Closed-Task Summary

The Scotland Yard project has been implemented such that all the unit tests pass. The file MyGameStateFactory returns an instance of a class implementing GameState without exposing the creation logic to the caller.

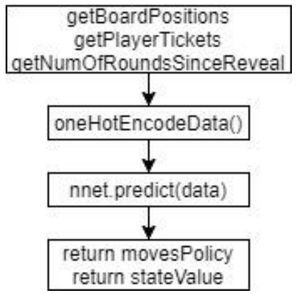
The constructor of the MyGameState class does all the required checks for a valid game state, checking for no inappropriate tickets on players and checking for any winning game states. All the possible moves that can be done by players are also calculated here, using getSingleMoves and getDoubleMoves as helper functions. The “advance” method correctly executes the functionality for movement of both Mr X and the detectives. This is in addition to doing other background tasks like giving tickets and changing the numerical location of players.

MyModelFactory was implemented in addition to MyGameStateFactory. “chooseMove” is written such that when called, the MyGameState attribute is set to the new instance returned when “advance” is called on the old object. As there are observers depending on if the game has ended, we update them using the event handler “onModelChange” given by the anonymous class definition.

Open Task Summary

Our AI implementation uses two neural networks (mrX and detective) and the Monte-Carlo Tree Search algorithm.

Both neural networks have the same architecture(in **FIG 1**).
The inputs and outputs are:



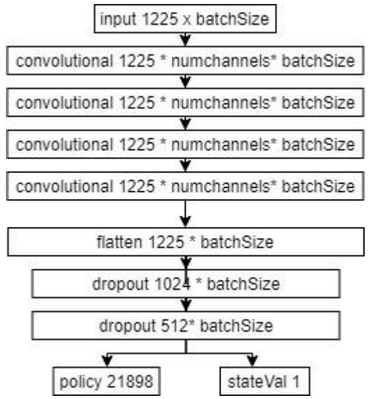
movesPolicy is a List<Float> of length *21898, every Float represents how good a move from the input state.
stateValue is a Float representing how good the input state is.

The Monte-Carlo tree search is a stochastic depth-first searching algorithm. It recursively searches the tree from the root node until it reaches a leaf node. At each stage, it chooses the node that maximises the UCB function. Once a leaf node is reached, the value of the leaf node is backpropagated up the tree to the root node and various parameters about each node are updated: number of visits, Q values, etc

The Game class acts as an interface to the Scotland-Yard API. This is so calls to the Gamestate from the Coach class and the MCTS class, can be encapsulated.

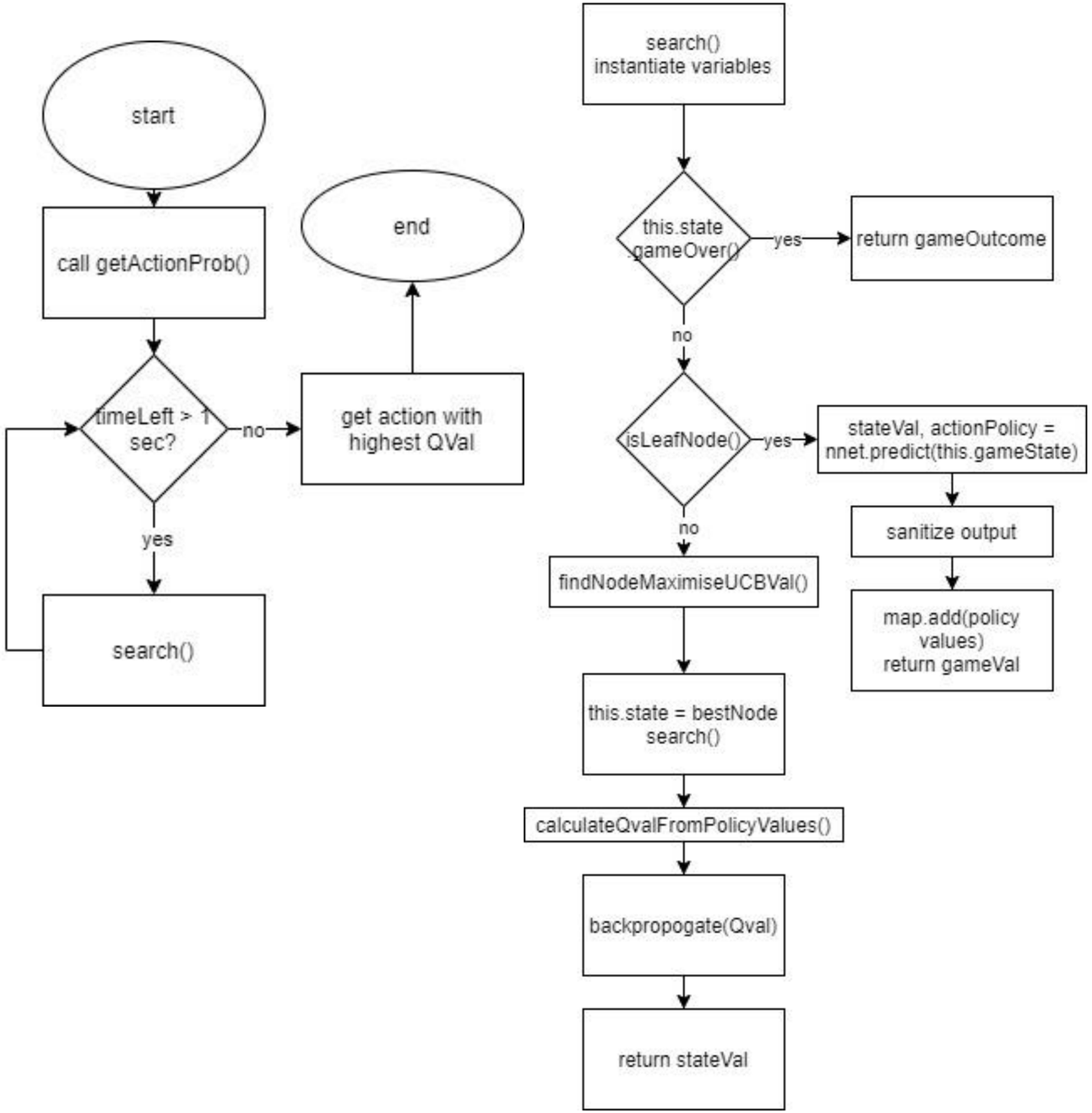
The Arena class implements the self-playing functionality, to test post-training neural networks against previous versions.

FIG 1

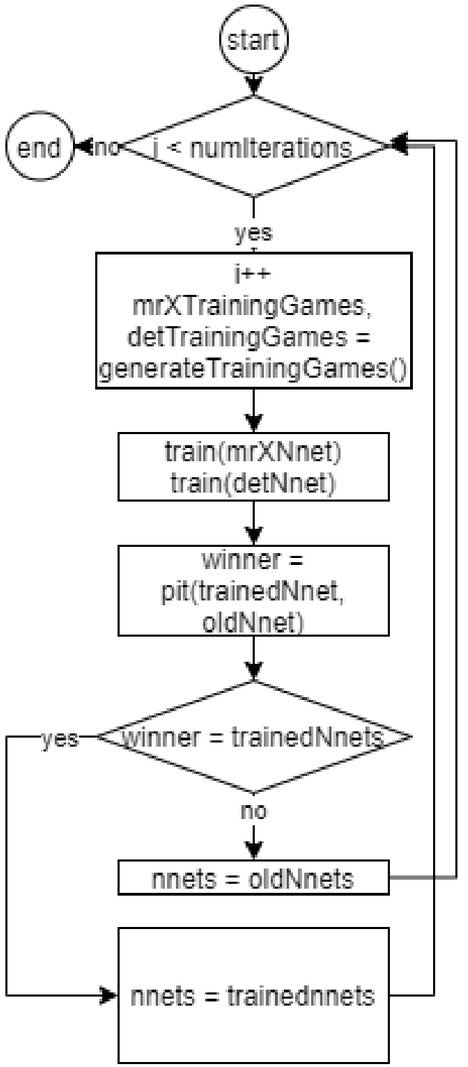


*21898 is the number of playable moves in Scotland Yard. This number is calculated on startup from moveArcs.txt. Every possible move is mapped to an integer, which we use to encode the output of the neural network

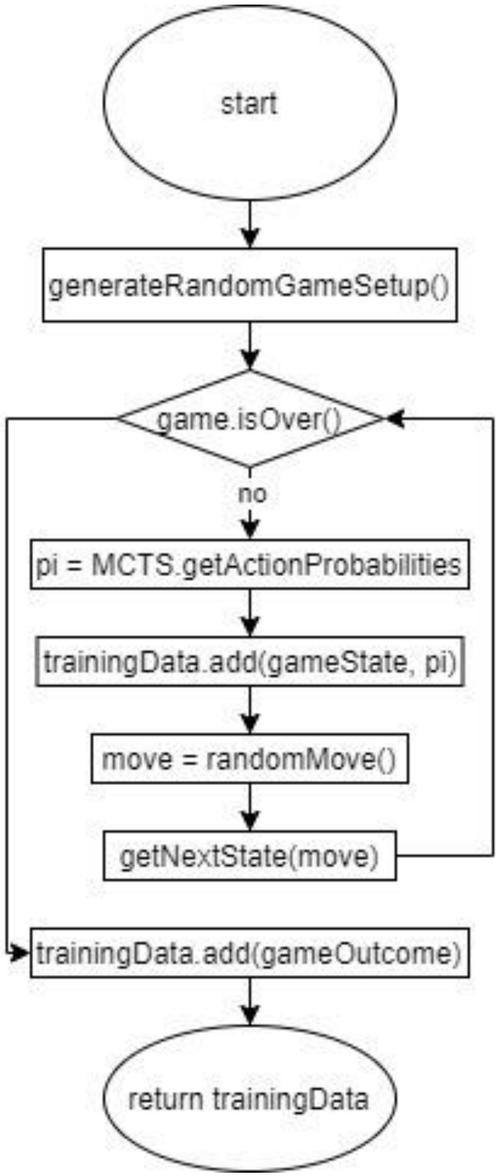
Monte-Carlo tree search Algorithm



Training loop



Game Generator



Achievements

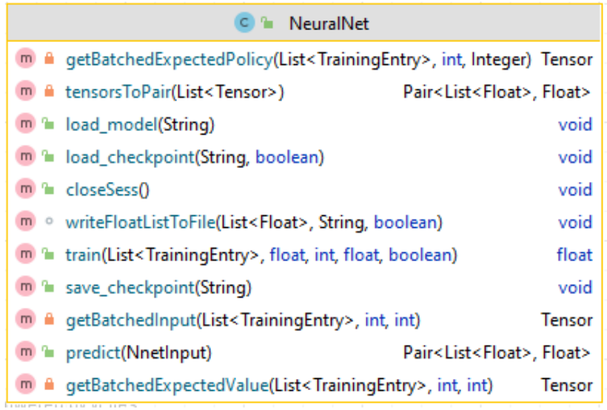
With regards to the AI, we tried 2 different approaches simultaneously - a neural network and an attempt with the minimax algorithm. The minimax algorithm was to use the shortest distance between detectives as a relative scoring function, however, on realising implementation, we instead chose to use the neural network.

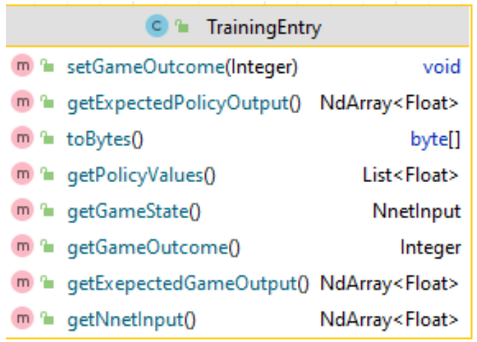
Although our implementation is functional, it does not play the game well because we did not have sufficient compute resources to train the neural network. Although the loss output of the training did decrease at a slightly decreasing rate (**fig 2) ,** indicating successful training, I suspect the model was overfitting. This was also due to poorly optimised hyper-parameters, namely: batch size, learning rate, dropout and the exploration constant in the MCTS.

fig 2

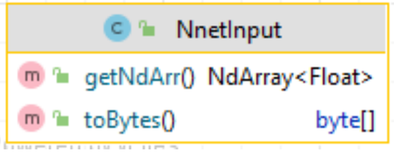
```
10.025051
9.630428
9.267572
9.018944
8.757070
8.114998
7.819105
7.580572
7.165511
7.198066
6.952410
5.850492
6.422998
5.534809
5.549112
4.351683
4.721137
5.314124
3.881793
3.466810
4.227333
4.026546
3.147265
3.134438
2.201959
3.054477
3.144745
2.335940
2.727668
4.699642
3.136915
2.671750
2.105469
2.796822
1.732100
```

Class Diagrams





TrainingEntry contains all the data needed to train a neural network. The input to the train() function is a List<TrainingEntry> of length batch size



Nnet input is the input to the network when performing a prediction.