UNIVERSITY OF CALIFORNIA

Los Angeles

An Application of Recurrent Neural Network: Prediction of Items' Price by Description

A thesis submitted in partial satisfaction
of the requirements for the degree Master of Applied Statistics

by

Beichen Su

2018

ABSTRACT OF THE THESIS

An Application of Recurrent Neural Network:
Prediction of Items' Price by Description

by

Beichen Su
Master of Applied Statistics
University of California, Los Angeles, 2018
Professor Yingnian Wu, Chair

In this application, a predictive model is built for finding the price of an item from their characteristics, especially the description. The implementation of word vectors, optimized gradient descent and long short-term memory are employed to tackle the problem. And the relatively low loss for both training and test set proves this is a reasonable approach.

The thesis of Beichen Su is approved.

University of California, Los Angeles

2018

Contents

List of Figures

1 Introduction

Setting the price for a product is getting harder at scale. This situation is getting worse in the second-hand market, that people have no idea about the value of the product they are trying to sale to achieve buyer and seller satisfaction. For example, thousands of brands, seasonal pricing trend and fashion of the clothing, the technology employed for the electronic device should all be considered as effective factor to set the price from human perspective. But what exact price should a seller tag on the product?

The traditional way might be searching the price of the related product, asking retailers' suggestion and make a discount from the original price. However, it will take a lot of effort, which might be more expensive compared to the product itself. In this thesis, machine learning algorithms will be employed to tackle this problem.

The price-prediction data is obtained on Kaggle, and the provider Mercari, Japan's biggest community-powered shopping app offers a price suggesting challenge to solve this problem as anyone can sell any legal product on their site[1]. By solving this challenge with reasonable error, sellers will benefit a lot from saving their effort to think about the price to set and the efficiency of the market will be proved significantly.

[1]Kaggle Mercari Price Suggestion Challenge: www.kaggle.com/c/mercari-price-suggestion-challenge

2 Problem definition

In this application, the ultimate task is to predict the item price from all predictors. This is a typical regression problem, and for a regression situation, there are many methods available to implement, if the numeric input predictors and target variable are given. However, the main challenge is that none of the predictors is the number which can be simply feed into a model such as linear regression, random forest and neural network.

Now the problem is transformed from a simple regression case to a feature extraction case. Categorical predictors like brand and category name need to be encoded to numerical value. Especially, the natural language processing need to be performed for the item description, which are almost sentences, so that proper word vector representation will be introduced for each word in the vocabulary at first, and a recurrent neural network, especially Long Short Term Memory(LSTM) will be employed to conduct the sentence vector for each observation.

3 Methodology

In this section, the main approaches used to conduct the result are introduced below. The structure of neural network will not be emphasized here.

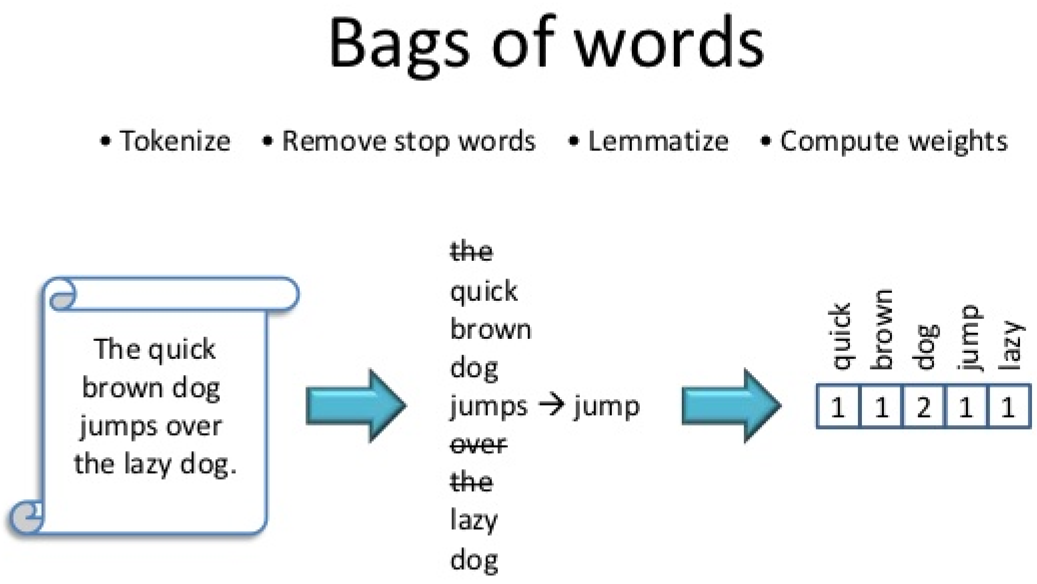
3.1 Word to Vector

To begin with encoding for description, we must realize that there are millions of sentences stored in this description column, providing information of the product. Sentence, sometimes can be treated as factor levels when they are short, and the dimension of levels is not too big. In this case, discussion of different product by their sellers varies and the information in the sentence need to be understood properly.

3.1.1 Bag of words

A typical approach would be Bag of Words[1]. Sentences are tokenized into single word, and the unique words from all sentences which sometimes is called corpus, will be collected to form a vocabulary. Optimization such as limit the minimum word count and remove the punctuation and stop words, can be performed. And a vector with length of the vocabulary will be created for each sentence. The count of unique words in the sentence will be placed in the corresponding index in the vector. A demo is given below:

Figure 1: Bag of word demo



Bags of words
• Tokenize • Remove stop words • Lemmatize • Compute weights
The quick brown dog jumps over the lazy dog.
~~the~~
quick
brown
dog
jumps → jump
~~over~~
~~the~~
lazy
dog
quick brown dog jump lazy
1 1 2 1 1

Source: https://www.slideshare.net/mgrcar/text-and-text-stream-mining-tutorial-15137759

It's worth noticing that the frequency of words in a sentence is preserved whereas the order is lost. This could introduce sever problem with some situation where two sentences have the exact same words but opposite meaning. On the other hand, this approach will create a huge vector for each sentence, with length of the vocabulary and most of the element is 0, that is, a sparse matrix will be introduced causing decrease training efficiency. The worst part is that the relationship between similar words cannot be observed from this distributed representation, which means the sentence vector actually does not have a real meaning except for the binding with the vocabulary. Meaningful word vector encoding is in demand and the word to vector approach will be introduced below.

3.1.2 W2V

In order to find the meaningful representation of the words, including the words' relationship with the context, the word2vec model[2] is introduced.

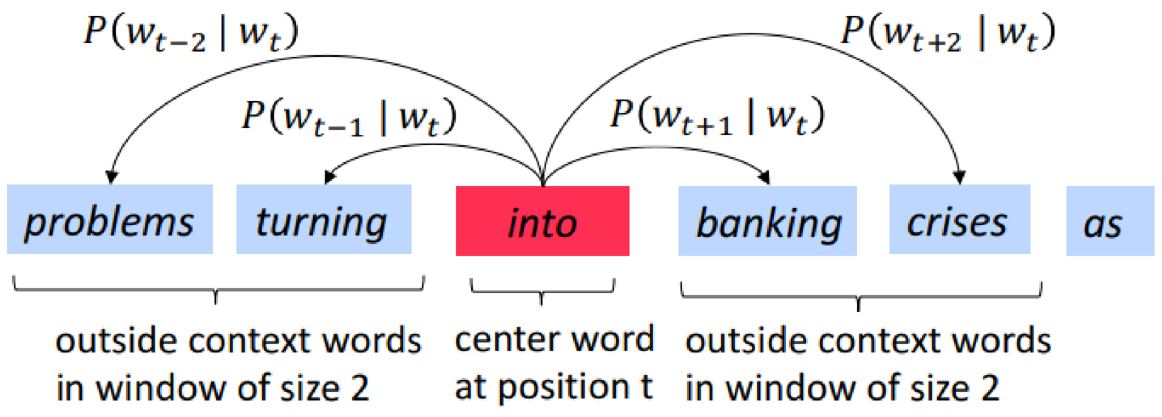
First, we need to consider the similarity of the word vector. Suppose two word vectors are given, a straight forward way to measure the similarity between them is the dot product.

By the definition of the dot product, given two vector u and v, the dot product is defined by: $u \cdot v = |u||v|cos(\theta)$. As the angle θ between those two vectors is getting smaller, the value of the dot product is getting larger, which implies that the larger the dot product, the larger the similarity.

From this perspective, we can build a model to train the word vector based on the similarity and the intuition that similar words appear in the similar position in the context. The skip gram w2v model is introduced here, and the main idea about the skip gram model is that given a word and its position in the sentence, we are trying to predict the words around it. That is, at position t, which is the current center, we have word w_t, and we are trying to maximize the probability of occurrence of surrounding word such as $P(w_{t-1}|w_t), P(w_{t+1}|w_t)$ given this w_t, as shown in the demo below with window size 2.

And hence a maximization problem is defined, such that we want to maximize the product of the $P(w_{t-2}|w_t), P(w_{t-1}|w_t), P(w_{t+1}|w_t), P(w_{t+2}|w_t)$ at each time step t while we are treating a sentence as sequence. To go over a sentence, we need multiply all of this product at each

Figure 2: Skip gram demo



Source: http://web.stanford.edu/class/cs224n/lectures/lecture2.pdf

time step, where the higher the probability the better the word vector. That is we want to maximize the likelihood:

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \leq i \leq m, i \neq 0} P(w_{t+i}|w_t;\theta)$$

Where t for each position in the sentence. m for the window size, normally 2. And θ for all the parameter needed to tune. It's always convenient to transform the Likelihood function to a log likelihood in order to remove the multiply sign and by adding a negative sign and average over the sentence length, a loss function is obtained for each sentence.

$$Loss(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq i \leq m, i \neq 0} log(P(w_{t+i}|w_t;\theta))$$

With loss function defined, we need $P(w_{t+i}|w_t)$ to minimize the loss. Therefore, for each unique word in the vocabulary, define two vectors: v_w when the word w is the center word and u_w when the word w is in the context. The Softmax function gives the probability of occurrence of the context word w_{t+1}given current center word w_t:

$$P(w_{t+1}|w_t) = \frac{e^{u_{t+1}^T v_t}}{\sum_{w \in Vocab} exp(u_w^T v_t)}$$

Then with a one hot input vector with vocabulary length will be passed in to a single layer neural network without activation and gradient descent to minimize the loss function. Unlike the normal usage of neural network, we use the weight matrix as our word vector.

3.2 Gradient descent

The gradient descent is the way we train the parameter of a neural network in the backward propagation.

Suppose we have loss function: $L(\theta)$, and for all sample in the training set, the loss is given by $L(\theta) = \frac{1}{n}\sum_{i=1}^{n} L_i(\theta)$. And we can update our parameter by $\theta_{next} = \theta_{current} - \eta L'(\theta)$. As long as the number of training sample grows, this step takes a huge amount of computation.

3.2.1 Stochastic gradient descent

The stochastic gradient descent[3] is a way to optimize the gradient, by update and computes the gradient just using only a few training examples from training. That is we can feed the neural network with a mini-batch at each step and update our parameter accordingly. Supose the size of our batch is k, then $\theta_{next} = \theta_{current} - \eta \sum_{i=1}^{k} L_i'(\theta)$, where η is the learning rate or step size in each update, which is always small take the step carefully.

3.2.2 Momentum

Sometimes the Stochastic gradient descent method might have trouble to navigate itself to the true direction of minimum when in one dimension the contour curve is steeper than all other dimension, which results in a lot of oscillation while updating the parameters.

The Momentum [4] is an acceleration method to help Stochastic gradient descent converge by reducing the oscillation. The gradient always follows the steepest direction, while it's not the best case as described above, so that it's even better to let the gradient move along the direction of momentum.

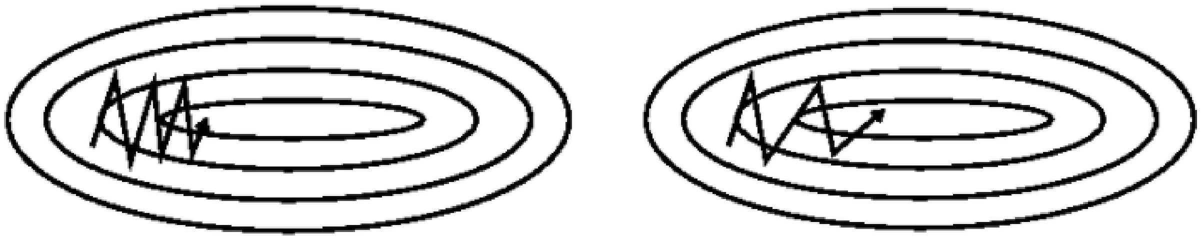


Figure 3: SGD without Momentum(left) v.s SGD with Momentum(right)

Source: http://ruder.io/optimizing-gradient-descent/

The updated rule is given by:

$$v_t = \gamma v_{t-1} + \eta g_t$$

$$\theta_t = \theta_{t-1} - v_t$$

where g_t is the averaged gradient given by the current batch and v_t is the momentum with a constant parameter γ associate with it, usually 0.9. Basically with momentum, we are pushing gradient to the minimum of loss function as we push a ball down from a mountain, and the momentum is accumulated, moving even faster to the local minimum.

3.2.3 Adagrad

The Adagrad optimaizer[5] modifies the gradient descent as the magnitudes of the average gradient g_t is pretty uneven and we want the model to be adaptive to this situation. And a sum of gradient magnitude at each step is introduced in the model:

$$G_t = G_{t-1} + g_t^2$$

$$\theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{G_t + \epsilon}}$$

3.2.4 RMSprop and Adam

The RMSprop[6] and Adam[7] are further optimizer based on the intuition of Adagrad. The G_t in Adagrad is the sum of gradient over all time steps and it's better to focus on the recent time steps to guide the gradient into the appropriate direction of descent, the G_t is modified by the following for the RMSprop:

$$G_t = \beta G_{t-1} + (1 - \beta) g_t^2$$

While the Adam optimizer combine the idea of momentum and RMSprop, that momentum is included in the calculation of gradient and the sum of gradient is applied to make the average gradient for the recent time steps, the update rules is given by:

$$v_t = \gamma v_{t-1} + (1 - \gamma) g_t$$

$$G_t = \beta G_{t-1} + (1 - \beta) g_t^2$$

$$v_t = \frac{v_t}{1 - \gamma}$$

$$G_t = \frac{G_t}{1 - \beta}$$

$$\theta_{t+1} = \theta_t - \eta \frac{v_t}{\sqrt{G_t + \epsilon}}$$

3.3 Batch Normalization

It's sometimes problematic that the distribution of the output of each fully connected layer, h_l keeps changing during the training. A stabilized method, Batch Normalization[8], is introduced by adding a normalization layer between h_l and h_{l-1}.

Let h_{li} be the element in the l layer. Compute mean and variance of all h_{li}, which is given by μ_l, σ_l. The normalized element in the l layer is given by $\hat{h_{li}} = \frac{h_{li}-\mu_l}{\sigma_l}$, and finally the output is given by a linear transformation: $h_{lb} = \alpha + \beta\hat{h_{li}}$. The parameter α, β is learnt from gradient descent.

3.4 Long short-term memory

Sometimes we need computer to have memory of the past to predict what would happen in the future. And this is something normal forward fully connected neural network lacks of.

Recurrent neural network solves this problem by looping through the input and persist the information, and this is suitable for list like sequential data such as sentences.

However, as the sequential gets long, it's very hard to back propagate using the gradient descent and it's hard for a recurrent neural network to connect the information when the gap between two data in a sequence is big.

Therefore, Long short term memory(LSTM)[9] is introduced to solve above problem. This is a special recurrent neural network with carefully design to solve the long term dependency issue cased by a normal recurrent neural network. LSTM, Just like recurrent neural network, the unfolded network has a chain structure, but the difference is that unlike the normal recurrent neural network with a single activation function, LSTM introduced input, forget and output gate, which ensure that only valuable information will be passed through. A comparison between a normal recurrent neural network cell and LSTM cell is given below.

Figure 4: RNN Cell

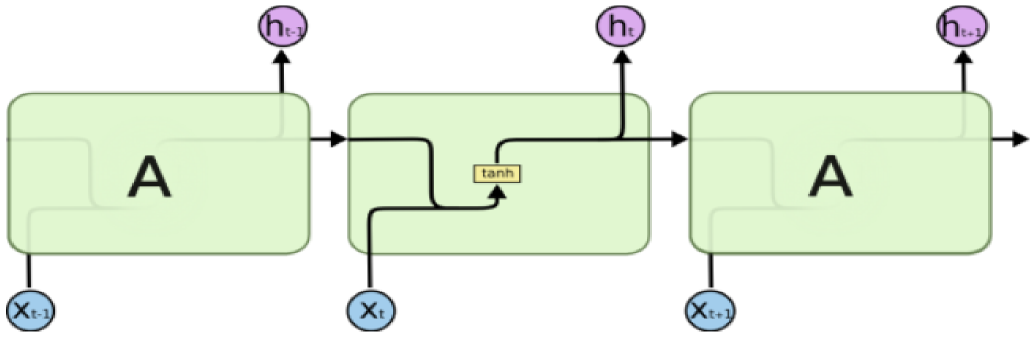
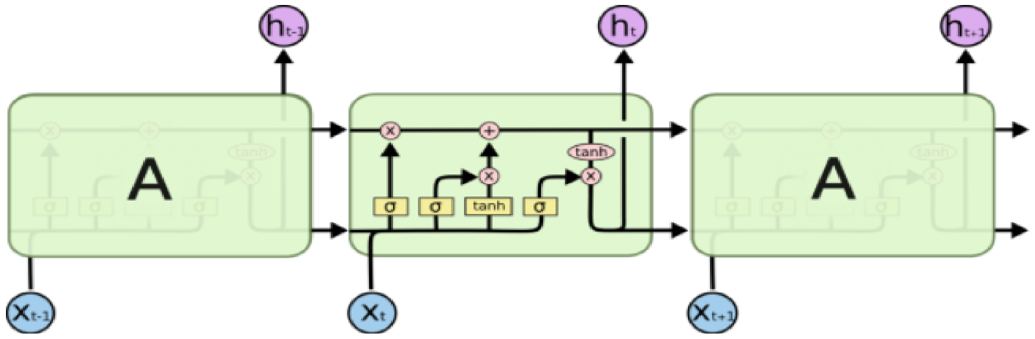


Figure 5: LSTM Cell



Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

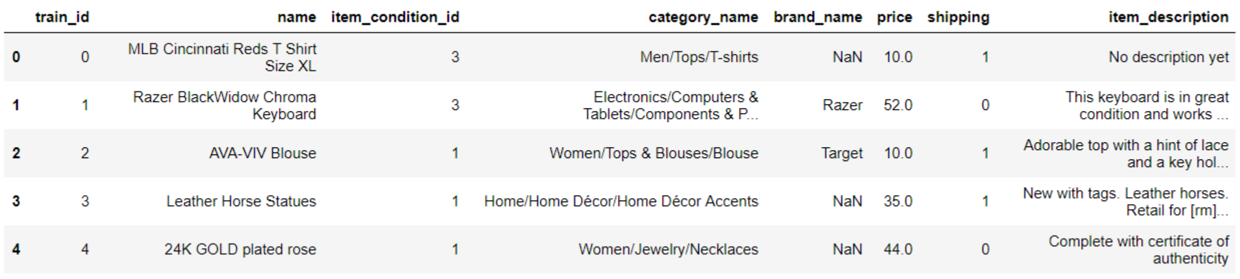
4 Data prepossessing

Before any mathematical modeling and analysis, understanding the data structure is always the first step. In this section, the exploratory analysis will be performed, where the distribution of the price and the relationship between the depend variable price and other predictors will be studied. Also, missing value will be handled according to specific situation.

The raw data is given by 2 sets, a training sets containing all predictor column and the dependent variable price, while the price is left to be computed in the test set. To begin with, the data processing will be applied to the training set.

The head of the data is given by:

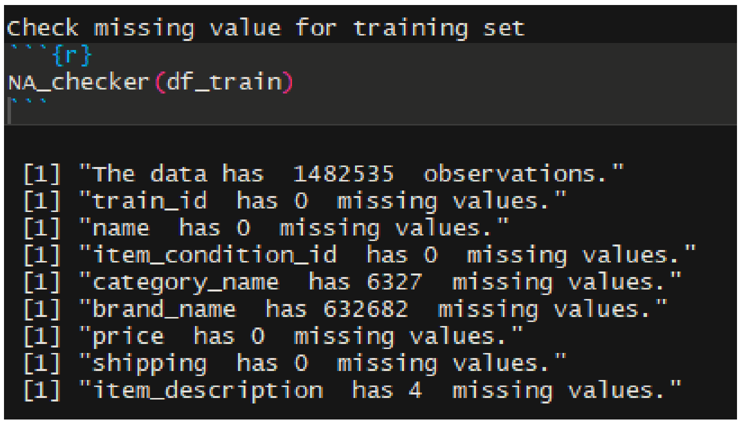
Figure 6: Head of the data



	train_id	name	item_condition_id	category_name	brand_name	price	shipping	item_description
0	0	MLB Cincinnati Reds T Shirt Size XL	3	Men/Tops/T-shirts	NaN	10.0	1	No description yet
1	1	Razer BlackWidow Chroma Keyboard	3	Electronics/Computers & Tablets/Components & P...	Razer	52.0	0	This keyboard is in great condition and works ...
2	2	AVA-VIV Blouse	1	Women/Tops & Blouses/Blouse	Target	10.0	1	Adorable top with a hint of lace and a key hol...
3	3	Leather Horse Statues	1	Home/Home Décor/Home Décor Accents	NaN	35.0	1	New with tags. Leather horses. Retail for [rm]...
4	4	24K GOLD plated rose	1	Women/Jewelry/Necklaces	NaN	44.0	0	Complete with certificate of authenticity

The missing value of the training set is given by:

Figure 7: Missing value for training data



~~~
Check missing value for training set
```{r}
NA_checker(df_train)
```

 [1] "The data has  1482535  observations."
 [1] "train_id  has 0  missing values."
 [1] "name  has 0  missing values."
 [1] "item_condition_id  has 0  missing values."
 [1] "category_name  has 6327  missing values."
 [1] "brand_name  has 632682  missing values."
 [1] "price  has 0  missing values."
 [1] "shipping  has 0  missing values."
 [1] "item_description  has 4  missing values."
~~~

In general, all the predictors are categorial and the test is to regression on the item price. Carefully encoding and filling for missing values are the first step, which will be performed below.

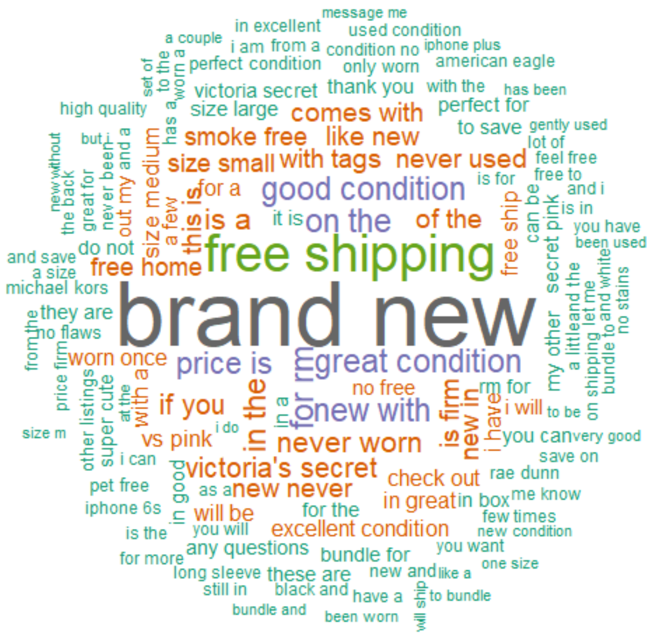
4.1 Item name and description

It's worth to mention that though only 4 NA reported the item description, the missing value in this column is presented as "No description yet" or "No description". There are 82731 missing values for the description.

As both item name and item description describe what the item is, and luckily, we have no missing item name, the simple combination of the item name and description will provide information about this item and the NAs in the description will be filled by item name accordingly.

At this step, name and description will be merged, meanwhile NAs and "No description" will be replaced by the item name. The name column will be dropped, and all the description of the item will be stored in the item description.

Figure 8: Word cloud for item name and description

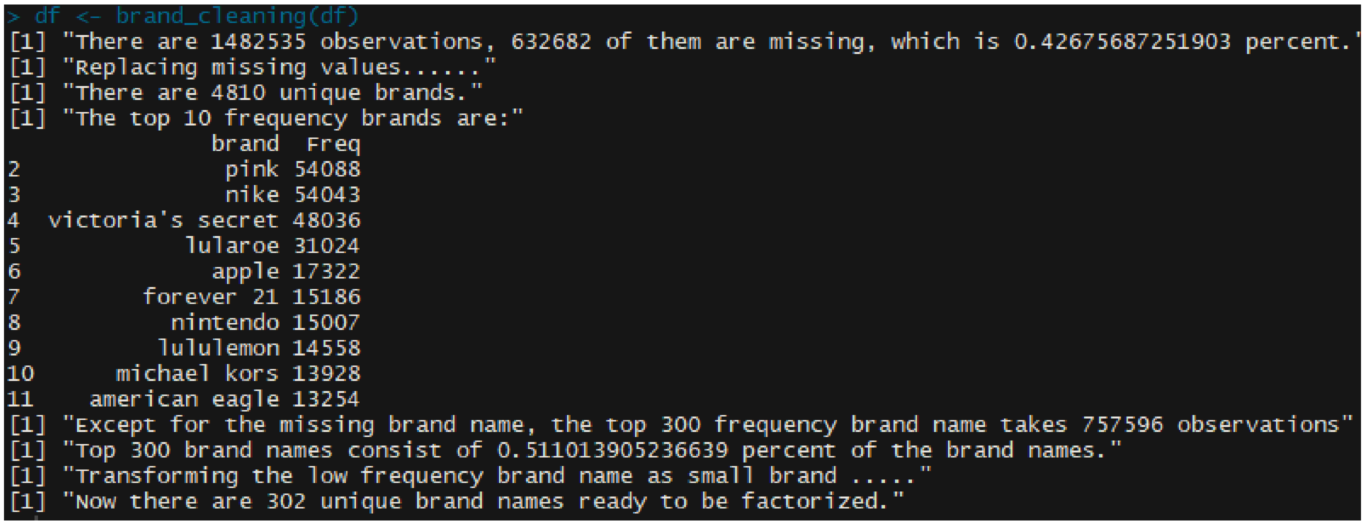


4.2 Brand name

As discussed in the introduction, the brand name takes a significant part in the selling price. However, 632,682 of them are missing, and the reason might be that the items don't really have a noticeable brand, or the sellers forget the brand. I think the first reason should be the majority, but to fill those NA, "missing brand name" will be placed.

Different from the description, the brand name are just a few words normally, and doesn't really have any literally meaning by the explanation. A good approach should be treat them as factor with one hot encode to pass in to the model as numeric values. However, this might not be the case in this data, because there are too many brands. So that the NAs are filled by "missing brand", and to lower the dimension of the one hot encoding, I picked top 300 brands and except for them and the missing ones, the remaining will be replaced by "small brand", as they don't really contribute much. A brief report for the operation on brand names is provided below:

Figure 9: Brand name report

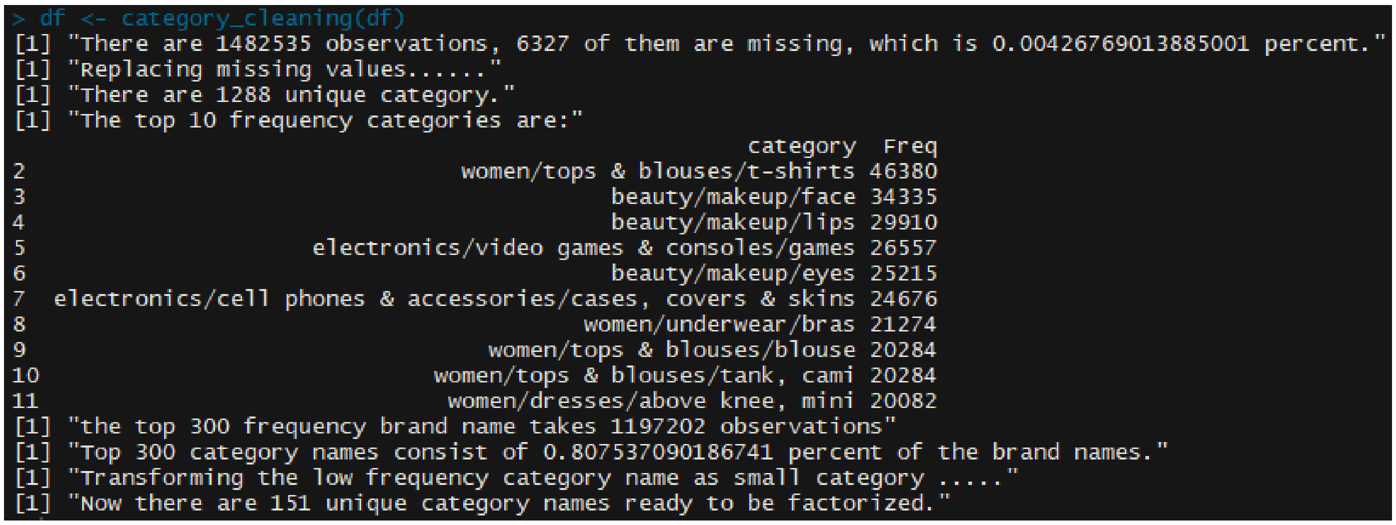


```
> df <- brand_cleaning(df)
[1] "There are 1482535 observations, 632682 of them are missing, which is 0.42675687251903 percent.
[1] "Replacing missing values......"
[1] "There are 4810 unique brands."
[1] "The top 10 frequency brands are:"
              brand  Freq
2              pink 54088
3              nike 54043
4  victoria's secret 48036
5           lularoe 31024
6             apple 17322
7        forever 21 15186
8          nintendo 15007
9         lululemon 14558
10     michael kors 13928
11   american eagle 13254
[1] "Except for the missing brand name, the top 300 frequency brand name takes 757596 observations"
[1] "Top 300 brand names consist of 0.511013905236639 percent of the brand names."
[1] "Transforming the low frequency brand name as small brand ....."
[1] "Now there are 302 unique brand names ready to be factorized."
```

4.3 Category name

Like the brand name, the category column consists of many sub-categories and missing value. As there are only 1288 unique categories, top 150 will be kept and the remaining will be replaced by a level called "small category". And therefore 151 levels will be left for factorization. A brief report is given below:

Figure 10: Category name report



```
> df <- category_cleaning(df)
[1] "There are 1482535 observations, 6327 of them are missing, which is 0.00426769013885001 percent."
[1] "Replacing missing values......"
[1] "There are 1288 unique category."
[1] "The top 10 frequency categories are:"
                                                     category  Freq
2                                 women/tops & blouses/t-shirts 46380
3                                          beauty/makeup/face 34335
4                                          beauty/makeup/lips 29910
5                      electronics/video games & consoles/games 26557
6                                          beauty/makeup/eyes 25215
7  electronics/cell phones & accessories/cases, covers & skins 24676
8                                        women/underwear/bras 21274
9                                   women/tops & blouses/blouse 20284
10                               women/tops & blouses/tank, cami 20284
11                               women/dresses/above knee, mini 20082
[1] "the top 300 frequency brand name takes 1197202 observations"
[1] "Top 300 category names consist of 0.807537090186741 percent of the brand names."
[1] "Transforming the low frequency category name as small category ....."
[1] "Now there are 151 unique category names ready to be factorized."
```

4.4 Item condition and shipping

The item condition is originally encoded from good to bad as 1 to 5. This dummy encoding works fine for this case as the number reflects badness of a product. So, I will leave it there.

On the other hand, the shipping column consists only 0 and 1 originally. This one hot encoding is also a good implementation that the two levels are carefully discriminated.

5 Models

A word to vector model based on skip gram algorithm and a deep neural network model for predicting the price will be introduced.

5.1 W2V model

This is the first step to let the computer understand the meaning of item description in order to extract higher dimensional feature for prediction the price.

To begin building this model, all sentences are tokenized into list of lists of word tokens. Then punctuation and stop words, such as "this", "that" and "the" are removed from the tokens, as they generally will not provide real meanings. The filtered tokens are ready to be feed in to the word2vec model, implemented by Gensim[2], a python natural language processing package. This word2vec model takes the the list of list of tokens as the main input and the desired word vector length can be specified with a minimum word count given, which will filter the vocabulary again and only the word with higher frequency will be vectored.

Two models are trained based on the skip gram algorithm, and the first with the minimum word count 10 while the second takes 50. The result of those two models can be visualized by calling the most similar word given a word, and the word vectors with the biggest dot product will be returned. The training step explanation with code is following:

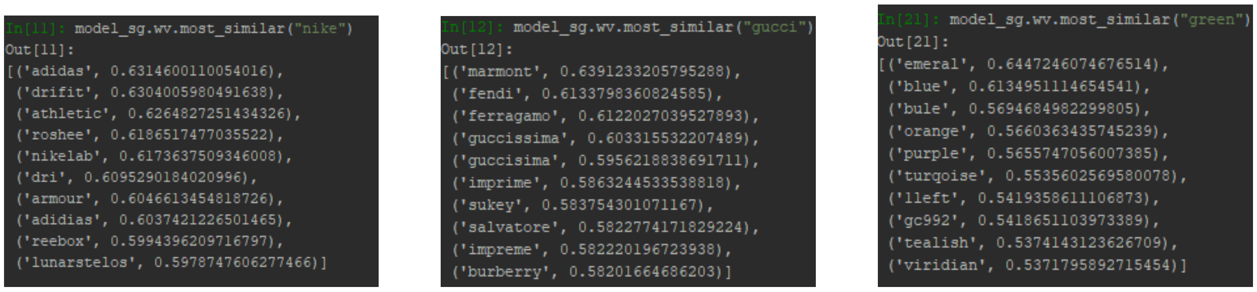
```
description_token = sentence_tokenizer(description, stop_words)


model_sg_1 = gensim.models.Word2Vec(description_token,
                                   min_count=50, size=300, sg=1)
```

The sentence will be tokenized for each observation, meaningless words which is provided

[2]https://radimrehurek.com/gensim/index.html

by the stop words will be removed. Then the Gensim[10] package in python provides a Word2Vec function to training the model. Minimum count will not take words with less than 50 occurance into the vocabulary, 300 denotes the word vector size and sg for using the skip gram algorithm.

The sample output is given below:

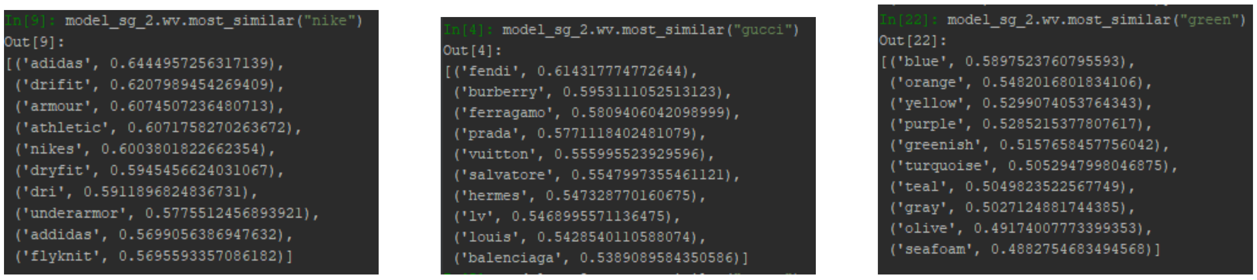


```
In[11]: model_sg.wv.most_similar("nike")
Out[11]:
[('adidas', 0.6314600110054016),
 ('drifit', 0.6304005980491638),
 ('athletic', 0.6264827251434326),
 ('roshee', 0.6186517477035522),
 ('nikelab', 0.6173637509346008),
 ('dri', 0.6095290184020996),
 ('armour', 0.6046613454818726),
 ('adidias', 0.6037421226501465),
 ('reebox', 0.5994396209716797),
 ('lunarstelos', 0.5978747606277466)]
```

```
In[12]: model_sg.wv.most_similar("gucci")
Out[12]:
[('marmont', 0.6391233205795288),
 ('fendi', 0.6133798360824585),
 ('ferragamo', 0.6122027039527893),
 ('guccissima', 0.603315532207489),
 ('guccisima', 0.5956218838691711),
 ('imprime', 0.5863244533538818),
 ('sukey', 0.58375430107167),
 ('salvatore', 0.5822774171829224),
 ('impreme', 0.58222019672393 8),
 ('burberry', 0.5820166468620 3)]
```

```
In[21]: model_sg.wv.most_similar("green")
Out[21]:
[('emeral', 0.6447246074676514),
 ('blue', 0.6134951114654541),
 ('bule', 0.5694684982299805),
 ('orange', 0.5660363435745239),
 ('purple', 0.5655747056007385),
 ('turqoise', 0.5535602569580078),
 ('lleft', 0.5419358611106873),
 ('gc992', 0.5418651103973389),
 ('tealish', 0.5374143123626709),
 ('viridian', 0.5371795892715454)]
```

Figure 11: W2V outcome with minimum word count 10



```
In[9]: model_sg_2.wv.most_similar("nike")
Out[9]:
[('adidas', 0.6444957256317139),
 ('drifit', 0.6207989454269409),
 ('armour', 0.6074507236480713),
 ('athletic', 0.6071758270263672),
 ('nikes', 0.6003801822662354),
 ('dryfit', 0.5945456624031067),
 ('dri', 0.5911896824836731),
 ('underarmor', 0.5775512456893921),
 ('addidas', 0.5699056386947632),
 ('flyknit', 0.5695593357086182)]
```

```
In[4]: model_sg_2.wv.most_similar("gucci")
Out[4]:
[('fendi', 0.614317774772644),
 ('burberry', 0.5953111052513123),
 ('ferragamo', 0.5809406042098999),
 ('prada', 0.577111840248107 9),
 ('vuitton', 0.555995523929596),
 ('salvatore', 0.5547997355461121),
 ('hermes', 0.547328770160675),
 ('lv', 0.546899557113647 5),
 ('louis', 0.5428540110588074),
 ('balenciaga', 0.5389089584350586)]
```

```
In[22]: model_sg_2.wv.most_similar("green")
Out[22]:
[('blue', 0.5897523760795593),
 ('orange', 0.5482016801834106),
 ('yellow', 0.5299074053764343),
 ('purple', 0.5285215377807617),
 ('greenish', 0.5157658457756042),
 ('turquoise', 0.5052947998046875),
 ('teal', 0.5049823522567749),
 ('gray', 0.5027124881744385),
 ('olive', 0.4917400777339935 3),
 ('seafoam', 0.4882754683494568)]
```

Figure 12: W2V outcome with minimum word count 50

It nice to observe that both models did a decent job on recognizing the similar word vectors. As we are trying to predict the price based on the items description, the brand name is quite important to learn. In the outcome shown above, in the searching of similar word based on the dot product similarity, the different brand "Nike" and "Gucci" gives impressive outputs. The similar words associated with "Nike" are other sport brands such as "Adidas" and "Under armor", as well as some unique feature of "Nike" product, such as "Dry fit". On the other hand, the similar words for a luxury brand "Gucci" are given by other Luxury brands, such as "Fendi", "Burberry" and "Prada". This result shows the power of those word vectors, where the meaning is assigned to each word so that the further training has a strong background to rely on. For reason of reducing the cost of calculation and as we have more than 1,400,000 sentences for our corpus, I choose the second model with minimum word count 50 for further training, which has about 10,000 less words in the vocabulary.

5.2 LSTM model

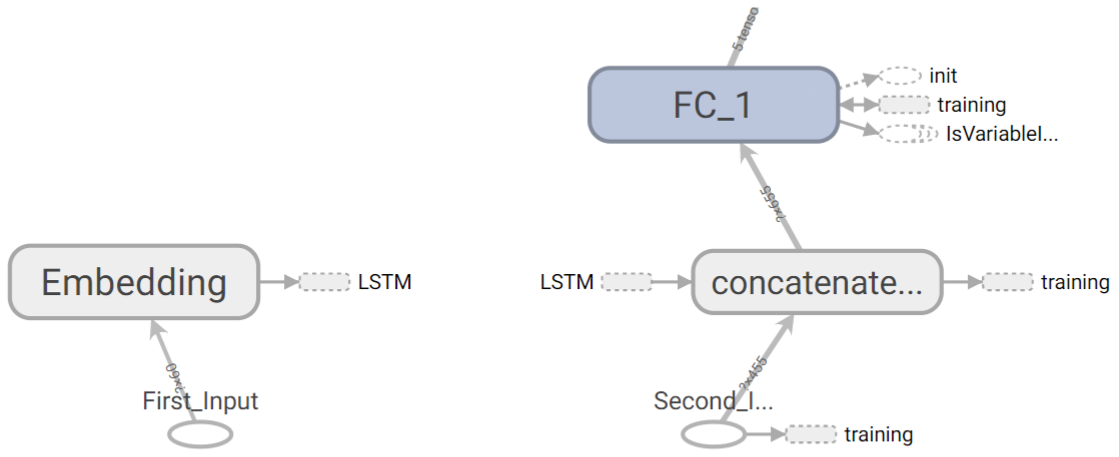
As the word vector is well developed, we have to consider the representation of the sentence. The sentence is actually a list or sequence of words, and the order of those words contributes a big amount to the grammar perspective, which can be further considered as the information in a sentence from a high level.

A human being reads a sentence from the beginning to the end, and therefore conducts the meaning of the sentence after all the information in the sentence, such as the meaning of the words, the grammar, and even the background of the context. We are expecting the computer to do the same, and this is where LSTM comes in.

If we feed a LSTM cell by a sentence, by the implemented input, forget and output gate, only useful meaning of the words will be passed through in our expectation. And the final output layer will produce a high dimensional feature which can be considered as the vector representation of the sentence.

However, unlike the W2V, we don't really have a guide to regulate the similarity of a sentence as we use the dot product to measure the similarity of two words and we need to adjust the sentence vector all the way from the target variable price. So, the LSTM served as a sentence reader in the network structure and the output high dimension vector for the sentence will be concatenated with other predictors, such as category name, item condition and shipping status. All of them will be treated as predictors for predicting the price of the item. The only difference between this structure and normal neural network for regression is that the sentence vector is not provided and is the output by the LSTM layer.

Figure 13: LSTM layer by tensorboard



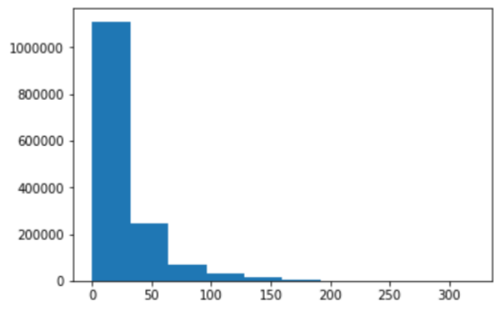
It's necessary to mention the embedding layer between the sequence input layer between and the LSTM layer. This is a mapping from the given index in the sequence into a higher dimension space. In this natural language processing case, the appropriate embedding matrix should be a matrix consist of meaningful word vectors, or simply a look up table for word vectors.

Besides, LSTM takes a fixed length of input sequence, whereas the length of sentences varies. So, it's important to pad all sentence to a fix length. This may cause loss of information if some sentences have bigger length than this threshold and we have to fill 0 to the sentences with less length than the threshold. As only a forward LSTM layer employed, 0s will be padded into the beginning of the sentence and we can choose the threshold of the maximum length from the distribution of length of the sentences. We can observe that most of the sentences in the description has length less than 60.

5.3 Neural network regression model

A neural network model is employed to predict the price of item from the brand name, category name, item condition, shipping status and lastly the output feature of LSTM. The

Figure 14: Sentences' length distribution



model employed batch normalization method to normalize the distribution of output of each dense layer and several drop out layer to prevent from over fitting. The whole structure is given by:

And as a remark, the main structure of the Keras[11] implementation of neural network on top of tensorflow[12] is given by:

```
main_input = Input(shape=(60,), dtype='int32', name='First_Input')
```

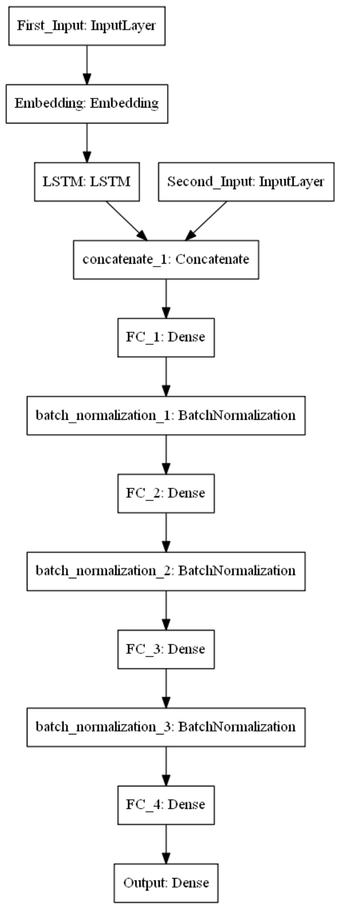
The main input is the padded description sequence with max length 60. The input will be fed to the embedding layer in order to look up the word vector will be load once we fit the model.

```
x = Embedding(output_dim=vector_size, input_dim=word_num,
              input_length=60, weights=[embedding_matrix],
              trainable=False, name="Embedding")(main_input)
```

The embedding layer takes the W2V look up table as the embedding matrix. The dimension of the embedding matrix is given by the length of vocabulary and the length of word vectors. We don't want to make any change to the embedding matrix as it's meaningful vector from W2V model, and the output is the word vector for each word in each sequence, which will be fed into LSTM layer.

```
lstm_out = LSTM(200, recurrent_dropout=0.1,
                dropout=0.1, name="LSTM")(x)
```

Figure 15: Neural network regression model structure


First_Input: InputLayer
Embedding: Embedding
LSTM: LSTM
Second_Input: InputLayer
concatenate_1: Concatenate
FC_1: Dense
batch_normalization_1: BatchNormalization
FC_2: Dense
batch_normalization_2: BatchNormalization
FC_3: Dense
batch_normalization_3: BatchNormalization
FC_4: Dense
Output: Dense

The output vector of length 200 stands for a dimension 200 for the sentence vector.

```
auxiliary_input = Input(shape=(455,), name="Second_Input")
```

The second level input including all predictors except for the description are loaded.

```
x = keras.layers.concatenate([lstm_out, auxiliary_input])
```

The sentence vector will be combined with the second level input and will be feed forward to the predictive neural network.

```
x = Dense(256, activation='relu', name="FC_1")(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
```

Fully connected, batch normalization and dropout layers are followed in order to perform high dimensional feature extraction. This is repeated for several times.

```
main_output = Dense(1, activation='relu', name='Output')(x)
model = Model(inputs=[main_input, auxiliary_input],
                  outputs=[main_output])
model.compile(loss="mean_squared_logarithmic_error",
                  optimizer='Adagrad', metrics=["mae"])
```

The output layer has dimension 1 as we are doing a regression. A model will be created and compiled, with defined loss and optimizer.

```
model.fit([train_x_1, train_x_2], y_train,
            batch_size=5000, epochs=5, validation_split=0.3,
            callbacks=[tensorboard, csv_logger])
```

Finally we train the model with input given and validation set configured.

The result of the predictive model will be analyzed in the next section.

6 Analysis of the result

As a remark, this application goes through the following steps. The training process takes all 1,482,535 observations in to the cleaning stage in R and output a .csv file with item name and description combined, and minority level in brand name and category name removed. A W2V model is created from the description as training corpus with minimum word count 60 and saved. Then the data is load to Python and one hot encoding is performed on the brand name and category name. After the encoding, those encoded categorical predictor except description are combined as the second level input of the neural network model for prediction. On the other hand, a word embedding matrix for the embedding layer is created from the word token and the pre-trained W2V model and sentence sequences are padded into maximum length 60 as the first level input for the LSTM part as illustrated in the structure diagram above.

There is 10% of the data are separated before the training as the test set and in the Keras implementation provides 30% of the remaining data is used for validation.

I used mean absolute error as a metric after each epochs and from the Kaggle competition, the Root Mean Squared Logarithmic Error is employed as the loss function of the gradient descent.

$$\epsilon = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(log(p_i + 1) - log(a_i + 1))^2}$$

p_i for predicted price and a_i for actual price.

The data in this application is quite huge initially and even bigger after the encoding and W2V. A GPU version of Tensorflow and a API on top of Tensorflow, Keras with a GTX1070 GPU are employed to boosting the calculation. Because the training size is huge and the limitation of my GPU memory, it's not possible to train over lots of epochs, and for all the model I used 5 epochs and 5000 for batch size.

My main interest of the tuning are the type of optimizer, the neurons in each fully connected layer and the dimension of the LSTM layer output. Several models are built and the hyper-parameter search is perform based on one factor at a time principle. The structure of the network is fix as shown in the diagram.

6.1 Adjusting the optimizer

In this section, I fixed the other parameter except for optimizers, and their information is given as following:

Number of Neurons	Batch Size	Dimension of LSTM output
256	5000	200

There optimizer are used to build the model: Stochastic gradient descent(SGD), Adagrad and Adam. Their performance are given below:

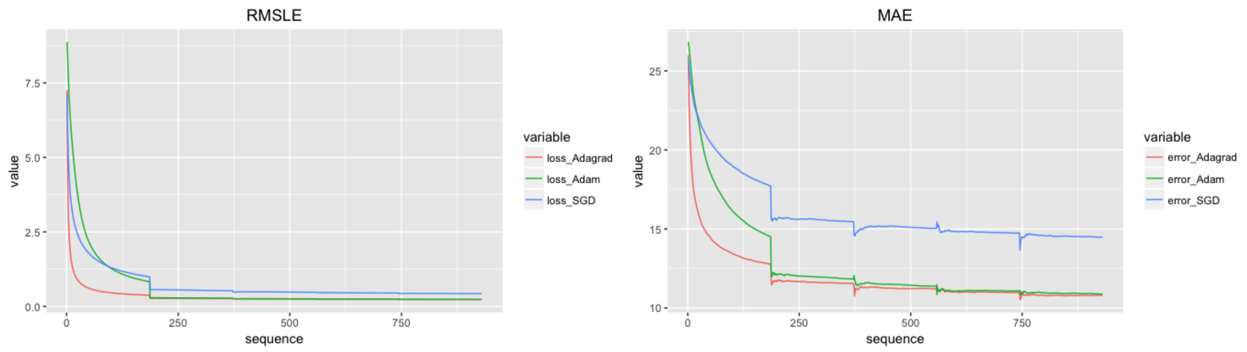


Figure 16: The training performance of 3 models over all batches over 5 epoches

As can be observed in the graph, the training loss almost keeps decreasing for 3 optimizers, however the SGD always gives the biggest loss. It's not surprising that the other two are the optimized version of SGD. Due to the lack of epoch and the validation metrics are only calculated at the end of epochs, I will just test the performance of three models on the test set, which is given by the table below.

Metrics	SGD	Adam	Adagrad
RMSLE	0.64646	0.48055	0.47761
MAE	14.33908	10.71064	10.77767

It's reasonable to choose the Adagrad as the optimizer because it has the minimum target loss RMSLE and the MAE is similar to Adam.

6.2 Adjust the number of neurons

By fixing the optimizer to Adagrad, now some adjustment is applied to the number of neurons. I am not sure if 256 neurons is enough or there is improvement possible by adding more neurons. On the other hand, more neurons may cause over fitting. I trained 3 models with 64, 128 and 256 neurons in each fully connected layer. And the training data for each batch over 5 epochs is given by:

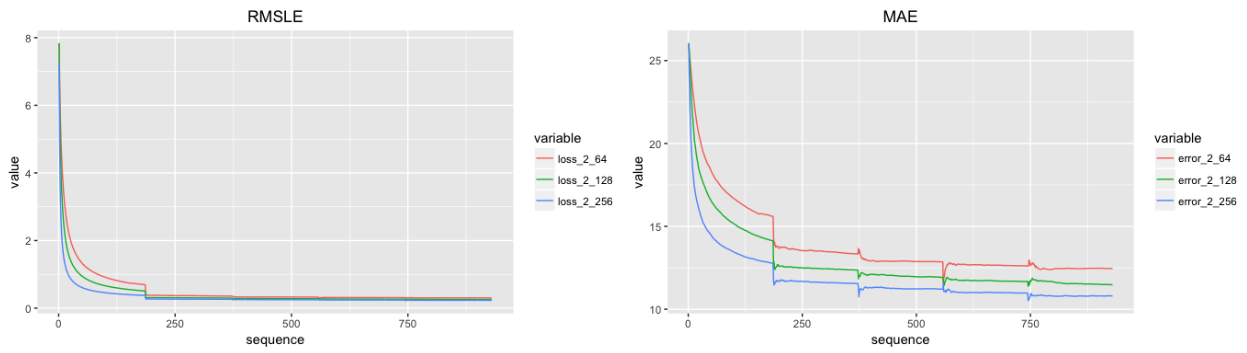


Figure 17: The training performance of 3 models over all batches over 5 epoches

From the both metrics, it appears that there is no doubt to increase the number of neurons. Even more layers and go deeper might gain the lower the training loss. To check the over fitting case, the metrics for the test data is given by:

Metrics	64 Neurons	128 Neurons	256 Neurons
RMSLE	0.52652	0.49284	0.47761
MAE	12.03442	11.17601	10.77767

Even on the test set, the increasing of neurons still improves the performance, which shows that the model can be further developed by adding more neurons and even more layers. However, model with more neurons and layers was crashed on my PC due to the limitation of the GPU memory.

7 Conclusion

The implementation of this model provide a practical implementation of machine learning in real world problem. A solution is conducted to predict the price as long as the seller input the information about the product. This improves the trading efficiency significantly as in the past people have to search over the database to find someone selling the similar item while this is always hard and time consuming.

Currently people achieved about 0.4 RSMLE for the test set on Kaggle and it looks like there are still some improvement can be achieved.

For the part of word to vector, some suggesting that using a big corpus may improve the value of word vectors and there is a model from Google with a size about 1.4GB. On one hand the size of this model prevent me from using it as my embedding matrix. On the other hand, I think the training corpus used is appropriate as they are all about item description.

For the neural network, as shown in adjusting the neurons in the fully connected layer, I am expecting more neurons to improve the result with a better machine or multiple GPU. Also, there can be improvement by tuning of the hyper-parameter, including the batch size, more epochs, the output shape of LSTM and max length of sentence padding.

A R code

A.1 Data Cleaning

```r
# Data preparation and cleaning
# Load packages
setwd("C:/Users/Lala No.5/Desktop/Final_Thesis")
library(readr)
library(ggplot2)
library(tm)

# Read data
df_train <- read_tsv("train.tsv")
df_test <- read_tsv("test.tsv")

# Check NA in data and report
NA_checker <- function(df) {
  col <- colnames(df)
  n <- length(col)
  print(paste("The data has " , dim(df)[1] , " observations."))
  for(i in 1 : n) {
    missed <- length(which(is.na(df[col[i]])))
    print(paste(col[i], " has", missed, " missing values."))
  }
}

# Fill na for description
description_name_merge <- function(df){
  ind <- which(grepl("No description",df$item_description))
  ind <- c(ind,which(is.na(df$item_description)))
  print(paste(length(ind), "missing values found in description,",
              length(ind)/length(df$train_id), "percent."))
  print("Filling those description by item name......")
  df$item_description[ind] <- df$name[ind]

  print("Merging the item name and description for the remaining")
  df$item_description[-ind] <- paste(df$name[-ind],
                                     df$item_description[-ind])
  return(df)
}


#####
# For both brand name and category name, fill na, mark as missing
# take top 300 frequency names and keep as factor column
# mark the remaining as not important brand name or category name
# This is the first general approach, category can be further
# implemented as sentence and pass in to the lstm,
# but the dimension reduction is necessary
#####

# Deal with brand name
brand_cleaning <- function(df){
  brand <- df$brand_name
  brand <- tolower(brand)
  n <- length(brand)
  ind <- which(is.na(brand))



  print(paste("There are", n, "observations,", length(ind) ,
              "of them are missing, which is",
              length(ind)/n, "percent."))
  print("Replacing missing values......")
  print(paste("There are", length(unique(brand)),
              "unique brands."))
  brand[ind] <- "missing brand name"

  brand_tb <- as.data.frame(sort(table(brand), decreasing = T))
  print("The top 10 frequency brands are:")
  print(brand_tb[2:11,])
  print(paste("Except for the missing brand name,
              the top 300 frequency brand name takes",
              sum(brand_tb$Freq[2:301]), "observations"))
  print(paste("Top 300 brand names consist of",
              sum(brand_tb$Freq[2:301])/n,
              "percent of the brand names."))
  print("Transforming the low frequency brand name as small brand .....")
  top300b <- as.character(brand_tb$brand[1:301])
  ind_small <- which(!brand %in% top300b)
  brand[ind_small] <- "small brand"
  print(paste("Now there are", length(unique(brand)),
              "unique brand names ready to be factorized."))
  df$brand_name <- brand
  return(df)
}


# Deal with category name
category_cleaning <- function(df){
  category <- df$category_name
  category <- tolower(category)
  n <- length(category)
  ind <- which(is.na(category))


  print(paste("There are", n, "observations,", length(ind) ,
              "of them are missing, which is",
              length(ind)/n, "percent."))
  print("Replacing missing values......")
  print(paste("There are", length(unique(category)), "unique category."))
  category[ind] <- "missing category name"

  category_tb <- as.data.frame(sort(table(category), decreasing = T))
  print("The top 10 frequency categories are:")
  print(category_tb[2:11,])
  print(paste("the top 300 frequency brand name takes",
              sum(category_tb$Freq[1:150]), "observations"))
  print(paste("Top 300 category names consist of",
              sum(category_tb$Freq[1:150])/n,
              "percent of the brand names."))
  print("Transforming the low frequency category
        name as small category .....")
  top300b <- as.character(category_tb$category[1:150])
  ind_small <- which(!category %in% top300b)
  category[ind_small] <- "small category"
  print(paste("Now there are", length(unique(category)),
              "unique category names ready to be factorized."))
  df$category_name <- category
  return(df)
}

df <- df_train
df <- description_name_merge(df)
df <- brand_cleaning(df)
df <- category_cleaning(df)
df$name <- NULL
df$train_id <- NULL

write_csv(df,"train_clean.csv")
```

A.2 Word cloud

```
setwd("C:/Users/Lala No.5/Desktop/Final_Thesis")
library(readr)
library(tm)
library(SnowballC)
library(wordcloud)
library(RWeka)
library(quanteda)

df <- read_csv("train_clean.csv")
ind <- sample(1:1482535, 100000, replace = FALSE)
description <- df$item_description
corpus <- Corpus(VectorSource(description))
corpus <- tm_map(corpus, PlainTextDocument)
corpus <- tm_map(corpus, function(x) iconv(enc2utf8(x),
                                             sub = "byte"))
corpus <- tm_map(corpus, stripWhitespace)
corpus <- tm_map(corpus, removePunctuation)
corpus <- tm_map(corpus, tolower)
corpus <- tm_map(corpus, removeWords,
                 stopwords('english'))
corpus <- tm_map(corpus, removeWords,
                 c("this", "the", "just", "one", "get", "can"))
corpus <- tm_map(corpus, stemDocument)

# Bigrams
minfreq_bigram<-2

token_delim <- " \\t\\r\\n.!?,;\"()"
bitoken <- NGramTokenizer(corpus,
                          Weka_control(min=2,max=2,
                                       delimiters = token_delim))
```

```
two_word <- data.frame(table(bitoken))
sort_two <- two_word[order(two_word$Freq,decreasing=TRUE),]
wordcloud(sort_two$bitoken,sort_two$Freq,random.order=FALSE,
          scale = c(2,0.35),min.freq = minfreq_bigram,
          colors = brewer.pal(8,"Dark2"),max.words=150)



wordcloud(corpus, max.words = 100, random.order = FALSE,
          rot.per = 0.35, use.r.layout = FALSE,
          colors = (brewer.pal(9,"Pastel1")))









# create the corpus object from the item_description column
description <- description[ind]
dcorpus <- corpus(df$item_description)

dcorpus <- tm_map(dcorpus, removeWords, stopwords('english'))
dcorpus <- tm_map(dcorpus, removeWords, c("this", "the", "just", "one", "get",
    "can"))
dfm2 <- dcorpus %>%
  corpus_sample(size = floor(ndoc(dcorpus) * 0.15)) %>%
  dfm(
    ngrams = 2,
    ignoredFeatures = c("rm", stopwords("english")),
    remove_punct = TRUE,
    remove_numbers = TRUE,
    concatenator = " "
  )
```

```
set.seed(456)
textplot_wordcloud(dfm2, min.freq = 2000, random.order = FALSE,
                   rot.per = .25,
                   colors = RColorBrewer::brewer.pal(8,"Dark2"))
```

B Python code

B.1 Word2vec

```
# load packages
import os
import pandas as pd
import nltk
import gensim
import pickle
import re

# read data
os.chdir('C:\\Users\Lala No.5\Desktop\Final_Thesis')
df = pd.read_csv("train_clean.csv")

# transform dataframe to list
description = df.item_description.values.tolist()
# read stopwords
stop_words = set(nltk.corpus.stopwords.words('english'))


# Function to filter stop words from tokens for each sentence
def sentence_filter(sentence_token, stop_words):
    filtered = []
    for token in sentence_token:
        if not token in stop_words:
            filtered.append(token)
    return filtered

# Clean the sentence
def sentence_cleaner(text):
    text = text.lower()
    text = re.sub(r"[^A-Za-z0-9^,!.\/'+-=]", " ", text)
    text = re.sub(r"what's", "what is ", text)
    text = re.sub(r"\'s", " ", text)
    text = re.sub(r"\'ve", " have ", text)
    text = re.sub(r"can't", "cannot ", text)
    text = re.sub(r"n't", " not ", text)
    text = re.sub(r"i'm", "i am ", text)
    text = re.sub(r"\'re", " are ", text)
    text = re.sub(r"\'d", " would ", text)
    text = re.sub(r"\'ll", " will ", text)
    text = re.sub(r",", " ", text)
    text = re.sub(r"\.", " ", text)
    text = re.sub(r"!", " ! ", text)
    text = re.sub(r"\/", " ", text)
    text = re.sub(r"\^", " ^ ", text)
    text = re.sub(r"\+", " + ", text)
    text = re.sub(r"\-", " - ", text)
    text = re.sub(r"\=", " = ", text)
    text = re.sub(r"'", " ", text)
    text = re.sub(r"(\d+)(k)", r"\g<1>000", text)
    text = re.sub(r":", " : ", text)
    text = re.sub(r" e g ", " eg ", text)
    text = re.sub(r" b g ", " bg ", text)
    text = re.sub(r" u s ", " american ", text)
    text = re.sub(r"\0s", "0", text)
    text = re.sub(r" 9 11 ", "911", text)
    text = re.sub(r"e - mail", "email", text)
    text = re.sub(r"j k", "jk", text)
    text = re.sub(r"\s{2,}", " ", text)
    return text

# Input description, tokenize each sentence and
# return token for each sentence without stopping words
def sentence_tokenizer(description, stop_words):
    value = []
    for sentence in description:
        sentence = sentence.lower()
        sentence = sentence_cleaner(sentence)
        sentence_token = nltk.word_tokenize(sentence)
        filtered_token = sentence_filter(sentence_token,
                                          stop_words)
        value.append(filtered_token)
    return value



# Tokenize the sentence
description_token = sentence_tokenizer(description, stop_words)

# Skip gram model with minimum word count = 10
# and output vector of length 300
model_sg_1 = gensim.models.Word2Vec(description_token,
                                    min_count=10, size=300, sg=1)

# Skip gram model with minimum word count = 50
# and output vector of length 300
model_sg_2 = gensim.models.Word2Vec(description_token,
                                    min_count=50, size=300, sg=1)
```

```
# CBOW(continuous bag of words) model
model_cbow = gensim.models.Word2Vec(description_token,
                                    min_count=10, size=300)

# save and reload
# For tokens of description
with open("decription_token.txt", "wb") as fp:  # Pickling
    pickle.dump(description_token, fp)

with open("decription_token.txt", "rb") as fp:  # Unpickling
    description_token = pickle.load(fp)

# For both models
model_sg_1.save('model_sg_1')
model_sg_2.save('model_sg_2')
```

B.2 LSTM and Deep Learning

```
# Load packages
import gensim
import pickle
import keras
import os
import math
import pydot
import graphviz
import random
import pandas as pd
import tensorflow as tf
import numpy as np
from numpy import array
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing.text import Tokenizer
from keras.backend.tensorflow_backend import set_session
from keras.layers import Input, Embedding, LSTM, Dense,\
    BatchNormalization, Dropout
from keras.models import Model, load_model
from keras.utils import plot_model
from keras import losses
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from matplotlib.pyplot import plot

# GPU configuration, allow GPU memory
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.9
set_session(tf.Session(config=config))

# Set working directory
os.chdir("C:\\Users\Lala No.5\Desktop\Final_Thesis")
# Add path to Graphviz package
os.environ["PATH"] += os.pathsep + \
                      'C:/Program Files (x86)/Graphviz2.38/bin/'
# Read data
print("Loading data...")
df = pd.read_csv("train_clean.csv")
# Number of observations
n = df.shape[0]
print("Number of total observations:", n)

# Numeric column and reshape
condition = np.array(df.item_condition_id).reshape(n, 1)
shipping = np.array(df.shipping).reshape(n, 1)
price = np.array(df.price).reshape(n, 1)
```

```
## One-Hot encoding for categorical column
category_name = np.array(df.category_name)
brand_name = np.array(df.brand_name)

# One-hot brand name
print("One hot encoding for brand and category name...")
values_brand = array(brand_name)
label_encoder_brand = LabelEncoder()
integer_encoded_brand = label_encoder_brand.\
    fit_transform(values_brand)
onehot_encoder_brand = OneHotEncoder(sparse=False)
integer_encoded_brand = integer_encoded_brand.\
    reshape(len(integer_encoded_brand),1)
onehot_encoded_brand = onehot_encoder_brand.\
    fit_transform(integer_encoded_brand)

# One-hot category name
values_cate = array(category_name)
label_encoder_cate = LabelEncoder()
integer_encoded_cate = label_encoder_cate.\
    fit_transform(values_cate)
onehot_encoder_cate = OneHotEncoder(sparse=False)
integer_encoded_cate = integer_encoded_cate.\
    reshape(len(integer_encoded_cate),1)
onehot_encoded_cate = onehot_encoder_cate.\
    fit_transform(integer_encoded_cate)

# Conmbine all those column except for description as the second input
print("Combining the second level inputs...")
train_x_2 = np.hstack([condition, shipping, onehot_encoded_brand,
                       onehot_encoded_cate])

# Release some memory
```

```
print("Releasing memory...")
del df, condition, brand_name, category_name, values_brand, \
    label_encoder_brand, integer_encoded_brand\
    , onehot_encoder_brand, values_cate, label_encoder_cate,\
    integer_encoded_cate, onehot_encoder_cate

## Deal with description:
# Read the sentence token, this is a list of list of token
print("Loading description tokens...")
with open("decription_token.txt", "rb") as fp:  # Unpickling
    description_token = pickle.load(fp)

# Load the pretrained w2v model with skip gram
model_sg = gensim.models.Word2Vec.load('model_sg_2')
print("Loading W2V model(Skip gram)...")

# Restore tokens of description to sentence
description_sentence = [' '.join(sentence) for
                        sentence in description_token]
del description_token
# Fit keras tokenizer and obtain sequence for token of sentence
word_num = len(model_sg.wv.vocab)
tokenizer = Tokenizer(word_num)
tokenizer.fit_on_texts(description_sentence)
description_token_sequence = tokenizer.\
    texts_to_sequences(description_sentence)
# The tokenizer.word_index is a dictionary
# which maps word into index in sequence above

# Create a embedding matrix to map the index in
# description_token_sequence to the word vector

print("Creating embedding matrix...")
```

```
vector_size = 300
embedding_matrix = np.zeros((word_num, vector_size))
for word, i in tokenizer.word_index.items():
    if word in list(model_sg.wv.vocab):
        vector = model_sg.wv[word]
        embedding_matrix[i] = vector
    if i > word_num:
        break

# Pad the sentences to a fixed length
print("Padding sentence to same length")
max_length = 60
description_token_sequence = pad_sequences(
    description_token_sequence, maxlen=max_length)
del model_sg
del tokenizer



# Model building part
print("Start building model...")
# First level nput layer for sentence sequence
main_input = Input(shape=(60,), dtype='int32', name='First_Input')

# This embedding layer will encode the input sequence
# into a sequence of dense 300-dimensional vectors.
# Using pretrained word2vec skip gram model
x = Embedding(output_dim=vector_size, input_dim=word_num,
              input_length=60, weights=[embedding_matrix],
              trainable=False, name="Embedding")(main_input)

# A LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(200, recurrent_dropout=0.1,
                    dropout=0.1, name="LSTM")(x)

# Second level input layer for other predictors
auxiliary_input = Input(shape=(455,), name="Second_Input")

# Merge layer to concatenate LSTM output,
# ie, sentence vector with second layer input
x = keras.layers.concatenate([lstm_out, auxiliary_input])

# Fully-connected layer and batch normalization layers
x = Dense(256, activation='relu', name="FC_1")(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(256, activation='relu', name="FC_2")(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(256, activation='relu', name="FC_3")(x)
x = BatchNormalization()(x)
x = Dense(256, activation='relu', name="FC_4")(x)
x = Dropout(0.5)(x)
# Output layer for price
main_output = Dense(1, activation='relu', name='Output')(x)
# Define model and compile
model = Model(inputs=[main_input, auxiliary_input],
                  outputs=[main_output])
model.compile(loss="mean_squared_logarithmic_error",
                  optimizer='Adagrad', metrics=["mae"])

# Prepare input data
train_x_1 = description_token_sequence
train_x_2 = train_x_2
X = np.hstack([train_x_1,train_x_2])
y = price
```

```
# Cut the data into train, validation and test set
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.1, random_state=42)
del train_x_1, train_x_2, X
train_x_1 = X_train[:,0:60]
train_x_2 = X_train[:,60:515]
test_x_1 = X_test[:,0:60]
test_x_2 = X_test[:,60:515]



# Collect the training result for tensorboard visualization
tensorboard = keras.callbacks.TensorBoard(
    log_dir='C:\\Users\Lala No.5\Desktop\Final_Thesis\graph',
    histogram_freq=0,
    write_graph=True, write_images=True)
csv_logger = keras.callbacks.CSVLogger('log.csv',
                                       append=True, separator=',')

# Train the model
model.fit([train_x_1, train_x_2], y_train,
          batch_size=5000, epochs=5, validation_split=0.3,
          callbacks=[tensorboard, csv_logger])
# Save the model
model.save('model_deep.h5')
# Plot and save the model structure
plot_model(model, to_file='model1.png')

# model_adam = load_model('model_1_Adam.h5')
# model_adagrad = load_model("model_Adagrad.h5")
# model_sgd = load_model("model_SGD.h5")
# pred_adam = model_adam.predict([test_x_1,test_x_2])
# pred_sgd = model_sgd.predict([test_x_1,test_x_2])
# pred_adagrad = model_adagrad.predict([test_x_1,test_x_2])
```

```
#
#
# def RMSLE(y, y_pred):
#   terms_to_sum = [(math.log(y_pred[i] + 1) - math.log(y[i] + 1))
#  ** 2.0 for i,pred in enumerate(y_pred)]
#   return (sum(terms_to_sum) * (1.0/len(y))) ** 0.5
#
# def MAE(y,y_pred):
#     return np.mean(np.abs(y-y_pred))
#
#
# model_2_64 = load_model("model_2_64.h5")
# model_2_128 = load_model("model_2_128.h5")
# pred_2_64 = model_2_64.predict([test_x_1,test_x_2])
# pred_2_128 = model_2_128.predict([test_x_1,test_x_2])
#
# RMSLE(y_test,pred_2_64)
# RMSLE(y_test,pred_2_128)
# MAE(y_test,pred_2_64)
# MAE(y_test,pred_2_128)
```

References

[1] Michael McTear, Zoraida Callejas, and David Griol. The conversational interface. *Springer*, 6(94):102, 2016.

[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[3] Thomas S Ferguson. An inconsistent maximum likelihood estimate. *Journal of the American Statistical Association*, 77(380):831–834, 1982.

[4] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[6] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[11] François Chollet et al. Keras. https://github.com/keras-team/keras, 2015.

[12] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.