KENN – Knowledge Enhanced Neural Networks

KENN (Knowledge Enhanced Neural Networks) is a library for python 2.7 built on top of TensorFlow 1 that permits to enhance neural networks models with logical constraints (clauses). It does so by adding a new final layer, called **Knowledge Enhancer (KE)**, to the existing neural network. The KE change the original predictions of the standard neural network enforcing the satisfaction of the constraints. Additionally, it contains *clause weights*, learnable parameters which represent the strength of the constraints.

This is an implementation of the model presented in our paper: Knowledge Enhanced Neural Networks.

If you use this software for academic research, please, cite our work using the following BibTeX:

```
@InProceedings{10.1007/978-3-030-29908-8_43,
author="Daniele, Alessandro
and Serafini, Luciano",
editor="Nayak, Abhaya C.
and Sharma, Alok",
title="Knowledge Enhanced Neural Networks",
booktitle="PRICAI 2019: Trends in Artificial Intelligence",
year="2019",
publisher="Springer International Publishing",
address="Cham",
pages="542--554",
isbn="978-3-030-29908-8"
}
```

How to create KENN docker container:

1. Install docker. See official docker website

2. From the terminal, load the KENN image:
   ```
   docker load < image_path/kenn_docker_v1.0.tar.gz
   ```
   This instruction tells docker to load the KENN image in the list of available images. To verify that the KENN image was correctly loaded:
   ```
   docker images -a
   ```

3. Create and run the container of KENN:
   ```
   docker run -d -t --name kenn kenn:1.0
   ```
   This instruction runs the container. The `-d` option tells docker to run the container in the detached mode, the `-t` option let you access the container using TTY, i.e. the terminal. Finally `--name kenn` specifies that the name of the container is kenn. You can change the name as you prefer. Notice however that if you use another name you would need to change the name in the next steps accordingly.

How to try the example provided with the docker image:

The docker image of KENN contains an example of usage with the Yeast dataset. To run the example:

```
docker exec -w /home/Yeast kenn python training.py
```

How to use the docker container with your own python code:

Like before, you need to execute your code using the docker `exec` command. However, this time you also need to copy your python script and the required data inside the container:

```
docker cp local_path/my_file.py kenn:desired_path/my_file.py
```

This instructions tells docker to copy `my_file.py` from the specified `local_path` to a desire location inside the container. This step must be repeated for all the files and data. A faster alternative is to copy directly the entire project folder using the same instruction:

```
docker cp local_path/my_project_folder kenn:desired_path/my_project_folder
```

Finally, to run your code:

```
docker exec -w desired_path/my_project_folder kenn python my_file.py
```

The `-w` option tells docker the working directory from where the code must be executed.

How to use KENN library in your python code:

To use KENN, you can start from existing TensorFlow code. Few changes must be done to add the logical clauses. Here a simple example, where we highlighted the 5 changes that are typically needed (for simplicity, we omit most of the standard TensorFlow code):

```
import tensorflow as tf

# *** 1 ***
from kenn import Knowledge_base as kb

# *** 2 ***
clauses = kb.read_knowledge_base(kb_file_name)

# Standard TensorFlow code
# ....

# Calculate preactivations
z = tf.matmul(h, w) + bias

# Standard code:
# y_hat = tf.nn.sigmoid(z)

# *** 3 ***
_, y_hat = kb.knowledge_enhancer(z, clauses)

# Loss definition
loss = tf.losses.mean_squared_error(y_hat, y)

# optimizer
train_step = tf.train.RMSPropOptimizer(learning_rate).minimize(loss)

# *** 4 ***
clauses_clip_ops = kb.clip_weigths(clauses)

with tf.Session(config=config) as sess:
    sess.run(tf.global_variables_initializer())

    for _ in range(number_of_steps):
```

```
            sess.run(train_step)

            # *** 5 ***
            sess.run(clauses_clip_ops)

        # Evaluations and standard operations ...

        # *** 6 (optional) ***
        learned_kb = kb.kb_to_string(sess, clauses)

        with open(kb_output_file, 'w') as kb_file:
            kb_file.writelines(learned_kb)
```

Example explained

In the previous example, we applies 5 changes to the standard TensorFlow code. Following, the details.

1. Import knowledge_base module

The first change is trivial, we need to import the library:

```
from kenn import Knowledge_base as kb
```

2. Read the knowledge base file

```
clauses = kb.read_knowledge_base(kb_file_name)
```

The `read_knowledge_base` function takes as input the path of the file containing the logical constraints. Following, an example of knowledge base file:

```
Dog,Cat,Animal,Car,Truck,Chair

1.5:nDog,Animal
_:nCat,Animal
2.0:nDog,nCat
_:nCar,Animal
_:nAnimal,Dog,Cat
```

The first row contains a list of predicates separated with a comma with no spaces. Each predicate must start with a capital letter. The second row must be empty. Other rows contain the clauses.

Each clause is in a separate row and must be written respecting this properties:

- logical disjunctions are represented with commas
- if a literal is negated, it must be preceded by the lowercase 'n'
- they must contain only predicates specified in the first row
- there shouldn't be spaces

Additionally, each clause must be preceded by a positive weight that represents the strength of the clause. More precisely, the weight could be a numeric value or an underscore: in the first case, the weight is fixed and determined by the specified value, in the second case the weight is learned during training.

For example, the third line represents the clause

$$\neg \text{Dog}(x) \vee \text{Animal}(x)$$

and it tells us that a dog should also be an animal. In this case, the clause weight is fixed to the value 1.5. A more interesting clause is the last one, that tells us that in our domain only cats and dogs are animals. Moreover, the corresponding weight is learned and if the constraint is not satisfied in the training set, KENN learn to ignore it.

3. Knowledge Enhancer (KE)
This is the most relevant change to the original code. Usually, in the last layer of a neural network, we apply an activation function to the preactivations calculated by previous layer.

```
y_hat = tf.nn.sigmoid(z)
```

Here, instead, we feed the preactivations to the knowledge enhancer:

```
_, y_hat = kb.knowledge_enhancer(z, clauses)
```

The `knowledge_enhancer` function takes as input the preactivations of the final layer and the set of clauses previously generated by the `read_knowledge_base` function (see 2.). It defines a new layer that changes the results by enforcing the clauses satisfaction. More precisely, it returns a tuple containing the preactivations (useful when using the `cross_entropy_with_logits` loss function) and final activations of the KE layer.

At this point, we can use the output of the `knowledge_enhancer` as we would have used the original activations: we can define the loss function directly on such predictions.

NB: in order to work properly, the preactivations tensor given as input to `knowledge_enhancer` should be a matrix whose rows represent a possible grounding. The columns represent the predicates of the knowledge base and it should be in the same order as specified in the first row of knowledge base file.

4-5. Clauses weights clipping
Another change that is needed is to define the `clauses_clip_ops`.

```
clauses_clip_ops = kb.clip_weigths(clauses)
```

Such operation sets to zero the clause weights that became negative. It should be called after each train step in order to keep the clauses weights positive:

```
_ = sess.run(train_step)
sess.run(clauses_clip_ops)
```

6. Saving learned clauses weigths
Optionally, by calling `kb_to_string` function, it is possible to obtain a string containing the clauses with the learned weights and storing it into a file.

```
# *** 6 (optional) ***
learned_kb = kb.kb_to_string(sess, clauses)

with open(kb_output_file, 'w') as kb_file:
    kb_file.writelines(learned_kb)
```

The content of the string will be something like this:

```
1.5:nDog,Animal
1.301:nCat,Animal
2.0:nDog,nCat
3.02:nCar,Animal
0.0:nAnimal,Dog,Cat
```

As you can see, these are the same clauses given as input through the knowledge base file (see 2.) with the only difference that the underscores are substituted by actual numbers, that are the learned clause weights. For instance, in this example the method learned that the last constraint is not taken into consideration by KENN.

NB: notice that writing this last piece of code is not mandatory, but reading the learned clause weights could help the user to better understand the importance of each clause in the final predictions made by KENN, increasing in this way the interpretability.