Ideas

# Cython

Fix the bug where cython is not working in the submissions. This will probably become an issue with more complex models.

# Proper Q function approximation

Note that SARSA is very similar to Q-learning, but Nantas said Q-learning is much better in partially observed tasks.

https://webdocs.cs.ualberta.ca/~sutton/book/ebook/node89.html

<http://artint.info/html/ArtInt_272.html>

# Binary features generation

* Autoencode more past states
* Add more layers to the autoencoder
* Add non-linear input to the autoencoder like x\*x, x\*x\*x, sin/cos/tan, log, exp, x1\*x2, etc...
* Encode states from train and test set
* Alternatively, cluster the states into classes and use those as states. This is probably worse than autoencoders though.
* Alternative to binary states, Kanerva coding. I don’t really know what it is but it can work.

# Eligibility traces

<https://webdocs.cs.ualberta.ca/~sutton/book/ebook/node72.html>  
2 pseudo methods described below, should also program the proper version!

## Pseudo-eligibility traces method 1

Do something like Eligibility traces, but update only states that did not produce a reward.

set e(s) to 1 for each visited state

if reward != 0:

for all s that e(s) == 1:

update q-value

e(s) = 0

## Pseudo-eligibility traces method 2

Use our classical linreg, but train the weights as such:

always remember the last N actions performed in order

do action according to the linreg function

observe immediate reward R

for i in xrange(len(N)):

if N[i]==0: # action 0 was performed i steps ago

for all weights W of action 0:

W = W \* R \* (N-i)/N \* eta

elif N[i]== all other actions:

do same shit

note: do this for reward != 0 to speed it up

# Overfitting

When we have some sensible results, do something about overfitting on the train set.

* Regularization l1/l2
* Dropout
* Train it 20 times, each result save into a pickle, then test it 20 times and choose the weights that have the best performance on the test set.
* Train to models, one on train set and one on test set. Then make them vote for the best result.
* Make efficient online learning that can tweak the weights and parameters on the go.

# Get more ideas

* <http://www.inf.ed.ac.uk/teaching/courses/mlpr/2015/slides/09_optimization-2x2.pdf>
* <http://www.rl-competition.org/>
* glue.rl-community.org
* The book http://link.springer.com/book/10.1007%2F978-3-642-27645-3
  + read the continues states section of the book - Reinforcement Learning in Continuous State and Action Spaces
  + evolutionary approach like in the book chapter - Evolutionary Computation for Reinforcement Learning

# If nothing else works

* try the random approach to update weights instead of gradients - maybe if the weights become too complicated
* try the random approach but compute the gradients of the score change
  + update the weights in the grads direction until it stops improving, then random again until it improves, then update the weights and so on...
* Create a NN architecture, run many runs with randomized weights, then train a new NN to map the old weights to final score, from the new NN get parameters that shoudl give highest score
* Robbins-Monro conditions for shrinking the learning rate of Q-learning