1.> Domain Background

In this Project we are going to implement Recurrent Deep Q-Learning for PAC-MAN.

Historical Information -> When playing the game one controls a dot-eating character called Pac-man through a maze. The maze if filled with dots (representing foods), which reward the Pac-man with a positive score when eaten. Collision with one of the player-chasing ghosts results in a loss and consequently the end-of-game. After eating special capsules the ghosts turns to blue so the player has the opportunity to also eat ghosts (resulting in a positive reward) for a short period of time. After that Blue enemies flash white to signal that they are about to become dangerous again. These capsules are generally found in the corners of the game grid. Finally the game is won by eating all dots.

Domain Background-> In this Pac-Man game implementation it does not include multiple lives. An agent situated in this environment performs actions in discrete time steps. Every time step, the environment undergoes a stochastic transition of state depending on the action chosen by the agent. Every state transition results in a reward.
A completed Pac-man game forms a sequence of state transitions, actions and rewards. The goal of the agent is to find an action-selection policy maximizing the cumulative reward over this sequence.
Here Deep Q-learning used to find an optimal policy to win the game Pac-man and maximize its game score.

Related Paper is included with the zip file.
Always want to make intelligent model that play game better than human that learn from his mistakes.

OpenAI Gym Research -> https://gym.openai.com/envs/MsPacman-v0
Tensorflow + Keras + OpenAI Gym implementation of 1-step Q Learning from "Asynchronous Methods for Deep Reinforcement Learning" -> https://github.com/coreylynch/async-rl

2.> Problem Statement ->

An implementation of the Pac-man game is needed which enables relative easy exchange of states, actions and rewards between the game and the algorithm. In addition, to effectively learn a policy using DQN a vast amount of iterations will be needed. Hence, it is helpful during training if
simulations on the Pac-man game can run faster than real-time.

3.> Datasets and Inputs

We are considering an RGB image of 20*21 tiles.
The code dependent on OpenCV, Python(3.5), Tensorflow and OpenAI gym in AWS Ubuntu instance with g2 large support.

4.> Solution Statement

In this project we take the data driven approach, and attempt to train deep neural network to play the game Pac-Man, using no specific game knowledge, only features extracted from raw-pixel images using a convolutional neural network paired with recurrent neural network layers. Where originally Q-learning uses a data table containing the Q-values of state-action pairs. However, due to the enormous size of the state space in Pac-man, the required amount of memory and the number of iterations needed for convergence make it practically impossible to set-up such table.
The advantage of using Deep Q-learning (DQN) is that the Q-values for state-action pairs can be interpolated by a convolutional neural network instead of a data table.
A state transition from state s to s' s a result of action a and the obtained reward r together form an experience tuple <s, a, r, s'>. At the beginning of the learning progress, the Q-network returns an arbitrary number. After a number of experiences, the algorithm updates Q so it converges to the expected discounted future reward.

In Deep Q-learning two neural networks are used: the Q-network and the target network. Every training iterations, the Q-network is updated by back-propagation and the earlier described update function. The target network is used when Q-values are calculated, and is an earlier copy of the Q-network. Once every set amount of iterations the target network is substituted by an up-to-date version of the Q-network.

5.> Benchmark Model

On several thousand reinforcement learning iterations with our ConvNet-LSTM and Inception LSTM architecture with activation depths of 8, 16 and 32.
The resulting average step loss over 288 games for both models drop in average loss within the first 100 games, after it which it seems to level off.
On an RGB image of (10*20 game grid). The agent starts winning after approximately 2000 episodes of training. Continuing to train the model after 2000 episodes slowly increases the win rate. A trained model was capable of winning 95% of the games.

6.> Project Design

This one will built on top of pre-existing Python implementation of Pac-Man. The implementation provides 5 moves: left, right, up, down and stay. The score starts at 0 and changes at each timestep as follows:

Each score drops by 1.
Eating food increases the score by 10.
Eating a ghost increases the score by 300.
Eating the last piece of food (winning the game) increases the score by 500.
Being eaten by ghost (losing the game) decreases the score by 500.

The goal of modeling is to insert as little as possible 'oracle' knowledge. This means that the algorithm should learn the environment on its own without explicitly being told how the environment works. Therefore, the model and its input (game states) does not contain explicit information
such as coordinates and information especially useful for pathfinding.

The observations of game states presented to the learning algorithm should ideally match observations made by a human when playing the game. In this implementation it does not use raw pixel values, but defines states as a tensor representation of the game grid. By doing this the dimensionality of the input data decreases significantly, but very little 'oracle' knowledge is told to the algorithm. From the Markov property the transitions of state only depend on the present state and that prior history is not relevant.

Applied to the Pac-man game this means that no memory is needed to store all past game frames. However, all ghosts in the Pac-man game move in a certain direction, but lack any acceleration. This way a state includes both
positional and directional information. Each game frame consists of a grid or matrix containing all 6 key-elements: Pac-man, walls, dot, capsules, hosts and scared-ghosts.

It is helpful to add a new dimension to the state in which the locations of elements are represented by independent matrices. In each matrix a 0 or 1 respectively express the existence or absence of the element on its corresponding matrix. As a consequence, each frame contains the locations of all game-elements represented in a W*H*6 tensor, where W and H are the respective width and height of the game grid.
Conclusively a state is represented by a tuple of two of these tensors together representing the last two frames, resulting in an input dimension of W * H * 6 * 2.

The obtained reward by the RL agent corresponds with the difference in game score. In this set-up Pac-man is rewarded with a small positive reward for eating dots and a high positive reward when eating scared ghosts or winning. A loss (eaten by ghosts) results in a high negative reward. A small negative reward is given over time, in order to advocate quick solutions.

In order to improve the learning progress experiences are randomly sampled using Experience Replay. The replay memory in the original DeepQN tensorflow implementation was reimplemented as a queue, using the native deque class in Python, which enables for easy sampling and straight forward freeing of memory. Every iteration the algorithm stores an experience tuple <s, a, r',s'> in this deque. In order to prevent that the specified maximum of used replay memory is exceeded, experience tuples are popped from this queue when more than mem size experiences are added.