Implementations of Query Reformulator based on Tensorforce

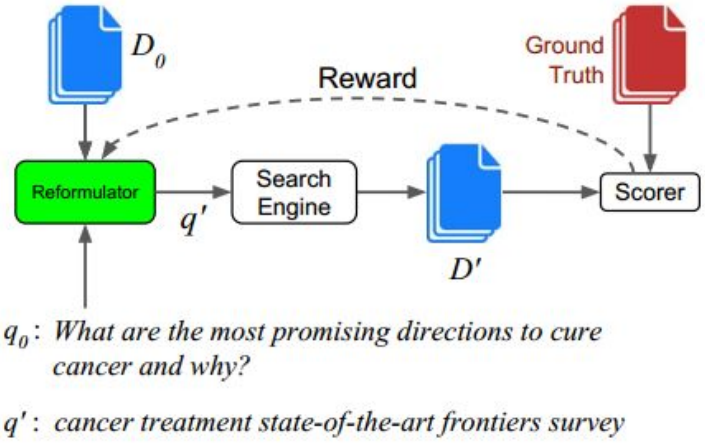
by Yifan Tian, Sixi Lu, Shailender Singh@UIUC

Overview

In this project we demonstrate reinforcement learning by performing Query Reformulation (QR)[1] task on an open source reinforcement learning library, Tensorforce[2]. QR task is to create a system that reformulates a query to maximize the number of relevant documents returned. We extend the existing source code of reformulation and introduce tensorforce library to exploit the environment and agent interaction.

1. Query Reformulator[1]:

[1] introduces a query reformulation system which is based on a neural network that rewrites a query to maximize the number of relevant documents returned.



The basic process is described as below.

For an initial query q0, a set of documents D0 is retrieved from a search engine. Then the reformulator will select certain terms from q0 and D0 ,and produce a reformulated query q' which is then sent to the search engine. Documents D' are returned, and a reward is computed against the set of ground-truth documents. The reformulator is trained with a deep reinforcement learning system to produce a query, or a series of queries, to maximize the expected return.

2. Tensorforce [2]

TensorForce is an open source reinforcement learning library built on top of TensorFlow, which has four core components.

Environment <-> Runner <-> Agent <-> Model

(1) A reinforcement learning environment provides the API to a simulated or real environment as the subject for optimization. It could be anything from video games (e.g. Atari) to robots or trading systems. The agent interacts with this environment and learns to act optimally in its dynamics.

(2) A reinforcement learning agent provides methods to process states and return actions, to store past observations, and to load and save models. Most agents employ a Model which implements the algorithms to calculate the next action given the current state and to update model parameters from past experiences.

(3) A "runner" manages the interaction between the Environment and the Agent.

Implementation 1:

[placeholder for the solution provided by yifan tan]

Latest code can be found at https://github.com/YifanTian/tensorforce_QR

Implementation 2:

Step by Step Implementation of a RL based Query Reformulation Application (by sixi lu)

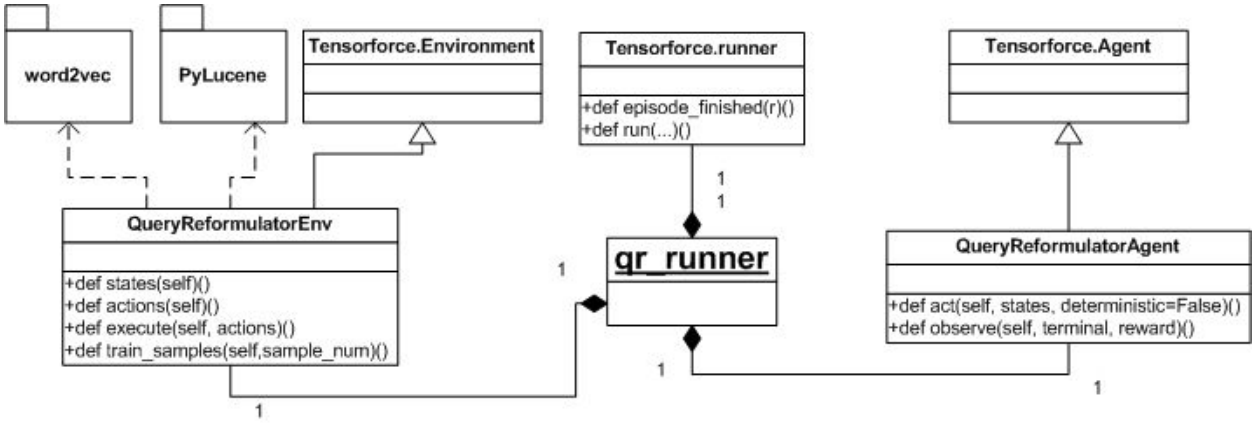
Step 1: build a customized RL application using tensorforce

a) Implementation details:

Definition of our environment, agent, actions.

An initial query is the natural language expression of the desired goal, and an agent (i.e. reformulator) learns to reformulate an initial query to maximize the expected return (we are using Recall to evaluate the agent's performance) through actions (i.e. selecting terms for a new query). The environment is a search engine which produces a new state (i.e. retrieved documents).

Classes Hierarchy



(1) qr_runner is the whole training process's start engine and it manages the interaction between our own implemented environment(QueryReformulatorEnv) and agent(QueryReformulatorAgent). It first calls the environment's reset() function to get an initial state, then it transfers this state to the agent's act(state) function to get an action(our QueryReformulatorAgent will return a random training sample as an action). Then it will call the env.execute(action), then, the env will return back a new state, reward and a terminal signal. The runner will repeat such process for certain episode times.

(2)QueryReformulatorEnv is a subclass of Tensorforce's Environment class. It is a simulated search engine environment. In our project, we use pyLucene as our search engine. It will execute a search query then return the matched document ids with performance metric(such as Recall, F1 and MAP, etc.) which will be used to evaluate our agent's performance.

There are four core functions of our QueryReformulatorEnv.

- States() is to define the state space. In our case, it is the initial searching terms plus the search engine returned expanded terms, and it is an array of term index in the pre-trained dictionary(D_cbow_pdw_8B_norm.pkl).
- Actions()returns the action space. In our case, we need to pick up a predefined number of candidate terms from the expanded term array.
- Execute() is to execute actions and returns the next state, a boolean indicating terminal, and a reward value. We are using Pylucene as our search engine, so we call a wrapper of it to retrieve a set of documents from which some candidate terms are selected as the initial query's expansion, meanwhile, the reward value is calculated based on our ground-truth data.
- train_samples() is used to generate a number of initial queries from a pre-loaded training data set. It is used by agent to simulate user queries.

(3)QueryReformulatorAgent is inherited from Tensorforce's Agent class, it usually process states and return actions. Most agents employ a Model which implements the algorithms to calculate the next action given the current state and to update model parameters from past experiences. In our case, we can use the tensorforce's existing model, such as policy gradient models.

(4) **Network spec in the model**. The original paper's neural network is composed by two CNN, one for initial query while the other is for candidate terms. However, currently Tensorforce does not support other than layer-stack networks to be defined in a JSON format. So we have to define our own network by subclassing Tensorforce's Network. But for qr_runner.py demo, we did not create a model. The purpose of it is to get familiar with tensorforce framework and create a customized RL application.

b) Demo:

https://github.com/bettermanlu/tensorforce4qr/blob/master/qr_runner.py

c) Sample output:

https://github.com/bettermanlu/tensorforce4qr/blob/master/qr_runner.sample.log

Step 2: build a basic RL based query reformulation

a) Implementation details:

In this part, we build Deep CNN network to estimate the probability of selecting candidate terms in the reformulated query. The network is defined using the tensorforce layers:

```
dict(type='conv2d',size=256,window=3,stride=2),
dict(type='pool2d',window=2,stride=2),
dict(type='conv2d',size=256,window=3),
dict(type='pool2d',window=2,stride=2),
dict(type='flatten'),
dict(type='dense', size=256, activation='tanh'),
dict(type='dense', size=256, activation='sigmoid')
```

To process the text information before putting them in the CNN, we need to define an embedding layer, and to extract the reformulated queries from the RL model, we need to translate the action id from the action spaces to vocabulary space and vise versa.

(1) Text embedding:

Tensorforce provide an embedding layer, but to convert the query text to the CNN input, we need define a new layer "Expansion", which is defined in 'Tensorforce/core/networks/layers.py'.

```
dict(type='embedding',indices=374557,size=500),
dict(type='expansion'),
```

(2) Action and State space

Detailed definition of the action and state are in the execute and reset function of qr_env2.py .

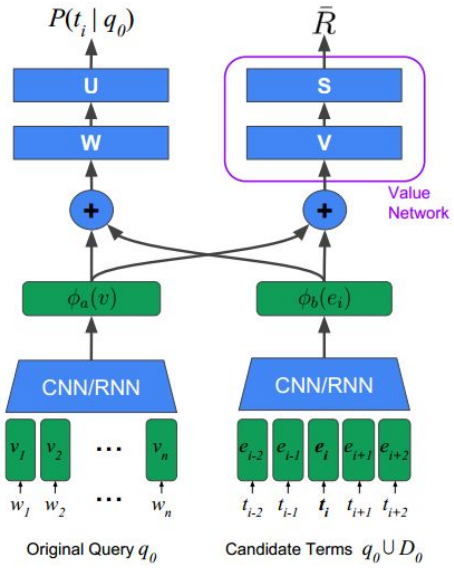
b) Demo:

https://github.com/bettermanlu/tensorforce4qr/blob/master/demo2/qr_demo2.py

c) Sample output:

https://github.com/bettermanlu/tensorforce4qr/blob/master/demo2/sample_output_demo2.txt

Step 3: build a customized network



In this part we tried to implement a deep CNN network with two kinds of inputs: original query and candidate terms, but Tensorforce does not support such kind of network definition at present, so we need to define a customized network using tensorflow.

The customized network is defined in qr_network.py (https://github.com/bettermanlu/tensorforce4qr/blob/master/demo3/qr_network.py), which contains a 4 layer cnn network for the query and two 4 layer cnn networks. Note that the customized network contains many layers and parameters, the parameters of each layer can be set in the config.yml

In the embedding layer, we build a new function instead of using the tensorforce's embedding layer. This allows us to use a pre-trained word2vec embeddings rather than learning the embedding by our model, which can make the RL algorithm converge faster.

Step 4: SL-CNN

In previous steps, we tried to learn a query reformulation agent using the reinforce algorithm. In this step, we add a minimized least square term in the loss function, following the definition in [1].

a) Implementation Details

In qr_cnn.py we implement a customized network using tensorflow, which has the same architecture with step3. We build the same loss function as the loss function in [1], and train the network using supervised learning, rather than using the tensorforce framework. This part is trying to re-implement the SL-CNN benchmark of the query reformulation.

Implementations: train_qr.py, qr_cnn.py

b)demo

https://github.com/bettermanlu/tensorforce4qr/blob/master/demo4/qr_cnn.py

https://github.com/bettermanlu/tensorforce4qr/blob/master/demo4/train_qr.py

c)sample output

https://github.com/bettermanlu/tensorforce4qr/blob/master/demo4/sample_output_train_qr.txt

Usage

Implementation 1's usage and video presentation can be found at here:

https://docs.google.com/presentation/d/1s2rOOunRtmTvLj9c419IiDXXVzXONYZacv336vR9ZzI/edit#slide=id.p3

https://mediaspace.illinois.edu/media/t/1_edr4fpp3

Implementation 2's latest code and usage instructions can be found at:
https://github.com/bettermanlu/tensorforce4qr

References

[1] Task-Oriented Query Reformulation with Reinforcement Learning. Rodrigo Nogueira and Kyunghyun Cho.

[2] tensorforce: https://github.com/reinforceio/tensorforce/

[3] pylucene: http://lucene.apache.org/pylucene/