1 Single Layer CNN

1.1 Introduction

At the early stage, Neural Network (NN) mostly appears in form of Multi-Layer Perceptrons (MLP), in which each layer is fully connected and the depth of layer determines its capacity on depicting the realistic. However, with the increase of layers, optimization functions sink more frequently into locally optimal solution, which deviates further from global optimal solution. Meanwhile, MLP faces vanishing gradient due to the *sigmoid* function.

To overcome vanishing gradient, other transmission function such as *ReLU*, *maxout* etc. replace *sigmoid* and construct the basic form of Deep Neural Network (DNN). Nevertheless, there is no difference between fully connected DNN and MLP in terms of structure, which brings a critical problem, the number of parameter incredibly. As a result, the model trains data slowly and easily encounter over-fitting. Under this background, Convolutional Neural Network (CNN, or ConvNet) which was invented in 1980 (inspired by biological processes), had been applied into research widely because of the development of GPU.

CNN restrict the number of parameter, focusing on mining local feature and preserving position relations among data. Therefore CNN has a space depth (compared with the Recurrent Neural Network (RNN) which has a time depth) and as a result, it is most commonly applied in computer vision field. Recent years, CNN was also applied in other field, such as natural language processing.

1.2 Structure

CNN consists of an input layer, several hidden layers (containing convolution layers, pooling layers, fully connected layers etc.) and a output layer. The single layer CNN is the most simple CNN model.

1.3 Single Layer CNN

1.3.1 Input Representation

To handle a sentence x with a single layer CNN model, we need to transform every word including punctuation into word vector. The sentence is represented as $x = \{x_1, x_2, ..., x_n\}$, where n is the length of the input sentence and each word vector $x_i \in \mathbb{R}^{d_w}$ is obtained by fetching from the word dictionary matrix.

Denote that t is the window size (t is odd). To ensure that every word vector will be used for the same times, we add $(t-1)/2$ word vectors at the beginning and at the end of the sentence x. Let us define the vector $r_i \in \mathbb{R}^{td_w \times 1}$ with

$$r_i = (x_{i-(t-1)/2}, ..., x_{i+(t-1)/2})^{\mathrm{T}} \tag{1}$$

The convolutional layer input matrix is defined as $R = [r_1, r_2, ..., r_n]$.

1.3.2 Convolution Layer

Convolutional layer applies a convolution operation to the input, in order to reduce the dimension of the data while extracting enough features. Note the convolution layer output matrix $R^* \in \mathbb{R}^{m \times n}$ defined with

$$R^* = f(W_f R + B_f) \tag{2}$$

where $W_f \in \mathbb{R}^{m \times td_w}$ is the filter matrix and m is the number of filters and B_f is a linear bias. Function f is the activate function (non linear), for example, $f = sigmoid()$, $f = ReLU()$ or $f = maxout()$ etc.

1.3.3 Pooling Layer

Pooling layers reduce the dimension of the data by combining the output neurons clusters at one layer into single neuron in the next layer. There is two type of pooling layer: max pooling and average pooling. For most of case, we use max pooling because average pooling might dilute the valuable extracted feature.

In case of max pooling, the feature vector $z \in \mathbb{R}^{m \times 1}$ is defined with

$$z_i = \max_j (R^*_{i,j}) \tag{3}$$

1.3.4 Output Layer

To compute the confidence score of each relation, the feature vector z needs to be transformed and fed into a softmax classifier to obtain the probability of each relation.

$$o = W_o z \tag{4}$$

where k is the number of possible output relation, and $W_o \in \mathbb{R}^{k \times n}$ is the transformation matrix and $o \in \mathbb{R}^{k \times 1}$ is the final output of the network.

1.3.5 Backpropagation Training

The softmax classifier is one of the basic functions used in the output layer. We define the parameter $\theta = (R, W_f, B_f, W_o)$ and we apply the softmax operation on vector o:

$$p(y = i | x; \theta) = \frac{exp(o_i)}{\sum_{j=1}^{k} exp(o_j)} \tag{5}$$

where y denotes the output relation type and $i \in [1, k]$

If we have a training set of N examples $\{x^{(i)}, y^{(i)}; i \in [1, N]\}$ and would like to learn the parameter θ of this model, we would begin by writing down the log-likelihood

$$J(\theta) = \sum_{i=1}^{N} \log p(y^{(i)}|x^{i}; \theta) \tag{6}$$

We can now obtain the maximum likelihood estimate of the parameters by maximizing $J(\theta)$ in terms of θ, using a method such as batch gradient descent as follows:

$$\theta \leftarrow \theta + \alpha \frac{\partial}{\partial \theta} J(\theta) \tag{7}$$

where α is the learning rate. Basically, when we are training on a large scale data set, we would choose rather stochastic gradient descent, which performs more efficiently.