Grounding in LTN (cont.)

This tutorial explains how to ground connectives and quantifiers. It expects some familiarity with the first tutorial on grounding non-logical symbols (constants, variables, functions and predicates).

In [1]:

```
import ltn
import numpy as np
import tensorflow as tf
```

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

Connectives

LTN suppports various logical connectives. They are grounded using fuzzy semantics. We have implemented the most common fuzzy logic operators using tensorflow primitives in `ltn.fuzzy_ops`. We recommend to use the following configuration:

- not: the standard negation $\neg u = 1 - u$,
- and: the product t-norm $u \wedge v = uv$,
- or: the product t-conorm (probabilistic sum) $u \vee v = u + v - uv$,
- implication: the Reichenbach implication $u \rightarrow v = 1 - u + uv$,

where u and v denote two truth values in $[0, 1]$. For more details on choosing the right operators for your task, read the complementary notebook.

In [2]:

```
Not = ltn.Wrapper_Connective(ltn.fuzzy_ops.Not_Std())
And = ltn.Wrapper_Connective(ltn.fuzzy_ops.And_Prod())
Or = ltn.Wrapper_Connective(ltn.fuzzy_ops.Or_ProbSum())
Implies = ltn.Wrapper_Connective(ltn.fuzzy_ops.Implies_Reichenbach())
```

The wrapper `ltn.Wrapper_Connective` allows to use the operators with LTN formulas. It takes care of combining sub-formulas that have different variables appearing in them (the sub-formulas may have different dimensions that need to be "broadcasted").

In [3]:

```
x = ltn.Variable('x',np.random.normal(0.,1.,(10,2))) # 10 values in R²
y = ltn.Variable('y',np.random.normal(0.,2.,(5,2))) # 5 values in R²

c1 = ltn.Constant([0.5,0.0], trainable=False)
c2 = ltn.Constant([4.0,2.0], trainable=False)

Eq = ltn.Predicate.Lambda(lambda args: tf.exp(-tf.norm(args[0]-args[1],axis=1))) # pre

Eq([c1,c2])
```

```
2021-09-24 17:02:27.556414: I tensorflow/core/common_runtime/pluggable_dev
2021-09-24 17:02:27.556508: I tensorflow/core/common_runtime/pluggable_dev
```

```
Metal device set to: Apple M1

systemMemory: 16.00 GB
maxCacheSize: 5.33 GB
```

Out[3]:

```
ltn.Formula(tensor=0.01775427535176277, free_vars=[])
```

In [4]:

```
Not(Eq([c1,c2]))
```

Out[4]:

```
ltn.Formula(tensor=0.9822457432746887, free_vars=[])
```

In [5]:

```
Implies(Eq([c1,c2]), Eq([c2,c1]))
```

Out[5]:

```
ltn.Formula(tensor=0.9824644327163696, free_vars=[])
```

In [6]:

```
# Notice the dimension of the outcome: the result is evaluated for every x.
And(Eq([x,c1]), Eq([x,c2]))
```

Out[6]:

```
ltn.Formula(tensor=[4.4328589e-03 1.8856935e-04 3.9932053e-03 2.6313865e
 9.7307027e-05 9.2917297e-04 1.2955565e-03 5.4630609e-03 5.0503397e-03],
```

In [7]:

```
# Notice the dimensions of the outcome: the result is evaluated for every x and y.
# Notice also that y did not appear in the 1st argument of `Or`;
# the connective broadcasts the results of its two arguments to match.
Or(Eq([x,c1]), Eq([x,y]))
```

Out[7]:

```
ltn.Formula(tensor=[[0.52351284 0.5660489  0.502086   0.54841334 0.8179424
 [0.12897979 0.279613   0.09810974 0.3607202  0.25842512]
 [0.45820022 0.35860953 0.32815725 0.4508675  0.6152009 ]
 [0.53618777 0.1908272  0.17470731 0.29135692 0.3339786 ]
 [0.25108603 0.342749   0.20701537 0.41542915 0.50837725]
 [0.11609086 0.11124759 0.0567009  0.7048274  0.15309101]
 [0.2687245  0.30622828 0.20789589 0.47038752 0.51323897]
 [0.3076315  0.3449486  0.25006413 0.45706868 0.59890985]
 [0.49300438 0.39925045 0.37364164 0.46637693 0.64494586]
 [0.28069535 0.30437022 0.3101134  0.28034654 0.38135415]], free_vars=['x'
```

Quantifiers

LTN suppports universal and existential quantification. They are grounded using aggregation operators. We recommend using the following two operators:

- existential quantification ("exists"):

  the generalized mean (`pMean`) $\mathrm{pM}(u_1,\ldots,u_n) = \left(\frac{1}{n}\sum\limits_{i=1}^{n} u_i^p\right)^{\frac{1}{p}} \qquad p \geq 1,$

- universal quantification ("for all"):
  the generalized mean of "the deviations w.r.t. the truth" (`pMeanError`)

  $$\mathrm{pME}(u_1,\ldots,u_n) = 1 - \left(\frac{1}{n}\sum\limits_{i=1}^{n}(1-u_i)^p\right)^{\frac{1}{p}} \qquad p \geq 1,$$

where $u_1,\ldots,u_n$ is a list of truth values in $[0,1]$.

In [8]:

```
Forall = ltn.Wrapper_Quantifier(ltn.fuzzy_ops.Aggreg_pMeanError(p=2),semantics="forall
Exists = ltn.Wrapper_Quantifier(ltn.fuzzy_ops.Aggreg_pMean(p=5),semantics="exists")
```

The wrapper `ltn.Wrapper_Quantifier` allows to use the aggregators with LTN formulas. It takes care of selecting the tensor dimensions to aggregate, given some variables in arguments.

In [9]:

```
x = ltn.Variable('x',np.random.normal(0.,1.,(10,2))) # 10 values in R²
y = ltn.Variable('y',np.random.normal(0.,2.,(5,2))) # 5 values in R²

Eq = ltn.Predicate.Lambda(lambda args: tf.exp(-tf.norm(args[0]-args[1],axis=1))) # pre

Eq([x,y])
```

Out[9]:

```
ltn.Formula(tensor=[[0.39133054 0.19750305 0.07793675 0.03600257 0.3931468
 [0.32958886 0.1688326  0.0260815  0.0207574  0.07129584]
 [0.1663024  0.19604874 0.01294116 0.02727364 0.04528237]
 [0.4060248  0.32324475 0.04142912 0.0436288  0.19317953]
 [0.33524016 0.09340193 0.03288566 0.0114226  0.05844909]
 [0.21770334 0.41895252 0.01864527 0.05260331 0.08970136]
 [0.07839028 0.59526646 0.009564   0.18803924 0.07536539]
 [0.05541972 0.27946767 0.00499568 0.09165297 0.03131543]
 [0.09379627 0.3161083  0.00786038 0.06133056 0.04147319]
 [0.20024009 0.5956605  0.02581004 0.08952475 0.17981495]], free_vars=['x'
```

In [10]:

```
Forall(x,Eq([x,y]))
```

Out[10]:

```
ltn.Formula(tensor=[0.21741337 0.29905307 0.02559453 0.06093419 0.111555
```

In [11]:

```
Forall((x,y),Eq([x,y]))
```

Out[11]:

```
ltn.Formula(tensor=0.1369343400001526, free_vars=[])
```

In [12]:

```
Exists((x,y),Eq([x,y]))
```

Out[12]:

```
ltn.Formula(tensor=0.3350968658924103, free_vars=[])
```

In [13]:

```
Forall(x, Exists(y, Eq([x,y])))
```

Out[13]:

```
ltn.Formula(tensor=0.28244274854660034, free_vars=[])
```

`pMean` can be understood as a smooth-maximum that depends on the hyper-paramer p:

- $p \to 1$: the operator tends to `mean`,
- $p \to +\infty$: the operator tends to `max`.

Similarly, `pMeanError` can be understood as a smooth-minimum:

- $p \to 1$: the operator tends to `mean`,
- $p \to +\infty$: the operator tends to `min`.

Therefore, p offers flexibility in writing more or less strict formulas, to account for outliers in the data depending on the application. Note that this can have strong implications for training (see complementary notebook). One can set a default value for p when initializing the operator, or can use different values at each call of the operator.

In [14]:

```
Forall(x,Eq([x,c1]),p=2)
```

Out[14]:

```
ltn.Formula(tensor=0.3222336173057556, free_vars=[])
```

In [15]:

```
Forall(x,Eq([x,c1]),p=10)
```

Out[15]:

```
ltn.Formula(tensor=0.22595995664596558, free_vars=[])
```

In [16]:

```
Exists(x,Eq([x,c1]),p=2)
```

Out[16]:

```
ltn.Formula(tensor=0.40868890285491943, free_vars=[])
```

In [17]:

```
Exists(x,Eq([x,c1]),p=10)
```

Out[17]:

```
ltn.Formula(tensor=0.6606776118278503, free_vars=[])
```

Diagonal Quantification

Given 2 (or more) variables, there are scenarios where one wants to express statements about specific pairs (or tuples) only, such that the i-th tuple contains the i-th instances of the variables. We allow this using `ltn.diag`. **Note**: diagonal quantification assumes that the variables have the same number of individuals.

In simplified pseudo-code, the usual quantification would compute:

```
for x_i in x:
    for y_j in y:
        results.append(P(x_i,y_j))
aggregate(results)
```

In contrast, diagonal quantification would compute:

```
    for x_i, y_i in zip(x,y):
        results.append(P(x_i,y_i))
    aggregate(results)
```

We illustrate `ltn.diag` in the following setting:

- the variable x denotes 100 individuals in $\mathbb{R}^{2\times 2}$,
- the variable l denotes 100 one-hot labels in $\mathbb{N}^3$ (3 possible classes),
- l is grounded according to x such that each pair (x_i, l_i), for $i = 0..100$ denotes one correct example from the dataset,
- the classifier $C(x, l)$ gives a confidence value in $[0, 1]$ of the sample x corresponding to the label l.

In [18]:

```
# The values are generated at random, for the sake of illustration.
# In a real scenario, they would come from a dataset.
samples = np.random.rand(100,2,2) # 100 R^{2x2} values
labels = np.random.randint(3, size=100) # 100 labels (class 0/1/2) that correspond to
onehot_labels = tf.one_hot(labels,depth=3)

x = ltn.Variable("x",samples)
l = ltn.Variable("l",onehot_labels)

class ModelC(tf.keras.Model):
    def __init__(self):
        super(ModelC, self).__init__()
        self.flatten = tf.keras.layers.Flatten()
        self.dense1 = tf.keras.layers.Dense(5, activation=tf.nn.elu)
        self.dense2 = tf.keras.layers.Dense(3, activation=tf.nn.softmax)
    def call(self, inputs):
        x, l = inputs[0], inputs[1]
        x = self.flatten(x)
        x = self.dense1(x)
        x = self.dense2(x)
        return tf.math.reduce_sum(x*l,axis=1)

C = ltn.Predicate(ModelC())
```

If some variables are marked using `ltn.diag`, LTN will only compute their "zipped" results (instead of the usual "broadcasting").

In [19]:

```
print(C([x,l]).tensor.shape) # Computes the 100x100 combinations
ltn.diag(x,l) # sets the diag behavior for x and l
print(C([x,l]).tensor.shape) # Computes the 100 zipped combinations
ltn.undiag(x,l) # resets the normal behavior
print(C([x,l]).tensor.shape) # Computes the 100x100 combinations
```

```
(100, 100)
(100,)
(100, 100)
```

In practice, `ltn.diag` is designed to be used with quantifiers. Every quantifier automatically calls `ltn.undiag` after the aggregation is performed, so that the variables keep their normal behavior outside of the formula. Therefore, it is recommended to use `ltn.diag` only in quantified formulas as follows.

In [20]:

```
Forall(ltn.diag(x,l), C([x,l])) # Aggregates only on the 100 "zipped" pairs.
                                # Automatically calls `ltn.undiag` so the behavior of
```

Out[20]:

```
ltn.Formula(tensor=0.32587695121765137, free_vars=[])
```

Guarded Quantifiers

One may wish to quantify over a set of elements whose grounding satisfy some **boolean** condition. Let us assume x is a variable from some domain and m is a mask function that returns boolean values (that is, 0 or 1, not continuous truth degrees in $[0,1]$) for each element of the domain.
In guarded quantification, one has quantifications of the form

$$(\forall x : m(x))\phi(x)$$

which means "every x satisfying $m(x)$ also satisfies $\phi(x)$", or

$$(\exists x : m(x))\phi(x)$$

which means "some x satisfying $m(x)$ also satisfies $\phi(x)$".

The mask m can also depend on other variables in the formula. For instance, the quantification $\exists y (\forall x : m(x,y))\phi(x,y)$ is also a valid sentence.

Let us consider the following example, that states that there exists an euclidian distance d below which all pairs of points x, y should be considered as similar: $\exists d\ \forall x, y : \mathrm{dist}(x,y) < d\ (\mathrm{Eq}(x,y)))$

In [22]:

```
Eq = ltn.Predicate.Lambda(lambda args: tf.exp(-tf.norm(args[0]-args[1],axis=1))) # pre

points = np.random.rand(50,2) # 50 values in [0,1]^2
x = ltn.Variable("x",points)
y = ltn.Variable("y",points)
d = ltn.Variable("d",[.1,.2,.3,.4,.5,.6,.7,.8,.9])
```

In [23]:

```
is_greater_than = ltn.Predicate.Lambda(lambda inputs: inputs[0] > inputs[1]) # boolean
eucl_dist = ltn.Function.Lambda(
      lambda inputs: tf.expand_dims(tf.norm(inputs[0]-inputs[1],axis=1),axis=1)
) # function measuring euclidian distance



Exists(d,
      Forall([x,y],
            Eq([x,y]),
            mask = is_greater_than([d, eucl_dist([x,y])])
      )
)
```

Out[23]:

```
ltn.Formula(tensor=0.7828403115272522, free_vars=[])
```

The guarded option is particularly useful to propagate gradients (see notebook on learning) over only a subset of the domains, that verifies the condition m.