Deep Learning-Based Sentiment Classification

1. Main Models

1.1. CNN

Just like the convolution in 2D image, we concatenate the word vectors for each word in the whole sentence and get a matrix, which is then be processed by a set of filters with different window sizes, but only on one dimension. A feature c_i in the feature map is generated by

$$c_i = tanh(w \cdot x_{i:i+h-2} + b) \in [c_1, c_2, \ldots, c_{n-h+1}]$$

And then we apply a max-over-time pooling operation (capturing the most important feature) to deal with variable sentence lengths:

$$\hat{c} = \max\{c\}$$

And finally, we'll use the softmax to calculate the probability distribution of each class.

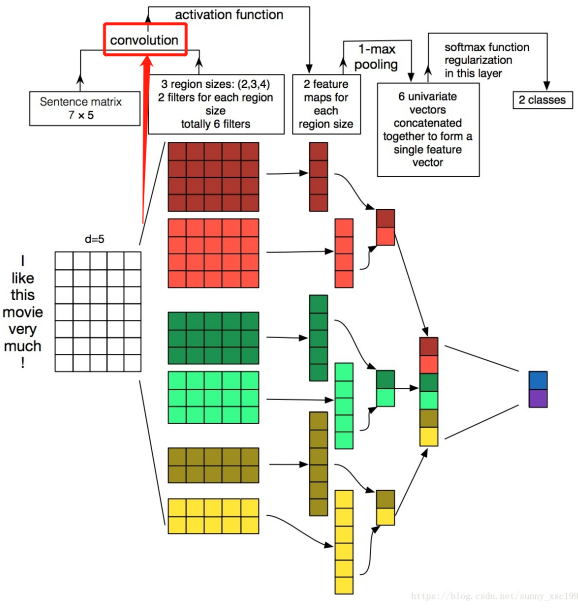

Figure 1. Model architecture with two channels for an example sentence

1.2. RNN

1.2.1. Vanilla RNN

RNN is a kind of neural network that can process inputs of variable length. A simple RNN will firstly concatenate word embeddings

$$x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)} \Rightarrow e = [e^{(1)}; e^{(2)}; e^{(3)}; e^{(4)}]$$

then get into the hidden layer

$$h = tanh(We + b_1)$$

and finally output a distribution over different classes, based on the element-wise max/mean of all hidden layers, i.e., sentence encoding.

$$\hat{y} = \text{softmax}(Uh + b_2)$$

Note that the built-in RNN function of PyTorch uses 0 vectors as the initial hidden state.

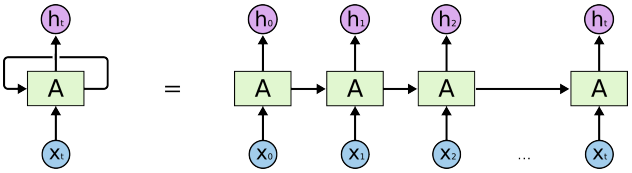

Figure 2. Recurrent Neural Networks and its unrolled form

1.2.2. Bidirectional RNN

For tasks like sentiment classification, we want the word representations contains information from both left and right context, rather than left only. So here we use bidirectional RNN to gain a concatenated representation of a word.

Instead of only running one RNN, we currently run two RNNs at the same time. Besides, to leverage the built-in function of torch for the dropout implementation, we used two-layer RNN, which just adds another layer to the original model.

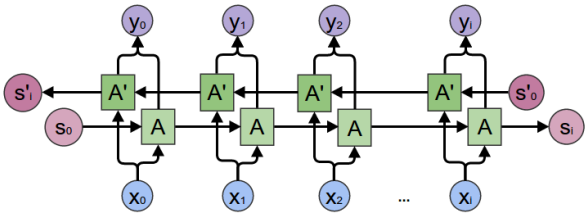

Figure 3. Unrolled Bidirectional RNN

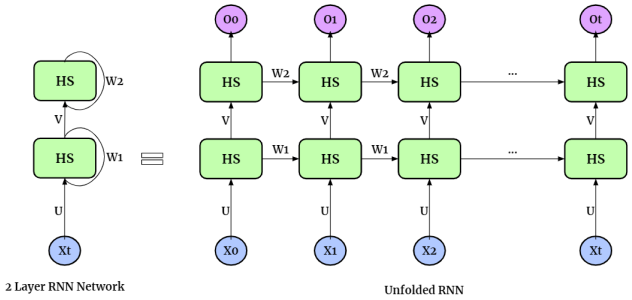

Figure 4. Two Layer RNN and its Unrolled Form

$$\text{Forward RNN} \quad h_{\rightarrow}^{(t)} = \text{RNN}_{FW}\left(h_{\rightarrow}^{(t-1)}, x^{(t)}\right)$$

$$\text{Backward RNN} \quad h_{\leftarrow}^{(t)} = \text{RNN}_{BW}\left(h_{\leftarrow}^{(t+1)}, x^{(t)}\right)$$

$$\text{Concatenated hidden states} \quad h^{(t)} = \left[h_{\rightarrow}^{(t)}, h_{\leftarrow}^{(t)}\right]$$

There are other variants of Vanilla RNN, e.g. LSTM and GRU, but since simple RNN does a good job here and we haven't encoutered problems like gradient vanishing, we didn't implement and discuss them in this report.

2. Common Techniques

2.1. Padding the Sentence Vectors

When we want to train either CNN or RNN with mini-batch using PyTroch, the function assumes each batch of the input instances is of the same size. Besides, for CNNs, if some of the sentences are too short, then the filters will fail to do convolution. So we have to pad our sentence vectors into the same length. In our implementation, this is achieved by defining customized $collate_fn$ for $Dataloader$.

2.2. Word Embeddings: Random Initialization

Firstly, we tried to initialize our word embeddings randomly use the built-in function of PyTorch, $torch.nn.Embedding$, which initialize all weights from $N(0, 1)$.

2.3. Word Embeddings: Pre-trained Model

Then for pre-trained models, we've tried both stanford Word2Vec, that were trained on 100 billion words from Google News with CBOW model, and Glove pre-trained embeddings, which were trained on Wikipedia 2014 and Gigaword 5. Both models are used through $gensim$ module. Words not present in the set of the pre-trained words will be simply ignored.

3. Regularization: Dropout

Neural Networks, especially the deep ones, are famous for their tendency to overfit, so regularization is important for deep learning methods. Here we only employed dropout on the penultimate layer of our RNN, which will randomly set some nodes to be 0 during training.

4. Experiment Result

4.1. Word Embeddings

Here are some results of the random initialization and pre-trained word embeddings for CNNs with 100 channels for each filter size.

Table 1. Random and Pre-train Embedding with CNN

Model	filters	with stem	Acc
CNN_Random	[3,4,5]	False	56.805
CNN_Random	[3,4,5]	True	55.7
CNN_Random	[2,3,4]	True	57.161
CNN_Random	[4,5,6]	True	57.237
CNN_CBOW300d	[3,4,5]	False	778
CNN_GloVe300d	[3,4,5]	False	56.591

Here are some results of the random initialization and pre-trained word embeddings for two layer Bidirectional RNNs with 100-dimension hidden state.

Table 2. Random and Pre-train Embedding with RNN

Model	dropout	with stem	Acc
RNN_Random	0.5	False	54.853
RNN_Random	0.5	True	54.565
RNN_CBOW300d	0.5	False	58.013
RNN_GloVe300d	0.5	False	57.595
RNN_GloVe300d	0.2	False	57.267

5. Summary and Thinking

From the experiment we found that, for CNN model with random initialized word embeddings, stemmed words outperform the non-stemmed words. Whereas for RNNs the story is totally different. However for pre-trained models,

both gained better performance with non-stemmed words, since the pre-trained models are trained on words with non-stemmed pattern.