TTIC 31210 Homework 1
Spr 2017

Hao Jiang

April 11, 2017

1 Word Average Sentence Classifier

1.1 Exposition

Given a sentence of N words $w_1, w_2, \ldots, w_N$, the simple word average sentence classifier is defined by the following formula

$$\mathbb{P}(c = 1) = \text{Sigmoid}(\frac{1}{N}(\sum_{i=1}^{N} \text{Embed}(w_i)) \cdot \mathbf{v})$$

, where Embed is word embedding function that converts a given word to $\mathbb{R}^d$, $\mathbf{v} \in \mathbb{R}^d$ a parameter, d a hyperparameter describing the size of embedding.

The function Embed is computed using a word embedding matrix E of size $W \times d$, where W is the vocabulary size. Each word w_i is assigned a unique index i, and the corresponding embedding is the i-th column vector in E. E is also a parameter that will be learned during the training process.

To learn the parameter E and $\mathbf{v}$, we use cross-entropy loss function for the sigmoid classifier, which is defined as following

$$L(x) = -c \log(x) - (1 - c) \log(1 - x)$$

, where x is the result from the *Sigmoid* function described above, $c \in 0, 1$ is the class label.

1.2 Implementation

I attach the code I use to implement this classifier. The implementation is based on my own ML framework. It includes the main program file `word_average.py` and library files under folder `ndnn/`.

The parameters are inited with Xavier, and the plain old SGD is used to update the parameters. The learning rate is set to be 0.01 and decay is 0.95. I also use mini-batching, which groups sentences of the same length together.

1.3 Experimentation

I set the word embedding dimension d to be 300 in this experiment. Figure 1 shows the result. The final test accuracy after 100 epoches is **0.8182**. It can be seen from the figure that Dev loss stop decreasing after 40 epoches, however it didn't increase much either. So it didn't trigger early stopping.

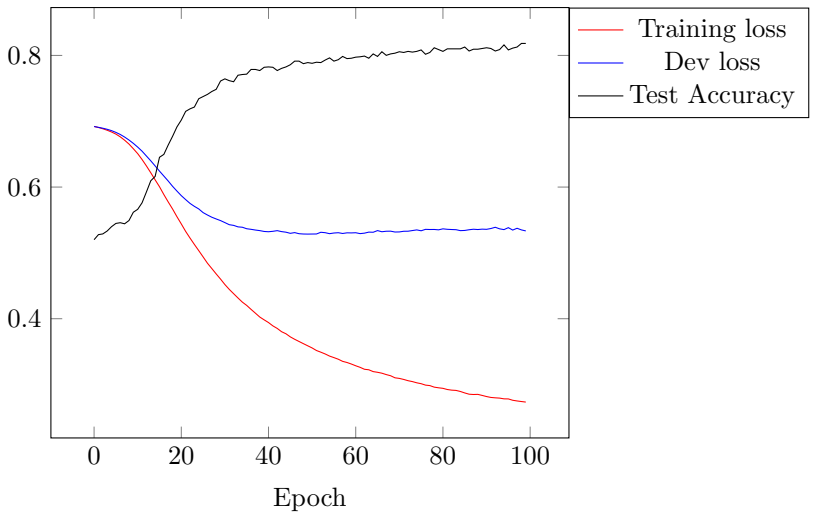

Figure 1: Training of Simple Word Average Classifier

Index	Largest Word	Largest Value	Smallest Word	Smallest Value
1	worst	3.10	Told	0.11
2	bad	2.91	Beneath	0.11
3	love	2.23	laptops	0.11
4	dull	2.10	Halos	0.11
5	enjoyable	2.09	worship	0.12
6	too	2.07	deliriously	0.12
7	heart	1.96	screens	0.12
8	best	1.88	re-creation	0.12
9	no	1.87	Yang	0.12
10	flat	1.87	abilities	0.12

Table 1: Top 10 words with largest and smallest norms

1.4 Analysis

The word with largest norm is "worst", with norm 3.107. The word with smallest norm is "Told", with norm 0.114.

In Table 1, we also list the top 10 words with largest norms, and top 10 words with smallest norms. An intuitive explanation is that for the words with obvious emotional inclination, e.g., "good" or "bad", the norm will be larger, while for neutral words the norm will be smaller.

2 Attention-Augmented Word Avreaging

2.1 Exposition

The attention-augmented word averaging classifier is described as following:

$$P(c = 1) = \text{Sigmoid}(\text{SoftMax}(W\mathbf{v})^T W\mathbf{w}) \tag{1}$$

$W = [Embed(w_1), Embed(w_2), \dots Embed(w_N)]$ is a matrix of size $N \times d$, where Embed is the same as above, N is the sentence length, and d is the hyperparameter controlling the word embedding dimension. $\mathbf{v}$ and $\mathbf{w}$ are both parameter

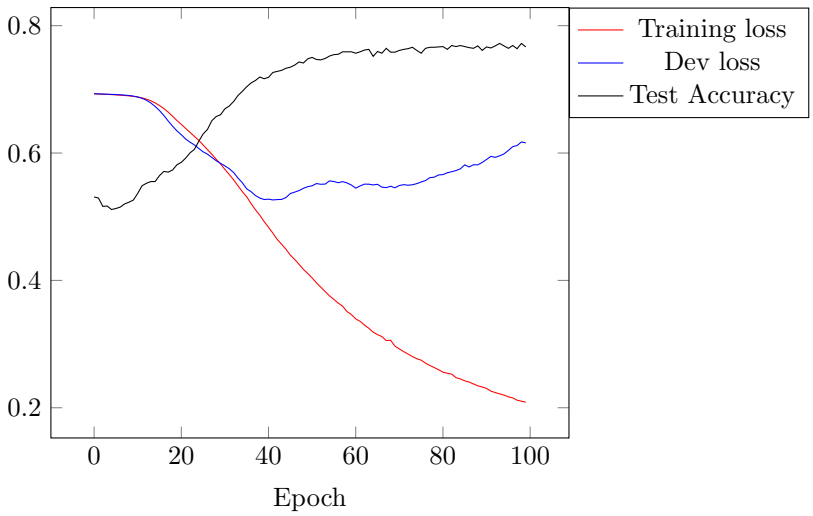

Figure 2: Training of Attention-Augmented Word Average Classifier

vectors of length d. In the training process, we need to learn the word embedding matrix E, as well as $\mathbf{v}$ and $\mathbf{w}$.

2.2 Implementation and Experimentation

The implementation is in `attention.py`. We again set the embedding dimension to 300, learning rate to 0.05 and decay to 0.95, and run 100 epoches. The experiment result is shown in Figure 2 The best test accuracy is **0.7721**, which is worse than the baseline. Around epoch 60, the dev loss begin increasing and early stopping is triggered.

2.3 Analysis

Examples of words with small, low-variance attention weights include **robots, Build, hovering, riddles, paycheck** Examples of words with large, low-variance attention weights include **Crummy, Rewarding, ROCKS, Imperfect, Weird** Examples of words with high-variance attention weights include **Schnieder, Wanders, Butterfingered, Criminal, Posey, Passionate, Ki-Deok**

These words have one thing in common: their frequency in the training text is very low. All the words appear only once or twice in the training text, which lead to the extreme distribution of the variance.

3 Enriching Attention Function

I tried three variations of attention augmented model, of which the details are described below. In all three cases, I didn't observe obvious change on the variation distribution of words. Words that appear only one or two times still dominate.

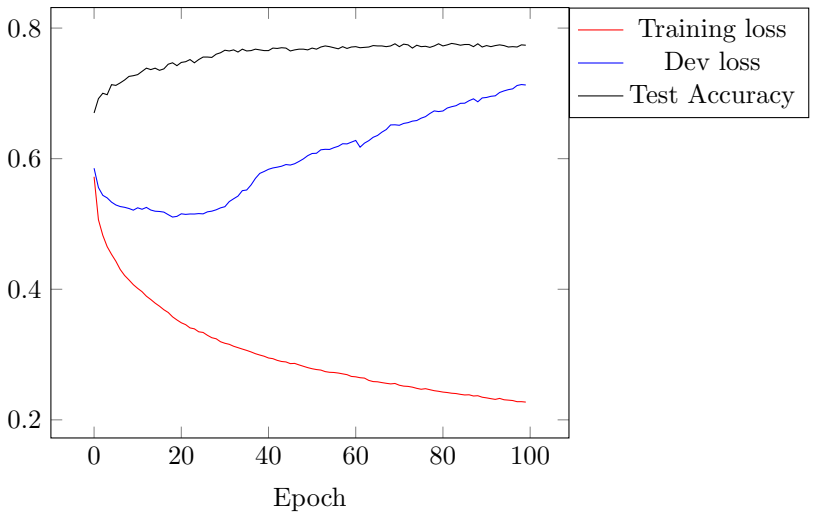

Figure 3: Training of Attention Variation 1 Classifier

3.1 Use Pre-trained Word Embedding

Instead of training from scratch, I use pretrained word embedding from the Glove project of Stanford NLP group. The dataset contains 1.9 million words trained from text corpus containing 840 billion words. The code is in `attention_var1.py` and the experiment result is shown in Figure 3. The training early stopped at around 30 epoches and achieves a test accuracy of **0.7699**

3.2 Use Nearby Words

When computing the attention weight of a word, I tries to take the nearby words into account. To do this, before applying SoftMax on $W\mathbf{v}$ in Equation (1) to compute the attention weight, I convolve a mask $g = [0.1, 0.2, 0.4, 0.2, 0.1]$ with each row vector of $W\mathbf{v}$, then apply SoftMax on the convolution result.

$$P(c = 1) = \text{Sigmoid}(\text{SoftMax}(\text{Conv}(W\mathbf{v}, g))^T W\mathbf{w})$$

The code is in `attention_var2.py` and the experiment result is shown in Figure 4. It can be seen that the training process early stopped at around 30 epoches and achieves test accuracy of **0.7935**.

3.3 Use Relative Position of the Word in Sentence

I introduce a parameter vector $\mathbf{r}$ to describe how the relative positon of a word in a sentence could affect the classification. The length of $\mathbf{r}$ is a fixed number L. Given $W\mathbf{v}$ of size N, we expand $\mathbf{r}$ to length N and do a elementwise-multiplication with $W\mathbf{v}$. Algorithm 1 describes how the Expand function works and thus the following formular describes the classifier

$$P(c = 1) = \text{Sigmoid}(\text{SoftMax}((W\mathbf{v} \otimes \text{Expand}(\mathbf{r}, N)))^T W\mathbf{w})$$

The code is in `attention_var3.py` and the experiment result is shown in Figure 5. The training process early stopped at around 80 epoches and achieves a test accuracy of **0.7699**.

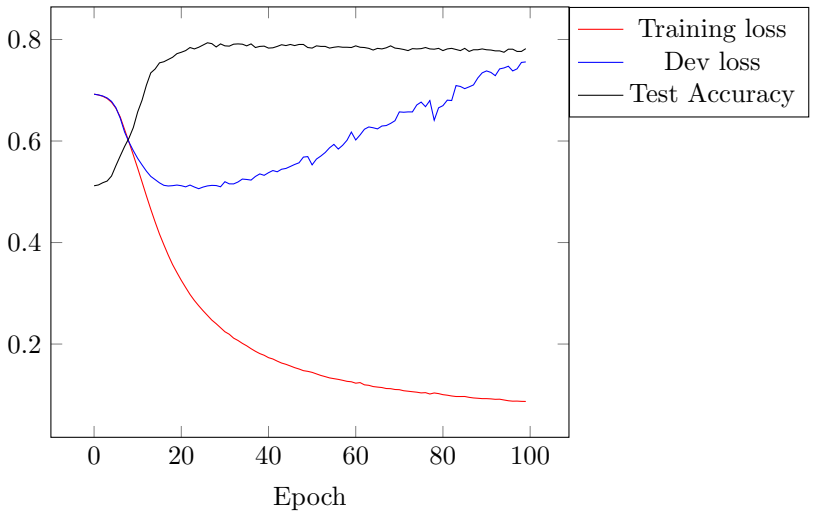

Figure 4: Training of Attention Variation 2 Classifier

Algorithm 1 Expand Function

```
function EXPAND(r, N)
    res = Array(N)
    L = r.length
    for i do = 0 to N-1
        res[i] = r[floor(i*L/N)]
    end forreturn res
end function
```

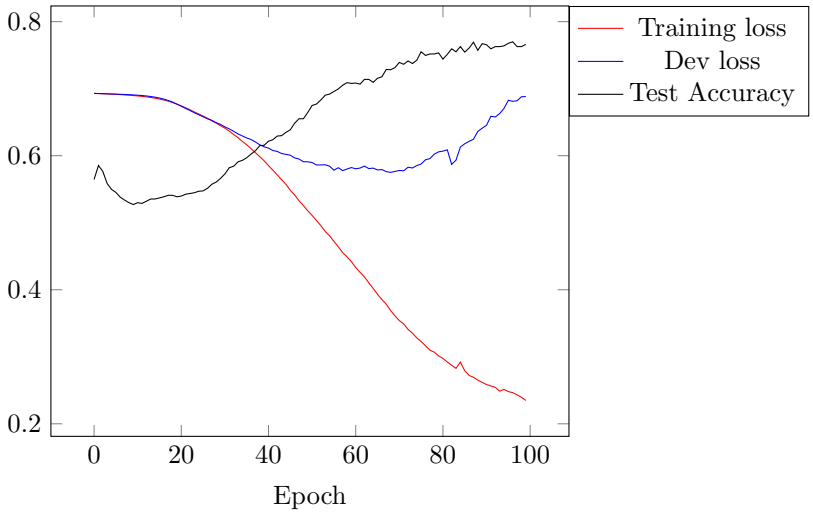

Figure 5: Training of Attention Variation 3 Classifier