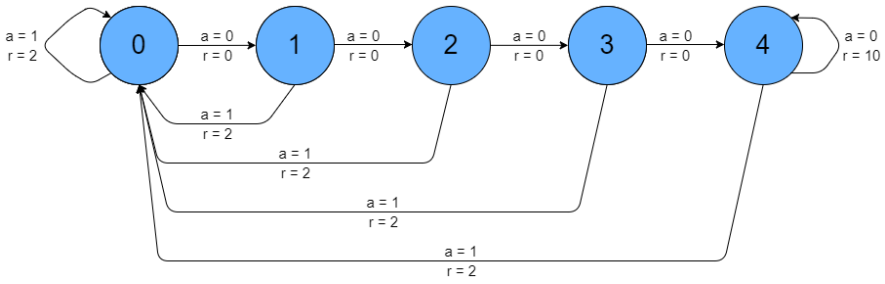
Reinforcement learning tutorial using Python and

By admin | Keras

Mar
03

18



In this post, I'm going to introduce the concept of reinforcement learning, and show you how to build an autonomous agent that can successfully play a simple game. Reinforcement learning is an active and interesting area of machine learning research, and has been spurred on by recent successes such as the AlphaGo system, which has convincingly beat the best human players in the world. This occurred in a game that was thought too difficult for machines to learn. In this tutorial, I'll first detail some background theory while dealing with a toy game in the Open AI Gym toolkit. We'll then create a Q table of this game using simple Python, and then create a Q network using Keras. If you'd like to scrub up on Keras, check out my introductory Keras tutorial. All code present in this tutorial is available on this site's Github page.

Reinforcement learning – the basics

Reinforcement learning can be considered the third genre of the machine learning triad – unsupervised learning, supervised learning and reinforcement learning. In supervised learning, we supply the machine learning system with curated (x, y) training pairs, where the intention is for the network to learn to map x to y. In reinforcement learning, we create

Search...

RECENT POSTS

Transfer learning in TensorFlow 2

An introduction to Global Averag in convolutional neural networks

Metrics and summaries in Tenso

An introduction to entropy, cross and KL divergence in machine lea

Google Colaboratory introductior how to build deep learning syster Google Colaboratory

RECENT COMMENTS

Andry on Neural Networks Tu A Pathway to Deep Learning

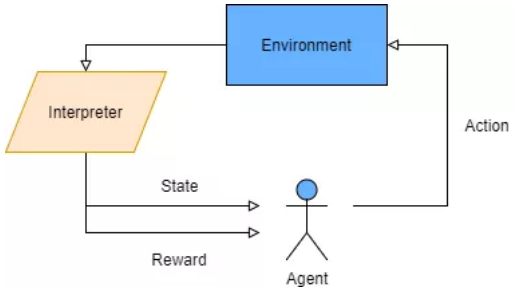
Sandipan on Keras LSTM tut How to easily build a powerfu learning language model

Andy on Neural Networks Tu Pathway to Deep Learning

Martin on Neural Networks Tu A Pathway to Deep Learning

uri on The vanishing gradient and ReLUs – a TensorFlow investigation

an *agent* which performs *actions* in an *environment* and the agent receives various *rewards* depending on what *state* it is in when it performs the action. In other words, an agent explores a kind of game, and it is trained by trying to maximize rewards in this game. This cycle is illustrated in the figure below:



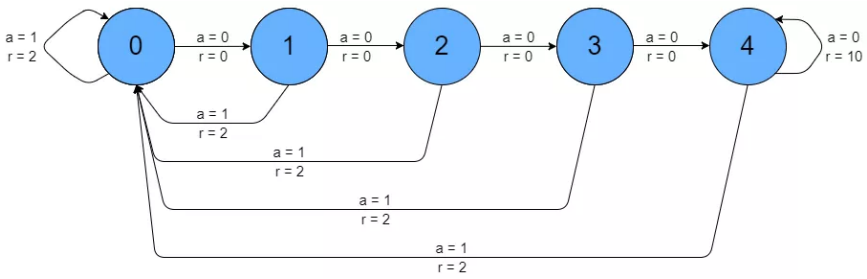
Reinforcement learning environment

As can be observed above, the agent performs some action in the environment. An interpreter views this action in the environment, and feeds back an updated state that the agent now resides in, and also the reward for taking this action. The environment is not known by the agent beforehand, but rather it is discovered by the agent taking incremental steps in time. So, for instance, at time *t* the agent, in state s_t, may take action *a*. This results in a new state s_{t+1} and a reward *r*. This reward can be a positive real number, zero, or a negative real number. It is the goal of the agent to learn which *state dependent action* to take which maximizes its rewards. The way which the agent optimally learns is the subject of reinforcement learning theory and methodologies.

To more meaningfully examine the theory and possible approaches behind reinforcement learning, it is useful to have a simple example in which to work through. This simple example will come from an environment available on Open AI Gym called NChain.

Open AI Gym example

The NChain example on Open AI Gym is a simple 5 state environment. There are two possible actions in each state, move forward (action 0) and move backwards (action 1). When action 1 is taken, i.e. move backwards, there is an immediate reward of 2 given to the agent – and the agent is returned to state 0 (back to the beginning of the chain). However, when a move forward action is taken (action 0), there is no immediate reward until state 4. When the agent moves forward while in state 4, a reward of 10 is received by the agent. The agent stays in state 4 at this point also, so the reward can be repeated. There is also a random chance that the agent's action is "flipped" by the environment (i.e. an action 0 is flipped to an action 1 and vice versa). The diagram below demonstrates this environment:



Open AI Gym's NChain environment

You can play around with this environment by first installing the Open AI Gym Python package – see instructions here. Then simply open up your Python command prompt and

have a play – see the figure below for an example of some of the commands available:

```
>>> import gym
>>> env = gym.make('NChain-v0')
>>> env.reset()
0
>>> env.step(1)
(0, 2, False, {})
>>> env.step(0)
(1, 0, False, {})
>>> env.step(0)
(0, 2, False, {})
>>> env.step(0)
(1, 0, False, {})
>>> env.step(0)
(2, 0, False, {})
>>> env.step(0)
(3, 0, False, {})
>>> env.step(0)
(4, 0, False, {})
>>> env.step(0)
(4, 10, False, {})
>>> env.step(0)
(4, 10, False, {})
>>> env.step(0)
(4, 10, False, {})
```

NChain Python playaround

If you examine the code above, you can observe that first the Python module is imported, and then the environment is loaded via the gym.make() command. The first step is to initalize / reset the environment by running env.reset() – this command returns the initial state of the environment – in this case 0. The first command I then run is env.step(1) – the value in the bracket is the action ID. As explained previously, action 1 represents a step back to the beginning of the chain (state 0). The step() command returns 4 variables in a tuple, these are (in order):

- The new state after the action
- The reward due to the action
- Whether the game is "done" or not – the NChain game is done after 1,000 steps
- Debugging information – not relevant in this example

As can be observed, starting in state 0 and taking step(1) action, the agent stays in state 0 and gets 2 for its reward. Next, I sent a series of action 0 commands. After every action 0 command, we would expect the progression of the agent along the chain, with the state increasing in increments (i.e. 0 -> 1 -> 2 etc.). However, you'll observe after the first step(0) command, that the agent stays in state 0 and gets a 2 reward. This is because of the random tendency of the environment to "flip" the action occasionally, so the agent *actually* performed a 1 action. This is just unlucky.

Nevertheless, I persevere and it can be observed that the state increments as expected, but there is no immediate reward for doing so for the agent until it reaches state 4. When in state 4, an action of 0 will keep the agent in step 4 *and* give the agent a 10 reward. Not only that, the environment allows this to be done repeatedly, as long as it doesn't produce an unlucky "flip", which would send the agent back to state 0 – the beginning of the chain.

Now that we understand the environment that will be used in this tutorial, it is time to consider what method can be used to train the agent.

A first naive heuristic for reinforcement learning

Keras
Loss functions
LSTMs
Metrics and summaries
Neural networks
NLP
Optimisation
PyTorch
Recurrent neural networks
Reinforcement learning
TensorBoard
TensorFlow
TensorFlow 2.0
Transfer learning
Weight initialization
Word2Vec

In order to train the agent effectively, we need to find a good *policy* π which maps states to actions in an optimal way to maximize reward. There are various ways of going about finding a good or optimal policy, but first, let's consider a naive approach.

Let's conceptualize a table, and call it a reward table, which looks like this:

$$\begin{bmatrix} r_{s_0,a_0} & r_{s_0,a_1} \\ r_{s_1,a_0} & r_{s_1,a_1} \\ r_{s_2,a_0} & r_{s_2,a_1} \\ r_{s_3,a_0} & r_{s_3,a_1} \\ r_{s_4,a_0} & r_{s_4,a_1} \end{bmatrix}$$

Each of the rows corresponds to the 5 available states in the NChain environment, and each column corresponds to the 2 available actions in each state – forward and backward, 0 and 1. The value in each of these table cells corresponds to some measure of reward that the agent has "learnt" occurs when they are in that state and perform that action. So, the value r_{s_0,a_0} would be, say, the sum of the rewards that the agent has received when in the past they have been in state 0 and taken action 0. This table would then let the agent choose between actions based on the summated (or average, median etc. – take your pick) amount of reward the agent has received in the past when taking actions 0 or 1.

This might be a good policy – choose the action resulting in the greatest previous summated reward. Let's give it a try, the code looks like:

```
def naive_sum_reward_agent(env, num_episodes=500):
    # this is the table that will hold our summated rewards for
    # each action in each state
    r_table = np.zeros((5, 2))
    for g in range(num_episodes):
        s = env.reset()
        done = False
        while not done:
            if np.sum(r_table[s, :]) == 0:
                # make a random selection of actions
                a = np.random.randint(0, 2)
            else:
                # select the action with highest cummulative reward
                a = np.argmax(r_table[s, :])
            new_s, r, done, _ = env.step(a)
            r_table[s, a] += r
            s = new_s
    return r_table
```

In the function definition, the environment is passed as the first argument, then the number of episodes (or number of games) that we will train the r_table on. We first create the r_table matrix which I presented previously and which will hold our summated rewards for each state and action. Then there is an outer loop which cycles through the number of episodes. The env.reset() command starts the game afresh each time a new episode is commenced. It also returns the starting state of the game, which is stored in the variable *s*.

The second, inner loop continues until a "done" signal is returned after an action is passed to the environment. The *if* statement on the first line of the inner loop checks to see if there are any existing values in the r_table for the current state – it does this by confirming

if the sum across the row is equal to 0. If it is zero, then an action is chosen at random – there is no better information available at this stage to judge which action to take.

This condition will only last for a short period of time. After this point, there will be a value stored in at least one of the actions for each state, and the action will be chosen based on which column value is the largest for the row state *s*. In the code, this choice of the maximum column is executed by the numpy *argmax* function – this function returns the index of the vector / matrix with the highest value. For example, if the agent is in state 0 and we have the r_table with values [100, 1000] for the first row, action 1 will be selected as the index with the highest value is column 1.

After the action has been selected and stored in *a*, this action is fed into the environment with env.step(a). This command returns the new state, the reward for this action, whether the game is "done" at this stage and the debugging information that we are not interested in. In the next line, the r_table cell corresponding to state *s* and action *a* is updated by adding the reward to whatever is already existing in the table cell.

Finally the state *s* is updated to *new_s* – the new state of the agent.

If we run this function, the r_table will look something like:

```
[[     0. 639006.]
 [     0. 129034.]
 [     0.  25418.]
 [     0.   4944.]
 [     0.   2762.]]
```

Examining the results above, you can observe that the most common state for the agent to be in is the first state, seeing as any action 1 will bring the agent back to this point. The least occupied state is state 4, as it is difficult for the agent to progress from state 0 to 4 without the action being "flipped" and the agent being sent back to state 0. You can get different results if you run the function multiple times, and this is because of the stochastic nature of both the environment and the algorithm.

Clearly – something is wrong with this table. One would expect that in state 4, the most rewarding action for the agent would be to choose action 0, which would reward the agent with 10 points, instead of the usual 2 points for an action of 1. Not only that, but it has chosen action 0 for *all* states – this goes against intuition – surely it would be best to sometimes shoot for state 4 by choosing multiple action 0's in a row, and that way reap the reward of multiple possible 10 scores.

In fact, there are a number of issues with this way of doing reinforcement learning:

- First, once there is a reward stored in one of the columns, the agent will always choose that action from that point on. This will lead to the table being "locked in" with respect to actions after just a few steps in the game.
- Second, because no reward is obtained for most of the states when action 0 is picked, this model for training the agent has no way to encourage acting on *delayed reward* signal when it is appropriate for it to do so.

Let's see how these problems could be fixed.

Delayed reward reinforcement learning

If you want to be a medical doctor, you're going to have to go through some pain to get there. You'll be studying a long time before you're free to practice on your own, and the rewards will be low while you are doing so. However, once you get to be a fully fledged MD, the rewards will be great. During your time studying, you would be operating under a delayed reward or delayed gratification paradigm in order to reach that greater reward. However, you might only be willing to undertake that period of delayed reward for a given period of time – you wouldn't want to be studying forever, or at least, for decades.

We can bring these concepts into our understanding of reinforcement learning. Let's say we are in state 3 – in the previous case, when the agent chose action 0 to get to state 3, the reward was zero and therefore r_table[3, 0] = 0. Obviously the agent would not see this as an attractive step compared to the alternative for this state i.e. r_table[3, 1] >= 2. But what if we assigned to this state the reward the agent would received if it chose action 0 in state 4? It would look like this: r_table[3, 0] = r + 10 = 10 – a much more attractive alternative!

This idea of propagating possible reward from the best possible actions in future states is a core component of what is called *Q learning*. In Q learning, the Q value for each action in each state is updated when the relevant information is made available. The Q learning rule is:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

First, as you can observe, this is an *updating* rule – the existing Q value is added to, not replaced. Ignoring the α for the moment, we can concentrate on what's inside the brackets. The first term, *r,* is the reward that was obtained when action *a* was taken in state *s*. Next, we have an expression which is a bit more complicated. Ignore the γ for the moment and focus on $\max_{a'} Q(s',a')$. What this means is that we look at the next state *s'* after action *a* and return the maximum possible Q value in the next state. In other words, return the maximum Q value for the best possible action in the next state. In this way, the agent is looking forward to determine the best possible future rewards before making the next step *a*.

The γ value is called the *discounting* factor – this decreases the impact of future rewards on the immediate decision making in state *s*. This is important, as this represents a limited patience in the agent – it won't study forever to get that medical degree. So γ will always be less than 1. The – Q(s, a) term acts to restrict the growth of the Q value as the training of the agent progresses through many iterations. Finally, this whole sum is multiplied by a *learning rate* α which restricts the updating to ensure it doesn't "race" to a solution – this is important for optimal convergence (see my neural networks tutorial for more on learning rate).

Note that while the learning rule only examines the best action in the following state, in reality, discounted rewards still cascade down from future states. For instance, if we think of the cascading rewards from all the 0 actions (i.e. moving forward along the chain) and start at state 3, the Q reward will be $r + \gamma \max_a Q(s',a') = 0 + 0.95 * 10 = 9.5$ (with a γ = 0.95). If we work back from state 3 to state 2 it will be 0 + 0.95 * 9.5 = 9.025. Likewise, the cascaded, discounted reward from to state 1 will be 0 + 0.95 * 9.025 = 8.57, and so on. Therefore, while the immediate updating calculation only looks at the maximum Q value

for the next state, "upstream" rewards that have previously been discovered by the agent still cascade down into the present state and action decision. This is a simplification, due to the learning rate and random events in the environment, but represents the general idea.

Now that you (hopefully) understand Q learning, let's see what it looks like in practice:

```
def q_learning_with_table(env, num_episodes=500):
    q_table = np.zeros((5, 2))
    y = 0.95
    lr = 0.8
    for i in range(num_episodes):
        s = env.reset()
        done = False
        while not done:
            if np.sum(q_table[s,:]) == 0:
                # make a random selection of actions
                a = np.random.randint(0, 2)
            else:
                # select the action with largest q value in state s
                a = np.argmax(q_table[s, :])
            new_s, r, done, _ = env.step(a)
            q_table[s, a] += r + lr*(y*np.max(q_table[new_s, :]) - q_table
            s = new_s
    return q_table
```

This function is almost exactly the same as the previous naive r_table function that was discussed. The additions and changes are:

- The variables *y* which specifies the discounting factor γ and *lr* which is the Q table updating learning rate
- The line:

```
q_table[s, a] += r + lr*(y*np.max(q_table[new_s, :]) - q_table[s, a])
```

This line executes the Q learning rule that was presented previously. The np.max(q_table[new_s, :]) is an easy way of selecting the maximum value in the q_table for the row new_s. After this function is run, an example q_table output is:

```
[[ 0.          29.66281604]
 [ 0.          29.84359424]
 [ 0.          27.66807031]
 [28.95331666  0.        ]
 [ 0.          31.20152823]]
```

This output is strange, isn't it? Again, we would expect at least the state 4 – action 0 combination to have the highest Q score, but it doesn't. We might also expect the reward from this action in this state to have cascaded down through the states 0 to 3. Something has clearly gone wrong – and the answer is that there isn't enough *exploration* going on within the agent training method.

Q learning with ϵ-greedy action selection

If we think about the previous iteration of the agent training model using Q learning, the action selection policy is based solely on the maximum Q value in any given state. It is conceivable that, given the random nature of the environment, that the agent initially makes "bad" decisions. The Q values arising from these decisions may easily be "locked in" – and from that time forward, bad decisions may continue to be made by the agent because it can only ever select the maximum Q value in any given state, even if these values are not necessarily optimal. This action selection policy is called a *greedy* policy.

So we need a way for the agent to eventually always choose the "best" set of actions in the environment, yet at the same time allowing the agent to not get "locked in" and giving it some space to explore alternatives. What is required is the ϵ-greedy policy.

The ϵ-greedy policy in reinforcement learning is basically the same as the *greedy* policy, except that there is a value ϵ (which may be set to decay over time) where, if a random number is selected which is less than this value, an action is chosen completely at random. This step allows some random *exploration* of the value of various actions in various states, and can be scaled back over time to allow the algorithm to concentrate more on *exploiting* the best strategies that it has found. This mechanism can be expressed in code as:

```
def eps_greedy_q_learning_with_table(env, num_episodes=500):
    q_table = np.zeros((5, 2))
    y = 0.95
    eps = 0.5
    lr = 0.8
    decay_factor = 0.999
    for i in range(num_episodes):
        s = env.reset()
        eps *= decay_factor
        done = False
        while not done:
            # select the action with highest cummulative reward
            if np.random.random() < eps or np.sum(q_table[s, :]) == 0:
                a = np.random.randint(0, 2)
            else:
                a = np.argmax(q_table[s, :])
            # pdb.set_trace()
            new_s, r, done, _ = env.step(a)
            q_table[s, a] += r + lr * (y * np.max(q_table[new_s, :]) - q_t
            s = new_s
    return q_table
```

This code shows the introduction of the ϵ value – *eps*. There is also an associated *eps* decay_factor which exponentially decays eps with each episode *eps *= decay_factor*. The ϵ-greedy based action selection can be found in this code:

```
if np.random.random() < eps or np.sum(q_table[s, :]) == 0:
    a = np.random.randint(0, 2)
```

```
    else:
        a = np.argmax(q_table[s, :])
```

The first component of the *if* statement shows a random number being selected, between 0 and 1, and determining if this is below *eps*. If so, the action will be selected randomly from the two possible actions in each state. The second part of the *if* statement is a random selection if there are no values stored in the q_table so far. If neither of these conditions hold true, the action is selected as per normal by taking the action with the highest q value.

The rest of the code is the same as the standard *greedy* implementation with Q learning discussed previously. This code produces a q_table which looks something like the following:

```
[[41.20824652 35.96930955]
 [43.55900598 37.46930009]
 [41.56353501 42.67025479]
 [41.61145508 37.83951913]
 [59.73051977 42.22259272]]
```

Finally we have a table which favors action 0 in state 4 – in other words what we would expect to happen given the reward of 10 that is up for grabs via that action in that state. Notice also that, as opposed to the previous tables from the other methods, that there are no actions with a 0 Q value – this is because the full action space has been explored via the randomness introduced by the ϵ-greedy policy.

Comparing the methods

Let's see if the last agent training model actually produces an agent that gathers the most rewards in any given game. The code below shows the three models trained and then tested over 100 iterations to see which agent performs the best over a test game. The models are trained as well as tested in each iteration because there is significant variability in the environment which messes around with the efficacy of the training – so this is an attempt to understand average performance of the different models. The main testing code looks like:

```
def test_methods(env, num_iterations=100):
    winner = np.zeros((3,))
    for g in range(num_iterations):
        m0_table = naive_sum_reward_agent(env, 500)
        m1_table = q_learning_with_table(env, 500)
        m2_table = eps_greedy_q_learning_with_table(env, 500)
        m0 = run_game(m0_table, env)
        m1 = run_game(m1_table, env)
        m2 = run_game(m2_table, env)
        w = np.argmax(np.array([m0, m1, m2]))
        winner[w] += 1
        print("Game {} of {}".format(g + 1, num_iterations))
    return winner
```

First, this method creates a numpy zeros array of length 3 to hold the results of the winner in each iteration – the winning method is the method that returns the highest rewards after training and playing. The run_game function looks like:

```
def run_game(table, env):
    s = env.reset()
    tot_reward = 0
    done = False
    while not done:
        a = np.argmax(table[s, :])
        s, r, done, _ = env.step(a)
        tot_reward += r
    return tot_reward
```

Here, it can be observed that the trained table given to the function is used for action selection, and the total reward accumulated during the game is returned. A sample outcome from this experiment (i.e. the vector w) is shown below:

[13, 22, 65]

As can be observed, of the 100 experiments the ϵ-greedy, Q learning algorithm (i.e. the third model that was presented) wins 65 of them. This is followed by the standard greedy implementation of Q learning, which won 22 of the experiments. Finally the naive accumulated rewards method only won 13 experiments. So as can be seen, the ϵ-greedy Q learning method is quite an effective way of executing reinforcement learning.

So far, we have been dealing with explicit tables to hold information about the best actions and which actions to choose in any given state. However, while this is perfectly reasonable for a small environment like NChain, the table gets far too large and unwieldy for more complicated environments which have a huge number of states and potential actions.

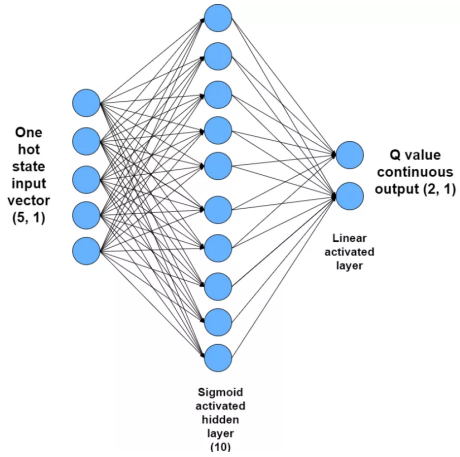
This is where neural networks can be used in reinforcement learning. Instead of having explicit tables, instead we can train a neural network to predict Q values for each action in a given state. This will be demonstrated using Keras in the next section.

Reinforcement learning with Keras

To develop a neural network which can perform Q learning, the input needs to be the current state (plus potentially some other information about the environment) and it needs to output the relevant Q values for each action in that state. The Q values which are output should approach, as training progresses, the values produced in the Q learning updating rule. Therefore, the loss or cost function for the neural network should be:

$$\text{loss} = (\underbrace{r + \gamma \max_{a'} Q'(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}})^2$$

The reinforcement learning architecture that we are going to build in Keras is shown below:



Reinforcement learning Keras architecture

The input to the network is the one-hot encoded state vector. For instance, the vector which corresponds to state 1 is [0, 1, 0, 0, 0] and state 3 is [0, 0, 0, 1, 0]. In this case, a hidden layer of 10 nodes with sigmoid activation will be used. The output layer is a linear activated set of two nodes, corresponding to the two Q values assigned to each state to represent the two possible actions. Linear activation means that the output depends only on the linear summation of the inputs and the weights, with no additional function applied to that summation. For more on neural networks, check out my comprehensive neural network tutorial.

Building this network is easy in Keras – to learn more about how to use Keras, check out my tutorial. The code below shows how it can be done in a few lines:

```
model = Sequential()
model.add(InputLayer(batch_input_shape=(1, 5)))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(2, activation='linear'))
model.compile(loss='mse', optimizer='adam', metrics=['mae'])
```

First, the model is created using the Keras Sequential API. Then an input layer is added which takes inputs corresponding to the one-hot encoded state vectors. Then the sigmoid activated hidden layer with 10 nodes is added, followed by the linear activated output layer which will yield the Q values for each action. Finally the model is compiled using a mean-squared error loss function (to correspond with the loss function defined previously) with the Adam optimizer being used in its default Keras state.

To use this model in the training environment, the following code is run which is similar to the previous ϵ-greedy Q learning methodology with an explicit Q table:

```
# now execute the q learning
y = 0.95
eps = 0.5
decay_factor = 0.999
r_avg_list = []
for i in range(num_episodes):
    s = env.reset()
    eps *= decay_factor
    if i % 100 == 0:
        print("Episode {} of {}".format(i + 1, num_episodes))
    done = False
```

```
    r_sum = 0
    while not done:
        if np.random.random() < eps:
            a = np.random.randint(0, 2)
        else:
            a = np.argmax(model.predict(np.identity(5)[s:s + 1]))
        new_s, r, done, _ = env.step(a)
        target = r + y * np.max(model.predict(np.identity(5)[new_s:new_s +
        target_vec = model.predict(np.identity(5)[s:s + 1])[0]
        target_vec[a] = target
        model.fit(np.identity(5)[s:s + 1], target_vec.reshape(-1, 2), epoc
        s = new_s
        r_sum += r
    r_avg_list.append(r_sum / 1000)
```

The first major difference in the Keras implementation is the following code:

```
if np.random.random() < eps:
    a = np.random.randint(0, 2)
else:
    a = np.argmax(model.predict(np.identity(5)[s:s + 1]))
```

The first condition in the *if* statement is the implementation of the ϵ-greedy action selection policy that has been discussed already. The second condition uses the Keras model to produce the two Q values – one for each possible state. It does this by calling the model.predict() function. Here the numpy identity function is used, with vector slicing, to produce the one-hot encoding of the current state *s*. The standard numpy argmax function is used to select the action with the highest Q value returned from the Keras model prediction.

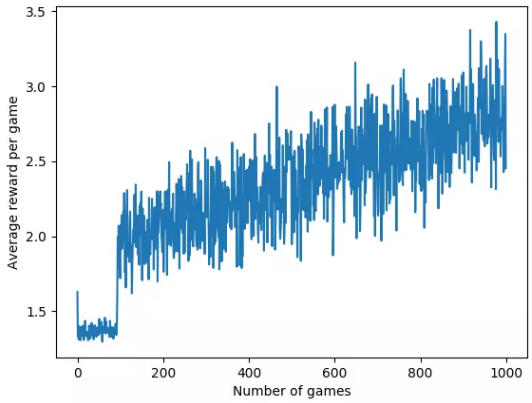
The second major difference is the following four lines:

```
target = r + y * np.max(model.predict(np.identity(5)[new_s:new_s + 1]))
target_vec = model.predict(np.identity(5)[s:s + 1])[0]
target_vec[a] = target
model.fit(np.identity(5)[s:s + 1], target_vec.reshape(-1, 2), epochs=1, ve
```

The first line sets the target as the Q learning updating rule that has been previously presented. It is the reward *r* plus the discounted maximum of the predicted Q values for the new state, *new_s*. This is the value that we want the Keras model to learn to predict for state *s* and action *a* i.e. Q(s,a). However, our Keras model has an output for each of the two actions – we don't want to alter the value for the other action, only the action *a* which has been chosen. So on the next line, *target_vec* is created which extracts both predicted Q values for state *s*. On the following line, only the Q value corresponding to the action *a* is changed to *target* – the other action's Q value is left untouched.

The final line is where the Keras model is updated in a single training step. The first argument is the current state – i.e. the one-hot encoded input to the model. The second is our target vector which is reshaped to make it have the required dimensions of (1, 2). The third argument tells the fit function that we only want to train for a single iteration and finally the *verbose* flag simply tells Keras not to print out the training progress.

Running this training over 1000 game episodes reveals the following average reward for each step in the game:



Reinforcement learning in Keras – average reward improvement over number of episodes trained

As can be observed, the average reward per step in the game increases over each game episode, showing that the Keras model is learning well (if a little slowly).

We can also run the following code to get an output of the Q values for each of the states – this is basically getting the Keras model to reproduce our explicit Q table that was generated in previous methods:

State 0 – action [[62.734287 61.350456]]

State 1 – action [[66.317955 62.27209]]

State 2 – action [[70.82501 63.262383]]

State 3 – action [[76.63797 64.75874]]

State 4 – action [[84.51073 66.499725]]

This output looks sensible – we can see that the Q values for each state will favor choosing action 0 (moving forward) to shoot for those big, repeated rewards in state 4. Intuitively, this seems like the best strategy.

So there you have it – you should now be able to understand some basic concepts in reinforcement learning, and understand how to build Q learning models in Keras. This is just scraping the surface of reinforcement learning, so stay tuned for future posts on this topic (or check out the recommended course below) where more interesting games are played!

Recommended online course – If you're more of a video based learner, I'd recommend the following inexpensive Udemy online course in reinforcement learning: Artificial Intelligence: Reinforcement Learning in Python

---

About the Author