Representational Learning With Deep Neural Networks
(using Yelp Dataset)

Prateek Sharma
Master of Computer Science
Northeastern University

Introduction

Machine learning algorithms, rely on inputs being features and learn classification, regression, etc. on top of that. Most of these features are hand crafted, meaning, they are designed by humans, however for real world problems like where the data is a video or an image or words they do not work as well and its better to design features, for example, instead of hand crafting the image representations, we can learn them. This is known as representational learning. We can have a neural network which takes the image as an input and outputs a vector, which is the feature representation of the image. This is the representation learner. This can be followed by another neural network that acts as say a classifier.

There are various representational learning tools, one of the popular one for NLP application's is Google's word2vec [1], below is a brief description:

If one wants to feed words into machine learning models, one has to convert the words into some set of numeric vectors, a straight-forward way of doing this is to use a "one-hot method"[2] of converting the word into a sparse vector representation with only one element of the vector set to 1 for each of the word and the rest being zero. The result is a very sparse vector representation of words with a lot of zeros meaning for any two vectors the inner product(cosine similarity)[3] is zero and so it loses all information between similar words ,for instance, one might expect to see words "United" and "States" to appear close together, or "Soviet" and "Union", Or "food" and "eat", and so on. This method loses all such information and so there is no systematic way of measuring the similarity between words. One more problem with this method is that for very large corpus (i.e. for a lot of words) these vectors become very large and impractical to use.

Now word2vec developed by google takes the words and transform them from these high dimensional one hot vectors into lower dimensional vectors, (this process is called embedding), it also maintains the word context by taking the input word and then attempting to estimate the probability of other words appearing close to that word. This is called the skip-gram approach[4]. The alternative method, called Continuous Bag Of Words (CBOW)[4], does the opposite – it takes some context words as input and tries to find the single word that has the highest probability of fitting that context. A brief explanation of skip-gram approach is as below:

A gram is a group of n words, where n is the gram window size. So for the sentence "The cat sat on the mat", a 3-gram representation of this sentence would be "The cat sat", "cat sat on", "sat on the", "on the mat". The "skip" part refers to the number of times an input word is repeated in the data-set with different context words. These grams are fed into the word2vec context prediction system (a neural network). For instance, assume the input word is "cat", the word2vec tries to predict the context ("the", "sat") from this supplied input word. The word2vec system will move through all the supplied grams and input words and attempt to learn appropriate mapping vectors (embedding) which produce high probabilities for the right context given the input words.

The motivation behind this project is an intuition that “can a user be represented by his words”, like for example when we make a new friend on social media say Facebook, the first impression of the person comes through his posts, like what kind of posts they have written/shared, these helps us in a way to understand the person, like for example if two persons have expressed some racial comments about a religion it is highly likely they have more or less same views about that particular religion, i.e. in some way these two persons are similar in their religious tolerance.

Motivated by these ideas, and the concept of dense vector representation of words in word2vec, in our project we have tried to find similarity between users by the words that they have used to give reviews (on Yelp), using Yelp dataset[5], we have tried to predict similar users based on the similarity of words they have used in their reviews, for example if a person has used words such as “garlic, noisy, fantastic, crowded” and other person has used words as “mushroom, garlic, nice, many people” then for our model these are similar users, we also build a recommendation system which finds a “similar user” to the “user” whose review is used as an input to the model, and recommends this “user” a business(for example a restaurant).

For our Neural network the inputs are the reviews written by the user on Yelp, and the output layer has multiple nodes, one for each user (multi-class classification) ,the yelp dataset has different json files which contains different information, for us the most important ones were the review.json which had the detailed user reviews for the business given by the user, the user_id which is also present in user.json, the business_id also present in business.json , these id’s helps to determine which user has given review and for which business, user.json contains the metadata about the user like their name, year since they have been yelping, number of friends etc., business. json contains the metadata for the particular business like business_id, the name of the business, address of business etc., the review. json file is used to build the model, the business.json file is used to get the information about the business that is recommended by the model, and user.json file is used to show the details of the similar users predicted by the model.

To build our Neural Network we used Keras API in Python, we were able to successfully represent each user by the corpus of their words, the model was able to identify similar users (people who have used similar words for review) and was also able to recommend business to the user.

Methods

For this project as we wanted to learn the similarity of the user’s based on the review(words) they have written, we needed to understand the context between the words i.e. how likely a word is to appear after a particular word, and not only that but given a couple of previous words what is the probability of the next word and so on, for these types of interpretation where we have a sequence(or frame) of input, simple feed-forward neural networks does not perform very well.

One of the more complicated architectures, which is known to perform very well on text data, is the Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM)[6].

RNNs are designed to learn from sequences of data, where there is some kind of time dependency. For example, they are used for time-series analysis, where each data point has some relation to those immediately before and after. By extension, they work very well for language data, where each word is related to those before and after it in a sentence. One of the problems of traditional RNN is that they can only learn short range context (normally 2-3 words) but as in a sentence its more useful to find longer ranges context a newer modified architecture known as RNN-LSTM was used which performs better by using purpose-built memory cells to store information. These cells have different gates like forget gate, input gate, output gate, these gates help the network to determine what and how much of previous information has to be used for present tasks and what previous information should be forgotten i.e. how much of previous information will help in deciding what the present interpretation should be. These techniques help RNN-LSTM to exploit longer range context, for example what the next word in a sentence should be given a number of previous words.

A brief explanation of the basics of RNN and LSTM network is as follows:

Recurrent Neural Network (RNN)[7] is a class of artificial neural networks where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior, unlike feedforward neural network RNN's can use their internal memory to process arbitrary sequence of input,

Given an input sequence $x = (x_1,,,,,,x_T)$, a standard recurrent neural network (RNN) computes the hidden vector sequence $h = (h_1,,,,,,,,h_T)$ and output vector sequence $y = (y_1,,,,,,y_T)$ by iterating the following equations from $t = 1$ to T:

$h_t = H(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$ (1)
$y_t = W_{hy}h_t + b_y$ (2)

where the W terms denote weight matrices (e.g. W_{xh} is the input-hidden weight matrix), the b terms denote bias vectors (e.g. b_h is hidden bias vector) and H is the hidden layer function. H is usually an elementwise application of a sigmoid function.

A modified RNN architecture known as LSTM (Long Short-Term Memory) [8] which uses purpose-built memory cells to store information, is better than conventional RNN's at finding and exploiting long range context. It uses the following equations:

$h_t = H(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$ (1)
$y_t = W_{hy}h_t + b_y$ (2)

these two steps are same as above but here H is implemented by the following composite function:
$i_t = \sigma (W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i)$ (3)
$f_t = \sigma (W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f)$ (4)
$c_t = f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$ (5)
$o_t = \sigma (W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o)$ (6)
$h_t = o_t \tanh(c_t)$ (7)

where σ is the logistic sigmoid function, and i, f, o and c are respectively the input gate, forget gate, output gate and cell activation vectors, all of which are the same size as the hidden vector H. The weight matrices from the cell to gate vectors (e.g. W_{si}) are diagonal, so element m in each gate vector only receives input from element m of the cell vector.

Basically for LSTM we have the following sequence of steps which determines the cell state, first "forget gate" (4) determines which information will be thrown away from the cell state, next we have to determine which information has to be stored in the cell state, this has three steps first the “input gate” (3) decides which values to update and then a “tanh layer” (5) creates a vector of new candidate values, then we multiply the old state by “forget gate”(5) forgetting the things we need to forget, we then add this to the new candidate values calculated in the previous step and this gives us the new state of the cell. Finally we have to decide what we are going to output , this is done by “output layer” (6) which runs a sigmoid layer over the new cell state to determine which part of the cell state we are going to output from the cell.

We trained our recurrent model and it worked well enough, but it was slow to train. One way to speed up the training time was to improve our network architecture. To achieve this we added a “Convolutional” layer to our network. Convolutional Neural Networks (CNNs)[10] come from image processing. They pass a “filter” over the data, and calculate a higher-level representation. They have been shown to work surprisingly well for text [9], even though they have none of the sequence processing ability of LSTMs. They are also faster, as the different filters can be calculated independently of each other. LSTMs by contrast are hard to parallelize, as each calculation depends on many previous ones.

Here is a brief explanation of Convolutional Neural Network(CNN) [10], Convolutional Neural Network is a variant of standard neural network, where instead of fully connected hidden layers CNN introduces a special network structure which consists of alternating so-called convolution and pooling layers.

Convolution Layer: Every input feature map (assume i is the total number), O_i (i=1,.....,i) is connected to many feature maps (assume j in the total number), Q_j (j=1,......,j) in the convolution ply based on a number of local weight matrices (I * J in total), $w_{i,j}$(i=1,....,I;j=1,.....,J). The mapping can be represented as the well-known convolution operation in signal processing. Assuming input feature maps are all one dimensional, each unit of one feature map in the convolution ply can be computed as:

$q_{j,m} = \sigma\ (\Sigma(i=1...I)\ \Sigma(\ n=1...F)\ o_{i,n+m-1} w_{i,j,n} + w_{0,j}),\ (j=1....j)$

where $o_{i,m}$ is the m-th unit of the i-th input feature map O_i, $q_{j,m}$ is the m-th unit of the j-th feature map $q_{j,m}$ in the convolution ply, $w_{i,j,n}$ is the n^{th} element of the weight vector, $w_{i,j}$ which connects the i^{th} input feature map to the j^{th} feature map of the convolution ply. F is called the filter size, which determines the number of frequency bands in each input feature map that each unit in the convolution ply receives as input.

Pooling Layer: A pooling operation is applied to the convolution ply to generate its corresponding pooling ply. The pooling ply is also organized into feature maps, and it has the same number of feature maps as the number of feature maps in its convolution ply, but each map is smaller. The purpose of the pooling ply is to reduce the resolution of feature maps. This means that the units of this ply will serve as generalizations over the features of the lower convolution ply, and, because these generalizations will again be spatially localized in frequency, they will also be invariant to small variations in location. This reduction is achieved by applying a pooling function to several units in a local region of a size determined by a parameter called pooling size. It is usually a simple function such as maximization or averaging. The pooling function is applied to each convolution feature map independently.

Now back to our model, by the addition of a CNN layer before the LSTM, we allow the LSTM to see sequences of chunks instead of sequences of words. For example, the CNN might learn the chunk “I loved this” as a single concept and “friendly guest house” as another concept. The LSTM stacked on top of the CNN could then see the sequence [“I loved this”, “friendly guest house”] (a “sequence” of two items) and learn which users have written them, instead of having to learn the longer and more difficult sequence of the six independent items [“I”, “loved”, “this”, “friendly”, “guest”, “house”].

Now a brief explanation of how we build our model.

To build our model we first had to prepare our data with the review.json file. From this file we extracted the users (identified by user_id) and then for each of the user we append the reviews given by him in a dictionary i.e. the keys of the dictionary were the users and the values were corresponding reviews given by the user (also before appending the reviews we clean it by removing the special characters from it such as the commas, punctuations and so on as these does not have any affect in the learning model). Now we separate the reviews and the corresponding user's into different lists, where the review list is used as input to the neural network while the corresponding user's list is used as the class labels for the output layer of our neural network. These input data (reviews) and the corresponding class labels(the user's) are saved in pickle file(pickle helps to dump the data into a file, and we can later load this dumped data), this file is later loaded by the program that builds up our model so that we will not have to prepare data over and over again.

Now before feeding our input layer with the reviews and label our output layer with multiple classes (the users) we first had to convert the reviews and user's to integer values. Also from the huge corpus of words in the reviews we take only words which occur some minimum number of times in the corpus as the neural network cannot learn from rare words and it also increases the processing time. All this was done with the help of Tokenizer module of Keras.

Next we build our model, we first added an empty sequential model which allows to add several layers to the model, we then added an embedding layer which transformed the input words from high dimensional sparse vectors to dense vector representation, we then added an LSTM layer which helps in understanding context/relation between the input words , the weights connecting these layers are in fact the new learned vector representation of the words, after this we added a fully connected output layer with the number of neurons equal to the number of user's (each neuron representing a particular user). Also a point to note is that the output user label's were converted to one hot vector representation so that each neuron identifies a single user.

Now to avoid overfitting we used "dropout" argument in the LSTM layer which is a regularization method, a brief explanation of this method is as follows:

Dropout is a technique where randomly selected neurons are ignored during training. They are "dropped-out" randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data (overfitting). This reliant on context for a neuron during training is referred to complex co-adaptations.One can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network. The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to over fit the training data.

The activation function used in the output layer was "softmax" which squashes the probabilities of all the output classes so that they sum to 1, and this has proven to be pretty useful for multi-class classification (as was our case).

In the output layer we use "categorical_crossentropy" as a loss function, which measures the distance between what the model believes the output distribution should be and what the actual distribution really is by comparing the one hot encoded vectors of multiple classes(class labels) to the probability vector for these classes calculated by softmax activation function, this method is particularly useful when we have multiple classes.

Now for updating the weights of the model we used the "adam optimizer" which is an extension of the stochastic gradient descent algorithm. Adam optimizer maintains per parameter learning rate instead of a fixed learning rate used by stochastic gradient descent and it improves performance on problems with sparse gradients (like natural language, computer vision)[11].

Results

It turns out that a single user has given multiple reviews, so we had an intuition that to identify a user by his words we can use a subset of reviews given by him/her instead of using all the reviews given by him/her.

So we prepared two sets of data, one of them contained a list of all the reviews given by a user (to prepare this data it took more than 24hours because of the huge size of the dataset) and other contained a subset of reviews given by them. We trained two different models and then we gave these two different models a review written by a "user" and the two models predicted different "similar users" for this "user".

Below is the detail of the user (actual user) whose review was given as an input to the model and for whom "similar users" and a "recommended business" were predicted by the model:

Actual User:

The name of the user is Tiffany , he/she has given 3 reviews.

Review given by this User (This is input review to the model) :

Terrible! This dunkin location is never properly staffed. Waited forever and then finally just ordered some donuts instead of what I actually wanted just to get the hell out of there. The manager/owner needs to hire more people.

Results for when I ran for the model where all the reviews given by the user is taken into account and we see that the reviews are related in some context:

Similar User (predicted by the model):

The name of the user is Katrina , he/she has given 1 review.

Review of Predicted/Similar User:

I had a bite of the kids' pizza and it didn't seem anything exceptional. The dough was a little off. Service was a little slow.

Clearly we see that the words used by the users have a similar tone to it, both the users are not happy with the service, with one of them suggesting to hire more people while the other saying service was slow.

Results for when I ran for the model where a subset of the reviews given by the user is taken into account:

Similar User(predicted by the model):

The name of the user is TJ , he/she has given 261 reviews.

Review of Predicted User:

My bf and I were there on 7/01/11. We each got the Coalition package and had a blast shooting the full-auto SAW. UFC fighter Jake Shields was there but by far our fav person at The Gun Store was our Range Instructor Tom. He was so informative and pleasant. Thanks again, Tom. The Gun Store, you guys rock!

We see from these reviews do not have the same tone, so its better to use more data.

Below is the recommended business, which was recommended to the "Actual User" whose review is input to the neural network.

Business the "Actual User" went:

Dunkin' Donuts , full address is 6295 S Rainbow Blvd , Las Vegas , NV – 89118

Recommended Business for this user:

Pepperoncinis . full address is 105 Seminary Ave , Oakdale , PA - 15071

We see that the business recommended is a restaurant and not other type of business as both the user's (actual user and similar/predicted user) , have reviewed restaurants (in their reviews) so this shows a success of the recommendation system as it recommends similar type of business i.e. restaurants in this case, for example if the input review is about a restaurant then the model will recommend restaurant only and not say a shopping center.

Discussion

With the above results, it is clear that for similarity measure between users based on their words it is important to have more data defining a user (as using all the reviews of users instead of a subset of reviews gives better results), the obvious reason for this is the more better we understand a user by his words the more better we can predict similar user and an easy way for this is to use more data, also if we only use a subset of their reviews we can miss on some important features.

Now we trained our neural network model with all the reviews given by the first 40000 users and it took half an hour to train the model(with CNN architecture), but running with more data was failing because of hardware capability of my machine, so in future to make a model which can train over all the users I can get AWS machines and run the model on them.
We see from the results that the recommendation system, recommends a restaurant which is in a different city, this is because the "actual user" and the "similar user" can be present in different cities, but this is a problem as the recommended restaurant may not be in his/her city especially if the "actual user" belongs to a small city where there are fewer restaurants, so to overcome this we will have to take into account the

geographical distribution of the user’s (this information is also present in the business.json file of yelp dataset) and we will group the user’s according to their geographical location and predict a similar user and the recommended business from within that location only.

Now one of the possible situations where our model can fail is if the “actual user” for whom we want to predict a “similar user”, had used very few words in their reviews so it might become very difficult for the model to find this user a suitable “similar user”, and the predicted “similar” user by our model might have been talking about something completely different so it again comes down to the data, the more the better.

References

[1] https://code.google.com/archive/p/word2vec/

[2] https://en.wikipedia.org/wiki/One-hot
[3] https://en.wikipedia.org/wiki/Cosine_similarity
[4] https://iksinc.wordpress.com/tag/continuous-bag-of-words-cbow/

[5] https://www.yelp.com/dataset/download
[6] Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling, by Hasim Sak, Andrew Senior, Francoise Beaufays (Google USA).
[7] Speaker Independent Speech Recognition with Neural Networks and Speech Knowledge Paper by, Yoshua Bengio (Dept. of Computer Science McGill University, Montreal Canada H3A2A7), Renato De Mori (Dept. Computer Science McGill University), Regis Cardin (Dept. of Computer Science McGill University).
[8] SPEECH RECOGNITION WITH DEEP RECURRENT NEURAL NETWORKS Paper by, Alex Graves, Abdel-rahman Mohamed and Geoffrey Hinton (Department of Computer Science, University of Toronto).
[9] Kim, Y. (2014). Convolutional Neural Networks for Sentence Classification. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)
[10] DEEP CONVOLUTIONAL NEURAL NETWORKS FOR LVCSR Paper by, Tara N. Sainath, Abdel-rahman Mohamed, Brian Kingsbury, Bhuvana Ramabhadran (Department of Computer Science, University of Toronto).
[11] ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION by Diederik P. Kingma (University of Amsterdam, OpenAI) and Jimmy Lei Ba(University of Toronto).

Appendix

The implementation details for our model is as follows:

Below is sample code which translates the words to integer vectors:

```
1. tokenizer = Tokenizer(num_words=20000)
2. tokenizer.fit_on_texts(texts) ## texts are the list of user reviews which is our corpus of words
3. sequences = tokenizer.texts_to_sequences(texts)
4. data = pad_sequences(sequences, maxlen=300)
```

The first line says that from the corpus of words (reviews) we only want the top most 20000 most common words (and words which do not make this cutoff are considered as rare words and are ignored), the second line calculates the frequencies of the words in the corpus (reviews) , the third line converts our list of texts to list of integers (with lower values given to most common words and higher values to less common words), now step 3 will result different length of list of integers for the list of reviews (this in fact is a vector representing the words in the review) but we want them to be of same length so we use step 4 to pad our vectors with 0’s and also limit the size of each vector to be 300 i.e. for the review vectors which have a lot of words we take only top 300 words out of them and review vectors with less words will be padded with zeros),this step in fact makes the inputs (i.e. the training examples) to our neural network to be of same size.

Now for our class labels(the user’s) of the neural networks we represent them as one hot encoded vectors. Below is a sample code.

```
1. encoder = LabelEncoder()
2. encoder.fit(users_str)  ## user_str is the list of users
3. encoded_users = encoder.transform(users_str)
4. users = np_utils.to_categorical(encoded_users)
```

First we encode the strings to integers using the scikit-learn class LabelEncoder. (steps 1-3).Then convert the vector of integers to a one hot encoding using the Keras function to_categorical() (step 4).

After preparing our data we now build our neural network.

```
1. model = Sequential()
2. model.add(Embedding(20000, 128, input_length=300))
3. model.add(LSTM(128, dropout=0.2))
4. model.add(Dense(len(users, activation='softmax'))
5. model.compile(loss='categorical_crossentropy', optimizer='adam',    metrics=['accuracy'])
6. model.fit(data, np.array(users), validation_split=0.5, epochs=3)
```

The first step creates an empty sequential model and is used to add several layers to the model. In step two, we add an Embedding layer. This layer lets the network expand each word to a larger vector, allowing the network to represent words in a meaningful way. We pass 20000 as the first argument, which is the size of our vocabulary (remember, we told the tokenizer to only use the 20 000 most common words earlier), and 128 as the second, which means that each word can be expanded to a vector of size 128. We give it an input_length of 300, which is the length of each of our sequences, this layer in fact converts each word to a dense vector representation (as it lowers the dimension of vector from length 300 to a dense vector of length 128) that we need to train our neural network with. In the third step we add the LSTM layer, the first argument is 128, which is the size of our word embedding’s (the second argument from the Embedding layer). We add a 20% chance of dropout with the next argument. Dropout is a regularization method where input and recurrent connections to LSTM units are probabilistically excluded from activation and weight updates while training a network, this helps in reducing overfitting and improving model performance. These two layers in fact perform the word2Vec transformation where the embedding layer helps to reduce the dimension of the vector and the LSTM layer helps in understanding the context/relation between words, and the weights connecting these layers is in fact the new learned vector representation of the word. In the next step we add a Dense layer which is the most simplest type of neural network layer where all neurons in the layer are connected to each other, this is in fact the output layer , the number of neurons in this layer is equal to the number of users each representing different users (multi-class classification) and we use the softmax function as the activation function for this layer. In line five, we compile the model. This prepares the model to be run on the backend graph library (in our case, TensorFlow). We use loss= “categorical_crossentropy”,. We use the adam optimizer which is an extension of the stochastic gradient descent algorithm, interested in the accuracy metric (how many user predictions our neural network gets correct). In the last step we train or fit our network, we pass the tokenized data(reviews) as the input to the network and the labels (users) for our output layers as the second argument , we use validation_split =0.5 to tell our neural network that it should take half of the data to learn from, and that it should test itself on the other half. The last argument we pass, epochs=3, means that the neural network should run through all of the available training data three times.

Now we explain of how we added Convolutional Neural Network to our Neural Network Architecture.

```
model = Sequential()
model.add(Embedding(20000, 128, input_length=300))
model.add(Dropout(0.2))
model.add(Conv1D(64, 5, activation='relu'))
model.add(Dropout(0.2))
model.add(MaxPooling1D(pool_size=4))
model.add(LSTM(128))
model.add(Dropout(0.2))
model.add(Dense(len(users), activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(data, np.array(users), validation_split=0.5, epochs=3)
```

We add a dropout layer directly after the Embedding layer so as to avoid overfitting, we add this in the beginning so that the all the calculations performed by the network are more robust. Following this, we add a convolutional layer which passes a filter over the text to learn specific chunks or windows, the first parameter is the output channels of the convolutional layer, the second parameter is the kernel_size which is actually the moving window size (in our case it is 5*5), and the activation function used is relu , the activation function of reLu is f(x)=Max(0,x) and we use it here as its easy to calculate the gradient for this function and it is faster than other general activation function like sigmoid without compromising on the accuracy and also it works pretty well with convolutional neural network. After this, we have a MaxPooling layer (which has only one parameter the pooling size), which combines all of the different chunked representations into a single chunk. The rest of the model is the same.

After building the model, we store the model in pickle file and use it in two other files one of them is used to predict a similar user to a given user by feeding the model the review written by them and the other recommends a business, sample code is below:

```
1. with open("keras_text1.pickle", "rb") as f:
   texts_load= pickle.load(f)

2. with open("keras_user1.pickle", "rb") as f:
   users_load = pickle.load(f)

3. texts=texts_load[0]
4. users=users_load[0]
5. with open("keras_tokenizer.pickle", "rb") as f:
   tokenizer = pickle.load(f)

6. with open("keras_encoder.pickle", "rb") as f:
   encoder= pickle.load(f)

7. model = load_model("yelp_recomender_model.hdf5")
8. sequences = tokenizer.texts_to_sequences(texts)
9. data = pad_sequences(sequences, maxlen=300)
10. predictions = model.predict(data)
11. user_integer = np.argmax(predictions, axis=1)
12. similar_user=encoder.inverse_transform(user_integer)
13. print "Actual Users  : \n", users
14. print "Similar Users  : \n", similar_user[0]
```

The first step loads the file which has all the reviews which are stored in a list, line two loads all the corresponding users who are also stored in a list, steps 3 and 4 takes one of the user and the review written by them, step 5 loads the tokenizer which was used to build the model, in step 6 we load the encoder which was used during the training of the model, step 7 loads the neural network model which we have trained, in step 8 and 9 we again convert the reviews into vectors of integers (with the loaded tokenizer) and do the required padding , these steps prepare our test data for the neural network, in step 10 we give our model this prepared data for it to predict a user, in step 11 and 12 we convert back the predicted user to the string (as originally they were as strings only), in step 13 we print the user for whom we are finding the similar user and whose review was given as input to the model, and last step prints the user who is similar to this user.

We further printed the reviews written by the similar user and actual user, the review of actual user is simply texts of step 3, while to get the review of the predicted user we looped through the review.json file of yelp dataset to find the corresponding review, a sample code is below:

```
1. path_data='/Users/prateeksharma/Desktop/Machine Learning Project/dataset/'
2. yelp_files=['business','checkin','photos','review','tip','user']
3. review_file=yelp_files[3]
4. file_test=path_data+review_file+'.json'

5. with open(file_test) as yelp_review:
       f1 = yelp_review.read().strip().split("\n")

6. review_similar_user=''
7. for id, line in enumerate(f1):
   temp=json.loads(line)
   uid=str(temp['user_id'].encode('utf-8'))
   if(similar_user==uid):
       review_ similar_user=temp['text'].encode('utf-8')
```

Steps 1-5 loads the review.json file , and then in step 6 we just define an empty string to which we will append the review, in step 7 we loop through the json file and for the user predicted by our neural network we get their review and then we printed it.

Now for business recommendation we have similar steps as above its just instead of looping through the review.json file we loop through the business.json file to get the business of the similar user and print it, as this business becomes the predicted business of the model, we also print the actual business for which the input user wrote the review.

Also a further step in above code was to loop through the user.json file and get the metadata of the predicted and actual user(the input to neural network) and print them as well.