Grounding in Logic Tensor Networks (LTN)

Real Logic

The semantics of LTN differs from the standard abstract semantics of First-order Logic (FOL) in that domains are interpreted concretely by tensors in the Real field. To emphasize the fact that LTN interprets symbols which are grounded on real-valued features, we use the term *grounding*, denoted by $\mathcal{G}$, instead of interpretation. $\mathcal{G}$ associates a tensor of real numbers to any term of the language, and a real number in the interval $[0,1]$ to any formula ϕ. In the rest of the tutorials, we commonly use "tensor" to designate "tensor in the Real field".

The language consists of a non-logical part (the signature) and logical connectives and quantifiers.

- **constants** denote individuals from some space of tensors $\bigcup_{n_1 \ldots n_d \in \mathbb{N}^*} \mathbb{R}^{n_1 \times \cdots \times n_d}$ (tensor of any rank). The individual can be pre-defined (data point) or learnable (embedding).
- **variables** denote sequence of individuals.
- **functions** can be any mathematical function either pre-defined or learnable. Examples of functions can be distance functions, regressors, etc. Functions can be defined using any operations in Tensorflow. They can be linear functions, Deep Neural Networks, and so forth.
- **predicates** are represented as mathematical functions that map from some n-ary domain of individuals to a real from $[0,1]$ that can be interpreted as a truth degree. Examples of predicates can be similarity measures, classifiers, etc.
- **connectives** -- not, and, or, implies -- are modeled using fuzzy semantics.
- **quantifiers** are defined using aggregators.

This tutorial explains how to ground constants, variables, functions and predicates.

In [1]:
```
import ltn
import tensorflow as tf
import numpy as np
```

```
Init Plugin
Init Graph Optimizer
Init Kernel
```

Constants

LTN constants are grounded as some real tensor. Each constant c is mapped to a point in $\mathcal{G}(c) \in \bigcup_{n_1 \dots n_d \in \mathbb{N}^*} \mathbb{R}^{n_1 \times \cdots \times n_d}$. Notice that the objects in the domain may be tensors of any rank. A tensor of rank 0 corresponds to a scalar, a tensor of rank 1 to a vector, a tensor of rank 2 to a matrix and so forth, in the usual way.

Here we define $\mathcal{G}(c_1) = (2.1, 3)$ and $\mathcal{G}(c_2) = \begin{pmatrix} 4.2 & 3 & 2.5 \\ 4 & -1.3 & 1.8 \end{pmatrix}$.

In [2]:
```
c1 = ltn.Constant([2.1,3], trainable=False)
c2 = ltn.Constant([[4.2,3,2.5],[4,-1.3,1.8]], trainable=False)
```

```
2021-09-24 16:44:05.428136: I tensorflow/core/common_runtime/pluggable_devi
2021-09-24 16:44:05.428252: I tensorflow/core/common_runtime/pluggable_devi
```

```
Metal device set to: Apple M1

systemMemory: 16.00 GB
maxCacheSize: 5.33 GB
```

Note that a constant can be set as learnable by using the keyword argument `trainable=True`. This is useful to learn embeddings for some individuals. The features of the tensor will be considered as trainable parameters (learning in LTN is explained in a further

notebook).

In [3]:
```
c3 = ltn.Constant([0.,0.], trainable=True)
```

You can access the TensorFlow value of a LTN constant or any LTN expression `x` by querying `x.tensor`.

In [4]:
```
print(c1)
print(c1.tensor)
print(c3)
print(c3.tensor)
```

```
ltn.Constant(tensor=[2.1 3. ], trainable=False, free_vars=[])
tf.Tensor([2.1 3. ], shape=(2,), dtype=float32)
ltn.Constant(tensor=<tf.Variable 'Variable:0' shape=(2,) dtype=float32, num
<tf.Variable 'Variable:0' shape=(2,) dtype=float32, numpy=array([0., 0.], d
```

Predicates

LTN Predicates are grounded in mappings that assign a value between zero and one to some n-ary space of input values. Predicates in LTN can be neural networks or any other function that achieves such a mapping.

There are different ways to construct a predicate in LTN:

- the default constructor `ltn.Predicate(model)` takes in argument a `tf.keras.Model` instance; it can be used to ground any custom function (succession of operations, Deep Neural Network, ...) that return a value in $[0, 1]$,
- the lambda constructor `ltn.Predicate.Lambda(function)` takes in argument a lambda function; it is appropriate for small mathematical operations with **no trainable weights** (non-trainable function) that return a value in $[0, 1]$.

The following defines a predicate P_1 using the similarity to the point $\vec{\mu} = \langle 2, 3 \rangle$ with $\mathcal{G}(P_1) : \vec{x} \mapsto \exp(-\|\vec{x} - \vec{\mu}\|^2)$, and a predicate P_2 defined using a Tensorflow model.

In [5]:
```
mu = tf.constant([2.,3.])
P1 = ltn.Predicate.Lambda(lambda x: tf.exp(-tf.norm(x-mu,axis=1)))

class ModelP2(tf.keras.Model):
    """For more info on how to use tf.keras.Model:
    https://www.tensorflow.org/api_docs/python/tf/keras/Model"""
    def __init__(self):
        super(ModelP2, self).__init__()
        self.dense1 = tf.keras.layers.Dense(5, activation=tf.nn.elu)
        self.dense2 = tf.keras.layers.Dense(1, activation=tf.nn.sigmoid) # re
    def call(self, x):
        x = self.dense1(x)
        return self.dense2(x)

modelP2 = ModelP2()
P2 = ltn.Predicate(modelP2)
```

One can easily query predicates using LTN constants and LTN variables (see further in this notebook to query using variables).

In [6]:
```
c1 = ltn.Constant([2.1,3],trainable=False)
c2 = ltn.Constant([4.5,0.8],trainable=False)

print(P1(c1))
```

```
ltn.Formula(tensor=0.904837429523468, free_vars=[])
```

NOTE:

- If an LTN predicate (or an LTN function) takes several inputs, e.g. $P_3(x_1, x_2)$, the arguments must be passed as a list (cf Tensorflow's conventions).
- LTN converts inputs such that there is a "batch" dimension on the first axis. Therefore, most operations should work with `axis=1` .

In [7]:
```
class ModelP3(tf.keras.Model):
    def __init__(self):
        super(ModelP3, self).__init__()
        self.dense1 = tf.keras.layers.Dense(5, activation=tf.nn.elu)
        self.dense2 = tf.keras.layers.Dense(1, activation=tf.nn.sigmoid) # re

    def call(self, inputs):
        x1, x2 = inputs[0], inputs[1] # multiple arguments are passed as a li
        x = tf.concat([x1,x2],axis=1) # axis=0 is the batch axis
        x = self.dense1(x)
        return self.dense2(x)

P3 = ltn.Predicate(ModelP3())
print(P3([c1,c2])) # multiple arguments are passed as a list
```

```
ltn.Formula(tensor=0.999293863773346, free_vars=[])
```

We can define trainable or non trainable 0-ary predicates (propositional variables) using `ltn.Proposition`. They are grounded as a mathematical constant in $[0, 1]$.

In [8]:
```
# Declaring a trainable 0-ary predicate with initial truth value 0.3
A = ltn.Proposition(0.3, trainable=False)
print(A)
```

```
ltn.Proposition(tensor=0.30000001192092896, trainable=False, free_vars=[])
```

For more details and useful ways to create predicates (incl. how to integrate multiclass classifiers as binary predicates), see the complementary notebook.

Functions

LTN functions are grounded as any mathematical function that maps n individuals to one individual in the tensor domains.

There are different ways to construct an LTN function in LTN:

- the default constructor `ltn.Function(model)` takes in argument a `tf.keras.Model` instance; it can be used to ground any custom function (succession of operations, Deep Neural Network, ...),
- the lambda constructor `ltn.Function.Lambda(function)` takes in argument a lambda function; it is appropriate for small mathematical operations with **no weight tracking** (non-trainable function).

The following defines the grounding of a function f_1 that computes the difference of two inputs with $\mathcal{G}(f_1) : \vec{u}, \vec{v} \mapsto \vec{u} - \vec{v}$ and a function f_2 that uses a deep neural network that projects a value to $\mathbb{R}^5$.

In [9]:
```
f1 = ltn.Function.Lambda(lambda args: args[0]-args[1])

class MyModel(tf.keras.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(5)
    def call(self, x):
        x = self.dense1(x)
        return self.dense2(x)

model_f2 = MyModel()
f2 = ltn.Function(model_f2)
```

One can easily query predicates using LTN constants and LTN variables (see further in this notebook to query using variables).

In [11]:
```
c1 = ltn.Constant([2.1,3], trainable=False)
c2 = ltn.Constant([4.5,0.8], trainable=False)
print(f1([c1,c2])) # multiple arguments are passed as a list
print(f2(c1))
```

```
ltn.Term(tensor=[-2.4  2.2], free_vars=[])
ltn.Term(tensor=[-1.2620945   2.789203    0.17895734 -3.853095    2.1472168
```

Variables

LTN variables are sequences of individuals/constants from a domain. Variables are useful to write quantified statements, e.g. $\forall x \ P(x)$. Notice that a variable is a sequence and not a set; the same value can occur twice in the sequence.

The following defines two variables x and y with respectively 10 and 5 individuals, sampled from normal distributions in $\mathbb{R}^2$.
In LTN, variables need to be labelled (see the arguments `'x'` and `'y'` below).

In [13]:
```
x = ltn.Variable('x',np.random.normal(0.,1.,(10,2)))
y = ltn.Variable('y',np.random.normal(0.,4.,(5,2)))

print(x)
```

```
ltn.Variable(label=x, tensor=[[-3.3031723  -1.1864299 ]
 [ 0.88466865  0.91883767]
 [ 0.04525238  1.4658298 ]
 [-2.4912388   0.1417496 ]
 [ 1.0241538  -0.24183248]
 [-0.27830693 -0.75147146]
 [ 1.9639659  -1.0334796 ]
 [-0.19559991 -0.48291555]
 [ 0.4123148  -0.82985425]
 [-0.62613714  0.9860003 ]], free_vars=['x'])
```

Evaluating a term/predicate with one variable of n individuals yields n output values, where the i-th output value corresponds to the term calculated with the i-th individual.

For example, if x has 10 individuals, and y has 5 individuals, $P_3(x,y)$ returns 10×5 values. LTN keeps track of these dimensions and how they are connected to the variables using the attribute `free_vars` .

In [16]:
```
# Notice that the outcome is a 2 dimensional tensor where each cell
# represents the satisfiability of P3 with each individual in x and in y.
res1 = P3([x,y])
print(res1)
print(res1.take('x',2).take('y',0)) # gives the result calculated with the 3r
```

```
ltn.Formula(tensor=[[0.16020344 0.99925524 0.23648833 0.11888503 0.09269554]
 [0.37003732 0.9999889  0.46002787 0.6520825  0.3044056 ]
 [0.31776428 0.9999651  0.37986833 0.49643877 0.24240157]
 [0.2072321  0.99957293 0.24654248 0.16752328 0.12342633]
 [0.40251592 0.99999356 0.47012708 0.7431883  0.3429705 ]
 [0.32867122 0.9999753  0.37844056 0.4107827  0.23043956]
 [0.4734048  0.99999833 0.59228337 0.93088204 0.5602832 ]
 [0.33399373 0.99997556 0.377496   0.42079565 0.2359993 ]
 [0.37253213 0.99998915 0.42888838 0.60755545 0.29154813]
 [0.29656985 0.9999354  0.3264082  0.36363715 0.20904014]], free_vars=['x', 'y
ltn.Formula(tensor=0.3177642822265625, free_vars=[])
```

In [19]:
```
# This is also valid with the outputs of `ltn.Function`
res2 = f1([x,y])
print(res2.tensor.shape)
print(res2.free_vars)
print(res2.take('x',2).take('y',0)) # gives the result calculated with the 3r
```

```
(10, 5, 2)
['x', 'y']
ltn.Term(tensor=[7.6769648 2.0221434], free_vars=[])
```

In [21]:
```
res3 = P3([c1,y])
print(res3) # Notice that no axis is associated to a constant.
```

```
ltn.Formula(tensor=[0.3644095  0.9999945  0.71159357 0.83346635 0.45896447]
```

Variables made of trainable constants

We can create a variable made of trainable constants. In this case, we need to define the variable within the scope of a `tf.GradientTape` object (used for training, see tutorial 3). The tape will track the weights between the variable and the constants.

In [23]:
```
c1 = ltn.Constant([2.1,3], trainable=True)
c2 = ltn.Constant([4.5,0.8], trainable=True)

with tf.GradientTape() as tape:
    # Notice that the assignation must be done within a tf.GradientTape.
    # Tensorflow will keep track of the gradients between c1/c2 and x.
    # Read tutorial 3 for more details.
    x = ltn.Variable.from_constants("x", [c1,c2], tape=tape)
    res = P2(x)
tape.gradient(res.tensor,c1.tensor).numpy() # the tape keeps track of gradien
```

Out[23]:
```
array([-0.00143811,  0.01174497], dtype=float32)
```