The problem we are trying to solve is to get the reinforcement learning package RL4J (contained in the DL4J repository) to interface with environments created in AnyLogic. RL4J implements several canonical reinforcement learning algorithms from the machine learning literature and can in principle be applied to any task as long as this task can be brought into a format which the algorithm will understand. For instance, we can find examples of it being applied to classical problems from the machine learning literature like Cartpole which we are working on but also more sophisticated tasks like learning to play Atari games. A brief description of the Cartpole task is provided here: `https://gym.openai.com/envs/CartPole-v0/`.

We are leaving the the RL4J package completely unchanged and are simply replacing the environments from the examples contained in the repository (which are taken from something called OpenAI and converted from Python to Java) with example environments created in AnyLogic.

The sequence of events is as follows (using the Cartpole as an example):

We create an instance of the MDP ("Markov decision process")-agent we created in AnyLogic. This is an exact reproduction of the Cartpole problem with the same state-variables, available actions, parameters etc. See the AnyLogic model for details. The state variables of the MDP environment are initialised and its general characteristics (number of available actions, dimensions and bounds of the state-space) are fed into RL4J along with a set of parameters for the learning algorithm and the desired configuration of the neural nets we want to train (both explained in the model file). RL4J uses this input to define the training task, i.e. it creates the necessary neural nets and defines the learning algorithm. The training then begins.

In the beginning, the learning algorithm instructs the MDP (the Cartpole which lives inside AnyLogic) to take *fully random* actions for a number of periods in order to build up a buffer of experience. The results of these actions (the reward, the new value of the state variables, whether or not the pole fell over) are stored for later updates of the neural net. After the initial phase of random action, the algorithm begins properly. It receives an observation from the Cartpole (i.e. its position, velocity, and the angle of the pole). This information is fed into the current neural net which in turn calculates the expected value of taking the one or the other action given the situation.

The decision as to which action will be performed is then taken by a so-called ε-greedy policy. This means that with a certain probability ε, the action taken will be random and with probability $1 - \varepsilon$, the action with the maximum expected value (in terms of expected future rewards) is taken. This is intended to encourage a certain degree of exploration to avoid algorithms to get stuck in a local maximum. Initially the value of ε is relatively high but then declines over time as the neural net becomes better at the task. In either case, the ε-greedy policy produces a recommended action given the current state which is in turn fed back to the Cartpole inside AnyLogic.

The instructed action is performed by the AnyLogic Cartpole, giving rise to a reward, a new state (since both the cart and the pole will have moved) and a boolean variable indicating whether or not the pole fell. *This entire experience is saved* for use in updating the network and the new values of the state variables are fed back to the neural net in order to determine the next action. If the pole has fallen over, all state variables are reset and a new episode/epoch begins.

As soon as a sufficient number of experiences have been recorded, we begin to update the neural net. Using a random batch of experiences (i.e. combinations of prior state, action, reward, posterior state) along with outputs from the target network (see below), the current network is updated using stochastic gradient descent. As soon as the buffer of experience is large enough, this updating is performed in every period, i.e. after every single step, always using a random sample of past experience.

While the current network is hence updated every step, we also have a second network called the *target network*. This network has the same configuration as the current network but is updated much less frequently (hence it is more stable). In our case, the current network is copied every 500 steps to become the new target network. The target network is used in the updating of the current network. According to the literature, using output from the (more stable) target network in updating the current network can help in ensuring convergence of the learning algorithm. The training runs for a pre-defined number of steps (periods) by the end of which the network has hopefully learned the ideal action to take given any state, i.e. it has learned to balance the pole. The learned policy is then saved.

Our problem is that the entire learning process takes place inside a single function called train() which can also be found in the model file. As soon as this function is called the AnyLogic simulation time stops, meaning that while state variables etc. are still updated (and in fact the learning algorithm seems to converge), the simulation is effectively no longer running and steps in the Cartpole environment are not rendered. Ideally we would like to get to a stage where, after each step of training, the AnyLogic simulation is incremented by one time-step before the next step of training is performed. For this purpose we have been experimenting with creating custom simulations but so far we have not been successful.