COSI134: Final Project

Meg Kobashi

December 2017

Usage

Train, evaluate, and classify with the following command:

```
python nn.py --cache_word2vec --word2vec_file="word2vec" --log
```

Optional command-line flags:

- `--cache_word2vec`...Whether to train the word2vec model. Default is false.
- `--word2vec_file`...The file containing the word2vec model. If `--cache_word2vec` is set, the trained model will be saved in this file. Otherwise, the existing model will be read from this file. Default is 'word2vec'.
- `--log`...Display Tensorflow training logs on the console. Default is false.

Logic

Network Architecture

The neural network that yields the highest accuracy consists of 4 layers, the input layer, two hidden layers, and the output layer. Each of the hidden layers have layer size 300. The predicted class would be the highest value emitted from the output layer. The loss function is sparse softmax cross entropy, and the optimizer algorithm is the Adam algorithm. Several optimizer algorithms were attempted, such as the gradient descent optimizer, but this required more training steps to tune the hyper-parameters.

Features

The features were word embeddings for the raw text from the dataset. First, $Arg1$, $Connective$, $Arg2$ was tokenized into individual words. Then, we took the average of the word embeddings for each word in a single part of the sentence, so there were three vectors for $Arg1$, $Connective$, $Arg2$. These three vectors

were concatenated to construct a feature for that instance. The input to the neural network was input of an array of these features across all instances. This approach allows us to consider the former and latter parts of the sentence to determine what comes between them, and also encompasses information about each word for finer granularity.

Implementation

Tensorflow

The Tensorflow Estimator API in tf.estimator was used for the model. The API is well-abstracted, saving us the energy of specifying the dimensions for the parameters and matrix multiplication. It also provides the flexibility to customize the model configurations, including the loss function and activation functions for each layer. All of these configurations can be bundled in a "function", and can differ between training, evaluation, and testing. I encountered one drawback of framework while investigating the reason for low accuracy. Model training happens under the hood, so it does not support things like running evaluation on the development set after every iteration of training, which could have been beneficial for debugging.

Word Embeddings

The word embeddings which were used as features were generated using the Gensim library. A single instance has a feature of a single word embedding vector of length 300. The word2vec model for Gensim is trained on the training corpus, so it can provide word embeddings that are specific to this type of text. One disadvantage is that the training set is smaller compared to other word2vec models, such as Google word2vec which was trained on a larger dataset. Since we can only fetch word embeddings for words contained in the training set, having a smaller corpus poses a problem. We were not able to generate word embeddings for many words in the development and evaluation set since many of them were not in the model vocabulary. We compromised by inserting the word $UNKNOWN$ for every 5000 words in the training set, and used this word embedding for $UNKNOWN$ whenever we encounter a word not in the vocabulary. This $UNKNOWN$ word embedding was also used in place of $Arg1$, $Connective$, $Arg2$ if either of them was an empty string, allowing us to have the same size vectors for all instances.

Performance

The classifier yielded the highest accuracy of 0.529% with the following parameters:

- learning rate = 0.0001

- number of steps = 100000
- size of first hidden layer = 300
- size of second hidden layer = 300
- mini-batch size = 50

Problems

For a while, the classifier only yielded an accuracy of 13-16%. To investigate, I first confirmed that the loss was monotonically decreasing with each step of training by using batch instead of mini-batches. Next, I experimented with multiple parameters including the number of layers, layer size, and learning rate, none of which made a dramatic improvement. What solved the issue was understanding the subtle difference between the Tensorflow parameters *num_epochs* and *steps*. The network trains on a single mini-batch in a single step. Therefore, if you have 1000 instances, and mini-batch size of 100, it would take 10 steps to go through the entire data set once. On the other hand, the number of epochs is the number of times you iterate through the data set. These two parameters are actually related to one another with the following equation:

$$steps = (\#of instances/batch\_size) * num\_epochs$$

You only have to specify either the *steps* or *num_epochs* because Tensorflow would derive the missing one from the aforementioned calculation. If you specify both, then it'll take the minimum between the user-specified *steps* and the calculated *steps*. The cause of the low accuracy was that the *num_epochs* was too small that it capped the *steps*, stopping the training before it converged. Setting *num_epochs* to *None* and *steps* to a larger number, or vice versa, drastically helped learning.