NEURAL TENSOR NETWORK: EXPLORING RELATIONS AMONG TEXT ENTITIES

November 21, 2017

In this post, I will cover the Neural Tensor Network (NTN) as described in <i>Reasoning With Neural Tensor Networks for Knowledge Base Completion</i>. My implementation of NTN uses latest versions of Python 2.7, Keras 2.0, and Theano 0.9.

Skip directly to GitHub repository with code.

WHAT IS KNOWLEDGE BASE COMPLETION?

In knowledge base completion, the task is to identify a relationship between two entity pairs. For example, consider two entities pairs - `<cat, tail>` and `<supervised learning, machine learning>`: If we are asked to identify the relationship between the given two pairs - `<cat, R, tail>` and `<supervised learning, R, machine learning>` - then the first relationship can be best attributed as <i>has-type</i>, whereas the second relationship can be attributed as <i>instance-of</i>. So, we can redefine the two pairs as `<cat, has, tail>` and `<supervised learning, instance-of, machine learning>`. A neural tensor network (NTN) is trained on the database of entity-relationship pairs and is used to explore the additional relationship among the entities. This is achieved by representing each entity (i.e., each object or individual) in the database as a vector. These vectors can capture facts about that entity and how probable it is part of a certain relation. Each relation is defined through the parameters of a novel neural tensor network which can explicitly relate two entity vectors.

![Knowledge Base Diagram](image)

Predict new relationship triples using NTN.

NEURAL MODELS FOR REASONING OVER RELATIONS

The goal is to learn models for common sense reasoning, the ability to realize that some facts hold purely due to other existing relations. The NTN aims at finding the relationship among entities `<e1, e2>`, that is, predicting the relation `R` for `<e1, e2>` with some certainty. For instance, whether the relationship `<e1, R, e2> = (Bengal tiger, has part, tail)` is true and with what certainty. The Neural Tensor Network (NTN) replaces a standard linear neural network layer with a bilinear tensor layer that directly relates the two entity vectors across multiple dimensions. The model computes a score of how likely it is that two entities are in a certain relationship by the following NTN-based function:

http://deplearn-ai.com/2017/11/21/neural-tensor-network-expl...
$$g(e_1, R, e_2) = u^T_R f \left( e_1^T W_{1:2}^{[1:2]} e_2 + V_R \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} + b_R \right)$$

where is a standard nonlinearity applied element-wise, is a tensor and the bilinear tensor product results in a vector, where each entry is computed by one slice of the tensor. The other parameters for relation R are the standard form of a neural network: and .

**VISUALIZING NEURAL TENSOR LAYER**

The NTN models the relationship between two entities multiplicatively using the tensor variables. As shown above, the NTN is an extension to simple neural layer with the addition of these tensor variables. So, if we remove from the above figure, then we are left with the following terms

$$V_R \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} + b_R$$

which is a simple concatenation of entity vectors, along with the bias term.

**TRAINING OBJECTIVES**

The NTN is trained using contrastive max-margin objective function. Given the triplet in the training sample as , negative samples are created by randomly replacing the second entity as , where is a random index. Finally, the objective function is defined as

$$J(\Omega) = \sum_{i=1}^{N} \sum_{c=1}^{C} \max \left( 0, 1 - g \left( T^{(i)} \right) + g \left( T^{(c)} \right) \right) + \lambda \|\Omega\|_2^2,$$

where, is the regularization parameter.

**IMPLEMENTATION DETAILS**

Now, that we have seen the working of NTN, its time to dive into the implementation. One important point to be considered here is that each of the given relationships has its own set of tensor parameters. Let me give you a quick overview of what we need to do with the help of Keras.
Each of the relations is attributed to a separate Keras model which also adds the tensor parameters. For now, assume that the tensor layer is added in between model initialization and combination. Later in the post, I will explain the construction of the tensor layer. From the above figure, you can easily conclude that we need to process the training data in some way so that it can be passed to all the separate models simultaneously. What we want is to update only those tensor parameters that correspond to a specific relation. However, Keras doesn’t let us to update a separate model while leaving the rest. So, we need to divide the data into separate relations. Each training sample will consist of one instance of all the relations, that is, one pair of entities for each relation.

**Implementing the NTN Layer**

Let’s start by implementing the Neural Tensor Layer. The prerequisite for this section is writing custom layer in Keras. If you are not sure what that means, then have a look at the Keras documentation [writing-your-own-keras-layers](http://deplearn-ai.com/2017/11/21/neural-tensor-network-expl...).

We first initialize the NTN class with the parameters `inp_size`, `out_size`, and `activation`. The `inp_size` is the shape of the input variables, in our case the entities; the `out_size` is the number of tensor parameters (\(k\)), and `activation` is the activation function to be used (by default it is `tanh`).

```python
from ntn_input import *
from keras import activations

class ntn_layer(Layer):
    def __init__(self, inp_size, out_size, activation='tanh', **kwargs):
        super(ntn_layer, self).__init__(**kwargs)
        self.k = out_size
        self.d = inp_size
        self.activation = activations.get(activation)
        self.test_out = 0
```

The nomenclature of dimensions is kept same, that is, \(k\) corresponds to number of tensor parameter for each relation, and \(d\) is the shape of entity.

Now, we need to initialize the tensor layer parameters. In order to better understand what are we doing here, have a look at the following figure of the tensor network.
We initialize the four tensor's parameters, namely, $W$, $V$, $b$, and $U$ as follows:

def build(self,input_shape):
    self.W = self.add_weight(name='W',shape=(self.d, self.d, self.k), initializer='glorot_uniform', trainable=True)
    self.V = self.add_weight(name='V', shape=(self.k, self.d*2), initializer='glorot_uniform', trainable=True)
    self.b = self.add_weight(name='b', shape=(self.k,), initializer='zeros', trainable=True)
    self.U = self.add_weight(name='U', shape=(self.k,), initializer='glorot_uniform', trainable=True)
    super(ntn_layer, self).build(input_shape)

Here, we initialize the parameters with `glorot_uniform` sampling. In practice, this initialization results in better performance than other initialization. The `add_weight` function another parameter -- `trainable`, which can be set as `false` if we do not want to update a particular tuneable parameter. For example, we can set the $W$ parameter as non-trainable, and the NTN model will behave like a simple neural network, as discussed earlier.

Once, the parameters are initialized, it's time to implement the following equation:

$$g(e_1, R, e_2) = u^T_R f \left( e_1^T \left[ W_{[1:k]}^R e_2 + V^R e_2 \right] + b^R \right)$$

The above equation gives a score to each entity pair. As you can see, we have to iterate through $k$ tensor parameters (slices of tensor model). This is done by computing the middle products for each iteration, and finally, aggregating all these products. The following code snippet does this for you. Please do not change the name of the functions, as these are congruent to the Keras API.

def call(self,x,mask=None):
    e1=x[0] # entity 1
    e2=x[1] # entity 2
    batch_size = K.shape(e1)[0]
    V_out, h, mid_pro = [], [], []
    for i in range(self.k); # computing the inner products
        V_out = K.dot(self.V[i],K.concatenate([e1,e2])[i])
        temp = K.dot(e1,self.W[:,i,:])
        h = K.sum(temp*e2,axes=1)
        mid_pro.append(V_out+h+self.b[i])
tensor_bi_product = K.concatenate(mid_pro, axis=0)
tensor_bi_product = self.*self.activation(K.reshape(tensor_bi_product,(self.k,batch_size))).T

def compute_output_shape(self, input_shape):
    return (input_shape[0][0],self.k)

We have built the NTN layer which can be called like any other neural layer in Keras. Let’s see how to use the NTN layer on a real dataset.

**DATASET**

I will use the Wordbase and Freebase datasets, as mentioned in the paper. I have prepared the dataset (some part of preprocessing is taken from GitHub repositories), and can be processed as follows

```python
import ntn_input
data_name = 'wordbase' # 'wordbase' or 'freebase'
data_path = 'data'+data_name
raw_training_data = ntn_input.load_training_data(ntn_input.data_path)
raw_dev_data = ntn_input.load_dev_data(ntn_input.data_path)
entities_list = ntn_input.load_entities(ntn_input.data_path)
relations_list = ntn_input.load_relations(ntn_input.data_path)
indexed_training_data = data_to_indexed(raw_training_data,entities_list,relations_list)
indexed_dev_data = data_to_indexed(raw_dev_data,entities_list,relations_list)
(init_word_embs, entity_to_wordvec) = ntn_input.load_init_embs(ntn_input.data_path)

num_entities = len(entities_list)
num_relations = len(relations_list)
```

At this point you can print and see the entities along with their corresponding relationships. Now, we need to divide the dataset according to the relations, so that all individual Keras model can be updated simultaneously. I have included a pre-processing function that does this step for you. Negative samples are also added in this step. The negative samples are passed as the corrupt samples to the **prepare_data** function. If **corrupt_samples=1**, then one negative sample is added corresponding to each training sample. That means, the total training dataset will be doubled.

```python
import ntn_input
e1,e2,labels_train,t1,t2,labels_dev,num_relations = prepare_data(corrupt_samples)
```

The definition of NTN is stored in a file called **ntn**, which is easily imported for use.

**BUILDING THE MODEL**

To train the model we need to define the contrastive max-margin loss function.

```python
def contrastive_loss(y_true, y_pred):
    margin = 1
```
return K.mean(y_true * K.square(y_pred) + (1 - y_true) * K.square(K.maximum(margin - y_pred, 0)))

We should be able to call this custom loss function from the Keras compile function.

from ntn import *

def build_model(num_relations):
    Input_x, Input_y = [], []
    for i in range(num_relations):
        Input_x.append(Input(shape=(dimx,)))
        Input_y.append(Input(shape=(dimy,)))

    ntn, score = [], [] # storing separate tensor parameters
    for i in range(num_relations): # iterating through each slice 'k'
        ntn.append(ntn_layer[np_size=dimx, out_size=4][Input_x[i],Input_y[i]])
        score.append(Dense(1, activation='sigmoid')(ntn[i]))

    all_inputs = [Input_x[i] for i in range(num_relations)]
    all_inputs.extend([Input_y[i] for i in range(num_relations)]) # aggregating all the models

    model = Model(all_inputs, score) # aggregating all the models
    model.compile(loss=contrastive_loss, optimizer='adam')
    return model

Finally, we need to aggregate the data in order to train the model

e, t, labels_train, labels_dev = aggregate(e1, e2, labels_train, t1, t2, labels_dev, num_relations)
model.fit(e, labels_train, nb_epoch=10, batch_size=100, verbose=2)

At this point you could see that the model starts training, and the loss for each of the individual model decreases gradually. Furthermore, for calculating the accuracy of the NTN over the knowledge-base dataset, we need to compute the cost of all relations, and selecting the one with maximum score. The accuracy achieved is near to 88% (average), as mentioned in the paper.

WHAT'S NEXT?

In this post we have seen the construction of Neural Tensor Network for knowledge base completion. In the next post, we will see how NTN can be used in solving other NLP problems such as non-factoid based question answering.