COSI134A FALL 2019

Project2: Neural Discourse Relation Classification *Jiawei Zhang*

This project is my implementation of applying feedforward neural network and convolutional network on Penn Discourse Treebank discourse relation classification. On average, the program takes 350s to run on my mac, including loading data, feature representation, training and evaluating. The classifier's accuracy is around 52% with ~80% on explicit discourse relations and ~30% on non - explicit discourse relations.

Feature representation

Tokenize: remove punctations, turn into lower case.
Word embeddings: 1) **BEST** use Keras embedding layer with word vector length 300; 2) pre-trained word2vec models on train + dev set; 3) pre-trained Glove word vectors.
Features: First 100 words in arg1, first 100 words in arg2 and first one word in connective.

Neural network architecture:

FFN

Layer	Output Shape
embedding	(None, 201, 300)
flatten	(None, 60300)
dense	(None, 256)
dense	(None, 128)
Output	(None, 21)

CNN

Layer	Output Shape
embedding	(None, 201, 300)
conv1d	(None, 199, 50)
global_max_pooling1d	(None, 50)
dense	(None, 128)
Output	(None, 21)

Hyperparameters:

learning rate = 0.001, batch size = 100, epochs = 5

Code structure

train_w2v_model.py *get_selftrained_w2v_model()* train word2vec model on train + dev set, save model locally; *get_selftrained_wv_dict()* save word vector into a dict (add UNK and PAD);
preprocessing.py: *load_data()* load data for pretrained word embedding, *load_data_embed()* load data for keras embedding
model.py *train_embedding_ffn()*, *train_embedding_cnn()* are FFN, CNN models for Keras embedding; *train_ffn(), train_cnn()* are FFN, CNN models for pretrained embedding
main.py train FFN and CNN models, evaluate on test set, output prediction in json file
nn.py: 2 hidden layer feedforward neural network from scratch. *fwd_computation()* is for forward computation, *backpropagation()* is for back propagation, *train_ffn()* is to train a FFN model

How to run

python3.7 main.py arg1 arg2 to train on Keras FFN and CNN and evaluate the model performance.
arg1: word2vec, glove, keras_embedding
arg2: keras_ffn, cnn
e.g. **python3.7 keras_embedding keras_ffn**
python3.7 main.py word2vec sractch_ffn to train and evaluate the self-written FFN model.

PS: Download glove.6B.300d.txt in the same folder with code if you want to run glove embedding.
http://nlp.stanford.edu/data/glove.6B.zip

Experiment

FFN

1. Word embedding method

We use 3 different word embedding methods. 1) train word2vec model on train and dev data; 2) use pre-trained glove word vectors trained on Wikipedia; 3) use Keras embedding layer

	Word2vec	Glove	Keras embedding
Precision	0.27423	0.29705	0.52575
Recall	0.26084	0.27738	0.49882
F1	0.26737	0.28688	0.51193

The results show that Keras embedding layer is the best choice. The reason might be too less corpus to learn to good vector representation for word2vec, and much UNK for glove (around 1/3). Also, it's an end-to-end process, word vectors are also trained during the learning process.
Hyperparameters: Features = 100 + 100 + 1, word vector length = 300, learning rate = 0.001, hidden layer = Dense(256, 128), batch size = 100, epochs = 5

2. Number of words to use

	5 + 5 + 1	10 + 10 + 1	50 + 50 + 1	100 + 100 + 1
Precision	0.51707	0.52073	0.5207	0.52575
Recall	0.48936	0.49488	0.49567	0.49882
F1	0.50283	0.50747	0.50787	0.51193

There is no significant difference between these 4 feature representations. 100 words in arg1, 100 words in arg2 and 1 word in connective has the best performance on test data. We ignore numbers larger than 100 for its too time consuming. Hyperparameters: word vector length = 300, learning rate = 0.001, hidden layer = Desne(256, 128), batch size = 100, epochs = 5

3. Number of hidden layers

	1	2	3	4
Precision	0.51741	0.52575	0.52531	0.51824
Recall	0.49173	0.49882	0.49882	0.49251
F1	0.50424	0.51193	0.51172	0.50505

We can see that FFN model with 2 layers performs the best. Maybe because epoch is a little big large, they all had the situation of overfitting, the dev loss increased a little bit. Hyperparameters: Features = 100 + 100 + 1, word vector length = 300, learning rate = 0.001, batch size = 100, epochs = 5
1: Dense(128) 2: Dense(256, 128) 3: Dense(256, 128, 64) 4: Dense(256, 128, 64, 32)

4. Hidden layer neurons

	64, 32	128, 64	256, 128	512, 256
Precision	0.51741	0.51698	0.52575	0.51907
Recall	0.49173	0.49173	0.49882	0.4933
F1	0.50424	0.50404	0.51193	0.50586

We can tell that two layer combination of Dense(256, 128) is the best choice. (512, 256) is a little bit overfitting and time consuming. Hyperparameters: Features = 100 + 100 + 1, hidden layer = Desne(256, 128), word vector length = 300, learning rate = 0.001, batch size = 100, epochs = 5

5. Learning rate

	0.0001	0.001	0.01
Precision	0.5207	0.52575	0.49503
Recall	0.49567	0.49882	0.47124
F1	0.50787	0.51193	0.48284

We can tell that learning rate = 0.001 is the best choice. Hyperparameters: Features = 100 + 100 + 1, word vector length = 300, hidden layer = Desne(256, 128), learning rate = 0.001, batch size = 100, epochs = 5

6. Batch size

	50	100	200
Precision	0.51796	0.52575	0.51827
Recall	0.48857	0.49882	0.49173
F1	0.50284	0.51193	0.50465

As batch size get larger, it takes less time to train the model. Batch size 100 would be the best.
Hyperparameters: Features = 100 + 100 + 1, word vector length = 300, hidden layer = Desne(256, 128), learning rate = 0.001, epochs = 5

CNN

1. Word embedding

	Word2vec	Glove	Keras embedding
Precision	0.29139	0.25476	0.51581
Recall	0.27738	0.24271	0.48857
F1	0.28421	0.24859	0.50182

Also, Keras embedding layer is the best choice. Hyperparameters: Features = 100 + 100 + 1, word vector length = 300, learning rate = 0.001, batch size = 100, epochs = 5

2. Number of words to use

	5 + 5 + 1	10 + 10 + 1	50 + 50 + 1	100 + 100 + 1
Precision	0.50873	0.51377	0.51203	0.51664
Recall	0.48227	0.48015	0.48621	0.48936
F1	0.49515	0.49405	0.49879	0.50263

Same as FFN, 100 words in arg1, 100 words in arg2 and 1 word in connective has the best performance on test data.
Hyperparameters: word vector length = 300, learning rate = 0.001, batch size = 100, epochs = 5

3. Number of hidden layers

	1	2	3	4
Precision	0.51664	0.51209	0.51458	0.50705
Recall	0.48936	0.48385	0.49094	0.48148
F1	0.50263	0.49757	0.50143	0.49394

We can see that CNN model with 1 dense layer performs the best. Hyperparameters: Features = 100 + 100 + 1, word vector length = 300, learning rate = 0.001, batch size = 100, epochs = 5

4. Hidden layer neurons

	32	64	128	256
Precision	0.51205	0.50788	0.51664	0.50995
Recall	0.48542	0.48227	0.48936	0.48463
F1	0.49838	0.49475	0.50263	0.49697

We can tell that dense layer with 128 neurons performs best.

As for learning rate and batch size, from the experiment of FNN we can guess that they won't influence the model as much as hyperparameters above, so we just use the original one. The CNN architecture is shown below:

Layer	Output Shape
embedding	(None, 201, 300)
conv1d	(None, 199, 50)
global_max_pooling1d	(None, 50)
dense	(None, 128)
Output	(None, 21)

Above all, with the current feature representation, the deep learning models all get a similar performance. If we would like to get a higher accuracy, maybe we should try different feature representations.

Extra credit experiments:
We code feedforward neural network from scratch and compare it with Keras FFN model. The accuracy is slightly worse than Keras FFN. Also, the running time is much longer, might because there is no optimization on low level languages.

	Keras	Self
Precision	0.28418	0.27213
Recall	0.27029	0.39029
F1	0.27706	0.17152

Hyperparameters: Word embedding: self-trained word2vec, Features = 100 + 100 + 1, hidden layer = Desne(256, 128), word vector length = 300, learning rate = 0.001, batch size = 100, epochs = 1