CS 6375 – Machine Learning

Transfer Learning on Stack Exchange Tags

Contributors:-

|  |  |
| --- | --- |
| Name | Net ID |
| Rumela Saraswati | rxs145730 |
| Nagma Nishat | nxn151930 |
| Prasshant Srinivasan | kxs157030 |
| Sachin Kamasetty Venkatesha | sxk159030 |

**INTRODUCTION**

Stack Exchange is a platform of connected question and answer websites of diverse fields. The network currently has 150+ Q&A communities and serves millions of users per month. This project is based on Transfer Learning applied to predict tags on Stack Exchange. Transfer learning is the ease of learning new concepts depending on past concepts. This relates to the similarity of the tasks under consideration, more similar the tasks higher are the chances of applying past knowledge to master it. We have implemented various machine learning algorithms to learn training categories (e.g. diy, crypto) and then to transfer the knowledge in order to predict tags for the dataset which is based on physics.

**DATASET EXPLORATION**

This project is based on the Transfer Learning on Stack Exchange Tags on Kaggle (https://www.kaggle.com/c/transfer-learning-on-stack-exchange-tag). We obtained the data from the data provided for this competition on Kaggle. The data is categorized into 6 categories for training (e.g. cooking, biology etc.) and 1 dataset for testing i.e. test dataset. The training data contains the headers id, title, content and tags while the test data does not have the header tags. While examining the training data, we found that many posts have unique tags while others have repetitive tags. Also few tags appear more than in one domain. We also analyzed the average number of tags per post.

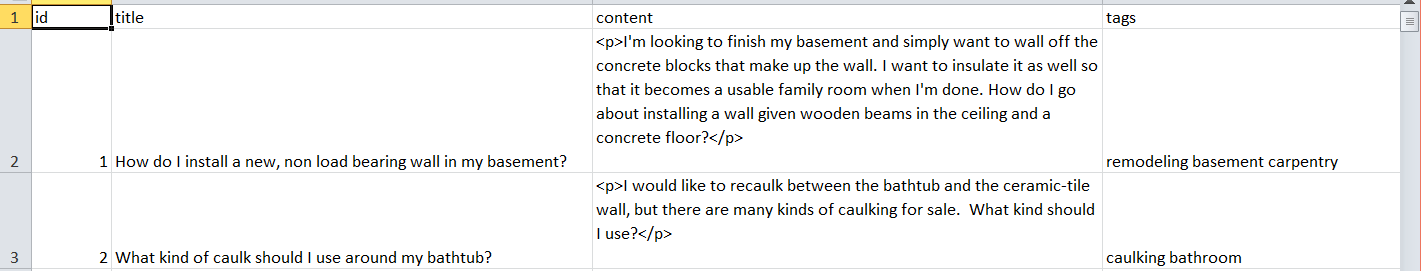
Following are some of the statistics we noted while trawling through the datasets:

Cooking tag count: 736  
Cooking unique tags: 8.  
Biology tag count: 678  
Biology unique tags: 32  
Crypto tag count: 392  
Crypto unique tags: 21  
DIY tag count: 734  
DIY unique tags: 28  
Robotics tag count: 231  
Robotics unique tags: 16  
Travel  tag count: 1645  
Travel unique tags: 212  
Total untagged questions: 56

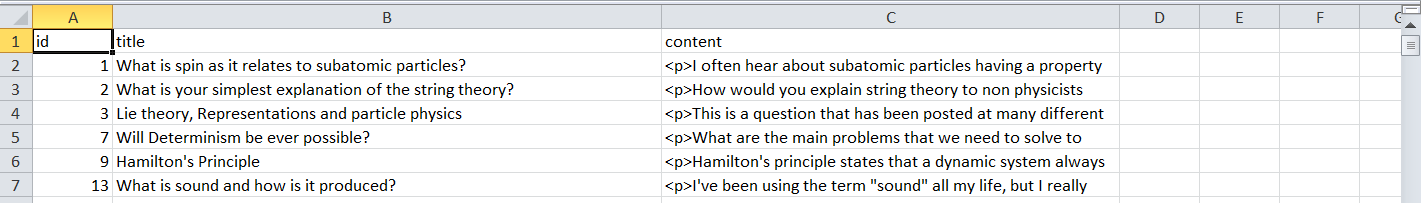
Tag Intersections from Cooking to other datasets:  
Biology - 27  
Crypto - 6  
Diy - 29  
Robotics - 3  
Travel - 15  
  
Total unique tags: 4268  
Tags found in all sets: 1('untagged')

Stack Exchange questions from the following topics are given for training:  
1. Biology - 13196 questions  
2. Cooking - 15404 questions  
3. Crypto - 10432 questions  
4. DIY - 25918 questions  
5. Robotics - 2771 questions  
6. Travel - 19729 questions  
The test data set consists of 81926 questions from Physics which the model will be tested on,  
Each question in the training set consists of the following fields:  
1. ID - The unique numeric identifier for the question2. Title - The question title or heading, which summarizes the question  
3. Content - The detailed explanation of the question  
4. Tags - The tags associated with the question

Below are snapshots of the training and test data:



1. Training Data



1. Test data

**RELATED WORK**

There have been some related work by Colbert and Weston in the area of Natural Language Processing. They have trained various tasks like SRL, POS by modeling language and demonstrated that learning tasks can often generalize the performance. Some research by Liu et. Al by using recurrent neural networks for text classification have proposed models for information sharing in order to model text.

**DATA PREPROCESSING**

The preprocessing of the data is broken into two parts primarily, *formatting* and *embedding*. The formatting part can further be broken into the following parts:

Formatting and Embedding for Word2Vec- Since the content had html tags, used Beautiful Soup to remove them. Converted all the content and title into tokens for input to the word vector model.

For RNN ,the pre-preprocessing for the Recurrent Neural Network is a Numpy Array. Each entry in the array is an array of Word Vector. For each question, we considered the top 200 words (content and title included). For each word, we obtain the word vector from the word vector model trained previously. If 200 words are not available, then 0 array are padded. This 200-word vector is an entry in the main input array. The input array consists of 10000 such 200 word vectors.

**Word Vectors:**

Words can be represented in several ways for processing in a machine learning algorithm. As they are, they are just a string of characters that don’t encode any semantic meaning. Also, if the vocabulary is represented as a matrix with each word being one dimension, then this can result in a vector space with extremely high dimensions, equal to the size of the vocabulary. Thus, it is helpful to convert the vocabulary into a format more accessible for computation.

The approach being followed in this project is to convert each word into a vector of a predetermined length, thus reducing the vector space dimensions, an idea generally known as word embedding. There are several algorithms that can be used for this conversion, such as GloVe and word2vec, we used Gensim’s Python implementation of word2vec.

The model generated by word2vec encodes not only all the words in the vocabulary in an n-dimensional vector space, it also retains the semantic meaning of the words in the vectors. Words that are similar in meaning will be closer in the vector space than words which are dissimilar. This is done by algorithm as it processes a corpus, by noting the context in which the words occur, for example commonly among other words, and using this to calculate the vectors. Word2Vec models are generally shallow neural networks.

The training of the word2vec model used in the project was done as follows:

Since the content had html tags,  Beautiful Soup was used to remove them. The content and title  were converted into tokens for input to the word vector model.

Gensim requires that a list of sentences be input into the model for training the word2vec model. Thus all the question titles and content were broken down into sentences, and the tags assigned were assumed to be sentences and these were fed into the training function.

For the purposes of this project, 100 dimensions were chosen for the vector space.

The resultant word2vec model performed quite well upon observation, for example, following are the top 10 similarity results for the word ‘india’:

[(u'usa', 0.7936956882476807), (u'australia', 0.7905254364013672), (u'pakistan', 0.7841647267341614), (u'canada', 0.7840760350227356), (u'sweden', 0.7829179763793945), (u'singapore', 0.7653782367706299), (u'philippines', 0.7649333477020264), (u'germany', 0.7568991184234619), (u'egypt', 0.7484690546989441), (u'mexico', 0.7454445362091064)]

As can be seen, the model has the learnt the concept of countries, and returned the names of countries as the most similar results, along with their vector cosine similarity scores.

**APPROACH**

**1) MULTINOMIAL NAÏVE BAYE’S FOR DOCUMENT CLASSIFICATION**

**Multinomial Naïve Bayes** is a probabilistic learning method for document classification. Mathematically, it is represented by P(c|d) = P(c) П 1≤k≤nd P(tk|c), where P(tk|c) is the conditional probability of tk occurring in a document that has class c and P(c) is the prior probability of document having class c. Here we used the weighted conditional parameters having weights log(P(tk|c)) and log(P(c)) respectively. This gives us a measure of how likely a document is to be present in a given class by simplifying it to log operations.

We used Multinomial Naïve Bayes as a crude approach to classify the categories of each document based on the title and content. By removing the stop words from the training data, and then using this learning method on the test data we have deduced the class of these documents. We have used accuracy as a measure of how well the documents are classified. This learning method did not give great results on the test data and we moved on to further training methods to get better results.

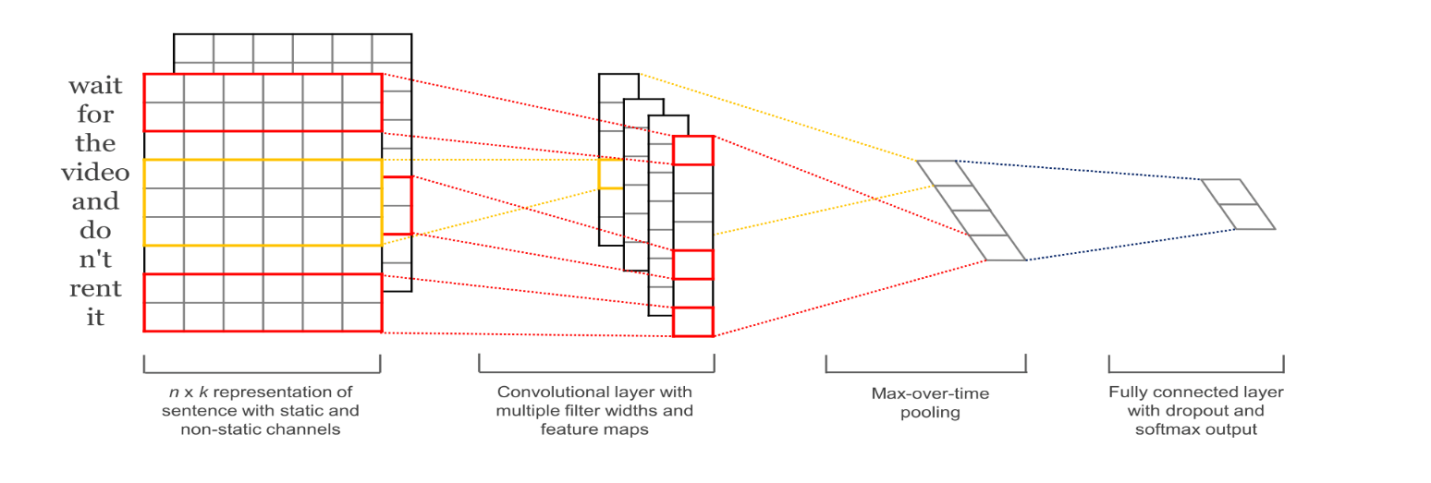
**2) CONVOLUTIONAL NEURAL NETWORKS**

The dataset we’ll use is the Kaggle dataset of Stack Exchange. The data is categorized into 6 categories for training (e.g. cooking, biology etc.) and 1 dataset for testing i.e. test dataset. The training data contains the headers id, title, content and tags while the test data does not have the header tags. While examining the training data, we found that many posts have unique tags while others have repetitive tags. Also, few tags appear more than in one domain. We also analyzed the average number of tags per post. The reported results are for 10-fold cross-validation on the data.

* Load positive and negative sentences from the raw data files.
* Clean the text data for various pre-processing techniques in the data\_helpers.py.
* Pad each sentence to the maximum sentence length, which turns out to be 256 words. We append special <PAD> tokens to all other sentences to make them 256 words. Padding sentences to the same length is useful because it allows us to efficiently batch our data since each example in a batch must be of the same length.
* Build a vocabulary index and map each word to an integer between 0 and 18,765 (the vocabulary size). Each sentence becomes a vector of integers.

THE MODEL

The network we will build looks roughly as follows:



Convolutional Neural Networks for Sentence Classification

The first layers embed words into low-dimensional vectors. The next layer performs convolutions over the embedded word vectors using multiple filter sizes. For example, sliding over 3, 4 or 5 words at a time. Next, we max-pool the result of the convolutional layer into a long feature vector, add dropout regularization, and classify the result using a softmax layer.

IMPLEMENTATION

To allow various hyperparameter configurations we put our code into a TextCNN class, generating the model graph in the init function.

To instantiate the class, we then pass the following arguments:

* sequence\_length – The length of our sentences. Remember that we padded all our sentences to have the same length (256 for our data set).
* num\_classes – Number of classes in the output layer, two in our case (positive implying physics and negative implying not physics).
* vocab\_size – The size of our vocabulary. This is needed to define the size of our embedding layer, which will have shape [vocabulary\_size, embedding\_size].
* embedding\_size – The dimensionality of our embeddings.
* filter\_sizes – The number of words we want our convolutional filters to cover. We will have num\_filters for each size specified here. For example, [3, 4, 5] means that we will have filters that slide over 3, 4 and 5 words respectively, for a total of 3 \* num\_filters filters.
* num\_filters – The number of filters per filter size

INPUT PLACEHOLDERS

We start by defining the input data that we pass to our network:

tf.placeholder creates a placeholder variable that we feed to the network when we execute it at train or test time. The second argument is the shape of the input tensor. None means that the length of that dimension could be anything. In our case, the first dimension is the batch size, and using None allows the network to handle arbitrarily sized batches. The probability of keeping a neuron in the dropout layer is also an input to the network because we enable dropout only during training. We disable it when evaluating the model.

EMBEDDING LAYER

The first layer we define is the embedding layer, which maps vocabulary word indices into low-dimensional vector representations. It’s essentially a lookup table that we learn from data.

CONVOLUTION AND MAX-POOLING LAYERS

Now we’re ready to build our convolutional layers followed by max-pooling. Remember that we use filters of different sizes. Because each convolution produces tensors of different shapes we need to iterate through them, create a layer for each of them, and then merge the results into one big feature vector.

DROPOUT LAYER

Dropout is the perhaps most popular method to regularize convolutional neural networks. The idea behind dropout is simple. A dropout layer stochastically “disables” a fraction of its neurons. This prevent neurons from co-adapting and forces them to learn individually useful features. The fraction of neurons we keep enabled is defined by the dropout\_keep\_prob input to our network. We set this to something like 0.5 during training, and to 1 (disable dropout) during evaluation.

SCORES AND PREDICTIONS

Using the feature vector from max-pooling (with dropout applied) we can generate predictions by doing a matrix multiplication and picking the class with the highest score. We could also apply a softmax function to convert raw scores into normalized probabilities, but that wouldn’t change our final predictions.

LOSS AND ACCURACY

Using our scores we can define the loss function. The loss is a measurement of the error our network makes, and our goal is to minimize it. The standard loss function for categorization problems it the cross-entropy loss. We also define an expression for the accuracy, which is a useful quantity to keep track of during training and testing.

CNN Tests Results for different values of embedding’s on 10-fold cross validation.

The data is divided into 25/75 ratio for test and train respectively.

|  |  |  |
| --- | --- | --- |
| Test Result | Embedding Parameters | Accuracy |
| 1. | For 3,4,5 word windows | 95.2381 |
| 2. | For 5,6,7word windows | 97.6190 |
| 3. | For 6,7,8 word windows | 92.8571 |

Loss function Analysis for CNN on the training data.

|  |  |  |
| --- | --- | --- |
| Test Result | Embedding Parameters | Step size, Loss function(cross entropy),Accuracy |
| 1. | For 3,4,5 word windows | step 200, loss 1.12745, acc 70.00 |
| 2. | For 5,6,7word windows | step 200, loss 0.599475, acc 90.00 |
| 3. | For 6,7,8 word windows | step 200, loss 1.46002, acc 70.00 |

3) **RECURRENT NEURAL NETWORKS**

Recurrent neural networks, specifically Long short-term memory networks were used as the model for classification. Recurrent Neural Networks are networks which include feedback loops, and thus have directed cycles. LSTM networks in particular have the characteristic of remembering values. The inputs to the LSTM are modeled as timesteps, with each input being given at a particular time or space. This is great for solving natural language processing problems, as the network can retain the context of the sentence in question when processing a particular word, and thus make better predictions. The given problem of predicting tags were transformed into a binary classification problem:

Each question had a particular tag attached at the end, and the output would be whether or not the tag is valid a tag for the question. All the words involved were replaced with their corresponding word vectors from the model created. For each question, each of its positive tags would be fed as input, along with an equal number of negative samples, where the tags do not belong to the question. This approach could not be followed for all the questions, as the systems we had quickly ran out of memory above a certain number of questions, and thus the number of questions were limited to 1000, which gave us around 4900 samples for the cooking dataset. The following configuration was used for deriving the final result:

model = Sequential()

model.add(LSTM(100, input\_shape=(250,100), return\_sequences='true'))

model.add(LSTM(50))

model.add(Dense(1, activation='relu'))

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

Given above is the code for constructing the network using Keras, with a TensorFlow backend. As can be seen, there are two LSTM layers, with an Dense layer with ReLU(Rectified Linear Unit) activation. The input shape is 250 \* 100 because each question was allocated 200 words each, and were either padded to the required length if they were smaller than this length, or cut down if smaller.

The rest of the input is simply the tag, repeated 50 times in order to give it more weightage when training the algorithm.

The final output layer is simply the dense layer, with ReLU activation. Sigmoid activation was initially tried, however it was biased more towards high values than lower, also ReLU was seen to give a high accuracy, and hence it was chosen.

Other modifications considered, but not included due to loss in accuracy were:

1. Including drop-off layers
2. Including more Dense layers
3. Including a Convolutional Network before the LSTM

Including more LSTM layers with more units was considered, but this turned out to be computationally infeasible due to extremely long running times, since a computer with a GPU was not available for the purposes of this project.

The following results were obtained with the final model trained on the cooking dataset:

Cooking  accuracy: 55.00%

Cooking  F-1 score: 0.523441275979

Biology Accuracy: 0.536876355748  
Biology F1-Score: 0.647515271587  
Crypto Accuracy: 0.500776699029  
Crypto F1-Score: 0.64503658705  
DIY Accuracy: 0.524739583333  
DIY F1-Score: 0.392678868552  
Robotics Accuracy: 0.480599647266  
Robotics F1-Score: 0.361517615176  
Travel Accuracy: 0.552581778859  
Travel F1-Score: 0.522917267224

As can be seen, the accuracy is pretty low for the cooking dataset itself. This is indicative of the model not being able to learn the required information with the given data, or the model being not being complex enough to learn it. There was little to be done about this given the resources, as the computational power and memory for using more samples or training more complex networks were not at hand. However, it seen that there is transfer of learning with the other datasets, with surprisingly high F1- scores with the Crypto and Biology datasets. DIY and Robotics were low as expected, since they have little in common with cooking.

Unfortunately the accuracy was so low that the model could not be used for predictions on Physics in a format suitable for submission to Kaggle, as the tags predicted could not be selected based on any degree of confidence, the model being so unreliable.

However, this is still an optimistic result, since this is a problem in which even the gold standard of human classification would still be very low. Consider the task of being asked to predict tags on cryptography after learning about cooking.

**4) LATENT DIRICHLET ALLOCATION (LDA)**

Topic models helps in analyzing large amount of unlabeled data. A topic may be defined as a group of words that frequently occur together which are generated before the document is. **LDA** is a probabilistic mode with a generative process that generates documents based on topic modeling. LDA was used as an experimental method to get the probability distribution of topics and words. We achieved the likelihood of a post having a tag by estimating the similarity between the tag and the topic.

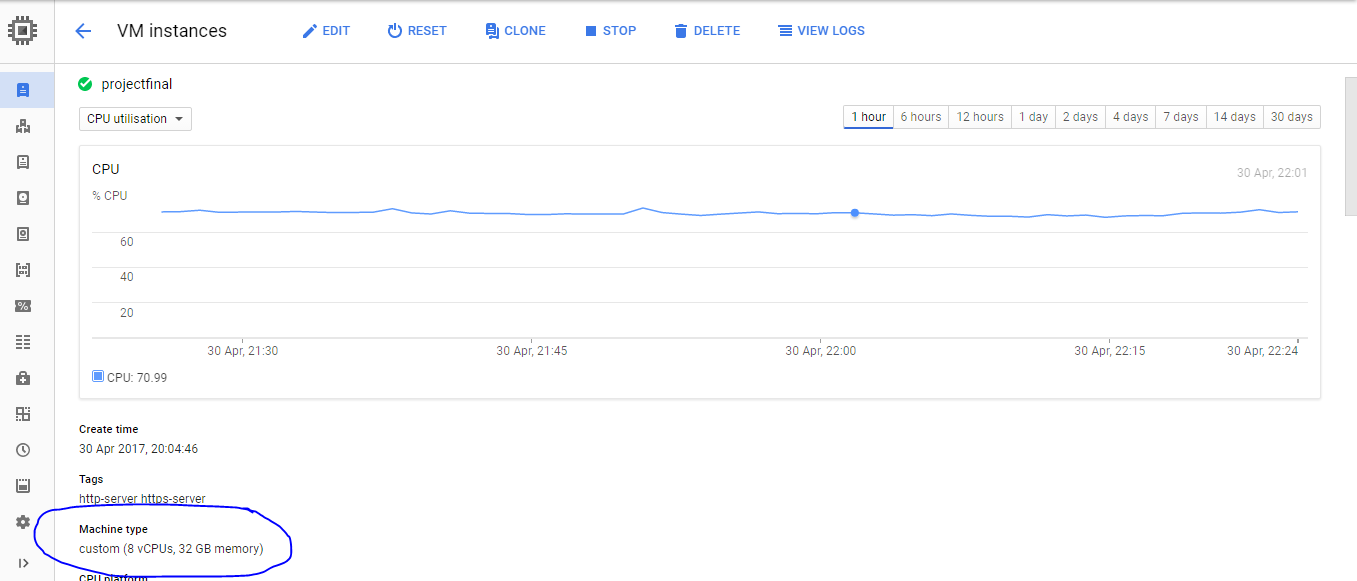
**CHALLENGES**  
1) Naïve Bayes – The folder structure in Naïve Bayes had to be the same for the training and the test data. Since the test data provided was only 1 folder, we had to simulate the folder structure by placing dummy files in the test folder.

2) LDA - The LDA approach used in this project was not suitable for predicting tags. It can be used to show the similarity between the topics and the words through a probability distribution but predicting tags can be achieved with other models like the Naïve Bayes, RNN and CNN. LDA approach is more suitable for predicting documents based on the topics.

3) The CNN seems to underfit over large word window sizes indicating it is unable to parse whole sentences at time.

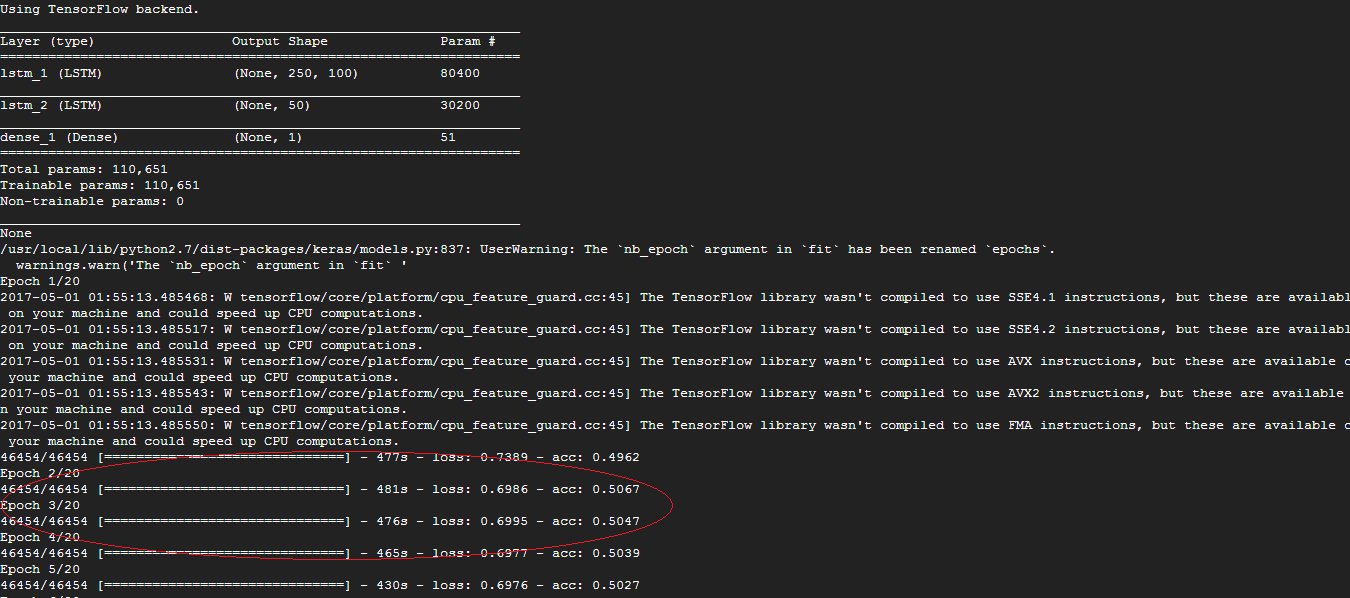
4) RNN - not enough memory or computational power to try learning with more samples or more complex networks.

5) Google Compute Engine - Since huge computation is required, we used Google Compute Engine. We create a Unix Virtual machine with the following configuration:



8vCPUs and 32GB memory

Even with this configuration and an input of 10000 questions (each question will have on an average of 2 tags and we add 2 wrong tags, which makes our input to around 40000 to 45000 sized array of 200 (number of words considered of each question) word vectors (each word vector is of 100 dimensions) it will take around 470s on an average for an epoch.



The above picture is a screenshot of a run of our module on Google Compute Engine.

Even with this configuration it takes around 2.5hrs to train the Recurrent Neural Network.

**CONCLUSION AND FUTURE WORK**

The LDA approach can be used for predicting the top k topics and documents based on those topics.

We observe that the CNN model for embedding parameters for 5,6,7 word windows gives us the best set of accuracy for classifying whether a question belongs to physics or not. For to attain transfer learning using CNN we devised the technique of combing the data from all the sample data from various categories except physics as negative data implying we would check f from the data we can infer the title and content if the new questions belong to the domain of physics or not. We trained it on various parameters and checked which of them gives the best accuracy. As we can see from above test results it is the embedding parameters of 5,6,7 which gives us the best model for this problem. We applied the softmax probability distribution for finding the best possible probability.

Recurrent Neural Networks can be tentatively used for the given task, considering that some transfer of learning was achieved with the limited number of samples and model complexity, however more work would be needed before they can be used for achieving extremely high accuracies in this area.

**CONTRIBUTION OF TEAM MEMBERS**

Rumela – Naïve Bayes and LDA

i) Using the preprocessed data from CNN built the Multinomial Naïve Bayes classifier to perform document classification

ii) Used accuracy as a measure to show the result of the document classification

iii) Performed experimental analysis using LDA for topic modeling.

Nagma - CNN

i) Pre-processed data for sentences built to use them as positive and negative label

ii) Built the various CNN MODELS using tensorflow

iii) Applied L2 regularization on the models.

Sachin – Pre-processing for RNN

i) Preprocessing for building word vector model and preprocessing for building RNN.

ii) Setting up and running on Google Cloud

Prasshant – Building RNN model

i) Building the word vector model

ii) Building the RNN models

iii) Experimentation with different RNN and deep learning model configurations

iv) Tag statistics exploration

v) Calculation of cross data-set accuracy and F-measure scores

**REFERENCES**

<https://rstudio-pubs-static.s3.amazonaws.com/79360_850b2a69980c4488b1db95987a24867a.html>

<https://en.wikipedia.org/wiki/Recurrent_neural_network>

<https://en.wikipedia.org/wiki/Long_short-term_memory>

<https://en.wikipedia.org/wiki/Word2vec>

<https://radimrehurek.com/gensim/models/word2vec.html>

<https://keras.io/getting-started/sequential-model-guide/>

<http://www.nltk.org/>

<https://arxiv.org/pdf/1510.03820.pdf>

https://arxiv.org/pdf/1408.5882.pdf