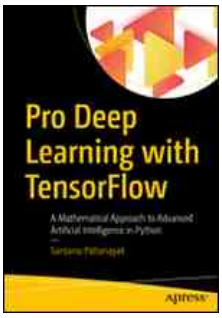

Pro Deep Learning with TensorFlow: A Mathematical Approach to Advanced Artificial Intelligence in Python

by Santanu Pattanayak

Chapter 4: Natural Language Processing Using Recurrent Neural Networks

S. Pattanayak, *Pro Deep Learning with TensorFlow*, https://doi.org/10.1007/978-1-4842-3096-1_4

Overview

In the modern age of information and analytics, natural language processing (NLP) is one of the most important technologies out there. Making sense of complex structures in language and deriving insights and actions from it is crucial from an artificial-intelligence perspective. In several domains, the importance of natural language processing is paramount and ever growing, as digital information in the form of language is ubiquitous. Applications of natural language processing include language translation, sentiment analysis, web search applications, customer service automation, text classification, topic detection from text, language modeling, and so forth. Traditional methods of natural language processing relied on the Bag of Word models, the Vector Space of Words model, and on-hand coded knowledge bases and ontologies. One of the key areas for natural language processing is the syntactic and semantic analysis of language. Syntactic analysis refers to how words are grouped and connected in a sentence. The main tasks in syntactic analysis are tagging parts of speech, detecting syntactic classes (such as verbs, nouns, noun phrases, etc.), and assembling sentences by constructing syntax trees. Semantics analysis refers to complex tasks such as finding synonyms, performing word-verb disambiguation, and so on.

Vector Space Model (VSM)

In NLP information-retrieval systems, a document is generally represented as simply a vector of the count of the words it contains. For retrieving documents similar to a specific document either the cosine of the angle or the dot product between the document and other documents is computed. The cosine of the angle between two vectors gives a similarity measure based on the similarity between their vector compositions. To illustrate this fact, let us look at two vectors $x, y \in \mathbb{R}^{2\times 1}$ as shown here:

$$x = \begin{bmatrix} 2 & 3 \end{bmatrix}^T$$
$$y = \begin{bmatrix} 4 & 6 \end{bmatrix}^T$$

Although vectors x and *y* are different, their cosine similarity is the maximum possible value of 1. This is because the two vectors are identical in their component compositions. The ratio of the first component to the second component for both vectors is 2/3, and hence content-composition-wise they are treated as being similar. Hence, documents with high cosine similarity are generally considered similar in nature.

Let's say we have two sentences:

$$Doc1 = [\ The\ dog\ chased\ the\ cat\ ]$$

$$Doc2 = [\ The\ cat\ was\ chased\ down\ by\ the\ dog\ ]$$

The number of distinct words in the two sentences would be the vector-space dimension for this problem. The distinct words are *The, dog, chased, the, cat, down, by*, and *was*, and hence we can represent each document as an eight-dimensional vector of word counts.

$$\begin{matrix} & 'The' & dog' & chased' & the' & cat' & down' & by' & 'was' \end{matrix}$$

$$Doc1 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{8\times 1}$$

$$Doc2 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \in \mathbb{R}^{8\times 1}$$

If we represent *Doc*1 by v_1 and *Doc*2 by v_2, then the cosine similarity can be expressed as

$$\cos(v_1, v_2) = \frac{(v_1^T v_2)}{\|v_1\|\|v_2\|} = \frac{1\times 1+1\times 1+1\times 1+1\times 1+1\times 1}{\sqrt{5}\sqrt{8}} = \frac{5}{\sqrt{40}}$$

Where $||v_1||$ is the magnitude or the l^2 norm of the vector v_1.

As stated earlier, cosine similarity gives a measure of the similarity based on the component composition of each vector. If the components of the document vectors are in somewhat similar proportion, the cosine distance would be high. It doesn't take the magnitude of the vector into consideration.

In certain cases, when the documents are of highly varying lengths, the dot product between the document vectors is taken instead of the cosine similarity. This is done when, along with the content of the document, the size of the document is also compared. For instance, we can have a tweet in which the words *global* and *economics* might have word counts of 1 and 2 respectively, while a newspaper article might have words counts of 50 and 100 respectively for the same words. Assuming the other words in both documents have insignificant counts, the cosine similarity between the tweet and the newspaper article would be close to 1. Since the tweet sizes are significantly smaller, the word counts proportion of 1:2 for *global* and *economics* doesn't really compare to the proportion of 1:2 for these words in the newspaper article. Hence, it doesn't really make sense to assign such a high similarity measure to these documents for several applications. In that case, taking the dot product as a similarity measure rather than the cosine similarity helps since it scales up the cosine similarity by the magnitude of the word vectors for the two documents. For comparable cosine similarities, documents with higher magnitudes would have higher dot product similarity since they have enough text to justify their word composition. The word composition for small texts might just be by chance and not be the true representation of its intended representation. For most applications where the documents are of comparable lengths, cosine similarity is a fair enough measure.

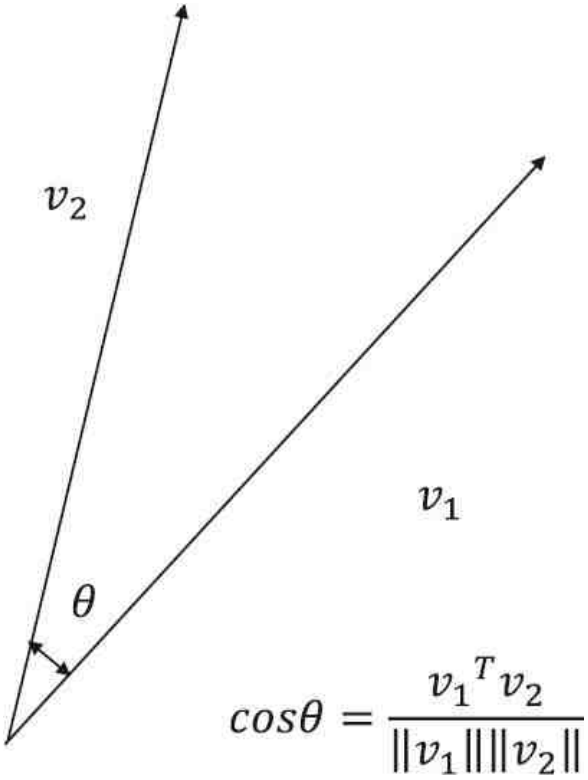

Figure 4-1: Cosine similarity between two word vectors

Figure 4-1 illustrates two vectors v_1 and v_2 with cosine similarity as the cosine of the angle θ between them.

At times, it makes sense to work with the distance counterpart of the cosine similarity. The cosine distance is defined as the square of the Euclidean distance between the unit vectors in the direction of the original vectors for which the distance needs to be computed. For two vectors v_1 and v_2 at an angle of θ between them, the cosine distance is given by $2(1 - cos\theta)$.

This can be easily deduced by taking the square of the Euclidean distance between the unit vectors $u_1 = \frac{v_1}{\|v_1\|}$ and $u_2 = \frac{v_2}{\|v_2\|}$ as shown here:

$$\begin{aligned}\|u_1 - u_2\|^2 &= (u_1 - u_2)^T (u_1 - u_2) \\ &= u_1^T u_1 + u_2^T u_2 - 2u_1^T u_2 \\ &= \|u_1\|^2 + \|u_2\|^2 - 2\|u_1\|\|u_2\| cos\theta\end{aligned}$$

Now, u_1 and u_2 being unit vectors, their magnitudes $|u_1||$ and $|u_2||$ respectively are both equal to 1 and hence

$$\|u_1 - u_2\|^2 = 1+1-2cos\theta = 2(1-cos\theta)$$

Generally, when working with document term-frequency vectors, the raw term/word counts are not taken and instead are normalized by how frequently the word is used in the corpus. For example, the term *the* is a frequently occurring word in any corpus, and it's likely that the term has a high count in two documents. This high count for *the* is likely to increase the cosine similarity, whereas we know that the term is a frequently occurring word in any corpus and should contribute very little to document similarity. The count of such words in the document term vector is penalized by a factor called *inverse document frequency*.

For a term word *t* occurring *n* times in a document *d* and occuring in *N* documents out of *M* documents in the corpus, the normalized count after applying inverse document frequency is

$$Normalized\ count = (Term\ freqency) \times (Inverse\ document\ frequency)$$

$$= n log\left(\frac{M}{N}\right)$$

As we can see, as *N* increases with respect to M, the $\log\left(\frac{M}{N}\right)$ component diminishes until it's zero for *M* = *N*. So, if a word is highly popular in the corpus then it would not contribute much to the individual document term vector. A word that has high frequency in the document but is less frequent across the corpus would contribute more to the document term vector. This normalizing scheme is popularly known as *tf -idf*, a short-form representation for term frequency inverse document frequency. Generally, for practical purposes, the *(N* +1) is taken as a denominator to avoid zeros that make the *log* function undefined. Hence, the inverse document frequency can be rephrased as $\log\left(\frac{M}{N+1}\right)$

Normalizing schemes are even applied to the term frequency *n* to make it non-linear. A popular such normalizing scheme is BM25 where the document frequency contribution is linear for small values of *n* and then the contribution is made to saturate as *n* increases. The term frequency is normalized as follows in BM25:

$$BM25(n) = \frac{(k+1)n}{k+n}$$

where *k* is a parameter that provides different shapes for different values of *k* and one needs to optimize *k* based on the corpus.

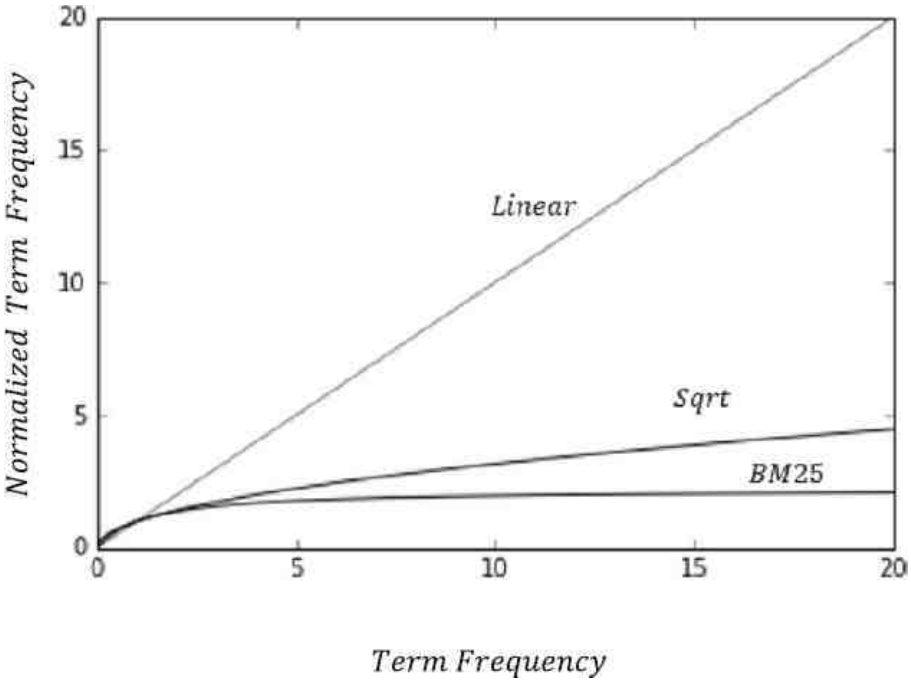

Figure 4-2: Normalized term frequency versus term frequency for different methods

In Figure 4-2, the normalized term frequency for different normalizing schemes has been plotted against the term frequency.

Square-root transformation makes the dependency sub-linear, whereas the BM25 plot for $k = 1.2$ is very aggressive and the curve saturates beyond a term frequency of 5. As stated earlier, the k can be optimized through cross-validation or other methods based on the problem needs.

Vector Representation of Words

Just as the documents are expressed as vectors of different word counts, a word in a corpus can also be expressed as a vector, with the components being counts the word has in each document.

Other ways of expressing words as vectors would be to have the component specific to a document set to 1 if the word is present in the document or zero if the word doesn't exist in the document.

$$\begin{array}{lcccccccccc} & \textit{'The'} & \textit{dog'} & \textit{chased'} & \textit{the'} & \textit{cat'} & \textit{down'} & \textit{by'} & \textit{'was'} & \\ \textit{Doc1} = [& 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0] & \in \mathbb{R}^{8\times 1} \\ \textit{Doc2} = [& 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1] & \in \mathbb{R}^{8\times 1} \end{array}$$

Reusing the same example the word *The* can be expressed as a two-dimensional vector $[1\ 1]^T$ in the corpus of two documents. In a huge corpus of documents, the dimensionality of the word vector would be large as well. Like document similarity, word similarity can be computed through either cosine similarity or dot product.

Another way to represent words in a corpus is to one-hot encode them. In that case, the dimensionality of each word would be the number of unique words in the corpus. Each word would correspond to an index that would be set to 1 for the word, and all other remaining entries would be set to 0. So, each would be extremely sparse. Even similar words would have entries set to 1 for different indexes, and hence any kind of similarity measure would not work.

To represent word vectors better so that the similarity of the words can be captured more meaningfully, and also to render less dimensionality to word vectors, Word2Vec was introduced.

Word2Vec

Word2Vec is an intelligent way of expressing a word as a vector by training the word against words in its neighborhood. Words that are contextually like the given word would produce high cosine similarity or dot product when their Word2Vec representations are considered.

Generally, the words in the corpus are trained with respect to the words in their neighborhood to derive the set of the Word2Vec representations. The two most popular methods of extracting Word2Vec representations are the CBOW (Continuous Bag of Words) method and Skip-Gram method. The core idea behind CBOW is expressed in Figure 4-3.

Continuous Bag of Words (CBOW)

The CBOW method tries to predict the center word from the context of the neighboring words in a specific window length. Let's look at the following sentence and consider a window of five as a neighborhood.

> *"The cat jumped over the fence and crosed the road"*

In the first instance, we will try to predict the word *jumped* from its neighborhood *The cat over the*. In the second instance, as we slide the window by one position, we will try to predict the word *over* from the neighboring words *cat jumped the fence*. This process would be repeated for the entire corpus.

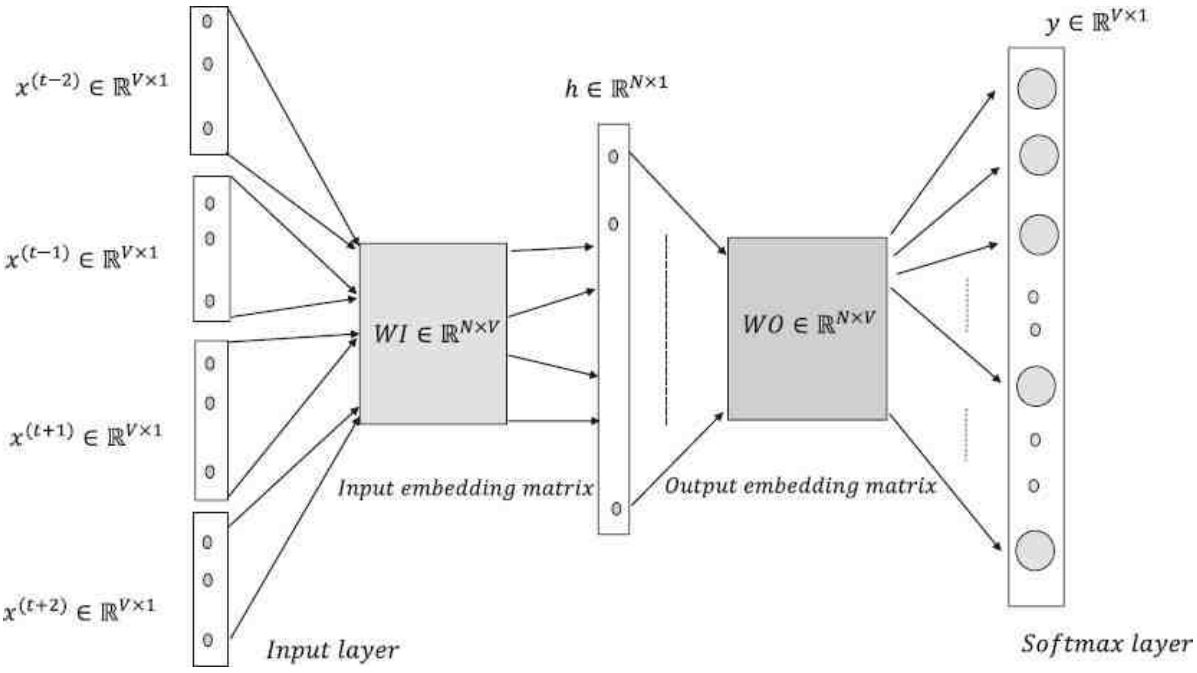

Figure 4-3: Continuous Bag of Words model for word embeddings

As shown in Figure 4-3, the Continuous Bag of Words model (CBOW) is trained on the context words as input and the center word as the output. The words in the input layer are expressed as one-hot encoded vectors where the component for the specific word is set to 1 and all other components are set to 0. The number of unique words *V* in the corpus determines the dimensionality of these one-hot encoded vectors, and hence $x^{(t)} \in \mathbb{R}^{V\times 1}$. Each one-hot encoded vector $x^{(t)}$ is multiplied by the input embedding matrix $WI \in \mathbb{R}^{N\times V}$ to extract the word-embeddings vector $u^{(k)} \in \mathbb{R}^{N\times 1}$ specific to that word. The index *k* in $u^{(k)}$ signifies that $u^{(k)}$ is the word embedded for the *kth* word in the vocabulary. The hidden layer vector *h* is the average of the input embedding vectors for all the context words in the window, and hence $h \in \mathbb{R}^{N\times 1}$ has the same dimension as that of the word-embedding vectors.

$$h = \frac{1}{l-1}\sum_{\substack{k=(t-2)\\ k\neq t}}^{(t+2)} (WI)x^{(k)}$$

where *l* is the length of the window size.

Just for clarity, let's say we have a six-word vocabulary—i.e., $V = 6$ —with the words being *cat, rat, chased, garden, the*, and *was*.

Let their one-hot encodings occupy the indexes in order so they can be represented as follows:

$$x_{cat} = \begin{bmatrix}1\\0\\0\\0\\0\\0\end{bmatrix} \quad x_{rat} = \begin{bmatrix}0\\1\\0\\0\\0\\0\end{bmatrix} \quad x_{chased} = \begin{bmatrix}0\\0\\1\\0\\0\\0\end{bmatrix} \quad x_{garden} = \begin{bmatrix}0\\0\\0\\1\\0\\0\end{bmatrix} \quad x_{the} = \begin{bmatrix}0\\0\\0\\0\\1\\0\end{bmatrix} \quad x_{was} = \begin{bmatrix}0\\0\\0\\0\\0\\1\end{bmatrix}$$

Let the input embedding where the embedding vector for each word is of dimensionality five be expressed as follows:

$$WI = \begin{array}{c} \begin{matrix} cat & rat & chased & garden & the & was \end{matrix} \\ \begin{bmatrix} 0.5 & 0.3 & 0.1 & 0.01 & 0.2 & 0.2 \\ 0.7 & 0.2 & 0.1 & 0.02 & 0.3 & 0.3 \\ 0.9 & 0.7 & 0.3 & 0.4 & 0.4 & 0.33 \\ 0.8 & 0.6 & 0.3 & 0.53 & 0.91 & 0.4 \\ 0.6 & 0.5 & 0.2 & 0.76 & 0.6 & 0.5 \end{bmatrix} \end{array}$$

Once we multiply the word-embedding matrix by the one-hot encoded vector for a word, we get the word-embedding vector for that word. Hence, by multiplying the one-hot vector for *cat* (i.e., x_{cat}) by the input embeddings matrix *WI*, one would get the first column of the *WI* matrix that corresponds to the cat, as shown here:

$$[WI][x_{cat}]$$

$$= \begin{bmatrix} 0.5 & 0.3 & 0.1 & 0.01 & 0.2 & 0.2 \\ 0.7 & 0.2 & 0.1 & 0.02 & 0.3 & 0.3 \\ 0.9 & 0.7 & 0.3 & 0.4 & 0.4 & 0.33 \\ 0.8 & 0.6 & 0.3 & 0.53 & 0.91 & 0.4 \\ 0.6 & 0.5 & 0.2 & 0.76 & 0.6 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.7 \\ 0.9 \\ 0.8 \\ 0.6 \end{bmatrix}$$

$\begin{bmatrix} 0.5 \\ 0.7 \\ 0.9 \\ 0.8 \\ 0.6 \end{bmatrix}$ is the word-embedding vector for the word *cat*.

Similarly, all the word-embedding vectors for the input words are extracted, and their average is the output of the hidden layer.

The output of the hidden layer *h* is supposed to represent the embedding of the target word.

All the words in the vocabulary have another set of word embedding housed in the output embedding matrix $WO \in \mathbb{R}^{V \times N}$. Let the word embeddings in *WO* be represented by $v^{(j)} \in \mathbb{R}^{N \times 1}$ where the index *j* denotes the *jth* word in the vocabulary in order as maintained in both the one-hot encoding scheme and the input embedding matrix.

$$WO = \begin{bmatrix} v^{(1)T} \rightarrow \\ v^{(2)T} \rightarrow \\ . \\ v^{(j)T} \rightarrow \\ . \\ v^{(V)T} \rightarrow \end{bmatrix}$$

The dot product of the hidden-layer embedding *h* is computed with each of the $v^{(j)}$ by multiplying the matrix *WO* by *h*. The dot product, as we know, would give a similarity measure for each of the output word embedding $v^{(j)} \forall j \in \{1,2,\ldots,N\}$ and the hidden-layer computed embedding h. The dot products are normalized to probability through a SoftMax and, based on the target word $w^{(t)}$, the categorical cross-entropy loss is computed and backpropagated through gradient descent to update the matrices' weights for both the input and output embedding matrices.

Input to the SoftMax layer can be expressed as follows:

$$[WO][h]=\begin{bmatrix} v^{(1)T} \rightarrow \\ v^{(2)T} \rightarrow \\ . \\ v^{(j)T} \rightarrow \\ . \\ v^{(V)T} \rightarrow \end{bmatrix}[h]=\begin{bmatrix} v^{(1)T}h & v^{(2)T}h \ldots. & v^{(j)T}h & v^{(V)T}h \end{bmatrix}$$

The SoftMax output probability for the *jth* word of the vocabulary $w^{(j)}$ given the context words is given by the following:

$$P\left(w=w^{(j)}/h\right)=p^{(j)}=\frac{e^{v^{(j)T}h}}{\sum_{k=1}^{V}e^{v^{(k)T}h}}$$

If the actual output is represented by a one-hot encoded vector $y=\begin{bmatrix} y_1 y_2 \ldots y_j \ldots y_n \end{bmatrix}^T \in \mathbb{R}^{V\times 1}$, where only one of the y_j is 1 (i.e., $\sum_{j=1}^{V} y_j = 1$), then the loss function for the particular combination of target word and its context words can be given by the following:

$$C=-\sum_{j=1}^{V} y_j \log\left(p^{(j)}\right)$$

The different $p^{(j)}$s are dependent on the input and output embeddings matrices' components, which are parameters to the cost function *C*. The cost function can be minimized with respect to these embedding parameters through backpropagation gradient-descent techniques.

To make this more intuitive, let's say our target variable is `cat`. If the hidden-layer vector *h* gives the maximum dot product with the outer matrix word-embeddings vector for `cat` while the dot product with the other outer word embedding is low, then the embedding vectors are more or less correct, and very little error or log loss will be backpropagated to correct the embedding matrices. However, let's say the dot product of *h* with `cat` is less and that of the other outer embedding vectors is more; the loss of the SoftMax is going to be significantly high, and thus more errors/log loss are going to be backpropagated to reduce the error.

Continuous Bag of Words Implementation in TensorFlow

The Continuous Bag of Words TensorFlow implementation has been illustrated in this section. The neighboring words within a distance of two from either side are used to predict the middle word. The output layer is a big SoftMax over the entire vocabulary. The word embedding vectors are chosen to be of size 128. The detailed implementation is outlined in Listing 4-1a. See also Figure 4-4.

Listing 4-1a: Continuous Bag of Words Implementation in TensorFlow

```
import numpy as np
import tensorflow as tf
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
%matplotlib inline

def one_hot(ind,vocab_size):
    rec = np.zeros(vocab_size)
    rec[ind] = 1
    return rec

def create_training_data(corpus_raw,WINDOW_SIZE = 2):
    words_list = []

    for sent in corpus_raw.split('.'):
    for w in sent.split():
        if w != '.':
            words_list.append(w.split('.')[0])    # Remove if delimiter is tied to the
```

```
                                                  end of a word

words_list = set(words_list)                      # Remove the duplicates for each word

word2ind = {}                                     # Define the dictionary for converting
                                                    a word to index
ind2word = {}                                     # Define dictionary for retrieving a
                                                    word from its index

vocab_size = len(words_list)                     # Count of unique words in the
                                                   vocabulary

for i,w in enumerate(words_list):                # Build the dictionaries
    word2ind[w] = i
    ind2word[i] = w

print word2ind
sentences_list = corpus_raw.split('.')
sentences = []

for sent in sentences_list:
    sent_array = sent.split()
    sent_array = [s.split('.')[0] for s in sent_array]
    sentences.append(sent_array)                # finally sentences would hold arrays of
                                                  word array for sentences

data_recs = []                                                # Holder for the input output
                                                                record
for sent in sentences:
    for ind,w in enumerate(sent):
        rec = []
        for nb_w in sent[max(ind - WINDOW_SIZE, 0) : min(ind + WINDOW_SIZE,
        len(sent)) + 1] :
            if nb_w != w:
                rec.append(nb_w)
            data_recs.append([rec,w])

x_train,y_train = [],[]

for rec in data_recs:
    input_ = np.zeros(vocab_size)
    for i in xrange(WINDOW_SIZE-1):
        input_ += one_hot(word2ind[ rec[0][i] ], vocab_size)
    input_ = input_/len(rec[0])
    x_train.append(input_)
    y_train.append(one_hot(word2ind[ rec[1] ], vocab_size))

return x_train,y_train,word2ind,ind2word,vocab_size
corpus_raw = "Deep Learning has evolved from Artificial Neural Networks, which has been
there since the 1940s. Neural Networks are interconnected networks of processing units
called artificial neurons that loosely mimic axons in a biological brain. In a biological
neuron, the dendrites receive input signals from various neighboring neurons, typically
greater than 1000. These modified signals are then passed on to the cell body or soma of
the neuron, where these signals are summed together and then passed on to the axon of the
neuron. If the received input signal is more than a specified threshold, the axon will
release a signal which again will pass on to neighboring dendrites of other neurons. Figure
2-1 depicts the structure of a biological neuron for reference. The artificial neuron units
are inspired by the biological neurons with some modifications as per convenience. Much
like the dendrites, the input connections to the neuron carry the attenuated or amplified
input signals from other neighboring neurons. The signals are passed on to the neuron, where
the input signals are summed up and then a decision is taken what to output based on the
total input received. For instance, for a binary threshold neuron an output value of 1 is
provided when the total input exceeds a pre-defined threshold; otherwise, the output stays
at 0. Several other types of neurons are used in artificial neural networks, and their
implementation only differs with respect to the activation function on the total input to
produce the neuron output. In Figure 2-2 the different biological equivalents are tagged in
the artificial neuron for easy analogy and interpretation."

corpus_raw = (corpus_raw).lower()
x_train,y_train,word2ind,ind2word,vocab_size= create_training_data(corpus_raw,2)

import tensorflow as tf
emb_dims = 128
learning_rate = 0.001

#-----------------------------------------------
# Placeholders for Input output
#-----------------------------------------------
```

```
x = tf.placeholder(tf.float32,[None,vocab_size])
y = tf.placeholder(tf.float32,[None,vocab_size])
#-------------------------------------------
# Define the Embedding matrix weights and a bias
#-------------------------------------------
W = tf.Variable(tf.random_normal([vocab_size,emb_dims],mean=0.0,stddev=0.02,dtype=tf.
float32))
b = tf.Variable(tf.random_normal([emb_dims],mean=0.0,stddev=0.02,dtype=tf.float32))
W_outer = tf.Variable(tf.random_normal([emb_dims,vocab_size],mean=0.0,stddev=0.02,dtype=tf.
float32))
b_outer = tf.Variable(tf.random_normal([vocab_size],mean=0.0,stddev=0.02,dtype=tf.float32))

hidden = tf.add(tf.matmul(x,W),b)
logits = tf.add(tf.matmul(hidden,W_outer),b_outer)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

epochs,batch_size = 100,10
batch = len(x_train)//batch_size

# train for n_iter iterations
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print 'was here'
    for epoch in xrange(epochs):
        batch_index = 0
        for batch_num in xrange(batch):
            x_batch = x_train[batch_index: batch_index +batch_size]
            y_batch = y_train[batch_index: batch_index +batch_size]
            sess.run(optimizer,feed_dict={x: x_batch,y: y_batch})
            print('epoch:',epoch,'loss :', sess.run(cost,feed_dict={x: x_batch,y: y_batch}))
    W_embed_trained = sess.run(W)

W_embedded = TSNE(n_components=2).fit_transform(W_embed_trained)
plt.figure(figsize=(10,10))
for i in xrange(len(W_embedded)):
    plt.text(W_embedded[i,0],W_embedded[i,1],ind2word[i])

plt.xlim(-150,150)
plt.ylim(-150,150)

--output--
('epoch:', 99, 'loss :', 1.0895648e-05)
```

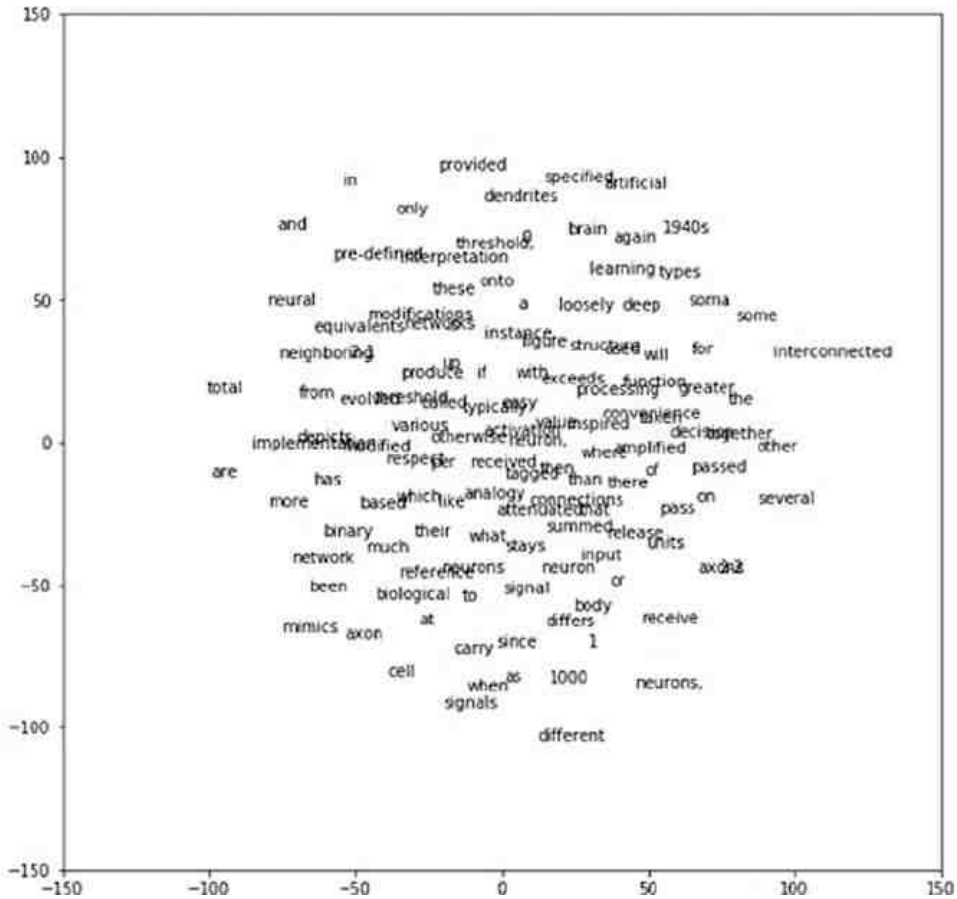

Figure 4-4: TSNE plot for the word-embeddings vectors learned from CBOW

The word embeddings learned have been projected to a 2D plane through the TSNE plot. The TSNE plot gives a rough idea of the neighborhood of a given word. We can see that the word-embeddings vectors learned are reasonable. For instance, the words *deep* and *learning* are very close to each other. Similarly, the words *biological* and *reference* are also very close to each other.

Skip-Gram Model for Word Embedding

Skip-gram models work the other way around. Instead of trying to predict the current word from the context words, as in Continuous Bag of Words, in Skip-gram models the context words are predicted based on the current word. Generally, given a current word, context words are taken in its neighborhood in each window. For a given window of five words there would be four context words that one needs to predict based on the current word. Figure 4-5 shows the high-level design of a Skip-gram model. Much like Continuous Bag of Words, in the Skip-gram model one needs to learn two sets of word embedding: one for the input words and one for the output context words. A Skip-gram model can be seen as a reversed Continuous Bag of Words model.

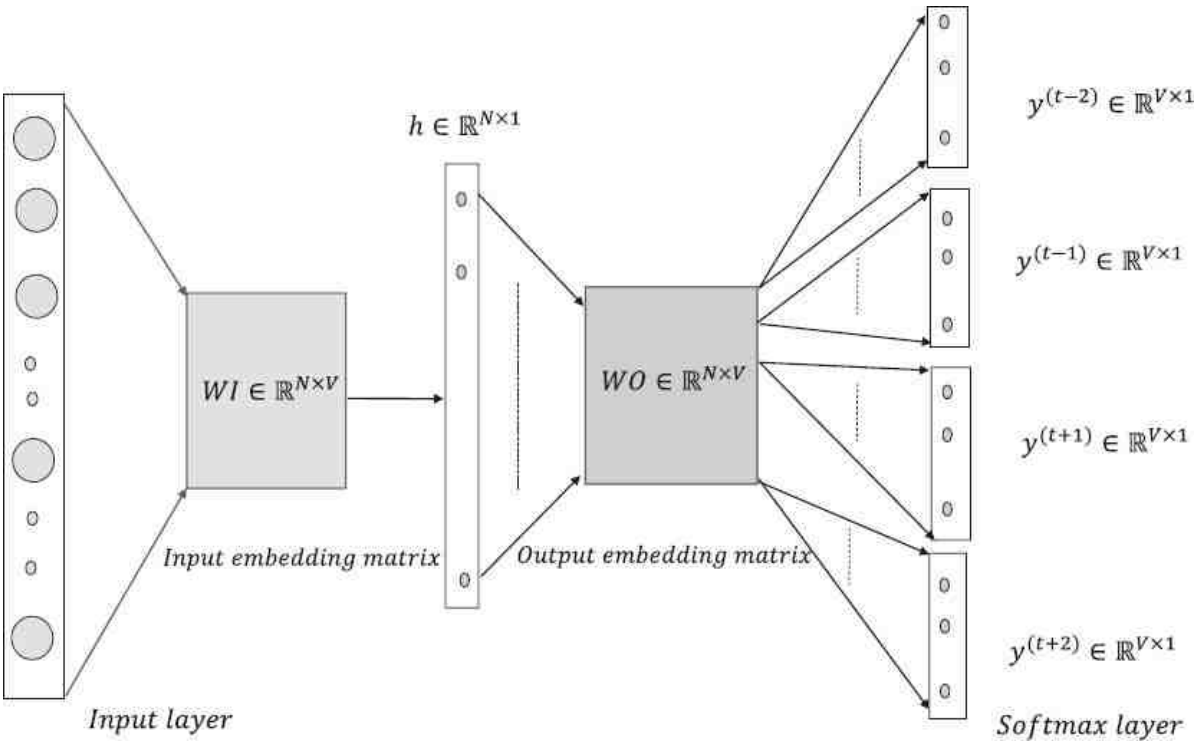

Figure 4-5: Skip-gram model for word embeddings

In the CBOW model the input to the model is a one-hot encoded vector $x^{(t)} \in \mathbb{R}^{V\times 1}$ for the current word, where *V* is the size of the vocabulary of the corpus. However, unlike CBOW, here the input is the current word and not the context words. The input $x^{(t)}$, when multiplied by the input word-embeddings matrix *WI*, produces the word-embedding vector $u^{(k)} \in \mathbb{R}^{N\times 1}$ given that $x^{(t)}$ represents the *kth* word in the vocabulary list. *N*, as before, represents the word-embeddings dimensionality. The hidden-layer output *h* is nothing but $u^{(k)}$.

The dot product of the hidden-layer output *h* is computed with every word vector $v^{(j)}$ of the outer embeddings matrix $WO \in \mathbb{R}^{V\times N}$ by computing [*WO*][*h*] just as in CBOW. However, instead of one SoftMax output layer, there are multiple SoftMax layers based on the number of context words that we are going to predict. For example, in Figure 4-5 there are four SoftMax output layers corresponding to the four context words. The input to each of these SoftMax layers is the same set of dot products in [*WO*][*h*] representing how similar the input word is to each word in the vocabulary.

$$[WO][h] = \left[v^{(1)T}h \;\; v^{(2)T}h \ldots. \;\; v^{(j)T}h \;\; v^{(V)T}h \right]$$

Similarly, all the SoftMax layers would receive the same set of probabilities corresponding to all the vocabulary words, with the probability of the *jth* word $w^{(j)}$ given the current or the center word $w^{(k)}$ being given by the following:

$$P\left(w = w^{(j)} / w = w^{(k)}\right) = p^{(j)} = \frac{e^{v^{(j)T}h}}{\sum_{k=1}^{V} e^{v^{(k)T}h}} = \frac{e^{v^{(j)T}u^{(k)}}}{\sum_{k=1}^{V} e^{v^{(k)T}u^{(k)}}}$$

If there are four target words, and their one-hot encoded vectors are represented by $y^{(t-2)}, y^{(t-1)}, y^{(t+1)}, y^{(t+2)} \in \mathbb{R}^{V\times 1}$, then the total loss function *C* for the word combination would be the summation of all four SoftMax losses as represented here:

$$C = -\sum_{\substack{m=t-2 \\ m\neq t}}^{t+2} \sum_{j=1}^{V} y_j^{(m)} \log\left(p^{(j)}\right)$$

Gradient descent using backpropagation can be used to minimize the cost function and derive the input and output embedding matrices' components.

Here are a few salient features about the Skip-gram and CBOW models:

- For Skip-gram models, the window size is not generally fixed. Given a maximum window size, the window size at each

current word is randomly chosen so that smaller windows are chosen more frequently than larger ones. With Skip-gram, one can generate a lot of training samples from a limited amount of text, and infrequent words and phrases are also very well represented.

- CBOW is much faster to train than Skip-gram and has slightly better accuracy for frequent words.
- Both Skip-gram and CBOW look at local windows for word co-occurrences and then try to predict either the context words from the center word (as with Skip-gram) or the center word from the context words (as with CBOW). So, basically, if we observe in Skip-gram that locally within each window the probability of the co-occurrence of the context word w_c and the current word w_t, given by $P(w_c/w_t)$, is assumed to be proportional to the exponential of the dot product of their word-embedding vectors. For example:

$$P(w_c / w_t) \propto e^{u^T v}$$

where u and v are the input and output word-embedding vectors for the current and context words respectively. Since the co-occurrence is measured locally, these models miss utilizing the global co-occurrence statistics for word pairs within certain window lengths. Next, we are going to explore a basic method to look at the global co-occurrence statistics over a corpus and then use SVD (singular value decomposition) to generate word vectors.

Skip-gram Implementation in TensorFlow

In this section, we will illustrate the Skip-gram model for learning word-vector embeddings with a TensorFlow implementation. The model is trained on a small dataset for easy representation. However, the model can be used to train large corpuses as desired. As illustrated in the Skip-gram section, the model is trained as a classification network. However, we are more interested in the word-embeddings matrix than in the actual classification of words. The size of the word embeddings has been chosen to be 128. The detailed code is represented in Listing 4-1b. Once the word-embeddings vectors are learned, they are projected via TSNE on a two-dimensional surface for visual interpretation.

Listing 4-1b: Skip-gram Implementation in TensorFlow

```
import numpy as np
import tensorflow as tf
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
%matplotlib inline

#-----------------------------------------------------------
# Function to one-hot encode the words
#-----------------------------------------------------------
def one_hot(ind,vocab_size):
    rec = np.zeros(vocab_size)
    rec[ind] = 1
    return rec

#-----------------------------------------------------------------------
# Function to create the training data from the corpus
#-----------------------------------------------------------------------
def create_training_data(corpus_raw,WINDOW_SIZE = 2):
    words_list = []

    for sent in corpus_raw.split('.'):
        for w in sent.split():
            if w != '.':
                words_list.append(w.split('.')[0])   # Remove if delimiter is tied to the
                                                       end of a word

    words_list = set(words_list)                     # Remove the duplicates for each word

    word2ind = {}                                    # Define the dictionary for converting
                                                       a word to index
    ind2word = {}                                    # Define dictionary for retrieving a
                                                       word from its index

    vocab_size = len(words_list)                     # Count of unique words in the vocabulary

    for i,w in enumerate(words_list):                # Build the dictionaries
        word2ind[w] = i
        ind2word[i] = w

    print word2ind
```

```
    sentences_list = corpus_raw.split('.')
    sentences = []

    for sent in sentences_list:
        sent_array = sent.split()
        sent_array = [s.split('.')[0] for s in sent_array]
        sentences.append(sent_array)              # finally sentences would hold arrays of
                                                    word array for sentences

    data_recs = []                                # Holder for the input output record

    for sent in sentences:
        for ind,w in enumerate(sent):
            for nb_w in sent[max(ind - WINDOW_SIZE, 0) : min(ind + WINDOW_SIZE,
            len(sent)) + 1] :
                if nb_w != w:
                    data_recs.append([w,nb_w])

    x_train,y_train = [],[]

    for rec in data_recs:
        x_train.append(one_hot(word2ind[ rec[0] ], vocab_size))
        y_train.append(one_hot(word2ind[ rec[1] ], vocab_size))

    return x_train,y_train,word2ind,ind2word,vocab_size

corpus_raw = "Deep Learning has evolved from Artificial Neural Networks which has been there
since the 1940s. Neural Networks are interconnected networks of processing units called
artificial neurons, that loosely mimics axons in a biological brain. In a biological neuron,
the Dendrites receive input signals from various neighboring neurons, typically greater
than 1000. These modified signals are then passed on to the cell body or soma of the neuron
where these signals are summed together and then passed on to the Axon of the neuron. If the
received input signal is more than a specified threshold, the axon will release a signal
which again will pass on to neighboring dendrites of other neurons. Figure 2-1 depicts the
structure of a biological neuron for reference.The artificial neuron units are inspired from
the biological neurons with some modifications as per convenience. Much like the dendrites
the input connections to the neuron carry the attenuated or amplified input signals from
other neighboring neurons. The signals are passed onto the neuron where the input signals
are summed up and then a decision is taken what to output based on the total input received.
For instance, for a binary threshold neuron output value of 1 is provided when the total
input exceeds a pre-defined threshold, otherwise the output stays at 0. Several other types
of neurons are used in artificial neural network and their implementation only differs

with respect to the activation function on the total input to produce the neuron output. In
Figure 2-2 the different biological equivalents are tagged in the artificial neuron for easy
analogy and interpretation."

corpus_raw = (corpus_raw).lower()
x_train,y_train,word2ind,ind2word,vocab_size= create_training_data(corpus_raw,2)

#-------------------------------------------------------------------------------
# Define TensorFlow ops and variable and invoke training
#-------------------------------------------------------------------------------
emb_dims = 128
learning_rate = 0.001
#-----------------------------------------------
# Placeholders for Input output
#-----------------------------------------------
x = tf.placeholder(tf.float32,[None,vocab_size])
y = tf.placeholder(tf.float32,[None,vocab_size])
#-----------------------------------------------
# Define the embedding matrix weights and a bias
#-----------------------------------------------
W = tf.Variable(tf.random_normal([vocab_size,emb_dims],mean=0.0,stddev=0.02,dtype=tf.
float32))
b = tf.Variable(tf.random_normal([emb_dims],mean=0.0,stddev=0.02,dtype=tf.float32))
W_outer = tf.Variable(tf.random_normal([emb_dims,vocab_size],mean=0.0,stddev=0.02,dtype=tf.
float32))
b_outer = tf.Variable(tf.random_normal([vocab_size],mean=0.0,stddev=0.02,dtype=tf.float32))

hidden = tf.add(tf.matmul(x,W),b)
logits = tf.add(tf.matmul(hidden,W_outer),b_outer)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

epochs,batch_size = 100,10
batch = len(x_train)//batch_size
```

```
# train for n_iter iterations
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print 'was here'
    for epoch in xrange(epochs):
        batch_index = 0
        for batch_num in xrange(batch):
            x_batch = x_train[batch_index: batch_index +batch_size]
            y_batch = y_train[batch_index: batch_index +batch_size]
            sess.run(optimizer,feed_dict={x: x_batch,y: y_batch})
            print('epoch:',epoch,'loss :', sess.run(cost,feed_dict={x: x_batch,y: y_batch}))
    W_embed_trained = sess.run(W)
W_embedded = TSNE(n_components=2).fit_transform(W_embed_trained)
plt.figure(figsize=(10,10))
for i in xrange(len(W_embedded)):
    plt.text(W_embedded[i,0],W_embedded[i,1],ind2word[i])
plt.xlim(-150,150)
plt.ylim(-150,150)

--output--

('epoch:', 99, 'loss :', 1.022735)
```

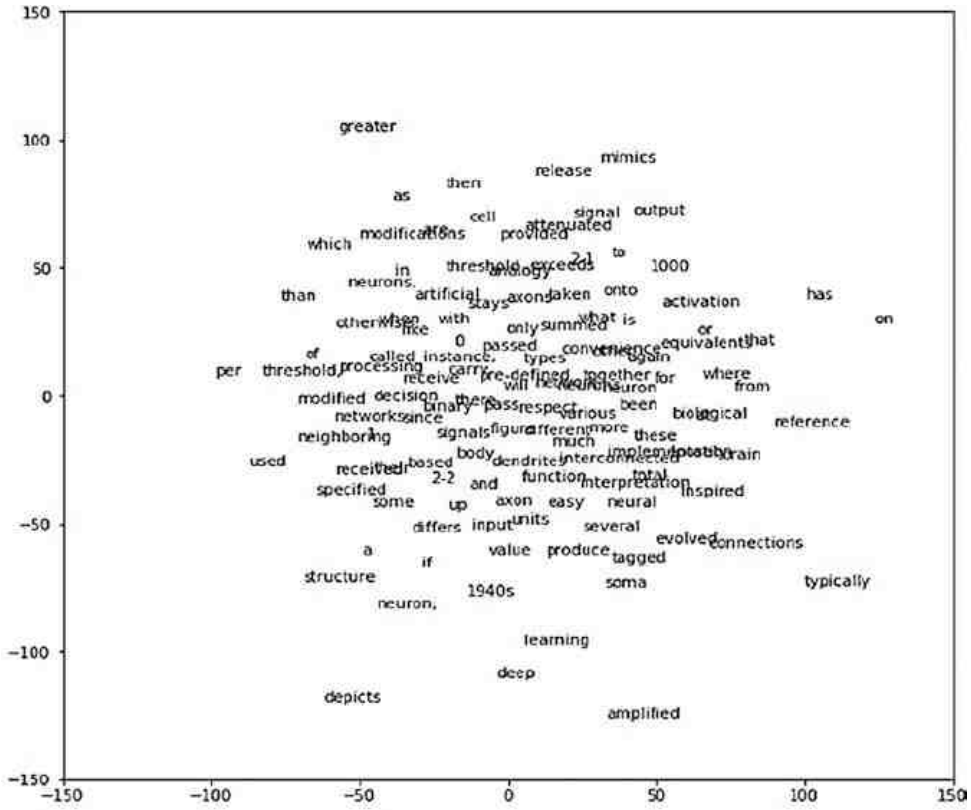

Figure 4-6: TSNE plot of word-embeddings vectors learned from Skip-gram model

Much like the word-embeddings vectors from the Continuous Bag of Words method, the embedding vectors learned from Skip-gram method are reasonable. For instance, the words *deep* and *learning* are very close to each other in Skip-grams too, as we can see from Figure 4-6. Also, we see other interesting patterns, such as the word *attenuated* being very close to the word *signal*.

Global Co-occurrence Statistics-based Word Vectors

The global co-occurrence methods, where the global counts of co-occurrences of words in each window over the whole corpus are collected, can be used to derive meaningful word vectors. Initially, we will look at a method that does matrix factorization through SVD (singular value decomposition) on the global co-occurrence matrix to derive a meaningful lower-dimensional representation of words. Later, we will look at the GloVe technique for word-vector representation, which combines the best of global co-occurrence statistics and the prediction methods of CBOW and/or Skip-gram to represent word vectors.

Let us consider a corpus:

'I like Machine Learning.'

'I like TensorFlow.'

'I prefer Python.'

We first collect the global co-occurrence counts for each word combination within a window of one. While processing the preceding corpus, we will get a co-occurrence matrix. Also, we make the co-occurrence matrix symmetric by assuming that whenever two words w_1 and w_2 appear together it would contribute to both probabilities $P(w_1/w_2)$ and $P(w_2/w_1)$, and hence we increment the count for both the count buckets $c(w_1/w_2)$ and $c(w_2/w_1)$ by one. The term $c(w_1/w_2)$ denotes the co-occurrence of the words w_1 and w_2, where w_2 acts as the context and w_1 as the word. For word-occurrence pairs, the roles can be reversed so that the context can be treated as the word and the word as the context. For this precise reason, whenever we encounter a co-occurring word pair (w_1, w_2), both count buckets $c(w_1/w_2)$ and $c(w_2/w_1)$ are incremented.

Coming to the incremental count, we need not always increment by 1 for a co-occurrence of two words. If we are looking at a window of *K* for populating the co-occurrence matrix, we can define a differential weighting scheme to provide more weight for words co-occurring at less distance from the context and penalize them as the distance increases. One such weighing scheme would be to increment the

co-occurrence counter by $\left(\frac{1}{k}\right)$, where *k* is the offset between the word and the context. When the word and the context are next to each other, then the offset is 1 and the co-occurrence counter can be incremented by 1, while when the offset is at maximum for a window of *K* the counter increment is at minimum at $\left(\frac{1}{k}\right)$.

In the SVD method of generating the word-vector embedding the assumption is that the global co-occurrence count $c(w_i/w_j)$ between a word w_i. and context w_j can be expressed as the dot product of the word-vector embeddings for the word w_i and for the context w_j. Generally, two sets of word embeddings are considered, one for the words and the other for the contexts. if $u_i \epsilon \mathbb{R}^{D\times 1}$ and $v_i \epsilon \mathbb{R}^{D\times 1}$ denote the word vector and the context vector for the *ith* word w_i in the corpus respectively, then the co-occurrence count can be expressed as follows:

$$c\left(w_i/w_j\right)=u_i^T v_j$$

Let's look at a three-word corpus and represent the co-occurrence matrix $X \in \mathbb{R}^{3\times 3}$ in terms of the dot products of the words and the context vectors. Further, let the words be $w_i, \forall i=\{1,2,3\}$ and their corresponding word vectors and context vectors be $u_i, \forall i=\{1,2,3\}$ and $v_i, \forall i=\{1,2,3\}$ respectively.

$$\begin{aligned} X &= \begin{bmatrix} c(w_1/w_1) & c(w_1/w_2) & c(w_1/w_3) \\ c(w_2/w_1) & c(w_2/w_2) & c(w_2/w_3) \\ c(w_3/w_1) & c(w_3/w_2) & c(w_3/w_3) \end{bmatrix} \\ &= \begin{bmatrix} u_1^T v_1 & u_1^T v_2 & u_1^T v_3 \\ u_2^T v_1 & u_2^T v_2 & u_2^T v_3 \\ u_3^T v_1 & u_3^T v_2 & u_3^T v_3 \end{bmatrix} \\ &= \begin{bmatrix} u_1^T \rightarrow \\ u_2^T \rightarrow \\ u_3^T \rightarrow \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} \\ &= [W][C] \end{aligned}$$

As we can see, the co-occurrence matrix turns out to be the product of two matrices, which are nothing but the word-vector embedding matrices for the words and contexts respectively. The word-vector embedding matrix $W \in \mathbb{R}^{3\times D}$ and the context-word embeddings matrix $C \in \mathbb{R}^{D\times 3}$, where D is the dimension of the word and context embeddings vectors.

Now that we know that the word co-occurrences matrix is a product of the word-vector embedding matrix and the context

embedding matrix, we can factorize the co-occurrence matrix by any applicable matrix factorization technique. Singular value decomposition (SVD) is a well-adopted method because it works, even if the matrices are not square or symmetric.

As we know from SVD, any rectangular matrix *X* can be decomposed into three matrices *U, Σ*, and *V* such that

$$X = [U][\Sigma][V^T]$$

The matrix *U* is generally chosen as the word-vector embeddings matrix *W* while ΣV^T is chosen as the context-vector embeddings matrix *C*, but there is no such restriction, and whichever works well on the given corpus can be chosen. One can very well chose W as $U\Sigma^{1/2}$ and C as $\Sigma^{1/2}V^T$. Generally, fewer dimensions in the data based on the significant singular values are chosen to reduce the size of *U*, Σ, and *V*. If $X \in \mathbb{R}^{m \times n}$, then $U \in \mathbb{R}^{m \times m}$. However, with truncated SVD we take only a few significant directions along which the data has maximum variability and ignore the rest as insignificant and/or noise. If we chose *D* dimensions, the new word-vector embeddings matrix $U' \in \mathbb{R}^{m \times D}$, where *D* is the dimension of every word-vector embedding.

The co-occurrence matrix $X \in \mathbb{R}^{m \times n}$ is generally obtained in a more generalized setting by making a pass through the entire corpus once. However, since the corpus might get new documents or contents over time, those new documents or contents can be processed incrementally. Figure 4-7 below illustrates the derivation of the word vectors or word embeddings in a three-step process.

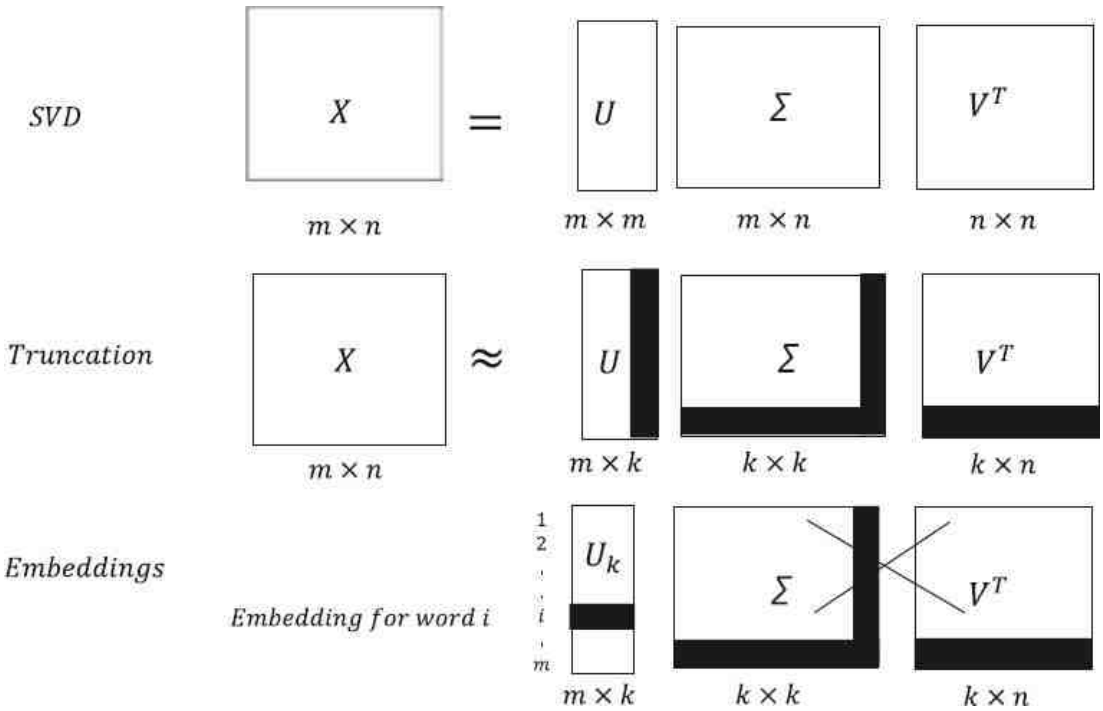

Figure 4-7: Extraction of word embeddings through SVD of word co-occurrence matrix

- In the first step, singular value decomposition (SVD) is performed on the co-occurrence matrix $X \in \mathbb{R}^{m \times n}$ to produce $U \in \mathbb{R}^{m \times m}$, which contains the left singular vectors, $\Sigma \in \mathbb{R}^{m \times n}$, which contains the singular values, and $V \in \mathbb{R}^{n \times n}$, which contains the right singular vectors.

$$[X]_{m \times n} = [U]_{m \times m}[\Sigma]_{m \times n}[V^T]_{n \times n}$$

- Generally, for word-to-word co-occurrence matrices, the dimensions *m* and *n* should be equal. However, sometimes instead of expressing words by words, words are expressed by contexts, and hence for generalization we have taken separate *m* and *n*.

- In the second step, the co-occurrence matrix is approximated by taking only *k* significant singular values from *Σ* that explain the maximum variance in data and by also choosing the corresponding *k* left singular and right singular vectors in *U* and *V*.

If we start with $U=\begin{bmatrix}u_1 u_2 & u_3 \ldots u_m\end{bmatrix}$, $\Sigma=\begin{bmatrix}\sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_m\end{bmatrix}$, $V^T=\begin{bmatrix}v_1^T \rightarrow \\ v_2^T \rightarrow \\ .. \\ v_n^T \rightarrow\end{bmatrix}$

after truncation, we would have

$U'=\begin{bmatrix}u_1 u_2 & u_3 \ldots u_k\end{bmatrix}$, $\Sigma'=\begin{bmatrix}\sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_k\end{bmatrix}$, $V'^T=\begin{bmatrix}v_1^T \rightarrow \\ v_2^T \rightarrow \\ .. \\ v_k^T \rightarrow\end{bmatrix}$

- In the third step, Σ and V'^T are discarded and the matrix $U' \in \mathbb{R}^{m \times k}$ is taken to the word-embeddings vector matrix. The word vectors have *k* dense dimensions corresponding to the chosen *k* singular values. So, from a sparse co-occurrence matrix we get a dense representation of the word-vector embeddings. There would be *m* word embeddings corresponding to each word of the processed corpus.

Mentioned in Listing 4-1c is the logic for building word vectors from a given corpus by factorizing the co-occurrence matrix of different words through SVD. Accompanying the listing is the plot of the derived word-vector embeddings in Figure 4-8.

Listing 4-1c

```
import numpy as np

corpus = ['I like Machine Learning.','I like  TensorFlow.','I prefer Python.']

corpus_words_unique = set()

corpus_processed_docs = []
for doc in corpus:
    corpus_words_ = []
    corpus_words = doc.split()
    print corpus_words
    for x in corpus_words:
        if len(x.split('.')) == 2:
            corpus_words_ += [x.split('.')[0]] + ['.']
        else:
            corpus_words_ += x.split('.')
    corpus_processed_docs.append(corpus_words_)
    corpus_words_unique.update(corpus_words_)
corpus_words_unique = np.array(list(corpus_words_unique))

co_occurence_matrix = np.zeros((len(corpus_words_unique),len(corpus_words_unique)))
for corpus_words_ in corpus_processed_docs:
    for i in xrange(1,len(corpus_words_)) :

        index_1 = np.argwhere(corpus_words_unique == corpus_words_[i])
        index_2 = np.argwhere(corpus_words_unique == corpus_words_[i-1])

        co_occurence_matrix[index_1,index_2] += 1
        co_occurence_matrix[index_2,index_1] += 1

U,S,V = np.linalg.svd(co_occurence_matrix,full_matrices=False)
print 'co_occurence_matrix follows:'
print  co_occurence_matrix
import matplotlib.pyplot as plt
for i in xrange(len(corpus_words_unique)):
    plt.text(U[i,0],U[i,1],corpus_words_unique[i])
plt.xlim((-0.75,0.75))
plt.ylim((-0.75,0.75))
plt.show()

    --output--


co_occurence_matrix follows:
[[ 0.  2.  0.  0.  1.  0.  0.  1.]
 [ 2.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.]
```

```
[ 1.  0.  0.  0.  0.  0.  1.  0.]
[ 0.  0.  1.  1.  0.  0.  0.  0.]
[ 0.  0.  0.  1.  1.  0.  0.  0.]
[ 1.  0.  0.  1.  0.  0.  0.  0.]]

Word-Embeddings Plot
```

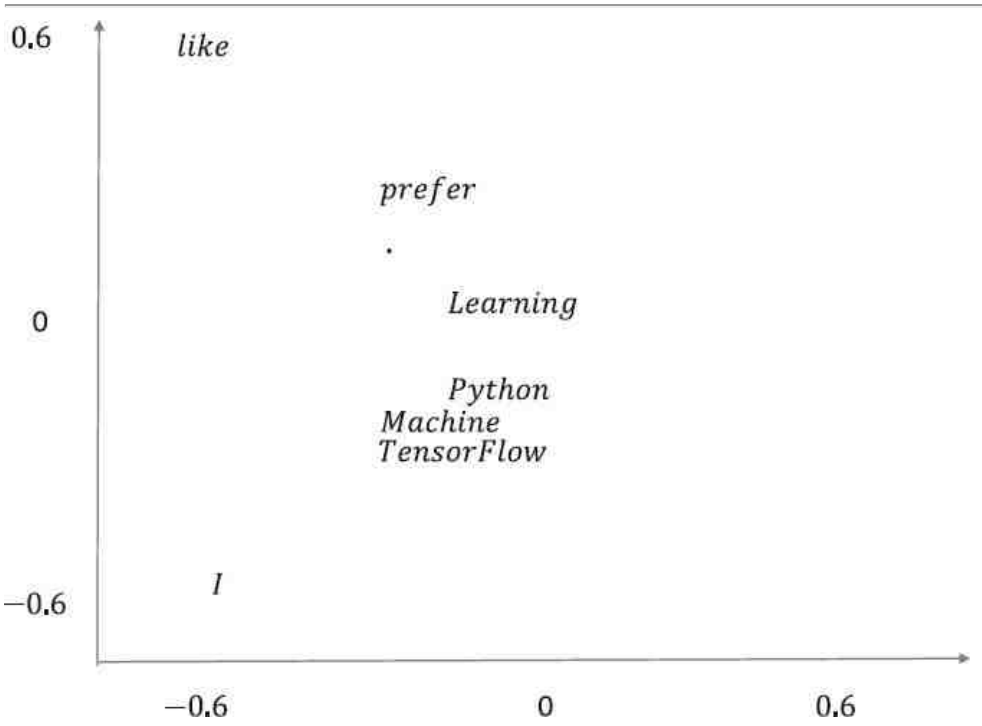

Figure 4-8: Word-embeddings plot

We can see a clear pattern in the plot of the word-vectors embedding in Figure 4-8, even with this small corpus. Here are a few of the findings:

- Common words like *I* and *like* are far away from others.
- *Machine, Learning, Python* and *Tensorflow*, being associated with different areas of learning, are clustered close to each other.

Next, we move on to global vectors, commonly known as GloVe, for generating word-vector embeddings.

GloVe

GloVe is a pre-trained, readily available word embedding vectors library from Stanford University. The training method for GloVe is significantly different from those for CBOW and Skip-gram. Instead of basing predictions on local-running windows for words, GloVe uses global word-to-word co-occurrence statistics from a corpus to train the model and derive the GloVe vectors. GloVe stands for *global vectors*. Pre-trained GloVe word embeddings are available at https://nlp.stanford.edu/projects/glove/. Jeffrey Pennington, Richard Socher, and Christopher D. Manning are the inventors of GloVe vectors, and they have documented GloVe vectors in their paper titled "GloVe: Global Vectors for Word Representation." The paper can be located at https://nlp.stanford.edu/pubs/glove.pdf.

Like SVD methods, GloVe looks at the global co-occurrence statistics, but the relation of the word and context vectors with respect to the co-occurrences count is a little different. If there are two words w_i and w_j and a context word w_k, then the ratio of the probabilities $P(w_k/w_i)$ and $P(w_k/w_j)$ provide more information than the probabilities themselves.

Let's consider two words, w_i = "*garden*" and w_j = *"market"*, and a couple of context words, $w_k \in$ {*"plants*","*shops*"}.The individual co-occurrences probability might be low; however, if we take the ratio of the co-occurrences probabilities, for instance

$$\frac{P(w_k = "plants" / w_i = "garden")}{P(w_k = "plants" / w_j = "market")}$$

The preceding ratio is going to be much greater than one indicating that *plants* are much more likely to be associated with *garden* than with *market*.

Similarly, let's consider w_k = "*shops*" and look at the ratio of the following co-occurrence probability:

$$\frac{P(w_k = "shops" / w_i = "garden")}{P(w_k = "shops" / w_j = "market")}$$

In this case, the ratio is going to be very small, signifying that the word *shop* is much more likely to be associated with the word *market* than with *garden*.

Hence, we can see that the ratio of the co-occurrence probabilities provides much more discrimination between words. Since we are trying to learn word vectors, this discrimination should be encoded by the difference of the word vectors, as in a linear vector space that's the best way to represent the discrimination between vectors. Similarly, the most convenient way to represent the similarity between vectors in a linear vector space is to consider their dot product, and hence the co-occurrence probability would be well represented by some function of the dot product between the word and the context vector. Taking all this into consideration helps one derive the logic for the GloVe vector derivation on a high level.

If u_i, u_j are the word-vector embeddings for the words w_i and w_j, and v_k is the context vector for word w_k, then the ratio of the two co-occurrence probabilities can be expressed as some function of the difference of the word vectors (i.e., $(u_i - u_j)$) and the context vector v_k. A logical function should work on the dot product of the difference of the word vector and the context vector, primarily because it preserves the linear structure between the vectors before it is manipulated by the function. Had we not taken the dot product, the function could have worked on the vectors in a way that would have disrupted the linear structure. Based on the preceding explanation, the ratio of the two co-occurrence probabilities can be expressed as follows:

(4.1.1) $$\frac{P(w_k / w_i)}{P(w_k / w_j)} = f\left(\left(u_i - u_j\right)^T v_k\right)$$

Where *f* is a given function that we seek to find out.

Also, as discussed, the co-occurrence probability $P(w_k / w_i)$ should be encoded by some form of similarity between vectors in a linear vector space, and the best operation to do so is to represent the co-occurrence probability by some function *g* of the dot product between the word vector w_i and context vector w_k, as expressed here:

(4.1.2) $$P\left(\frac{w_k}{w_i}\right) = g\left(u_i^T v_k\right)$$

Combining (4.1.1) and (4.1.2), we have

(4.1.3) $$\frac{g\left(u_i^T v_k\right)}{g\left(u_j^T v_k\right)} = f\left(\left(u_i - u_j\right)^T v_k\right)$$

Now the task is to determine meaningful functions *f* and *g* for the preceding equation to make sense. If we choose *f* and *g* to be the exponential function, it enables the ratio of the probabilities to encode the difference of the word vectors and at the same time keeps the co-occurrence probability dependent on the dot product. The dot product and difference of vectors keep the notion of similarity and discrimination of vectors in a linear space. Had the functions *f* and *g* been some kind of kernel functions then the measure of similarity and dissimilarity wouldn't have been restricted to a linear vector space and would have made the interpretability of word vectors very difficult.

Replacing *f* and *g* with the exponential function in (4.1.3), we get

$$\frac{e^{\left(u_i^T v_k\right)}}{e^{u_j^T v_k}} = e^{\left(u_i - u_j\right)^T v_k}$$

which gives us

(4.1.4) $$P(w_k / w_i) = e^{u_i^T v_k} \Rightarrow \log P(w_k / w_i) = u_i^T v_k$$

Interested readers with some knowledge of group theory in abstract algebra can see that the function $f(x) = e^x$ has been chosen so as to define a group homomorphism between the groups $(\mathbb{R},+)$ and $(\mathbb{R}>0,\times)$.

The co-occurrence probability of the word w_i and the context word w_k can be denoted as follows:

(4.1.5)
$$P(w_k/w_i)=\frac{c(w_i,w_k)}{c(w_i)}$$

where $c(w_i, w_k)$ denotes the co-occurrence count of word w_k with the context word w_k and $c(w_i)$ denotes the total occurences of the word w_i. The total count of any word can be computed by summing up its co-occurrences count with all other words in the corpus, as shown here:

$$c(w_i)=\sum_k c(w_i,w_k)$$

Combining (4.1.4) and (4.1.5), we get

$$\log c(w_i,w_k)-\log c(w_i)=u_i^T v_k$$

log $C(W_i)$ can be expressed as a bias b_i for the word w_i, and an additional bias $\tilde{b}_k$ for the word w_k is also introduced to make the equation symmetric. So, the final relation can be expressed as

$$\log c(w_i,w_k)=u_i^T v_k+\tilde{b}_k+b_i$$

Just as we have two sets of word-vector embeddings, similarly we see two sets of bias—one for context words given by $\tilde{b}_i$ and the other for words given by b_i, where i indicates the *ith* word of the corpus.

The final aim is to minimize a sum of the squared error cost function between the actual $logc(w_i, w_k)$ and the predicted $u_i^T v_k+\tilde{b}_k+b_i$ for all word-context pairs, as follows:

$$C(U,V,\tilde{B},B)=\sum_{i,j=1}^{V}\left(u_i^T v_k+\tilde{b}_k+b_i-logc(w_i,w_k)\right)^2$$

U and V are the set of parameters for the word-vector embeddings and context-vector embeddings. Similarly, $\tilde{B}$ and B are the parameters for the biases corresponding to the words and the contexts. The cost function $C(U,V,\tilde{B},B)$ must be minimized with respect to these parameters in $U,V,\tilde{B},B$

One issue with this scheme of least square method is that it weights all co-occurrences equally in the cost function. This prevents the model from achieving good results since the rare co-occurrences carry very little information. One of the ways to handle this issue is to assign more weight to co-occurrences that have a higher count. The cost function can be modified to have a weight component for each co-occurrence pair that is a function of the co-occurrences count. The modified cost function can be expressed as follows:

$$C(U,V,\tilde{B},B)=\sum_{i,j=1}^{V}h(c(w_i,w_k))\left(u_i^T v_k+\tilde{b}_k+b_i-logc(w_i,w_k)\right)^2$$

where h is the newly introduced function.

The function $h(x)$ (see Figure 4-9) can be chosen as follows:

$$h(x)=\left(\frac{x}{x_{\max}}\right)^{\alpha} \quad if\ x<x_{\max}$$

$$=1, elsewhere$$

One can experiment with different values of α, which acts as a hyper parameter to the model.

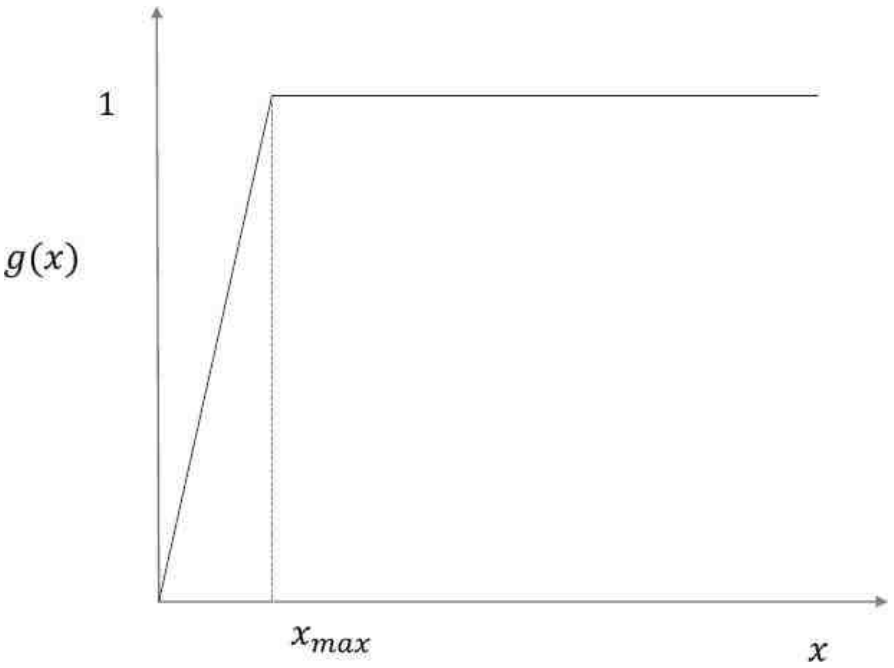

Figure 4-9: Weightage function for the co-occurrence counts

Word Analogy with Word Vectors

The good thing about word-vector embedding lies in its abilities to linearize analogies. We look at some analogies using the pre-trained GloVe vectors in Listing 4-2a, Listing 4-2b, and Listing 4-3c.

Listing 4-2a

```
import numpy as np
import scipy
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
%matplotlib inline
########################
# Loading glove vector
########################
EMBEDDING_FILE = '~/Downloads/glove.6B.300d.txt'

print('Indexing word vectors')
embeddings_index = {}
f = open(EMBEDDING_FILE)
count = 0
for line in f:
    if count == 0:
        count = 1
        continue
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %d word vectors of glove.' % len(embeddings_index))

-- output --

Indexing word vectors
Found 399999 word vectors of glove.
```

Listing 4-2b

```
king_wordvec = embeddings_index['king']
queen_wordvec = embeddings_index['queen']
man_wordvec   = embeddings_index['man']
woman_wordvec   = embeddings_index['woman']
```

```
pseudo_king = queen_wordvec -  woman_wordvec + man_wordvec
cosine_simi = np.dot(pseudo_king/np.linalg.norm(pseudo_king),king_wordvec/np.linalg.
norm(king_wordvec))
print 'Cosine Similarity',cosine_simi
--output --
Cosine Similarity 0.663537
```

Listing 4-2c

```
tsne = TSNE(n_components=2)
words_array = []
word_list = ['king','queen','man','woman']
for w in word_list:
    words_array.append(embeddings_index[w])
index1 = embeddings_index.keys()[0:100]
for i in xrange(100):
    words_array.append(embeddings_index[index1[i]])
words_array = np.array(words_array)
words_tsne = tsne.fit_transform(words_array)

ax = plt.subplot(111)
for i in xrange(4):
    plt.text(words_tsne[i, 0], words_tsne[i, 1],word_list[i])
    plt.xlim((-50,20))
    plt.ylim((0,50))

--output--
```

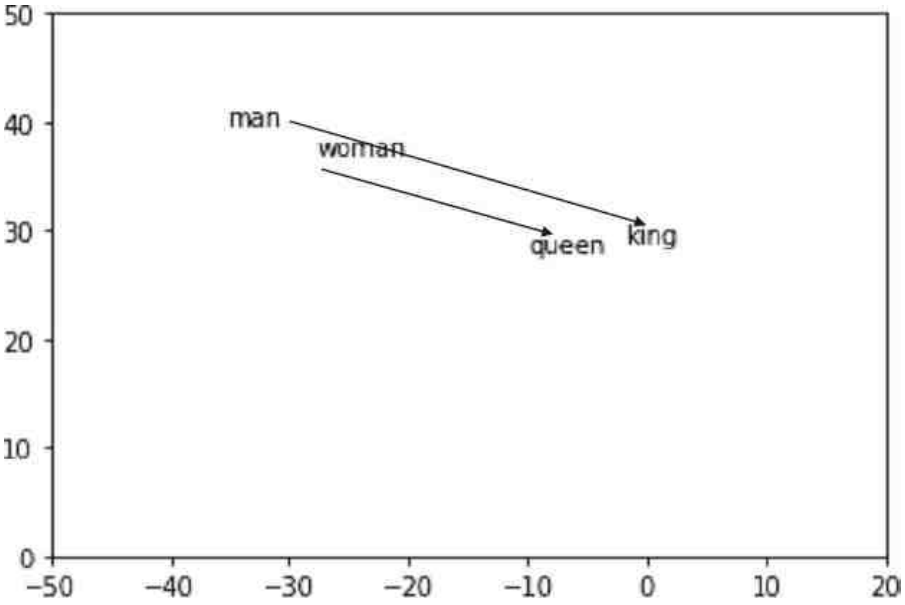

Figure 4-10: 2D TSNE vector plot for pre-trained GloVe vectors

In Listing 4-2a, the pre-trained GloVe vectors of dimensionality 300 are loaded and stored in a dictionary. We play around with the GloVe vectors for the words *king, queen, man*, and *woman* to find an analogy. Taking the word vectors for *queen, man*, and *woman*, a word vector *pseudo_king* is created as follows:

$$pseudo\_king = queen - woman + man$$

The idea is to see whether the preceding created vector somewhat represents the concept of *king* or not. The cosine of the angle between the word vectors *pseudo_king* and *king* is high at around 0.67, which is an indication that (*queen* – *woman* + *man*) very well represents the concept of *king*.

Next, in Listing 4-2c, we try to represent an analogy, and for that purpose, through TSNE, we represent the GloVe vectors of dimensionality 300 in a two-dimensional space. The results have been plotted in Figure 4-10. We can see that the word vectors for *king* and *queen* are close to each other and clustered together, and the word vectors for *man* and *woman* are clustered close to each other as well. Also, we see that the vector differences between *king* and *man* and those between *queen* and *woman* are almost parallelly aligned and of comparable lengths.

Before we move on to recurrent neural networks, one thing I want to mention is the importance of word embeddings for

recurrent neural networks in the context of natural language processing. A recurrent neural network doesn't understand text, and hence each word in the text needs to have some form of number representation. Word-embeddings vectors are a great choice since words can be represented by multiple concepts given by the components of the word-embeddings vector. Recurrent neural networks can be made to work both ways, either by providing the word-embeddings vectors as input or by letting the network learn those embeddings vectors by itself. In the latter case, the word-embeddings vectors would be aligned more toward the ultimate problem's being solved through the recurrent neural network. However, at times the recurrent neural network might have a lot of other parameters to learn, or the network might have very little data to train on. In such cases, having to learn the word-embeddings vectors as parameters might lead to overfitting or sup-optimal results. Using the pre-trained word-vector embeddings might be a wiser option in such scenarios.

Introduction to Recurrent Neural Networks

Recurrent neural networks (RNNs) are designed to utilize and learn from sequential information. The RNN architecture is supposed to perform the same task for every element of a sequence, and hence the term *recurrent* in its nomenclature. RNNs have been of great use in the task of natural language processing because of the sequential dependency of words in any language. For example, in the task of predicting the next word in a sentence, the prior sequence of words that came before it is of paramount importance. Generally, at any time step of a sequence, RNNs compute some memory based on its computations thus far; i.e., prior memory and the current input. This computed memory is used to make predictions for the current time step and is passed on to the next step as an input. The basic architectural principal of a recurrent neural network is illustrated in Figure 4-11.

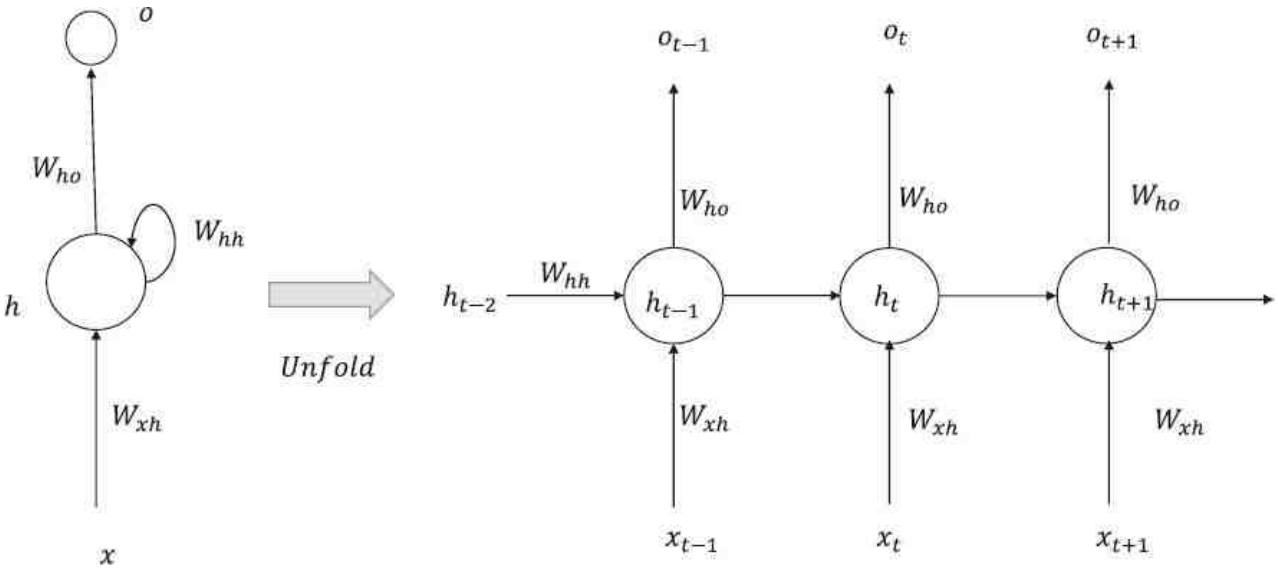

Figure 4-11: Folded and unfolded structure of an RNN

In Figure 4-11, the RNN architecture has been unrolled in time to depict the complete sequence. If we wish to process seven-word sequences of sentences, then the unfolded RNN architecture would represent a seven-layer feed-forward neural network, with the only difference being that the weights at each layer are common shared weights. This significantly reduces the number of parameters to learn in a recurrent neural network.

Just to get us familiar with the notations used, x_t,h_t, and o_t represent the input, computed memory or hidden states, and output respectively at time step *t*. W_{hh} represents the weights matrix from the memory states h_t at time *t* to the memory states h_{t+1} at time (*t*+1). W_{xh} represents the weight matrix from the input x_t to the hidden states h_t, whereas W_{ho} represents the weight matrix from the memory states h_t to o_t. The weight matrix W_{xh} acts as some sort of word-vector embeddings matrix when the inputs are presented in a one-hot encoded form. Alternately, in cases of one-hot encoded inputs one may choose to have a learnable separate embedding matrix so that when the one-hot encoded input vector passes through the embeddings layer its desired embeddings vector is presented as an output.

Now, let's drill down to each component in detail:

- The input x_t is a vector representing the input word at step *t*. For example, it can be a one-hot encoded vector with the component set to 1 for the corresponding word index in the vocabulary. It can also be the word-vector embedding from some pre-trained repository such as GloVe. In general, we assume $x_t \in \mathbb{R}^{D\times 1}$ Also, if we are looking to predict *V* classes then the output $y_t \in \mathbb{R}^{V\times 1}$.

- The memory or the hidden state vector h_t can have any length as per the user's choice. If the number of states chosen is n then $h_t \in \mathbb{R}^{n\times 1}$ and the weight matrix $W_{hh} \in \mathbb{R}^{n\times n}$.

- The weight matrix connecting the input to the memory states $W_{xh} \in \mathbb{R}^{n\times D}$ and the weight matrix connecting the memory states to the output $W_{ho} \in \mathbb{R}^{n\times V}$.

- The memory h_t at step t is computed as follows:

  $h_t = f\left(W_{hh}h_{t-1} + W_{xh}\,x_t\right)$, where f is a chosen non-linear activation function.

  The dimension of $\left(W_{hh}h_{t-1} + W_{xh}\,x_t\right)$ is n; i.e., $\left(W_{hh}h_{t-1} + W_{xh}\,x_t\right) \in \mathbb{R}^{n\times 1}$.

  The function f works element-wise on $\left(W_{hh}h_{t-1} + W_{xh}\,x_t\right)$ to produce h, and hence $\left(W_{hh}h_{t-1} + W_{xh}\,x_t\right)$ and h_t have the same dimension.

  If $W_{hh}h_{t-1} + W_{xh}\,x_t = \begin{bmatrix} s_{1t} \\ s_{2t} \\ . \\ . \\ s_{nt} \end{bmatrix}$ then the following holds true for h_t:

  $$h_t = \begin{bmatrix} f(s_{1t}) \\ f(s_{2t}) \\ . \\ . \\ f(s_{nt}) \end{bmatrix}$$

- The connections from the memory states to the output are just like the connections from a fully connected layer to the output layer. When a multi-class classification problem is involved, such as predicting the next word, then the output layer would be a huge SoftMax of the number of words in the vocabulary. In such cases the predicted output vector $o_t \in \mathbb{R}^{V\times 1}$ can be expressed as $SoftMax(W_{ho}h_t)$. Just to keep the notations simple, the biases have not been mentioned. In every unit, we can add bias to the input before it is acted upon by the different functions. So, o_t can be represented as

  $$o_t = SoftMax\left(W_{ho}h_t + b_o\right)$$

  where $b_o \in \mathbb{R}^{n\times 1}$ is the bias vector for the output units.

- Similarly, bias can be introduced in the memory units, and hence h_t can be expressed as

  $$h_t = f\left(W_{hh}h_{t-1} + W_{xh}\,x_t + b_h\right)$$

  where $b_h \in \mathbb{R}^{n\times 1}$ is the bias vector at the memory units.

- For a classification problem predicting the next word for a text sequence of T time steps, the output at each time step is a big SoftMax of V classes, where V is the size of the vocabulary. So, the corresponding loss at each time step is the negative log loss over all the vocabulary size V. The loss at each time step can be expressed as follows:

  $$C_t = -\sum_{j=1}^{V} y_t^{(j)} \log o_t^{(j)}$$

- To get the overall loss over all the time steps T, all such C_t needs to be summed up or averaged out. Generally, the average of all C_t is more convenient for stochastic gradient descent so as to get a fair comparison when the sequence length varies. The overall cost function for all the time steps T can thus be represented by the following:

  $$C = -\sum_{t=1}^{T}\sum_{j=1}^{V} y_t^{(j)} \log o_t^{(j)}$$

Language Modeling

In language modeling, the probability of a sequence of words is computed through the product rule of intersection of events. The probability of a sequence of words $w_1w_2w_3.....w_n$ of length n is given as follows:

$$P(w_1w_2w_3.....w_n) = P(w_1)P(w_2/w_1)P(w_3/w_1w_2)...P(w_3/w_1w_2)..... P(w_n/w_1w_2..w_{n-1})$$
$$= P(w_1)\prod_{k=2}^{n} P(w_k/w_1w_2..w_{k-1})$$

In traditional approaches, the probability of a word at time step k is generally not conditioned on the whole sequence of length $(k-1)$ prior to that but rather on a small window L prior to t. Hence, the probability is generally approximated as follows:

$$P(w_1w_2w_3.....w_n) \cong P(w_1)\prod_{k=2}^{n} P(w_k/w_{k-L}..w_{k-2}w_{k-1})$$

This method of conditioning a state based on L recent states is called the Markov assumption of chain rule probability. Although it is an approximation, it is a necessary one for traditional language models since the words cannot be conditioned on a large sequence of words because of memory constraints.

Language models are generally used for varied tasks related to natural language processing, such as sentence completion by predicting the next words, machine translation, speech recognition, and others. In machine translation, the words from another language might be translated to English but may not be correct syntactically. For example, a sentence in the Hindi language has been machine translated to produce the English sentence *"beautiful very is the sky* ". If one computes the probability of the machine-translated sequence (P(" *beautiful very is the sky* ")),itwould be very much less than that of the arranged counterpart, (P(" *The sky is very beautiful*"). Through language modeling, such comparisons of the probability of text sequences can be carried out.

Predicting the Next Word in a Sentence Through RNN Versus Traditional Methods

In traditional language models, the probability of a word appearing next is generally conditioned on a window of a specified number of previous words, as discussed earlier. To estimate probabilities, different n-grams counts are usually computed. From bi-gram and tri-gram counts, the conditional probabilities can be computed as follows:

$$P(w = w_2/w = w_1) = \frac{count(w_1,w_2)}{count(w_1)}$$

$$P(w = w_3/w = w_1, w = w_2) = \frac{count(w_1,w_2,w_3)}{count(w_1,w_2)}$$

In a similar manner, we can condition the word probabilities on longer sequences of words by keeping count of larger n-grams. Generally, when a match is not found on a higher n-gram—let's say four-gram—then a lower n-gram such as a three-gram is tried. This method is called back off and gives some performance gain over a fixed n-gram approach.

Since word prediction is conditioned on only a few previous words based on the chosen window size, this traditional method of computing probabilities through n-gram counts doesn't do as well as those models in which the whole sequence of words is taken into consideration for the next word prediction.

In RNNs, the output at each step is conditioned on all previous words, and hence RNNs do a better job than the n-gram models at language-model tasks. To understand this, let's look at the working principals of a generative recurrent neural network while considering a sequence $(x_1x_2x_3.....x_n)$ of length n.

The RNN updates its hidden state h_t recursively as $h_t = f(h_{t-1},x_t)$. The hidden state h_{t-1} has information accumulated for sequence of words $(x_1x_2x_3.....x_{t-1})$, and when the new word in the sequence x_t arrives the updated sequence information $(x_1x_2x_3.....x_t)$ is encoded in h_t through the recursive update.

Now, if we must predict the next word based on the word sequence seen so far, i.e., $(x_1x_2x_3.....x_t)$, the conditional probability

distribution one needs to look at is

$$P(x_{n+1} = o_i / x_1 x_2 x_3 \ldots .. x_t)$$

where o_i represents any generalized word in the vocabulary.

For neural networks this probability distribution is governed by the computed hidden state h_t based on the sequence seen so far, i.e., $x_1x_2x_3.....x$, and the model parameter V, which converts the hidden states to probabilities corresponding to every word in the vocabulary.

So,

$$P(x_{n+1} = o_i / x_1 x_2 x_3 \ldots .. x_t)$$

$$= P(x_{n+1} = o_i / h_t) \text{ or } P(x_{n+1} = o_i / x_t; h_{t-1})$$

The vector of probabilities for $P(x_{n+l} = o_i / h_t)$ corresponding to all indices i over the vocabulary is given by $Softmax(W_{ho}h_t)$.

Backpropagation Through Time (BPTT)

Backpropagation for recurrent neural networks is same as that for feed-forward neural networks, with the only difference being that the gradient is the sum of the gradient with respect to the log loss at each step.

First, the RNN is unfolded in time, and then the forward step is executed to get the internal activations and the final predictions for output. Based on the predicted output and the actual output labels, the loss and the corresponding error at each time step is computed. The error at each time step is backpropagated to update the weights. So, any weight update is proportional to the sum of the gradients' contribution from errors at all the T time steps.

Let's look at a sequence of length T and at the weight updates through BPTT. We take the number of memory states as n $(\text{i.e., } h_t \in \mathbb{R}^{n\times 1})$ and choose the input vector length as D $(\text{i.e., } x_t \in \mathbb{R}^{D\times 1})$. At each sequence step t we predict the next word from a vocabulary of V words through a SoftMax function.

The total cost function for a sequence of length T is given as

$$C = -\sum_{t=1}^{T}\sum_{j=1}^{V} y_t^{(j)} \log o_t^{(j)}$$

Let's compute the gradient of the cost function with respect to the weights connecting the hidden memory states to the output states layers—i.e., the weights belonging to the matrix W_{ho}. The weight w_{ij} denotes the weight connecting the hidden state i to the output unit j.

The gradient of the cost function C_t with respect to w_{ij} can be broken down by the chain rule of partial derivatives as the product of the partial derivative of the cost function with respect to the output of the jth unit $(\text{i.e., } \frac{\partial C_t}{\partial o_t^{(j)}})$, the partial derivative of the output of the jth unit with respect to the net input $s_t^{(j)}$ at the jth unit $(\text{i.e., } \frac{\partial o_t^{(j)}}{\partial s_t^{(j)}})$, and finally the partial derivative of the net input to the jth unit with respect to the concerned weight from the ith memory unit to the jth hidden layer $(\text{i.e., } \frac{\partial s_t^{(j)}}{\partial w_{ij}})$.

(4.2.1) $$\frac{\partial C_t}{\partial w_{ij}} = \frac{\partial C_t}{\partial o_t^{(j)}} \frac{\partial o_t^{(j)}}{\partial s_t^{(j)}} \frac{\partial s_t^{(j)}}{\partial w_{ij}}$$

(4.2.2) $$\frac{\partial s_t^{(j)}}{\partial w_{ij}} = h_t^{(i)}$$

Considering the SoftMax over vocabulary V and the actual output at time t as y_t $y_t = \left[y_t^{(1)} y_t^{(2)} .. y_t^{(V)}\right]^T \in \mathbb{R}^{V\times 1}$,

(4.2.3) $$\frac{\partial C_t}{\partial o_t^{(j)}} = -\frac{y_t^{(j)}}{o_t^{(j)}}$$

(4.2.4) $$\frac{\partial o_t^{(j)}}{\partial s_t^{(j)}} = o_t^{(j)}\left(1 - o_t^{(j)}\right)$$

Substituting the expressions for individual gradients from (4.2.2), (4.2.3), and (4.2.4) into (4.2.1), we get

(4.2.5) $$\frac{\partial C_t}{\partial w_{ij}} = -\frac{y_t^{(j)}}{o_t^{(j)}} o_t^{(j)}\left(1 - o_t^{(j)}\right) = -y_t^{(j)}\left(1 - o_t^{(j)}\right) h_t^{(i)}$$

To get the expression of the gradient of the total cost function C with respect to w_{ij}, one needs to sum up the gradients at each sequence step. Hence, the expression for the gradient $\frac{\partial C}{\partial w_{ij}}$ is as follows:

(4.2.6) $$\frac{\partial C}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial C_t}{\partial w_{ij}} = \sum_{t=1}^{T} -y_t^{(j)}\left(1 - o_t^{(j)}\right) h_t^{(i)}$$

So, we can see that determining the weights of the connections from the memory state to the output layers is pretty much the same as doing so for the fully connected layers of feed-forward neural networks, the only difference being that the effect of each sequential step is summed up to get the final gradient.

Now, let's look at the gradient of the cost function with respect to the weights connecting the memory states in one step to the memory states in the next step—i.e., the weights of the matrix W_{hh}. We take the generalized weight $u_{ki} \in W_{hh}$, where k and i are indexes of the memory units in consecutive memory units.

This will get a little more involved because of the recurrent nature of the memory units' connections. To appreciate this fact, let's look at the output of the memory unit indexed by i at step t—i.e., $h_t^{(i)}$:

(4.2.7) $$h_t^{(i)} = g\left(\sum_{l=1}^{N} u_{li} h_{t-1}^{(l)} + \sum_{m=1}^{D} v_{mi} x_t^{(m)} + b_{hi}\right)$$

Now, let's look at the gradient of the cost function at step t with respect to the weight u_{ki}

(4.2.8) $$\frac{\partial C_t}{\partial u_{ki}} = \frac{\partial C_t}{\partial h_t^{(i)}} \frac{\partial h_t^{(i)}}{\partial u_{ki}}$$

We are only interested in expressing $h^{(i)}_t$ as a function of u_{ki}, and hence we rearrange (4.2.7) as follows:

(4.2.9) $$\begin{aligned} h_t^{(i)} &= g\left(u_{ki} h_{t-1}^{(k)} + u_{ii} h_{t-1}^{(i)} + \sum_{\substack{l=1 \\ l \neq k,i}}^{N} u_{li} h_{t-1}^{(l)} + \sum_{m=1}^{D} v_{mi} x_t^{(m)}\right) \\ &= g\left(u_{ki} h_{t-1}^{(k)} + u_{ii} h_{t-1}^{(i)} + c_t^{(i)}\right) \end{aligned}$$

where

$$c_t^{(i)} = \sum_{\substack{l=1 \\ l \neq k,i}}^{N} u_{li} h_{t-1}^{(l)} + \sum_{m=1}^{D} v_{mi} x_t^{(m)}$$

We have rearranged $h_t^{(i)}$ to be a function that contains the required weight u_{ki} and kept $h^{(i)}_{t-1}$ since it can be expressed through recurrence as $g\left(u_{ki} h_{t-2}^{(k)} + u_{ii} h_{t-2}^{(i)} + c_{t-1}^{(i)}\right)$.

This nature of recurrence at every time step t would continue up to the first step, and hence one needs to consider the summation effect of all associated gradients from $t = t$ to $t = 1$. If the gradient of $h_t^{(i)}$ with respect to the weight u_{ki} follows the

recurrence, and if we take the derivative of $h_t^{(i)}$ as expressed in (4.2.9) with respect to u_{ki}, then the following will hold true:

(4.2.10) $$\frac{\partial h_t^{(i)}}{\partial u_{ki}} = \sum_{t'=1}^{t} \frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}} \frac{\bar{\partial} h_{t'}^{(i)}}{\partial u_{ki}}$$

Please note the bar for the expression $\frac{\bar{\partial} h_{t'}^{(i)}}{\partial u_{ki}}$. It denotes the local gradient of $h_t^{(i)}$ with respect to u_{ki}, holding $h^{(i)}_{t'-1}$ constant.

Replacing (4.2.9) with (4.2.8) we get

(4.2.11) $$\frac{\partial C_t}{\partial u_{ki}} = \sum_{t'=1}^{t} \frac{\partial C_t}{\partial h_t^{(i)}} \frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}} \frac{\bar{\partial} h_{t'}^{(i)}}{\partial u_{ki}}$$

The equation (4.2.11) gives us the generalized equation of the gradient of the cost function at time t. Hence, to get the total gradient we need to sum up the gradients with respect to the cost at each time step. Therefore, the total gradient can be represented by

(4.2.12) $$\frac{\partial C}{\partial u_{ki}} = \sum_{t=1}^{T} \sum_{t'=1}^{t} \frac{\partial C_t}{\partial h_t^{(i)}} \frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}} \frac{\bar{\partial} h_{t'}^{(i)}}{\partial u_{ki}}$$

The expression $\frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}}$ follows a product recurrence and hence can be formulated as

(4.2.13) $$\frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}} = \prod_{g=t'+1}^{t} \frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}}$$

Combining (4.2.12) and (4.2.13), we get

(4.2.14) $$\frac{\partial C}{\partial u_{ki}} = \sum_{t=1}^{T} \sum_{t'=1}^{t} \frac{\partial C_t}{\partial h_t^{(i)}} \left(\prod_{g=t'+1}^{t} \frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}} \right) \frac{\bar{\partial} h_{t'}^{(i)}}{\partial u_{ki}}$$

The computation of the gradient of the cost function C for the weights of the matrix W_{xh} can be computed in a manner similar to that for the weights corresponding to the memory states.

Vanishing and Exploding Gradient Problem in RNN

The aim of recurrent neural networks (RNNs) is to learn long dependencies so that the interrelations between words that are far apart are captured. For example, the actual meaning that a sentence is trying to convey may be captured well by words that are not in close proximity to each other. Recurrent neural networks should be able to learn those dependencies. However, RNNs suffer from an inherent problem: they fail to capture long-distance dependencies between words. This is because the gradients in instances of long sequences have a high chance of either going to zero or going to infinity very quickly. When the gradients drop to zero very quickly, the model is unable to learn the associations or correlations between events that are temporally far apart. The equations derived for the gradient of the cost function with respect to the weights of the hidden memory layers will help us understand why this vanishing-gradient problem might take place.

The gradient of the cost function C_t at step t for a generalized weight $u_{ki} \in W_{hh}$ is given by

$$\frac{\partial C_t}{\partial u_{ki}} = \sum_{t'=1}^{t} \frac{\partial C_t}{\partial h_t^{(i)}} \frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}} \frac{\partial h_{t'}^{(i)}}{\partial u_{ki}}$$

where the notations comply with their original interpretations as mentioned in the "Backpropagation Through Time (BPTT)" section.

The components summed to form $\frac{\partial C_t}{\partial u_{ki}}$ are called its temporal components. Each of those components measures how the

weight u_{ki} at step *t'* influences the loss at step *t*. The component $\frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}}$ backpropagates the error at step *t* back to step *t'*.

Also,

$$\frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}} = \prod_{g=t'+1}^{t} \frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}}$$

Combining the preceding two equations, we get

$$\frac{\partial C_t}{\partial u_{ki}} = \sum_{t'=1}^{t} \frac{\partial C_t}{\partial h_t^{(i)}} \left(\prod_{g=t'+1}^{t} \frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}} \right) \frac{\partial h_{t'}^{(i)}}{\partial u_{ki}}$$

Let's take the net input at a memory unit *i* at time step *g* to be $z_g^{(i)}$. So, if we take the activation at the memory units to be sigmoids, then

$$h_g^{(i)} = \sigma\left(z_g^{(i)}\right)$$

where σ is the sigmoid function.

Now,

$$\frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}} = \sigma'\left(z_g^{(i)}\right) \frac{\partial z_g^{(i)}}{\partial h_{g-1}^{(i)}} \quad = \sigma'\left(z_g^{(i)}\right) u_{ii}$$

Where σ' ($z_g^{(i)}$) denotes the gradient of $\sigma(z_g^{(i)})$ with respect to $z_g^{(i)}$.

If we have a long sequence, i.e., *t*=*t* and *t'*=k, then the following quantity would have many derivatives of the sigmoid function, as shown here:

$$\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}} = \prod_{g=k+1}^{T} \frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}} = \frac{\partial h_{k+1}^{(i)}}{\partial h_k^{(i)}} \frac{\partial h_{k+2}^{(i)}}{\partial h_{k+1}^{(i)}} \cdots \frac{\partial h_t^{(i)}}{\partial h_{(t-1)}^{(i)}} = \sigma'\left(z_{k+1}^{(i)}\right) u_{ii} \sigma'\left(z_{k+2}^{(i)}\right) u_{ii} .. \sigma'\left(z_t^{(i)}\right) u_{ii}$$

Combining the gradient expressions in product-notation form, this important equation can be rewritten as follows:

$$\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}} = \left(u_{ii}\right)^{t-k} \prod_{g=k+1}^{t} \sigma'\left(z_g^{(i)}\right) (1)$$

Sigmoid functions have good gradients only within a small range of values and saturate very quickly. Also, the gradients of sigmoid activation functions are less than 1. So, when the error from a long-distance step at *t* = *T* passes to a step at *t* = l there would be *(T* -l) sigmoid activation function gradients the error must pass through, and the multiplicative effect of (*T*-l) gradients of values less than 1 may make the gradient component $\frac{\partial h_T^{(i)}}{\partial h_1^{(i)}}$ vanish exponentially fast. As discussed, $\frac{\partial h_T^{(i)}}{\partial h_1^{(i)}}$ backpropagates the error at step *T* = *t* back to step *t* = *l* so that the long-distance correlation between the words at steps *t* = *l* and *t* = *T* is learned.

However, because of the vanishing-gradient problem, $\frac{\partial h_T^{(i)}}{\partial h_1^{(i)}}$ may not receive enough gradient and may be close to zero, and hence it won't be possible to learn the correlation or dependencies between the long-distance words in a sentence.

RNNs can suffer from exploding gradients too. We see in the expression for $\frac{\partial h_T^{(i)}}{\partial h_1^{(i)}}$ the weight u_{ii} has been

repeatedly multiplied *(T*-1) times. If $u_{tt} > l$, and for simplicity we assume $u_{tt} = 2$, then after 50 steps of backpropagation from the sequence step T to sequence step (*T*- 50) the gradient would magnify approximately 2^{50} times and hence would drive the model training into instability.

Solution to Vanishing and Exploding Gradients Problem in RNNs

There are several methods adopted by the deep-learning community to combat the vanishing-gradient problem. In this section, we will discuss those methods before moving on to a variant of RNN called long short-term memory (LSTM) recurrent neural networks. LSTMs are much more robust regarding vanishing and exploding gradients.

Gradient Clipping

Exploding gradients can be tackled by a simple technique called gradient clipping. If the magnitude of the gradient vector exceeds a specified threshold, then the magnitude of the gradient vector is set to the threshold while keeping the direction of the gradient vector intact. So, while doing backpropagation on a neural network at time t, if the gradient of the cost function *C* with respect to the weight vector *w* exceeds a threshold k, then the gradient *g* used for backpropagation is updated as follows:

- ***Step 1:*** Update $g \leftarrow \nabla C\left(w = w^{(t)}\right)$
- ***Step 2:*** If ||g|| > *k* then update $g \leftarrow \frac{k}{\|g\|} g$

Smart Initialization of the Memory-to-Memory Weight Connection Matrix and ReLU units

Instead of randomly initializing the weights in the weight matrix W_{hh}, initializing it as an identity matrix helps prevent the vanishing-gradient problem. One of the main reasons the for vanishing-gradient problem is the gradient of the hidden unit *i* at time *t* with respect to the hidden unit *i* at time *t'* where *t'*<< *t* is given by

$$\frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}} = \prod_{g=t'+1}^{t} \frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}}$$

In the case of the sigmoid activation function, each of the terms $\frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}}$ can be expanded as follows:

$$\frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}} = \sigma'\left(z_g^{(t)}\right) u_{ii}$$

where σ(.) denotes the sigmoid function and $z_g^{(t)}$ denotes the net input to hidden unit *i* at step *t*. The parameter u_{ii} is the weight connecting the *ith* hidden memory state at any sequence step *t* -1 to the *ith* hidden unit memory at sequence step *t*.

The greater the distance is between sequence steps *t'* and *t*, the more sigmoid derivatives the error would go through in passing from *t* to *t'*. Since the sigmoid activation function derivatives are always less than 1 and saturate quickly, the chance of the gradient going to zero in cases of long dependencies is high.

If, however, we choose ReLU units, then the sigmoid gradients would be replaced by ReLU gradients, which have a constant value of 1 for positive net input. This would reduce the vanishing-gradient problem since when the gradient is 1 the gradients would flow unattenuated. Also, when the weight matrix is chosen to be identity then the weight connection u_{ii} would be 1, and hence the quantity $\frac{\partial h_t^{(i)}}{\partial h_{t'}^{(i)}}$ would be 1 irrespective of the distance between sequence step *t* and sequence step *t'*. This means that the error that would be propagated from hidden memory state $h_t^{(i)}$ at time *t* to the hidden memory state $h_t^{(i)}$ at any prior time step *t'* would be constant irrespective of the distance from step *t* to *t'*. This will enable the RNN to learn associated correlations or dependencies between the long-distance words in a sentence.

Long Short-Term Memory (LSTM)

Long short-term memory recurrent neural networks, popularly known as LSTMs, are special versions of RNNs that can learn distant dependencies between words in a sentence. Basic RNNs are incapable of learning such correlations between distant words, as discussed earlier. The architecture of long short-term memory (LSTM) recurrent neural networks is quite a bit different than that of traditional RNNs. Represented in Figure 4-12 is a high-level representation of an LSTM.

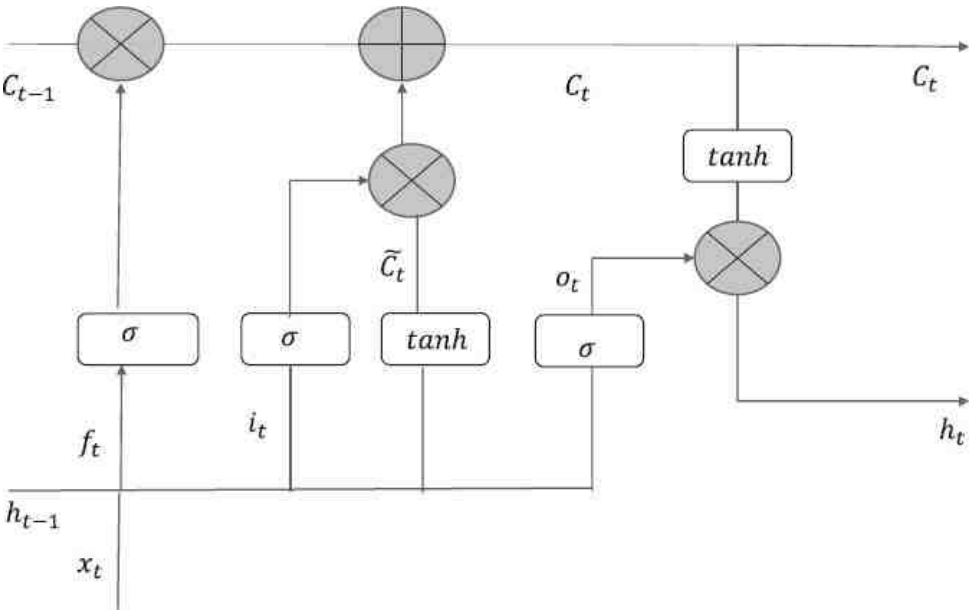

Figure 4-12: LSTM architecture

The basic building blocks of an LSTM and their functions are as follows:

- The new element in LSTMs is the introduction of the cell state C_t, which is regulated by three gates. The gates are composed of sigmoid functions so that they output values between 0 and 1.

  At sequence step *t* the input x_t and the previous step's hidden states h_{t-1} decide what information to forget from cell state C_{t-1} through the forget-gate layer. The forget gate looks at x_t and h_{t-1} and assigns a number between 0 and 1 for each element in the cell state vector C_{t-1}. An output of 0 means totally forget the state while an output of 1 means keep the state completely. The forget gate output is computed as follows:

  $$f_t = \sigma\left(W_f x_t + U_f h_{t-1} \right)$$

- Next, the input gate decides which cell units should be updated with new information. For this, like the forget-gate output, a value between 0 to 1 is computed for each cell state component through the following logic:

  $$i_t = \sigma\left(W_i x_t + U_i h_{t-1} \right)$$

  Then, a candidate set of new values is created for the cell state using x_t and h_{t-1} as input. The candidate cell state $\tilde{C}t$ is computed as follows:

  $$\tilde{C}t = \tanh\left(W_c x_t + U_c h_{t-1} \right)$$

  The new cell state C_t is updated as follows:

  $$C_t = f_t * C_{t-1} + i_t * \tilde{C}t$$

- The next job is to determine which cell states to output since the cell state contains a lot of information. For this, x_t and h_{t-1} are passed through an output gate, which outputs a value between 0 to 1 for each component in the cell state vector C_t. The output of this gate is computed as follows:

  $$o_t = \sigma\left(W_o x_t + U_o h_{t-1} \right)$$

- The updated hidden state h_t is computed from the cell state C_t by passing each of its elements through a *tanh* function and then doing an element-wise product with the output gate values:

  $$h_t = o_t * \tanh\left(C_t \right)$$

Please note that the symbol * in the preceding equations denotes element-wise multiplication. This is done so that, based on the gate outputs, we can assign weights to each element of the vector it is operated on. Also, note that whatever gate output values are obtained they are multiplied as is. The gate outputs are not converted to discrete values of 0 and 1. Continuous values between 0 and 1 through sigmoid provide smooth gradients for backpropagation.

The forget gate plays a crucial role in the LSTM, when the forget gate units output zero then the recurrent gradients becomes zero and the corresponding old cell state units are discarded. This way, the LSTM throws away information that it thinks is not going to be useful in the future. Also, when the forget- gate units output 1, the error flows through the cell units unattenuated and the model can learn long-distance correlations between temporally distant words. We will discuss more about this in the next section.

Another important feature of the LSTM is the introduction of the output gates. The output-gate unit ensures that not all information the cell state C_t units have is exposed to the rest of the network, and hence only the relevant information is revealed in the form of h_t. This ensures that the rest of the network is not impacted by the unnecessary data while that data in the cell state is still held back in the cell state to help drive future decisions.

LSTM in Reducing Exploding- and Vanishing -Gradient Problems

LSTMs don't suffer much from vanishing- or exploding-gradient problems. The main reason for this is the introduction of the forget gate f_t and the way the current cell state is dependent on it through the following equation:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

This equation can be broken down on a cell state unit-level that has a general index *i* as follows:

$$C_t^{(i)} = f_t^{(i)} C_{t-1}{}^{(i)} + i_t^{(i)} \tilde{C}_t^{(i)}$$

It is important to note here that $C_t^{(i)}$ is linearly dependent on $C_{t-1}{}^{(i)}$ and hence the activation function is the identity function with a gradient of 1.

The notorious component in recurrent neural network backpropagation that may lead to vanishing or exploding gradients is the component $\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}}$ when (*t-k*) is large. This component backpropagates the error at sequence step *t* to sequence step *k* so that the model learns long-distant dependencies or correlations.

The expression for $\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}}$, as we have seen in the vanishing and exploding gradient section, is given by the following:

$$\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}} = (u_{ii})^{t-k} \prod_{g=k+1}^{t} \sigma'\left(z_g^{(i)}\right)$$

A vanishing-gradient condition will arise when the gradients and/or weights are less than 1 since the product of (*t-k*) in them would force the overall product to near zero. Since sigmoid gradients and tanh gradients are most of the time less than 1 and saturate fast where they have near-zero gradients, this problem of vanishing gradients would be more severe with those. Similarly, exploding gradients can happen when the weight connection *u* between the *ith* hidden to the *ith* hidden unit is greater than 1 since the product of (*t-k*) in them would make the term $(u_{ii})^{t-k}$ exponentially large.

The equivalent of $\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}}$ in LSTM is the component $\frac{\partial C_t^{(i)}}{\partial C_k^{(i)}}$, which can also be expressed in the product format as follows:

(4.3.1) $$\frac{\partial C_t^{(i)}}{\partial C_k^{(i)}} = \prod_{g=t'+1}^{t} \frac{\partial C_g^{(i)}}{\partial C_{g-1}^{(i)}}$$

On taking the partial derivative on both sides of the cell state update equation $C_t^{(i)} = f_t^{(i)} C_{t-1}{}^{(i)} + i_t^{(i)} \tilde{C}_t^{(i)}$, one gets the important expression

(4.3.2) $$\frac{\partial C_t^{(i)}}{\partial C_{t-1}^{(i)}} = f_t^{(i)}$$

Combining (1) and (2), one gets

(4.3.3) $$\frac{\partial C_t^{(i)}}{\partial C_k^{(i)}} = \left(f_t^{(i)} \right)^{t-k}$$

The equation (4.3.3) says that if the forget-gate value is held near 1, LSTM will not suffer from vanishing- or exploding-gradient problems.

MNIST Digit Identification in TensorFlow Using Recurrent Neural Networks

We see an implementation of RNN in classifying the images in the MNIST dataset through LSTM. The images of MNIST dataset are 28 x 28 in dimension. Each image would be treated as having 28 sequence steps, and each sequence step would consist of a row in an image. There would not be outputs after each sequence step but rather only one output at the end of the 28 steps for each image. The output is one of the ten classes corresponding to the ten digits from 0 to 9. So, the output layer would be a SoftMax of ten classes. The hidden cell state of the final sequence step h_{28} would be fed through weights into the output layer. So, only the final sequence step would contribute to the cost function; there would be no cost function associated with the intermediate sequence steps. If you remember, backpropagation was with respect to the cost function C_t in each individual sequence step t when there was output involved at each sequence step, and finally the gradients with respect to each of those cost functions were summed together. Here, everything else remains the same and backpropagation would be done only with respect to the cost at sequence step 28, C_{28}. Also, as stated earlier, each row of an image would form data at a sequence step t and hence would be the input vector x_t. Finally, we would be processing images in a mini batch, and thus for each image in the batch a similar processing procedure would be followed, minimizing the average cost over the batch. One more important thing to note is that TensorFlow demands that there be separate tensors for each step within the mini-batch input tensor. The input tensor structure has been depicted in Figure 4-13 for ease of understanding.

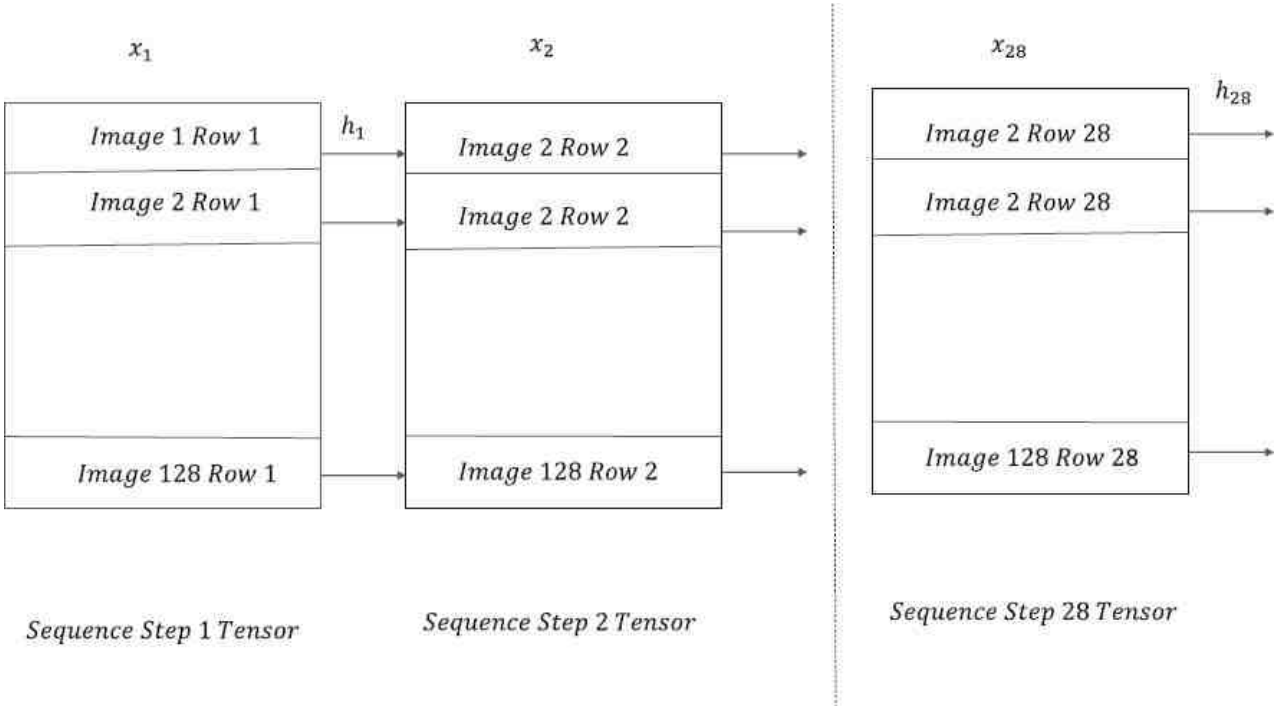

Figure 4-13: Input tensor shape for RNN LSTM network in Tensorflow

Now that we have some clarity about the problem and approach, we will proceed with the implementation in TensorFlow. The detailed code for training the model and validating it on the test dataset is illustrated in Listing 4-3.

Listing 4-3: TensorFlow Implementation of Recurrent Neural Network using LSTM for Classification

```
#Import the Required Libraries
import tensorflow as tf
from tensorflow.contrib import rnn
import numpy as np

# Import MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

# Batch Learning Parameters
```

```
learning_rate = 0.001
training_iters = 100000
batch_size = 128
display_step = 50
num_train = mnist.train.num_examples
num_batches = (num_train//batch_size) + 1
epochs = 2

# RNN LSTM Network Parameters
n_input = 28 # MNIST data input (img shape: 28*28)
n_steps = 28 # timesteps
n_hidden = 128 # hidden layer num of features
n_classes = 10 # MNIST total classes (0-9 digits)

# Define th forward pass for the RNN

def RNN(x, weights, biases):

    # Unstack to get a list of n_stepstensors of shape (batch_size, n_input) as illustrated
    in Figure 4-12
    x = tf.unstack(x, n_steps, 1)

    # Define a lstm cell
    lstm_cell = rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)

    # Get lstm cell output
    outputs, states = rnn.static_rnn(lstm_cell, x, dtype=tf.float32)

    # Linear activation, using rnn inner loop last output
    return tf.matmul(outputs[-1], weights['out']) + biases['out']

# tf Graph input
x = tf.placeholder("float", [None, n_steps, n_input])
y = tf.placeholder("float", [None, n_classes])
# Define weights
weights = {
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))
}
biases = {
    'out': tf.Variable(tf.random_normal([n_classes]))
}

pred = RNN(x, weights, biases)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

# Evaluate model
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    i = 0

    while i < epochs:
        for step in xrange(num_batches):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            batch_x = batch_x.reshape((batch_size, n_steps, n_input))
        # Run optimization op (backprop)
            sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
            if (step + 1) % display_step == 0:
                # Calculate batch accuracy
                acc = sess.run(accuracy, feed_dict={x: batch_x, y: batch_y})
                # Calculate batch loss
                loss = sess.run(cost, feed_dict={x: batch_x, y: batch_y})
                print "Epoch: " + str(i+1) + ",step:"+ str(step+1) +", Minibatch Loss= " + \
                      "{:.6f}".format(loss) + ", Training Accuracy= " + \
                      "{:.5f}".format(acc)
        i += 1
    print "Optimization Finished!"

    # Calculate accuracy
    test_len = 500
```

```
    test_data = mnist.test.images[:test_len].reshape((-1, n_steps, n_input))
    test_label = mnist.test.labels[:test_len]
    print "Testing Accuracy:", \
        sess.run(accuracy, feed_dict={x: test_data, y: test_label})
xx---output--xx

Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Epoch: 1,step:50, Minibatch Loss= 0.822081, Training Accuracy= 0.69531
Epoch: 1,step:100, Minibatch Loss= 0.760435, Training Accuracy= 0.75781
Epoch: 1,step:150, Minibatch Loss= 0.322639, Training Accuracy= 0.89844
Epoch: 1,step:200, Minibatch Loss= 0.408063, Training Accuracy= 0.85156
Epoch: 1,step:250, Minibatch Loss= 0.212591, Training Accuracy= 0.93750
Epoch: 1,step:300, Minibatch Loss= 0.158679, Training Accuracy= 0.94531
Epoch: 1,step:350, Minibatch Loss= 0.205918, Training Accuracy= 0.92969
Epoch: 1,step:400, Minibatch Loss= 0.131134, Training Accuracy= 0.95312
Epoch: 2,step:50, Minibatch Loss= 0.161183, Training Accuracy= 0.94531
Epoch: 2,step:100, Minibatch Loss= 0.237268, Training Accuracy= 0.91406
Epoch: 2,step:150, Minibatch Loss= 0.130443, Training Accuracy= 0.94531
Epoch: 2,step:200, Minibatch Loss= 0.133215, Training Accuracy= 0.93750
Epoch: 2,step:250, Minibatch Loss= 0.179435, Training Accuracy= 0.95312
Epoch: 2,step:300, Minibatch Loss= 0.108101, Training Accuracy= 0.97656
Epoch: 2,step:350, Minibatch Loss= 0.099574, Training Accuracy= 0.97656
Epoch: 2,step:400, Minibatch Loss= 0.074769, Training Accuracy= 0.98438
Optimization Finished!
Testing Accuracy: 0.954102
```

As we can see from the Listing 4-3 output, just by running 2 epochs an accuracy of 95% is achieved on the Test Dataset.

Next-Word Prediction and Sentence Completion in TensorFlow Using Recurrent Neural Networks

We train a model on a small passage from *Alice in Wonderland* to predict the next word from the given vocabulary using LSTM. Sequences of three words have been taken as input, and the subsequent word has been taken as output for this problem. Also, a two-layered LSTM model has been chosen instead of one. The sets of inputs and outputs are chosen randomly from the corpus and fed as a mini batch of size 1. We see the model achieves good accuracy and is able to learn the passage well. Later, once the model is trained, we input a three-word sentence and let the model predict the next 28 words. Each time it predicts a new word it appends it to the updated sentence. For predicting the next word, the previous three words from the updated sentence are taken as input. The detailed implementation of the problem has been outlined in Listing 4-4.

Listing 4-4: Next-Word Prediction and Sentence Completion in TensorFlow Using Recurrent Neural Networks

```
# load the required libraries
import numpy as np
import tensorflow as tf
from tensorflow.contrib import rnn
import random
import collections
import time

# Parameters
learning_rate = 0.001
training_iters = 50000
display_step = 500
n_input = 3

# number of units in RNN cell
n_hidden = 512

# Function to read and process the input file
def read_data(fname):
    with open(fname) as f:
        data = f.readlines()
    data = [x.strip() for x in data]
    data = [data[i].lower().split() for i in range(len(data))]
    data = np.array(data)
    data = np.reshape(data, [-1, ])
    return data

# Function to build dictionary and reverse dictionary of words.
def build_dataset(train_data):
```

```
    count = collections.Counter(train_data).most_common()
    dictionary = dict()
    for word, _ in count:
        dictionary[word] = len(dictionary)
    reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return dictionary, reverse_dictionary

# Function to one-hot the input vectors
def input_one_hot(num):
    x = np.zeros(vocab_size)
    x[num] = 1
    return x.tolist()

# Read the input file and build the required dictionaries
train_file = 'alice in wonderland.txt'
train_data = read_data(train_file)
dictionary, reverse_dictionary = build_dataset(train_data)
vocab_size = len(dictionary)

# Place holder for Mini-batch input output
x = tf.placeholder("float", [None, n_input, vocab_size])
y = tf.placeholder("float", [None, vocab_size])

# RNN output node weights and biases
weights = {
    'out': tf.Variable(tf.random_normal([n_hidden, vocab_size]))
}
biases = {
    'out': tf.Variable(tf.random_normal([vocab_size]))
}

# Forward pass for the recurrent neural network
def RNN(x, weights, biases):

    x = tf.unstack(x, n_input, 1)

    # 2 layered LSTM Definition
    rnn_cell = rnn.MultiRNNCell([rnn.BasicLSTMCell(n_hidden),rnn.BasicLSTMCell(n_hidden)])

    # generate prediction
    outputs, states = rnn.static_rnn(rnn_cell, x, dtype=tf.float32)

    # there are n_input outputs but
    # we only want the last output
    return tf.matmul(outputs[-1], weights['out']) + biases['out']

pred = RNN(x, weights, biases)

# Loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate).minimize(cost)

# Model evaluation
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
init = tf.global_variables_initializer()

# Launch the graph
with tf.Session() as session:
    session.run(init)
    step = 0
    offset = random.randint(0,n_input+1)
    end_offset = n_input + 1
    acc_total = 0
    loss_total = 0

    while step < training_iters:
        if offset > (len(train_data)-end_offset):
            offset = random.randint(0, n_input+1)

        symbols_in_keys = [ input_one_hot(dictionary[ str(train_data[i])]) for i in
        range(offset, offset+n_input) ]
        symbols_in_keys = np.reshape(np.array(symbols_in_keys), [-1, n_input,vocab_size])
        symbols_out_onehot = np.zeros([vocab_size], dtype=float)
        symbols_out_onehot[dictionary[str(train_data[offset+n_input])]] = 1.0
        symbols_out_onehot = np.reshape(symbols_out_onehot,[1,-1])
```

```
        _, acc, loss, onehot_pred = session.run([optimizer, accuracy, cost, pred], \
                                                feed_dict={x: symbols_in_keys, y: symbols_
                                                out_onehot})
        loss_total += loss
        acc_total += acc

        if (step+1) % display_step == 0:
            print("Iter= " + str(step+1) + ", Average Loss= " + \
                  "{:.6f}".format(loss_total/display_step) + ", Average Accuracy= " + \
                  "{:.2f}%".format(100*acc_total/display_step))
            acc_total = 0
            loss_total = 0
            symbols_in = [train_data[i] for i in range(offset, offset + n_input)]
            symbols_out = train_data[offset + n_input]
            symbols_out_pred = reverse_dictionary[int(tf.argmax(onehot_pred, 1).eval())]
            print("%s - Actual word:[%s] vs Predicted word:[%s]" % (symbols_in,symbols_
            out,symbols_out_pred))
        step += 1
        offset += (n_input+1)
    print("TrainingCompleted!")
# Feed a 3-word sentence and let the model predict the next 28 words
    sentence = 'i only wish'
    words = sentence.split(' ')
    try:
        symbols_in_keys = [ input_one_hot(dictionary[ str(train_data[i])]) for i in
        range(offset, offset+n_input) ]
        for i in range(28):
            keys = np.reshape(np.array(symbols_in_keys), [-1, n_input,vocab_size])
            onehot_pred = session.run(pred, feed_dict={x: keys})
            onehot_pred_index = int(tf.argmax(onehot_pred, 1).eval())
            sentence = "%s %s" % (sentence,reverse_dictionary[onehot_pred_index])
            symbols_in_keys = symbols_in_keys[1:]
            symbols_in_keys.append(input_one_hot(onehot_pred_index))
        print "Complete sentence follows!'
        print(sentence)
    except:
        print("Error while processing the sentence to be completed")

---output --

Iter= 30500, Average Loss= 0.073997, Average Accuracy= 99.40%
['only', 'you', 'can'] - Actual word:[find] vs Predicted word:[find]
Iter= 31000, Average Loss= 0.004558, Average Accuracy= 99.80%
['very', 'hopeful', 'tone'] - Actual word:[though] vs Predicted word:[though]
Iter= 31500, Average Loss= 0.083401, Average Accuracy= 99.20%
['tut', ',', 'tut'] - Actual word:[,] vs Predicted word:[,]
Iter= 32000, Average Loss= 0.116754, Average Accuracy= 99.00%
['when', 'they', 'met'] - Actual word:[in] vs Predicted word:[in]
Iter= 32500, Average Loss= 0.060253, Average Accuracy= 99.20%
['it', 'in', 'a'] - Actual word:[bit] vs Predicted word:[bit]
Iter= 33000, Average Loss= 0.081280, Average Accuracy= 99.00%
['perhaps', 'it', 'was'] - Actual word:[only] vs Predicted word:[only]
Iter= 33500, Average Loss= 0.043646, Average Accuracy= 99.40%
['you', 'forget', 'to'] - Actual word:[talk] vs Predicted word:[talk]
Iter= 34000, Average Loss= 0.088316, Average Accuracy= 98.80%
[',', 'and', 'they'] - Actual word:[walked] vs Predicted word:[walked]
Iter= 34500, Average Loss= 0.154543, Average Accuracy= 97.60%
['a', 'little', 'startled'] - Actual word:[when] vs Predicted word:[when]
Iter= 35000, Average Loss= 0.105387, Average Accuracy= 98.40%
['you', 'again', ','] - Actual word:[you] vs Predicted word:[you]
Iter= 35500, Average Loss= 0.038441, Average Accuracy= 99.40%
['so', 'stingy', 'about'] - Actual word:[it] vs Predicted word:[it]
Iter= 36000, Average Loss= 0.108765, Average Accuracy= 99.00%
['like', 'to', 'be'] - Actual word:[rude] vs Predicted word:[rude]
Iter= 36500, Average Loss= 0.114396, Average Accuracy= 98.00%
['make', 'children', 'sweet-tempered'] - Actual word:[.] vs Predicted word:[.]
Iter= 37000, Average Loss= 0.062745, Average Accuracy= 98.00%
['chin', 'upon', "alice's"] - Actual word:[shoulder] vs Predicted word:[shoulder]
Iter= 37500, Average Loss= 0.050380, Average Accuracy= 99.20%
['sour', '\xe2\x80\x94', 'and'] - Actual word:[camomile] vs Predicted word:[camomile]
Iter= 38000, Average Loss= 0.137896, Average Accuracy= 99.00%
['very', 'ugly', ';'] - Actual word:[and] vs Predicted word:[and]
Iter= 38500, Average Loss= 0.101443, Average Accuracy= 98.20%
["'", 'she', 'went'] - Actual word:[on] vs Predicted word:[on]
Iter= 39000, Average Loss= 0.064076, Average Accuracy= 99.20%
['closer', 'to', "alice's"] - Actual word:[side] vs Predicted word:[side]
Iter= 39500, Average Loss= 0.032137, Average Accuracy= 99.60%
['in', 'my', 'kitchen'] - Actual word:[at] vs Predicted word:[at]
```

```
Iter= 40000, Average Loss= 0.110244, Average Accuracy= 98.60%
[',', 'tut', ','] - Actual word:[child] vs Predicted word:[child]
Iter= 40500, Average Loss= 0.088653, Average Accuracy= 98.60%
["i'm", 'a', 'duchess'] - Actual word:[,] vs Predicted word:[,]
Iter= 41000, Average Loss= 0.122520, Average Accuracy= 98.20%
["'", "'", 'perhaps'] - Actual word:[it] vs Predicted word:[it]
Iter= 41500, Average Loss= 0.011063, Average Accuracy= 99.60%
['it', 'was', 'only'] - Actual word:[the] vs Predicted word:[the]
Iter= 42000, Average Loss= 0.057289, Average Accuracy= 99.40%
['you', 'forget', 'to'] - Actual word:[talk] vs Predicted word:[talk]
Iter= 42500, Average Loss= 0.089094, Average Accuracy= 98.60%
['and', 'they', 'walked'] - Actual word:[off] vs Predicted word:[off]
Iter= 43000, Average Loss= 0.023430, Average Accuracy= 99.20%
['heard', 'her', 'voice'] - Actual word:[close] vs Predicted word:[close]
Iter= 43500, Average Loss= 0.022014, Average Accuracy= 99.60%
['i', 'am', 'to'] - Actual word:[see] vs Predicted word:[see]
Iter= 44000, Average Loss= 0.000067, Average Accuracy= 100.00%
["wouldn't", 'be', 'so'] - Actual word:[stingy] vs Predicted word:[stingy]
Iter= 44500, Average Loss= 0.131948, Average Accuracy= 98.60%
['did', 'not', 'like'] - Actual word:[to] vs Predicted word:[to]
Iter= 45000, Average Loss= 0.074768, Average Accuracy= 99.00%
['that', 'makes', 'them'] - Actual word:[bitter] vs Predicted word:[bitter]
Iter= 45500, Average Loss= 0.001024, Average Accuracy= 100.00%
[',', 'because', 'she'] - Actual word:[was] vs Predicted word:[was]
Iter= 46000, Average Loss= 0.085342, Average Accuracy= 98.40%
['new', 'kind', 'of'] - Actual word:[rule] vs Predicted word:[rule]
Iter= 46500, Average Loss= 0.105341, Average Accuracy= 98.40%
['alice', 'did', 'not'] - Actual word:[much] vs Predicted word:[much]
Iter= 47000, Average Loss= 0.081714, Average Accuracy= 98.40%
['soup', 'does', 'very'] - Actual word:[well] vs Predicted word:
[well]Iter= 47500, Average Loss= 0.076034, Average Accuracy= 98.40%
['.', "'", "everything's"] - Actual word:[got] vs Predicted word:[got]
Iter= 48000, Average Loss= 0.099089, Average Accuracy= 98.20%
[',', "'", 'she'] - Actual word:[said] vs Predicted word:[said]
Iter= 48500, Average Loss= 0.082119, Average Accuracy= 98.60%
['.', "'", "'"] - Actual word:[perhaps] vs Predicted word:[perhaps]
Iter= 49000, Average Loss= 0.055227, Average Accuracy= 98.80%
[',', 'and', 'thought'] - Actual word:[to] vs Predicted word:[to]
Iter= 49500, Average Loss= 0.068357, Average Accuracy= 98.60%
['dear', ',', 'and'] - Actual word:[that] vs Predicted word:[that]
Iter= 50000, Average Loss= 0.043755, Average Accuracy= 99.40%
['affectionately', 'into', "alice's"] - Actual word:[,] vs Predicted word:[,]

Training Completed!

"Complete sentence follows!'
i only wish off together. alice was very glad to find her in such a pleasant temper , and
thought to herself that perhaps it was only the pepper that
```

We can see from the output of Listing 4-4 that the model is able to predict the actual words while training quite nicely. In the sentence-completion task, although the prediction didn't start off well with the first two predictions, it did an excellent job for the rest of the 28 characters. The generated sentence is exceptionally rich in grammar and punctuation. The model accuracy can be increased by increasing the sequence length and also by introducing predictions after every word in the sequence. Also, the training corpus was small. Word-prediction and sentence-completion quality will be further enhanced if the model is trained on a larger corpus of data. Listing 4-5 shows the passage from *Alice in Wonderland* that was used to train the model.

Listing 4-5

```
' You can't think how glad I am to see you again , you dear old thing ! ' said the Duchess ,
as she tucked her arm affectionately into Alice's , and they walked off together . Alice was
very glad to find her in such a pleasant temper , and thought to herself that perhaps it
was only the pepper that had made her so savage when they met in the kitchen . ' When I'm a
Duchess , ' she said to herself , ( not in a very hopeful tone though ) , ' I won't have any
pepper in my kitchen at all . Soup does very well without — Maybe it's always pepper that
makes people hot-tempered , ' she went on , very much pleased at having found out a new kind
of rule , ' and vinegar that makes them sour — and camomile that makes them bitter — and —
and barley-sugar and such things that make children sweet-tempered . I only wish people knew
that : then they wouldn't be so stingy about it , you know — 'She had quite forgotten the
Duchess by this time , and was a little startled when she heard her voice close to her ear
. ' You're thinking about something , my dear , and that makes you forget to talk . I can't
tell you just now what the moral of that is , but I shall remember it in a bit . ' ' Perhaps
it hasn't one , ' Alice ventured to remark . ' Tut , tut , child ! ' said the Duchess . '
Everything's got a moral , if only you can find it . ' And she squeezed herself up closer
to Alice's side as she spoke . Alice did not much like keeping so close to her : first ,
```

```
because the Duchess was very ugly ; and secondly , because she was exactly the right height
to rest her chin upon Alice's shoulder , and it was an uncomfortably sharp chin . However,
she did not like to be rude , so she bore it as well as she could .
```

Gated Recurrent Unit (GRU)

Much like LSTM, the gates recurrent units, popularly known as GRU, have gating units that control the flow of information inside. However, unlike LSTM, they don't have separate memory cells. The hidden memory state h_t at any time step t is a linear interpolation between previous hidden memory states h_{t-1} and candidate new hidden state $\tilde{h}_t$. The architectural diagram of a GRU is presented in Figure 4-14.

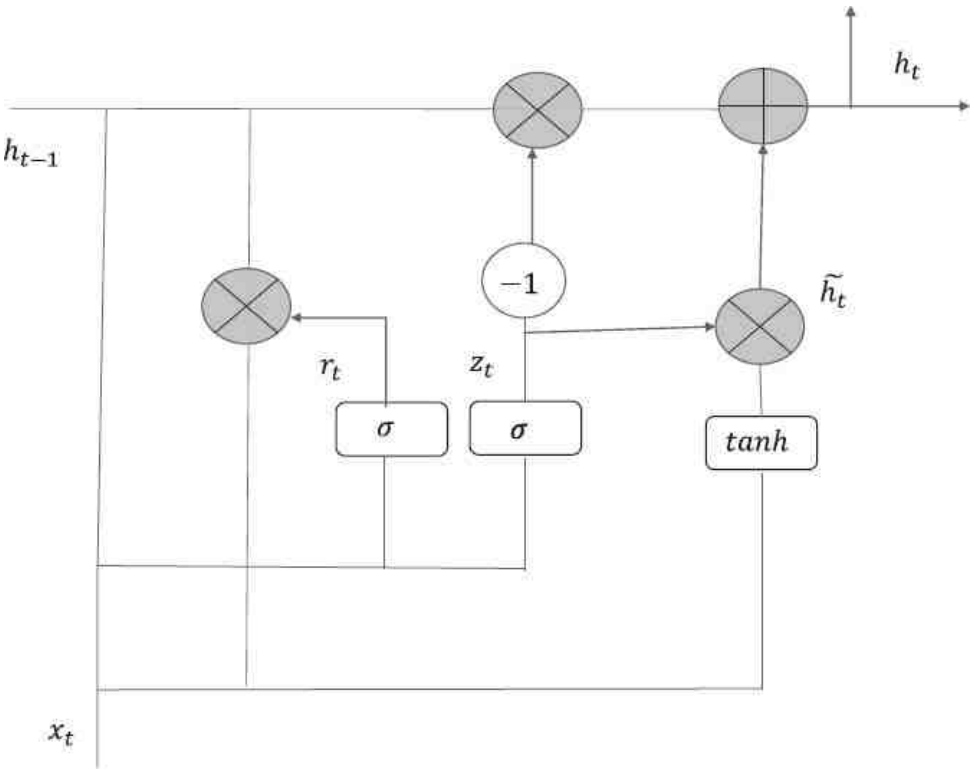

Figure 4-14: GRU architecture

The following are the high-level details as to how the GRU works:

- Based on the hidden memory state h_{t-1} and current input x_t, the reset gate r_t and the update gate z_t are computed as follows:

  $$r_t = \sigma\left(W_r h_{t-1} + U_r x_t\right)$$

  $$z_t = \sigma\left(W_z h_{t-1} + U_z x_t\right)$$

  The reset gate determines how important h_{t-1} is in determining the candidate new hidden state. The update gate determines how much the new candidate state should influence the new hidden state.

- The candidate new hidden state $\tilde{h}_t$ is computed as follows:

  (4.4.1) $$\tilde{h}_t = \tanh\left(r_t * U h_{t-1} + W x_t\right)$$

- Based on the candidate state and the previous hidden state, the new hidden memory state is updated as follows:

  (4.4.2) $$h_t = \left(1 - z_t\right) * h_{t-1} + z_t * \tilde{h}_t$$

The key point to stress in a GRU is the role of the gating functions, as discussed here:

- When the reset gate output units from r_t are close to zero, the previous hidden states for those units are ignored in the computation of the candidate new hidden state, as is evident from the equations in (4.4.1). This allows the model to drop information that would not be useful in the future.

- When the update gate output units from z_t are close to zero, then the previous step states for those units are copied over to the current step. As we have seen before, the notorious component in recurrent neural network backpropagation that may lead to vanishing or exploding gradients is the component $\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}}$, which backpropagates the error at sequence step *t* to sequence step *k* so that the model learns long-distance dependencies or correlations. The expression for $\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}}$, as we saw in the vanishing and exploding gradients section, is given by

$$\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}} = (u_{ii})^{t-k} \prod_{g=k+1}^{t} \sigma'\left(z_g^{(i)}\right)$$

  When *(t - k)* is large, a vanishing-gradient condition will arise when the gradients of the activations in the hidden state units and/or the weights are less than 1 since the product of *(t-k)* in them would force the overall product to near zero. Sigmoid and tanh gradients are often less than 1 and saturate fast where they have near-zero gradients, thus making the problem of vanishing gradient more severe. Similarly, exploding gradients can happen when the weight connection u_{ii} between the *ith* hidden to the *ith* hidden unit is greater than 1 since the product of *(t-k)* in them would make the term $(u_{ii})^{t-k}$ exponentially large for large values of *(t-k)*.

- Now, coming back to the GRU, when the update-gate output units in z_t are close to 0, then from the equation (4.4.2),

  (4.4.3) $h_t^{(i)} \approx h_{t-1}^{(i)} \quad \forall i \in K$

  where *K* is the set of all hidden units for which $z_t^{(i)} \approx 0$.

  On taking the partial derivative of $h_t^{(i)}$ with respect to $h_{t-1}^{(i)}$ in (4.4.3) we get the following:

$$\frac{\partial h_t^{(i)}}{\partial h_{t-1}^{(i)}} \approx 1$$

  This will ensure that the notorious term $\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}}$ is also close to 1 since it can be expressed as

$$\frac{\partial h_t^{(i)}}{\partial h_k^{(i)}} = \prod_{g=t'+1}^{t} \frac{\partial h_g^{(i)}}{\partial h_{g-1}^{(i)}}$$

$$= 1.1.1.\ldots.1\,(t-k)\,times = 1$$

  This allows the hidden states to be copied over many sequence steps without alteration, and hence the chances of a vanishing gradient diminish and the model is able to learn temporally long-distance association or correlation between words.

Bidirectional RNN

In a standard recurrent neural network we make predictions that take the past sequence states into account. For example, for predicting the next word in a sequence we take the words that appear before it into consideration. However, for certain tasks in natural language processing, such as parts of speech, tagging both past words and future words with respect to a given word is crucial in determining the given word's part of speech tag. Also, for parts of speech-tagging applications, the whole sentence would be availabel for tagging, and hence for each given word—barring the ones at the start and end of a sentence—its past and future words would be present to make use of.

Bidirectional RNNs are a special type of RNN that makes use of both the past and future states to predict the output label at the current state. A bidirectional RNN combines two RNNs, one of which runs forward from left to right and the other of which

runs backward from right to left. A high-level architecture diagram of a bidirectional RNN is depicted in Figure 4-15.

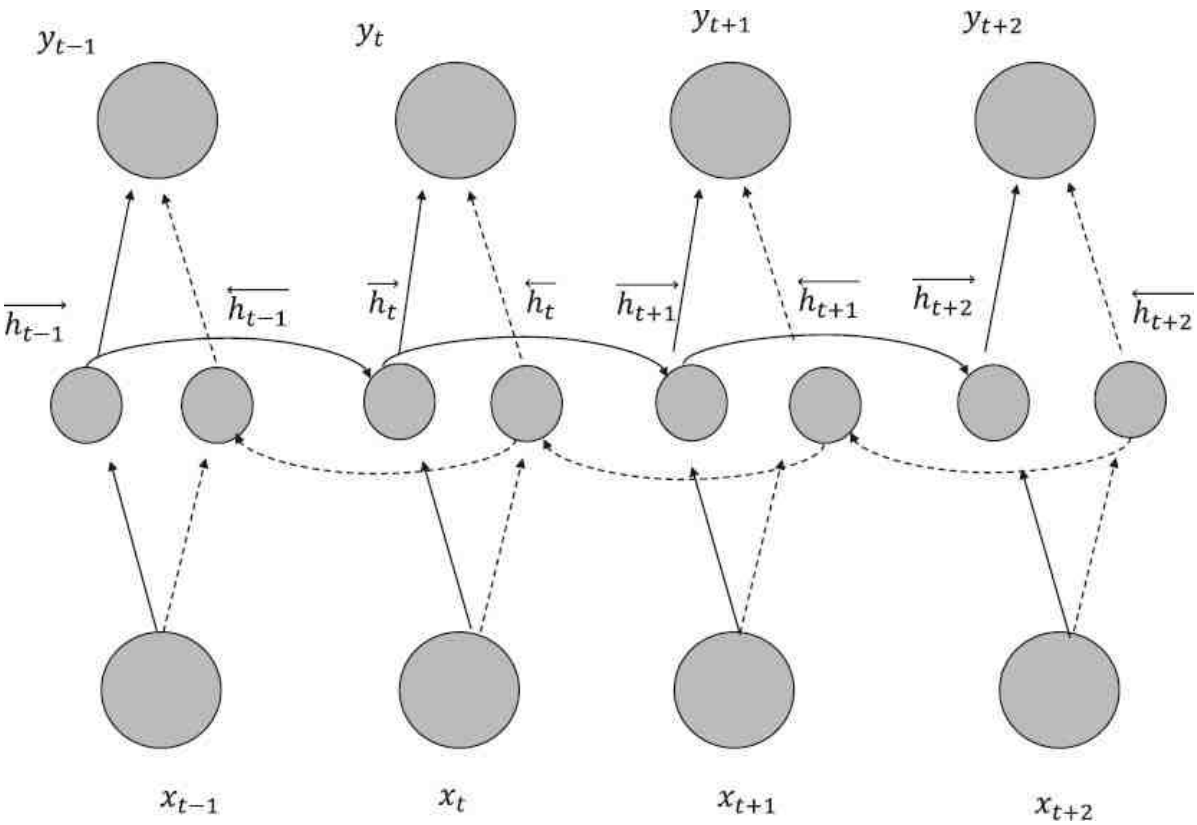

Figure 4-15: Bidirectional RNN architectural diagram

For a bidirectional RNN there are two hidden memory states for any sequence step *t*. The hidden memory states corresponding to the forward flow of information, as in standard RNN, can be represented as $\overrightarrow{h_t}$, and the ones corresponding to the backward flow of information can be denoted by $\overleftarrow{h_t}$. The output at any sequence step *t* depends on both the memory states $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$. The following are the governing equations for a bidirectional RNN.

$$\overrightarrow{h_t} = f\left(\overrightarrow{W_{xh}}x_t + \overrightarrow{W_{hh}}\overrightarrow{h_{t-1}} + \overrightarrow{b}\right)$$

$$\overleftarrow{h_t} = f\left(\overleftarrow{W_{xh}}x_t + \overleftarrow{W_{hh}}\overleftarrow{h_{t-1}} + \overleftarrow{b}\right)$$

$$y_t = g\left(U\left[\overrightarrow{h_t};\overleftarrow{h_t}\right] + c\right)$$

The expression $\left[\overrightarrow{h_t};\overleftarrow{h_t}\right]$ represents the combined memory state vector at time *t*. It can be obtained by concatenating the elements of the two vectors $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$.

$\overrightarrow{W_{hh}}$ and $\overleftarrow{W_{hh}}$ are the hidden state connection weights for the forward pass and the backward pass respectively. Similarly, $\overrightarrow{W_{xh}}$ and $\overleftarrow{W_{xh}}$ are the inputs to the hidden state weights for the forward and backward passes. The biases at the hidden memory state activations for forward and backward passes are given by $\overrightarrow{b}$ and $\overleftarrow{b}$ respectively. The term *U* represents the weight matrix from the combined hidden state to the output state, while *c* represents the bias at the output.

The function *f* is generally the non-linear activation function chosen at the hidden memory states. Chosen activation functions for *f* are generally sigmoid and tanh. However, ReLU activations are also being used now because they reduce the vanishing- and exploding-gradient problems. The function *g* would depend on the classification problem at hand. In cases of multiple classes, a SoftMax would be used, whereas for a two-class problem either a sigmoid or a two-class SoftMax could be used.

Summary

After this chapter, the reader is expected to have gained significant insights into the working principles of recurrent neural networks and their variants. Also, the reader should be able to implement RNN networks using TensorFlow with relative ease. Vanishing- and exploding-gradient problems with RNNs pose a key challenge in training them effectively, and thus many powerful versions of RNNs have evolved that take care of the problem. LSTM, being a powerful RNN architecture, is widely used in the community and has almost replaced the basic RNN. The reader is expected to know the uses and advantages of these advanced techniques, such as LSTM, GRU, and so forth, so that they can be accordingly implemented based on the problem. Pre-trained Word2Vec and GloVe word-vector embeddings are used by several RNN, LSTM, and other networks so that the input word to the model at each sequence step can be represented by its pre- trained Word2Vec or GloVe vector instead of learning these word-vector embeddings within the recurrent neural network itself.

In the next chapter, we will take up restricted Boltzmann machines (RBMs), which are energy-based neural networks, and various auto-encoders as part of unsupervised deep learning. Also, we will discuss deep-belief networks, which can be formed by stacking several restricted Boltzmann machines and training such networks in a greedy manner, and collaborative filtering through RBMs. I look forward to your participation in the next chapter.