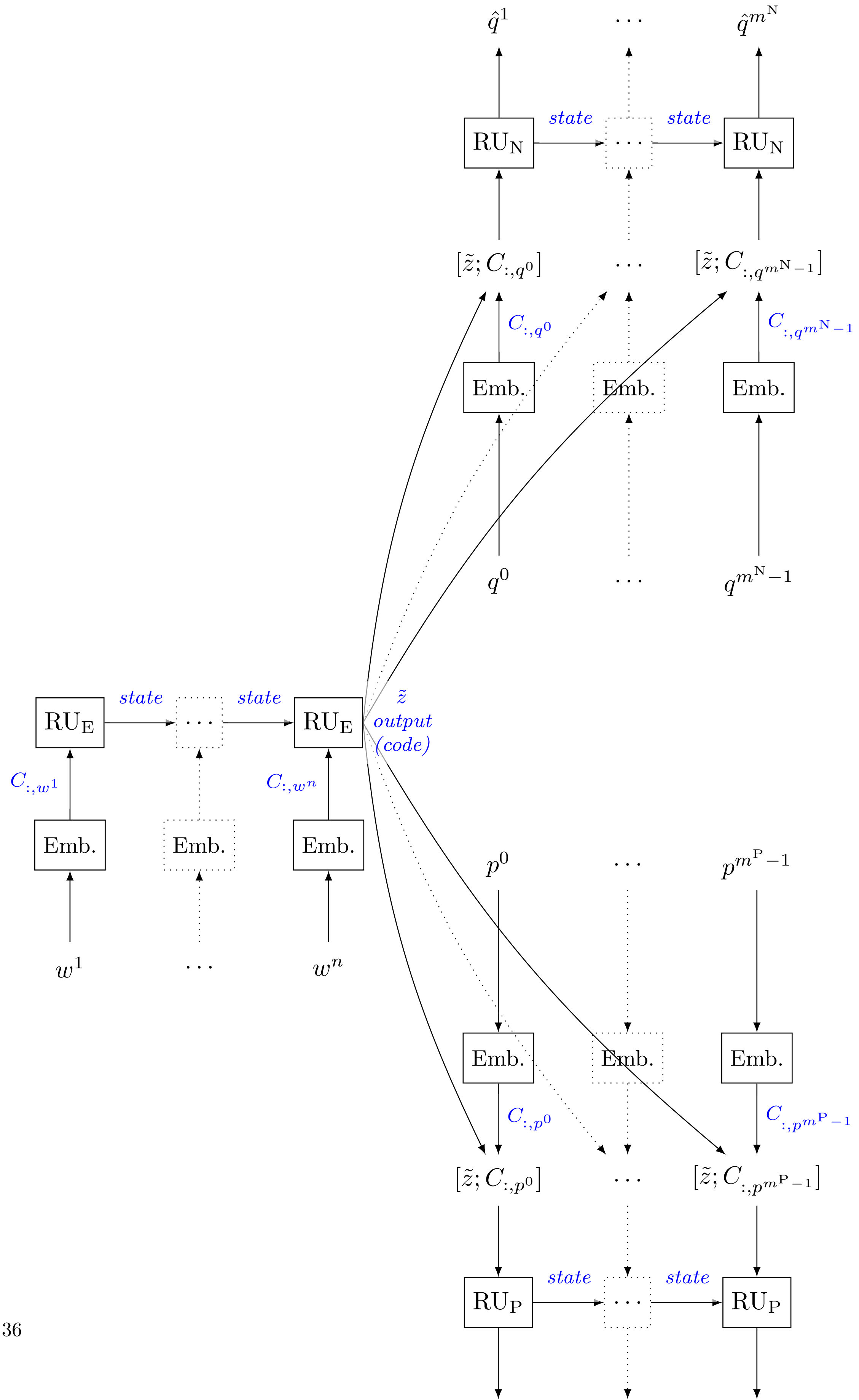
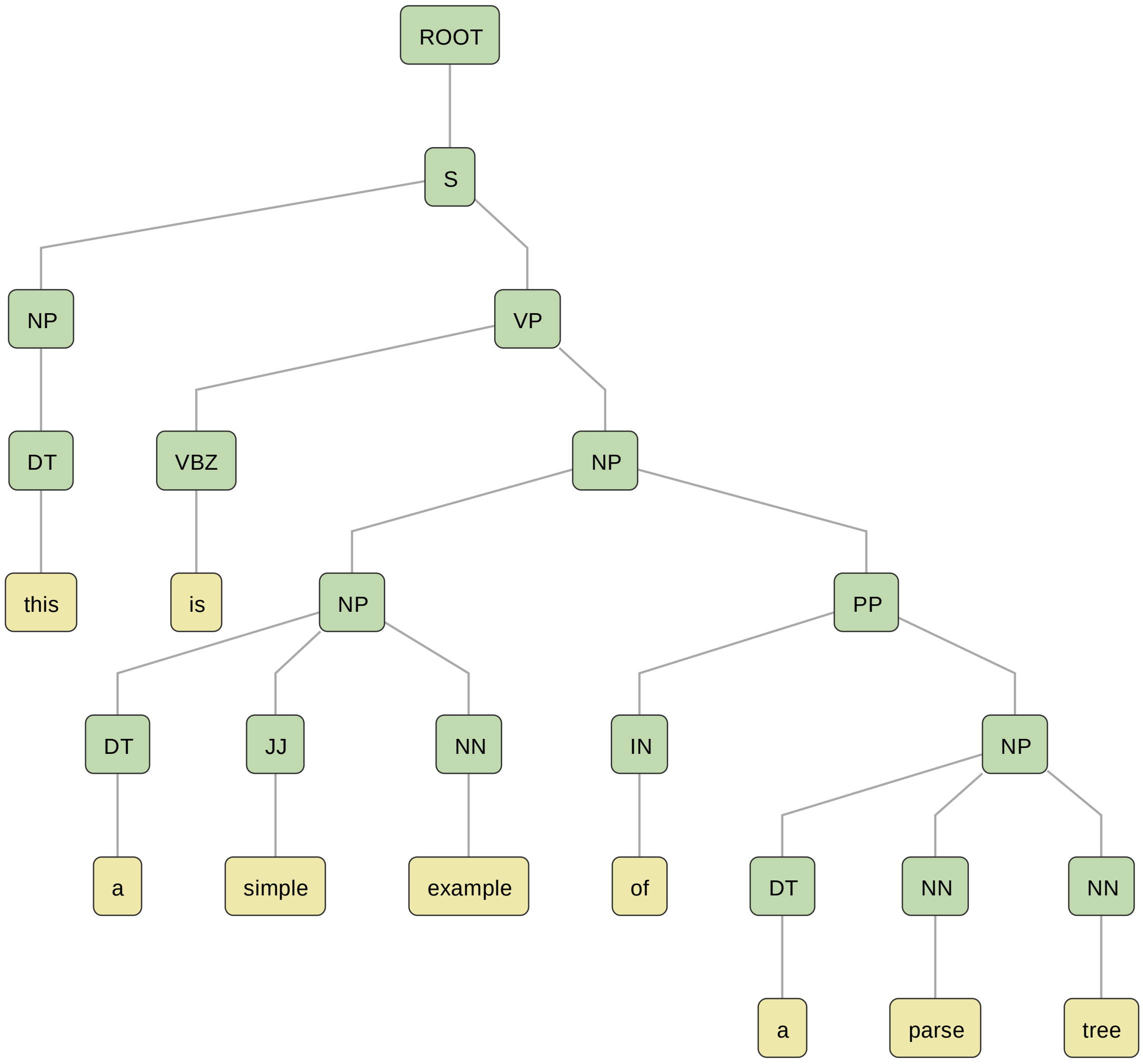


Figure 3.4: A constituency parse tree for the sentence: `This is a simple example of a parse tree`. In this diagram the leaf nodes are the input words, their intimidate parents are their POS tags, and the other nodes with multiple children represent sub-phrases of the sentence, for example NP is a Noun Phrase.



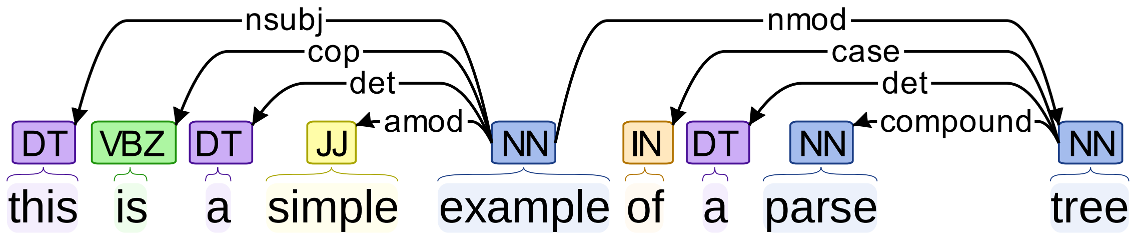
(an RNN) is a particular case of a structural model, just as a linked list is a particular type of tree. However, we will exclude sequential models them this discussion except where marked.

The initial work on structural models was done in the thesis of **socher2014recursive**. It builds on the work of **goller1996BPstructure** and **Pollack199077**, which present back-propagation through structure. Back-propagation can be applied to networks of any structure, as the chain-rule can be applied to any differentiable equation to find its derivative. Structured networks, like all other networks, are formed by the composition of differentiable functions, so are differentiable. In a normal network the same composition of functions is used for all input cases, whereas in a structured network it is allowed to vary based on the inputs. This means that structuring a network according to its parse tree is possible.

3.3.1 Constituency Parse Tree (Binary)

Tree structured networks work by applying a recursive unit (which we will call RV) function across pairs (or other groups) of the representations of the lower levels, to produce a combined representation. The network structure for an input of binary tree structured text is itself a binary

Figure 3.5: A dependency parse tree for the sentence `This is a simple example of a parse tree`, This flattened view may be misleading. `example` is at the peak of the tree, with direct children being: `this`,`is`,`a`,`simple`, and `tree`. `tree` has direct children being: `of`,`a`, and `parse`.



tree of RVs. Each RV (i.e. node in the graph) can be defined by the composition function:

$$f^{\mathrm{RV}}(\tilde{u},\tilde{v}) = \varphi\left(\begin{bmatrix} S & R \end{bmatrix}\begin{bmatrix} \tilde{u} \\ \tilde{v} \end{bmatrix} + \tilde{b}\right) \tag{3.3}$$

$$= \varphi\left(S\tilde{u} + R\tilde{v} + \tilde{b}\right) \tag{3.4}$$

where $\tilde{u}$ and $\tilde{v}$ are the left and right substructures embeddings (word embeddings at the leaf node level), and S and R are the matrices defining how the left and right children's representations are to be combined.

This is a useful form as all constituency parse trees can be converted into binary parse trees, via left-factoring or right factoring (adding new nodes to the left or right to take some of the children). This is sometimes called binarization, or putting them into Chomsky normal form. This form of structured network has been used in many words, including **socher2010PhraseEmbedding**, **SocherEtAl2011:RAE**, **SocherEtAl2011:PoolRAE**, **Socher2011ParsingPhrases** and **zhang2014BRAE**. Notice that S and R matrices are shared for all RVs, so all substructures are composed in the same way, based only on whether they are on the left, or the right.

3.3.2 Dependency Tree

The dependency tree is the other commonly considered parse-tree. Structured networks based upon the dependency tree have been used by Socher et al. (2014), **iyyer2014neural**, and **iyyer2014generating**. In these works rather than a using composition matrix for left-child and right-child, the composition matrix varies depending on the type of relationship of between the head word and its child. Each dependency relationship type has its own composition matrix. That is to say there are distinct composition matrices for each of `nsub`, `det`, `nmod`, `case` etc. This allows for multiple inputs to a single head node to be distinguished by their relationship, rather than their order. This is important for networks using a dependency parse tree structure as the relationship is significant, and the structure allows a node to have any number of inputs.

Consider a function $\pi(i,j)$ which returns the relationship between the head word at position i and the child word at position j. For example, using the tree shown in Figure 3.5, which has w^8 = `parse` and w^9 = `tree` then $\pi(8,9)$ = `compound`. This is used to define the composed representation for each RV:

$$f^{\mathrm{RV}}(i) = \varphi\left(W^{\mathrm{head}}C_{:,w^i} + \sum_{j\in\mathrm{children}(i)} W^{\pi(\mathrm{i,j})}\, f_{RV}(j) + \tilde{b}\right) \tag{3.5}$$

Here $C_{:,w^i}$ is the word embedding for w^i, and W^{head} encodes the contribution of the headword to the composed representation. Similarly, $W^{\pi(\mathrm{i,j})}$ encodes the contribution of the child words. Note that the terminal case is just $f_{RV}(i) = \varphi\left(W^{\mathrm{head}}C_{:,w^i} + \tilde{b}\right)$ when a node i has no children. This use of the relationship to determine the composition matrix, increases both the networks expressiveness, and also handles the non-binary nature of dependency trees.

A similar technique could be applied to constituency parse trees. This would be using the part of speech (e.g. VBZ, NN) and phrase tags (e.g. NP, VP) for the sub-structures to choose the weight matrix. This would, however, lose the word-order information when multiple inputs have the same tag. This would be the case, for example, in the right-most branch shown in Figure 3.4, where both `parse` and `tree` have the NN POS tag, and thus using only the tags, rather than the order would leave `parse tree` indistinguishable from `tree parse`. This is not a problem for the dependency parse, as word relationships unambiguously correspond to the role in the phrase's meaning. As such, allowing the dependency relationship to define the mathematical relationship, as encoded in the composition matrix, only enhances expressibility.

For even greater capacity for the inputs to control the composition, would be to allow every word be composed in a different way. This can be done by giving the child nodes there own composition matrices, to go with there embedding vectors. The composition matrices encode the relationship, and the operation done in the composition. So not only is the representation of the (sub)phrase determined by a relationship between its constituents (as represented by their embeddings), but the nature of that relationship (as represented by the matrix) is also determined by those same constituents. In this approach at the leaf-nodes, every word not only has a word vector, but also a word matrix. This is discussed in Section 3.4.

3.3.3 Parsing

The initial work for both contingency tree structured networks (**socher2010PhraseEmbedding**) and for dependency tree structured networks (**stenetorp2013transition**) was on the creation of parsers. This is actually rather different to the works that followed. In other works the structure is provided as part of the input (and is found during preprocessing). Whereas a parser must induce the structure of the network, from the unstructured input text. This is simpler for contingency parsing, than for dependency parsing.

When creating a binary contingency parse tree, any pair of nodes can only be merged if they are adjacent. The process described by **socher2010PhraseEmbedding**, is to consider which nodes are to be composed into a higher level structure each in turn. For each pair of adjacent nodes, an RV can be applied to get a merged representation. A linear scoring function is also learned, that takes a merged representation and determines how good it was. This is trained such that correct merges score highly. Hinge loss is employed for this purpose. The Hinge loss function works on similar principles to negative sampling (see the motivation given in Section 1.4.2). Hinge loss is used to cause the merges that occur in the training set to score higher than those that do not. To perform the parse, nodes are merged; replacing them with their composed representation; and the new adjacent pairing score is then recomputed. **socher2010PhraseEmbedding** considered both greedy, and dynamic programming search to determine the order of composition, as well as a number of variants to add additional information to the process. The dependency tree parser extends beyond this method.

Dependency trees can have child-nodes that do not correspond to adjacent words (non-projective language). This means that the parser must consider that any (unlinked) node be linked to any other node. Traditional transition-based dependency parsers function by iteratively predicting links (transitions) to add to the structure based on its current state. **stenetorp2013transition** observed that a composed representation similar to Equation (3.4), was an ideal input to a softmax classifier that would predict the next link to make. Conversely, the representation that is suitable for predicting the next link to make, is itself a composed representation. Note, that **stenetorp2013transition** uses the same matrices for all relationships (unlike the works discussed in Section 3.3.2). This is required, as the relationships must be determined from the links made, and thus are not available before the parse. **bowman2016fast**, presents a work an an extension of the same principles, which combines the parsing step with the processing of the data to accomplish some task, in their case detecting entailment.

3.3.4 Recursive Autoencoders

Recursive autoencoders are autoencoders, just as the autoencoder discussed in **??**, they reproduce their input. It should be noted that unlike the encoder-decoder RNN discussed in Section 3.2.1, they cannot be trivially used to generate natural language from an arbitrary embeddings, as they require the knowledge of the tree structure to unfold into. Solving this would be the inverse problem of parsing (discussed in Section 3.3.3).

The models presented in **SocherEtAl2011:PoolRAE** and **iyyer2014generating** are unfolding recursive autoencoders. In these models an identical inverse tree is defined above the highest node. The loss function is the sum of the errors at the leaves, i.e. the distance in vector space between the reconstructed words embeddings and the input word-embeddings. This was based on a simpler earlier model: the normal (that is to say, not unfolding) recursive autoencoder.

The normal recursive autoencoder, as used in **SocherEtAl2011:RAE** and **zhang2014BRAE** only performs the unfolding for a single node at a time during training. That means that it assesses how well each merge can individually be reconstructed, not the success of the overall reconstruction. This per merge reconstruction has a loss function based on the difference between the reconstructed embeddings and the inputs embeddings. Note that those inputs/reconstructions are not word embeddings: they are the learned merged representations, except when the inputs happen to be leaf node. This single unfold loss covers the auto-encoding nature of each merge; but does not give any direct assurances of the auto-encoding nature of the whole structure. However, it should be noted that while it is not trained for, the reconstruction components (that during training are applied only at nodes) can nevertheless be applied recursively from the top layer, to allow for full reconstruction.

Semi-supervised

In the case of all these autoencoders, except **iyyer2014generating**, a second source of information is also used to calculate the loss during training. The networks are being simultaneously trained to perform a task, and to regenerate their input. This is often considered as semi-supervised learning, as unlabelled data can be used to train the auto-encoding part (unsupervised) gaining a good representation, and the labelled data can be used to train the task output part (supervised) making that representation useful for the task. This is done by imposing an additional loss function onto the output of the central/top node.

- In **SocherEtAl2011:RAE** this was for sentiment analysis.
- In **SocherEtAl2011:PoolRAE** this was for paraphrase detection.
- In **zhang2014BRAE** this was the distance between embeddings of equivalent translated phrases of two RAEs for different languages.

The reconstruction loss and the supervised loss can be summed, optimised in alternating sequences, or the reconstructed loss can be optimised first, then the labelled data used for fine-tuning.

3.4 Matrix Vector Models

3.4.1 Structured Matrix Vector Model

SocherMVRNN proposed that each node in the graph should define not only a vector embedding, but a matrix defining how it was to be combined with other nodes. That is to say, each word and each phrase has both an embedding, and a composition matrix.

Consider this for binary constituency parse trees. The composition function is as follows:

$$f^{\mathrm{RV}}(\tilde{u}, \tilde{v}, U, V) = \varphi\left([S\ R][U\tilde{v}; V\tilde{u}] + \tilde{b}\right) \tag{3.6}$$

$$= \varphi\left(S\,U\tilde{v} + R\,V\tilde{u} + \tilde{b}\right) \tag{3.7}$$

$$F^{\mathrm{RV}}(U, V) = W\,[U; V] = W^{\mathrm{l}}U + W^{\mathrm{r}}V \tag{3.8}$$

f^{RV} gives the composed embedding, and F^{RV} gives the composing matrix. The S and R represent the left and right composition matrix components that are the same for all nodes (regardless of content). The U and V represent the per word/node child composition matrix components. We note that S and R could, in theory, be rolled in to U and R as part of the learning. The $\tilde{u}$ and $\tilde{v}$ represent the per word/node children embeddings, and W represents the method for merging two composition matrices.

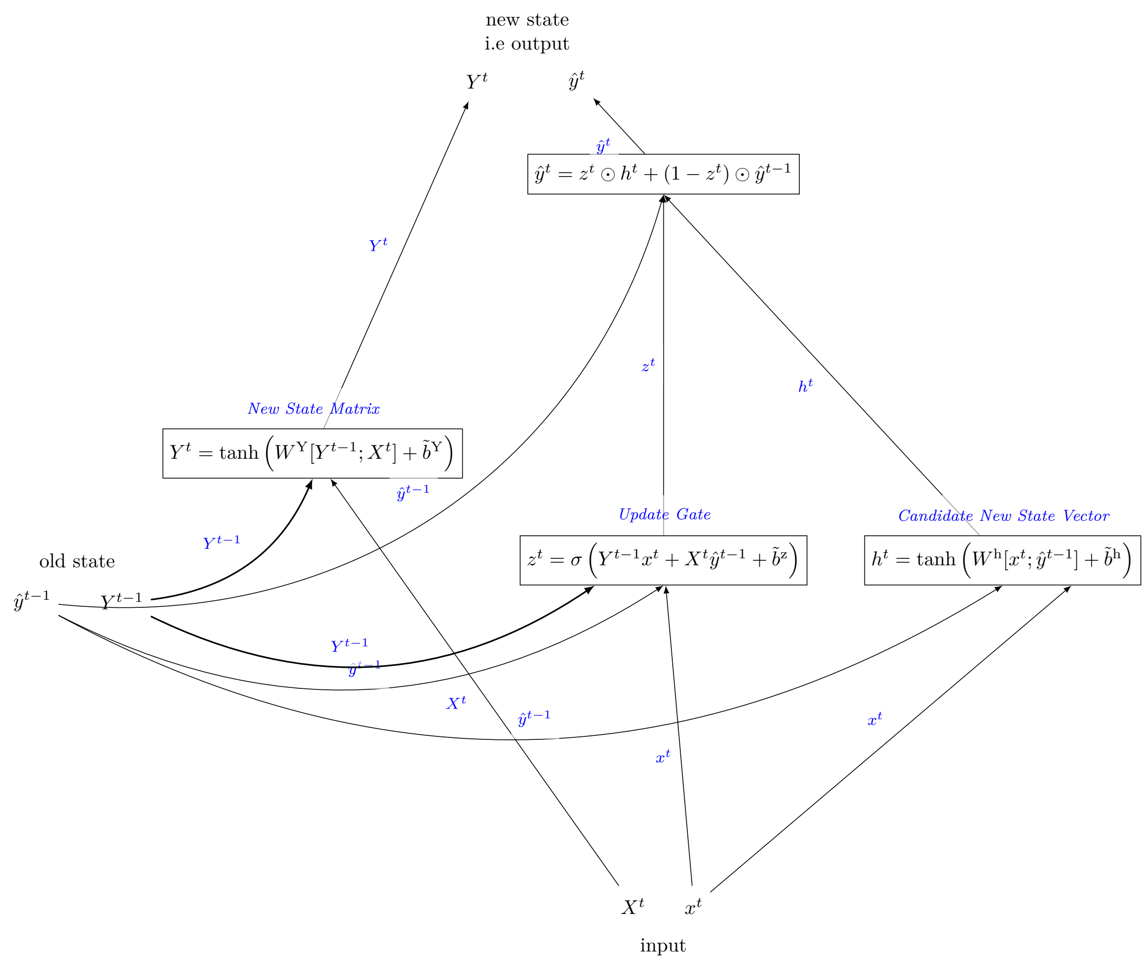
We note that one can define increasingly complex and powerful structured networks along these lines; though one does run the risk of very long training times and of over-fitting.

3.4.2 Sequential Matrix Vector Model

A similar approach, of capturing a per word matrix, was used on a sequential model by **rui2017mvrusemantic**. While sequential models are a special case of structured models, it should be noted that unlike the structured models discussed prior, this matrix vector RNN features a gated memory. This matrix-vector RNN is an extension of the GRU discussed in **??**, but without a reset gate.

In this sequential model, advancing a time step, is to perform a composition. This composition is for between the input word and the (previous) state. Rather than directly between two nodes in the network as in the structural case. It should be understood that composing with the state

Figure 3.6: A Matrix Vector recurrent unit



new state
i.e output
Y^t
$\hat{y}^t$
$\hat{y}^t$
$\hat{y}^t = z^t \odot h^t + (1 - z^t) \odot \hat{y}^{t-1}$
Y^t
z^t
h^t
New State Matrix
$Y^t = \tanh\left(W^{\mathrm{Y}}[Y^{t-1}; X^t] + \tilde{b}^{\mathrm{Y}}\right)$
$\hat{y}^{t-1}$
Update Gate
Candidate New State Vector
Y^{t-1}
old state
$z^t = \sigma\left(Y^{t-1}x^t + X^t\hat{y}^{t-1} + \tilde{b}^{\mathrm{z}}\right)$
$h^t = \tanh\left(W^{\mathrm{h}}[x^t; \hat{y}^{t-1}] + \tilde{b}^{\mathrm{h}}\right)$
$\hat{y}^{t-1}$
Y^{t-1}
Y^{t-1}
$\hat{y}^{t-1}$
X^t
$\hat{y}^{t-1}$
x^t
x^t
X^t
x^t
input

is not the same as composing the current input with the previous input. But rather as composing the current input with all previous inputs (though not equally).

As depicted in Figure 3.6 each word, w^t is represented by a word embedding $\tilde{x}^t$ and matrix: $\tilde{X}^{w^t}$, these are the inputs at each time step. The network outputs and states are the composed embedding $\hat{y}^t$ and matrix Y^t.

$$h^t = \tanh\left(W^{\mathrm{h}}[x^t; \hat{y}^{t-1}] + \tilde{b}^{\mathrm{h}}\right) \tag{3.9}$$

$$z^t = \sigma\left(Y^{t-1}x^t + X^t\hat{y}^{t-1} + \tilde{b}^{\mathrm{z}}\right) \tag{3.10}$$

$$\hat{y}^t = z^t \odot h^t + (1 - z^t) \odot \hat{y}^{t-1} \tag{3.11}$$

$$Y^t = \tanh\left(W^{\mathrm{Y}}[Y^{t-1}; X^t] + \tilde{b}^{\mathrm{Y}}\right) \tag{3.12}$$

The matrices W^{h}, W^{Y} and the biases $\tilde{b}^{\mathrm{h}}$, $\tilde{b}^{\mathrm{z}}$, $\tilde{b}^{\mathrm{Y}}$ are shared across all time steps/compositions. W^{Y} controls how the next state-composition Y^t matrix is generated from its previous value and the input composition matrix, X^t; W^{h} similarly controls the value of the candidate state-embedding h^t.

h^t is the candidate composed embedding (to be output/used as state). z_t is the update gate, it controls how much of the actual composed embedding ($\hat{y}^t$) comes from the candidate h^t and how much comes from the previous value ($\hat{y}^{t-1}$). The composition matrix Y^t (which is also part of the state/output) is not gated.

Notice, that the state composition matrix Y^{t-1} is only used to control the gate z^t, not to directly affect the candidate composited embedding h^t. Indeed, in fact one can note that all similarity to the structural method of **SocherMVRNN** is applied in the gate z^t. The method for calculating h^t is similar to that of a normal RU.

The work of **rui2017mvrusemantic**, was targeting short phrases. This likely explains the reason for not needing a forget gates. The extension is obvious, and may be beneficial when applying this method to sentences

3.5 Conclusion, on compositionality

It is tempting to think of the structured models as compositional, and the sequential models as non-compositional. However, this is incorrect.

The compositional nature of the structured models is obvious: the vector for a phrase is composed from the vectors of the words that the phrase is formed from.

Sequential models are able to learn the structures. For example, learning that a word from n time steps ago is to be remembered in the RNN state, to then be optimally combined with the current word, in the determination of the next state. This indirectly allows the same compositionality as the structured models. It has been shown that sequential models are indeed in-practice able to learn such relationships between words (**2017arXiv170909360W**). More generally as almost all aspects of language have some degree of compositionality, and sequential models work very well on most language tasks, this implicitly shows that they have sufficient representational capacity to learn sufficient degrees of compositional processing to accomplish these tasks.

In fact, it has been suggested that even some unordered models such as sum of word embeddings are able to capture some of what would be thought of as compositional information. **RitterPosition** devised a small corpus of short sentences describing containing relationships between the locations of objects. The task and dataset was constructed such that a model must understand some compositionality, to be able to classify which relationships were described. **RitterPosition** tested several sentence representations as the input to a naïve Bayes classifier being trained to predict the relationship. They found that when using sums of high-quality word embeddings as the input, the accuracy not only exceeded the baseline, but even exceeded that from using representation from a structural model. This suggests that a surprising amount of compositional information is being captured into the embeddings; which allows simple addition to be used as a composition rule. Though it being ignorant of word order does mean it certainly couldn't be doing so perfectly, however the presence of other words my be surprisingly effective hinting at the word order (**White2016a**), thus allow for more apparently compositional knowledge to be encoded than is expected.

To conclude, the compositionality capacity of many models is not as clear cut as it may initially seem. Further to that the requirement for a particular task to actually handle compositional reasoning is also not always present, or at least not always a significant factor in practical applications. We have discussed many models in this section, and their complexity varies significantly. They range from the very simple sum of word embeddings all the way to the the structured matrix models, which are some of the more complicated neural networks ever proposed.