The Hangman Game

Riyad Kettaf

June 2024

Contents

1 Introduction

The subject of this report is the presentation of some theoretical and empirical results regarding the game of Hangman. The principle of the game is very simple, at first you have $n \in \mathbb{N}$ blank spots representing a hidden word, $l \in \mathbb{N}$ lives, and while l is strictly bigger than 0, you get to uncover all the spots where the letter that you choose a in some alphabet $\mathcal{A}$ appears in the word if the letter is indeed in the word, and lose a life otherwise. Naturally, you win if you've uncovered all the spots in the word before l gets to 0.**We'll make the crucial assumption that the playing agent has knowledge of the remaining possible words after each question**. You can also guess the entire word at once. Many computer science fields appear in this report, but it will be centered around Information theory and deep learning methods.

2 Definitions

Throughout this report, we'll consider the following notations:
$n \in \mathbb{N}$ the word length
$\mathcal{A}$ the letter set
$\mathcal{D} \in \mathcal{A}^n$ the word set
X the random variable representing the words in $\mathcal{D}$ having distribution $x \to \mathbb{P}(X = x)$

Definition 1. *A strategy s is defined as a function of* $\mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{A}) \to \mathcal{A}$

Note 1. *This defines the possible letter choices when arriving at some state*

Definition 2. *We'll call S the set of strategies*

Definition 3. *Let* $P \subset \mathcal{D}$*, a strategy s is said to be optimal if it minimises the quantity:* $\sum_{x \in P} \mathbb{P}(X = x) l_s(x)$ *where* $x \to l_s(x)$ *is the amount of questions necessary to be able to guess x using strategy s.*

3 Winning probability

Abstract

In this section, we'll prove a result that gives the winning probability when playing the optimal strategy.

3.1 Tools

Definition 4. *We say that a mask* $m = (a, x) \in \mathcal{A} \times \{0, 1\}^n$ *is in a word* $w \in \mathcal{D}$ *if and only if :* $\forall i \in [1, n], x_i = 1 \implies w_i = a$

Note 2. *The case where for* $a \in \mathcal{A}$ *,* $m{=}(a, 0_n)$ *corresponds to the empty mask.*

Definition 5. *Let m be a mask, then* $\theta_m : P \to \{w \in P : m \text{ is in } w\}$ *where* $P \subset \mathcal{D}$

Definition 6. *Let* $r \in \mathcal{A}$*, then* $M_r{=}\{r\} \times \{0, 1\}^n$

3.2 Results

Theorem 1. *Let* $f : \mathbb{N} \times \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{A}) = \Omega \to [0, 1]$ *be recursively defined as follows, for (g,P,R)* [1] $\in \Omega$ [2]*:*

f(g,P,R)=0 if g=0
f(g,P,R)=1 if $|P| = 1$*, otherwise:*
f(g,P,R)= $\max_{r \in R} \sum_{m \in M_r} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x) f(g - \mathbb{1}(m \text{ is an empty mask}), \theta_m(P), R \backslash \{r\})$

then f defines the probability of winning playing the optimal strategy

[1] This vector represents the lives left, the remaining possible words left, and the remaining letters to choose from
[2] Inspired from: *Reams, Charles. 'Playing Perfect Hangman'. 1 Jan. 2009 : 149 – 153.*

Note 3. *We don't specify the case where $|R| = 0$ because it is not possible to get to this case before having $|P| = 1$*

Proof. We will prove this result by proving :
$\epsilon(g, P, R)$: "$f(g, P, R)$ is the probability of success of the optimal strategy" by induction on the set Ω with the product order relation.

Base case: Let $g = 0$ then $f(g, P, R) = 0$. $g = 0$ implies that we have no lives left, therefore the game is lost, and the probability of winning from that state is 0.

Let $P \subset \mathcal{D} : |P| = 1$ and $g > 0$. Then there is only word left possible, which means that the agent can immediately guess the word and win, which is why $f(g, P, R) = 1$.

Induction: Let $(g, P, R) \in \Omega \backslash \{\text{base cases}\}$, suppose that : $\forall (g', P', R') \preccurlyeq (g, P, R), \epsilon(g', P', R')$.
Let $r \in R$, let A be the event: 'winning after playing r'.
Let A_m be the event: 'winning after playing r, knowing that the pattern revealed in X playing r is m.

The events {Pattern m is revealed after playing $r, m \in M_r$} are disjoints, so we have
$\mathbb{P}(A) = \sum_{m \in M_r} \mathbb{P}(A_m)\mathbb{P}(\theta_m(P))$

Moreover, we have $\mathbb{P}(\theta_m(P)) = \sum_{x \in \theta_m(P)} \mathbb{P}(X = x)$ and
$\mathbb{P}(A_m) = f(g - \mathbb{1}(m \text{ is an empty mask}), \theta_m(P), R \backslash \{r\})$

We have now proven that for a given letter r played, the resulting probability of winning is $\sum_{m \in M_r} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x) f(g - \mathbb{1}(\text{m is an empty mask}), \theta_m(P), R \backslash \{r\})$

The optimal strategy minimises the average number of questions to ask needed to win. Therefore, on average, the probability of winning using the optimal strategy is going to be higher than with any other strategy, which is why the optimal strategy maximises the quantity given above. And thus $\Omega(g, P, R)$.

□

3.3 Complexity

Theorem 2. *The time complexity of f's computing is in $O(n^{|\mathcal{A}|}((2^n)^{|A|+1})) - 1))$*

Proof. For a given letter r, the number of recursive calls to f in the inductive case is 2^n as $|M_r| = 2^n$. So for a given sequence of letter, there will be (at most) as many recursive calls as there are nodes in a 2^n-ary tree of length $|\mathcal{A}|$. As the function effectively searches through all sequences of letters, we need to multiply this quantity by $|\{\text{strategies}\}| = n^{|\mathcal{A}|}$ □

Note 4. *The dictionary size doesn't appear in the complexity study but it is very much a relevant factor as we will take more guesses to get to a base case for large dictionaries*

This complexity study enables us to confidently say that it is computationally out of reach to calculate f over large alphabets or large dictionaries. However,we can make many practical improvements that noticeably reduce the computing time:

- In the implementation, it is important to keep values of $f(g, P, R)$ already calculated as there is a lot of redundancy naturally present with the recursive calls.
- On top of that, we can use **pruning**. What pruning enables us to do is ignoring letters that we know for certain won't maximise f at some state. Mathematically, if R_0 represents the sets of letters r for which I have already calculated

  $\sum_{m \in M_r} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x) f(g - \mathbb{1}(\text{m is an empty mask}), \theta_m(P), R \backslash \{r\})$

  Then if r represents the letter that I am currently computing the sum for, and that I have already recursively called f over a set A of masks, if

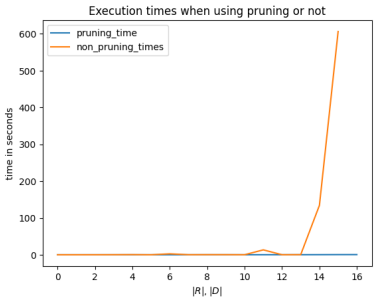
  $\sum_{m \in A} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x) f(g - \mathbb{1}(\text{m is an empty mask}), \theta_m(P), R \backslash \{r\}) +$

  $\sum_{m \in M_r \backslash A} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x)$

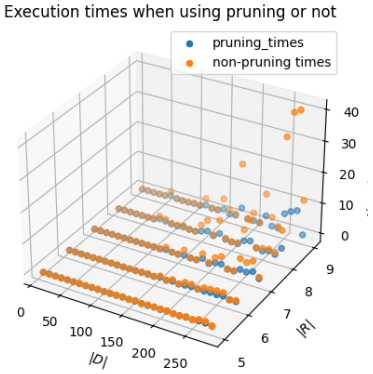
  is less than $\max_{r \in R_0} \sum_{m \in M_r} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x) f(g - \mathbb{1}(\text{m is an empty mask}), \theta_m(P), R \backslash \{r\})$ then we don't need to compute the rest of the sum for r and can skip to the next letter.

  We can also terminate the sum over R early if we find a letter r such that:

  $\sum_{m \in M_r} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x) f(g - \mathbb{1}(\text{m is an empty mask}), \theta_m(P), R \backslash \{r\}) = 1$



(a) Computing over samples of the english dictionary



(b) Computing over randomly generated dictionaries (with random g values)

We clearly see that using pruning enables us to compute values much quicker than without as $|D|$ or $|R|$ diverges.

4 Optimal strategy

Abstract

In this section, we'll present how to compute the optimal strategy and study it's complexity and see if we can make any practical improvements to how we may compute it .

4.1 The Optimal strategy

Theorem 3. *Let* $\phi : \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{A}) \to \mathbb{R}^+$ *be defined as follows:*

$$\phi : (P, R) \to 1 + \min_{r \in R} \sum_{m \in M_r} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x)\phi(\theta_m(P), R \backslash \{r\}) \text{ if } |P| > 1 \text{ and 0 otherwise} \quad (1)$$

Then ϕ *is the average number of questions playing the optimal strategy and* s *is defined as the letter in* R *that minimises the sum above*

Proof. To prove this theorem we'll prove something stronger:

$$\Delta_s(P, R) : "\epsilon(P, R) = 1 + \sum_{m \in M_{s(P,R)}} \sum_{x \in \theta_m(P)} \mathbb{P}(X = x)\phi(\theta_m(P), R \backslash \{s(P, R)\})"$$

if $|P| > 1$ and 0 otherwise is the average number of questions playing s (2)

for $(P, R) \in \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{A})$and s some strategy in S
Base case: Let $|P|$=1, then the number of questions to ask is 0 which is effectively what is returned by ϵ
Induction case:
Trying to find the average number of questions from state (P, R) is the same as finding the average number of questions after playing $s(P, R)$ *ie* computing the average number of questions for all possible outcomes weighted by the probability of those outcomes, which justifies the sum.[3]

As $\Delta_s(P, R)$ is true for all strategies s, therefore it is true for the optimal strategy, and by definition the optimal strategy will minimise the sum over R

□

4.2 Complexity

Theorem 4. ϕ*'s time complexity is* $O(n^{|\mathcal{A}|}((2^n)^{|A|+1})) - 1))$

ϕ and f are constructed in a similar manner, they both make practically the same recursive calls (if $g \neq 0$) and in worst case, they actually have the same complexity (*ie* if g never reaches 0)

Just as with f , we can make some great improvements to have a much better computation times, however we need to introduce some concepts of information theory first.

4.3 Information theory

[4]

Abstract

[3]This can't be accepted as a formal proof but enables us to be confident as to why the function is constructed in such a way

[4]See Thomas M.Cover's book *Elements of information theory* for proofs and more information

The motivation behind this section is to try to find what letter to explore first in ϕ's computing so that we can do efficient pruning.

Definition 7. *Let $x = (x_1, x_2, ...x_n) \in \mathcal{D}, i \in \mathcal{A}$ then f:(x,i) $\rightarrow (\mathbb{1}\{x_1 = i\}, \mathbb{1}\{x_2 = i\}...\mathbb{1}\{x_n = i\}) \in \{0,1\}^n$*

Definition 8. *Let $Y_t = \{f(X, i_s), s < t\}$* [5]

Definition 9. *Let H(X)=$-\sum_{x\in\mathcal{D}} \mathbb{P}(X = x)\log_2(\mathbb{P}(X = x))$, this quantity is called the entropy.*

Definition 10. *Let $\hat{X}_t \in \mathcal{D}$ be the random variable defining a word guess at time t*

Definition 11. *Let $\mathcal{D}_t$ be the remaining possible words (words that match the already uncovered pattern) at time t*

Theorem 5. *X can be guessed successfully at time t if and only if $H(X|Y_t) = 0$*

Proof. Fano's inequality gives us:

$$h_2(\mathbb{P}(\hat{X}_t \neq X)) + \mathbb{P}(\hat{X}_t \neq X)log_2(|D| - 1) \geqslant H(X|Y_t) \tag{3}$$

However,at time t we have $P(\hat{X}_t \neq X)$=0 if and only if we can successfully guess X at time t which finalises the proof. □

Theorem 6. $\forall t \geqslant 0, H(X|Y_{t+1}) = H(X|Y_t) - H(f(X, i_{t+1})|Y_t)$

Proof. Let $t \geqslant 0$, then $f(X, i_{t+1})$ is a function of X so $H(X|Y_t) = H(X, f(X, i_{t+1})|Y_t)$. Combining this with the chain rule we have:

$$H(X|Y_t) = H(X|Y_t, f(X, i_{t+1})) + H(f(X, i_{t+1})|Y_t) \tag{4}$$

Naturally, $Y_{t+1} = (Y_t, f(X, i_{t+1}))$, which gives us the result. □

The last result gives us theoretical motivation to choose entropy maximisation as a heuristic in the way that we explore letters in our computation of ϕ to have better pruning efficiency ie, **we'll choose to make recursive calls over letters that maximize $H(f(X, i_{t+1})|Y_t)$ first.**

4.4 Results

The way that we use pruning in our computation of ϕ is really similar to the way that we use pruning in our computation of f, the only two differences are that instead of checking if

$$\sum_{m\in A}\sum_{x\in\theta_m(P)} \mathbb{P}(X = x)\phi(\theta_m(P), R\backslash\{r\}) + \sum_{m\in M_r\backslash A}\sum_{x\in\theta_m(P)} \mathbb{P}(X = x) \tag{5}$$

is less than

$$\max_{r\in R_0}\sum_{m\in M_r}\sum_{x\in\theta_m(P)} \mathbb{P}(X = x)\phi(\theta_m(P), R\backslash\{r\}) \tag{6}$$

[5] $i = (i_1, i_2...i_{|\mathcal{A}|})$ represents the set of letters played

we check if

$$\sum_{m\in A}\sum_{x\in\theta_m(P)}\mathbb{P}(X=x)\phi(\theta_m(P),R\backslash\{r\})+\sum_{m\in M_r\backslash A}\sum_{x\in\theta_m(P)}\mathbb{P}(X=x) \tag{7}$$

is greater than

$$\min_{r\in R_0}\sum_{m\in M_r}\sum_{x\in\theta_m(P)}\mathbb{P}(X=x)\phi(\theta_m(P),R\backslash\{r\}) \tag{8}$$

. The second difference is that we explore letters with higher entropy first as mentioned in the previous section. Using this, we get the following graph:

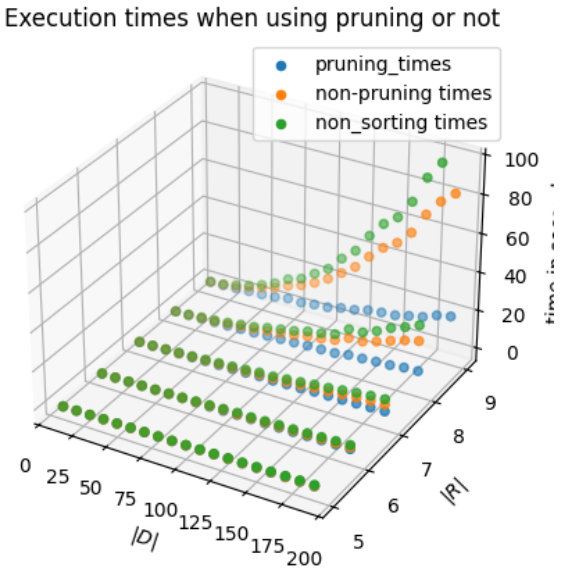


Figure 1: Computing over randomly generated dictionaries: *points are much more aligned in this graph because the randomness of g is removed*

We see that sorting according to decreasing entropy is vital to have efficient pruning.

5 Reinforcement learning

Abstract

In the previous section we detailed the optimal strategy and different solutions to lighten it's computation time, however,computing it over the entire English dictionary still seems out of reach as the problem is still exponential with pruning. This is the reason why we'll try to build an agent that is able to play hangman and find the word in a average number of turns slightly worse than when playing optimally.

5.1 Network architecture :Deep Q-network

Abstract

To build our agent that will try to play hangman with the least amount of turns on average, we'll build what we call a Deep Q-network, a type of deep learning paradigm that is used especially in reinforcement learning tasks.

5.2 Key concepts

A deep Q network is fundamentally based on 4 different objects:

1. States S: States describe the current situations of our agents, for instance, in the case of hangman a State would be:
   (a) $\hat{X}$ to describe the spots already uncovered by the agent
   (b) $\mathcal{D}_t$ to describe what remaining possible words there are.
   (c) $\mathcal{A}_t$ to describe what remaining letters it is possible to choose from.
2. Actions A: Actions describe what options does our agent have at some state, in our case that would be exactly $\mathcal{A}_t$
3. Reward functions R: They describe how we value each action's consequence on the game. For example, in the case of the hangman game, a basic reward function would be to return 1 if the letter chosen by the agent is in the word and -1 otherwise.
4. Q-Values : Q-values are functions that evaluate the expected future rewards starting from state S, taking action A, and then following an optimal policy. (More details on this below)

5.3 Network Overview

- Epsilon-Greedy Policy (exploration/exploitation): A deep Q-network chooses between doing two things at each time step: It decides between choosing a random action that hasn't been done yet and choosing the action chosen by the deep neural network (more below). The reason it does that is to avoid getting stuck in local optimums as if early on into training the qnet finds an action that yields good reward despite not being optimal, if we would not apply this policy it would keep on choosing this action despite the fact that other actions exist that yield better rewards. We also decrease the amount of times that the network resorts to this policy over the course of training to have converging behavior (coupled with the fact that exploration is less rewarding the more we train the model).
- LSTM : The deep neural network described above is a multi-layer LSTM network with the different elements that make up our State as our input layer and the chosen Action as our output layer. We choose this type of neural network because it is well suited to tackle problems where there are the successive inputs of the neural network that are dependent of each other.
- Training: Training works quite similarly to when training a supervised learning model, however there's not an immediately apparent loss function : we need to build our own:

To do this, we need to introduce the principle of batch: it is the set of state-action pairs collected by simply running through the neural network sampled from what we call the 'replay buffer' which stores the previous iterations during training.

Let N be the batch size. In the *key concepts* section we introduce the concept of 'Q-values' as an optimistic evaluation of future rewards. We can actually consider them in two different ways:

1. Predicted Q-values Q_{pred}: A prediction of future rewards going from state S, doing action A
2. Target Q-Values Q_{target}: Considers the immediate reward and γ discounted future Q-value when the agent plays optimally. It's mathematically given by Bellman's equation :

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \tag{9}$$

   with r being the reward when playing a at state s, γ the discount factor and $s \xrightarrow{a} s'$

The loss function will thus be the loss between our Predicted Q-Values and Target Q-Values for all state-action couples (s_i, a_i) in our batch. Mathematically it will minimize (using backpropagation and gradient descent):

$$\frac{1}{N} \sum_{i \in [1,N]} (Q_{pred}(s_i,a_i) - Q_{target}(s_i,a_i))^2 \tag{10}$$

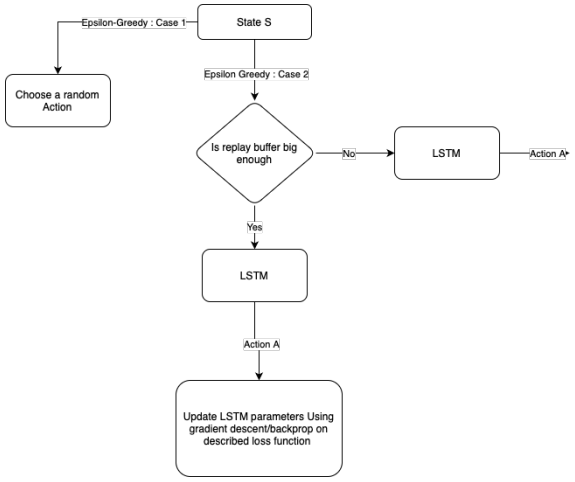
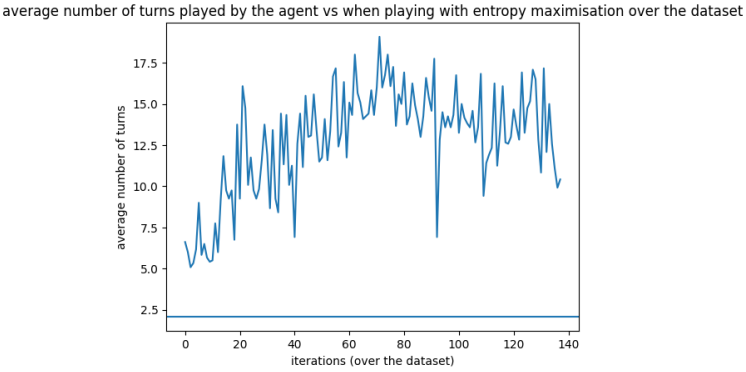


Figure 2: Diagram of Qnet construction

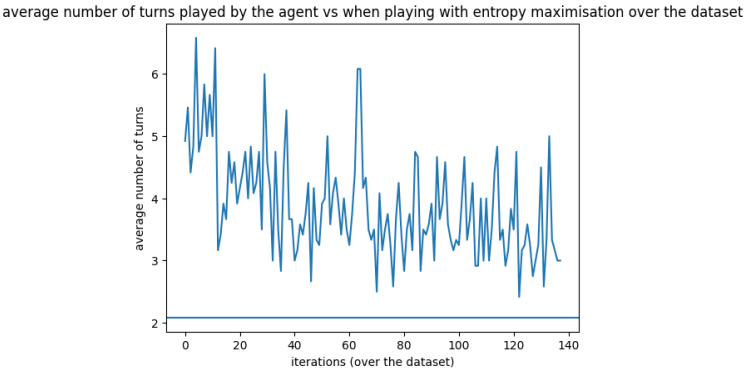
5.4 Results

Reinforcement learning is an idea that is really powerful as it does not require any data labelling and is fundamentally intuitive in the way that we imagine how deep learning models work. However

it is really sensitive to hyperparameter tuning and finding the good set of hyperparameters for our problems is a really hard task that would require extensive research. However we still managed to yield relatively good results. We trained an agent on a sample of size 120 of 5 letter English words.



(a) $\gamma = 0.99$



(b) $\gamma = 0.5$

5.5 Evil Hangman

Abstract

In the original version of the Hangman game, for each letter guess made by the player, the pattern revealed by the gamemaster relies only on the word that was chosen at the start of the game by the gamemaster. There's another version called the 'evil hangman' variant where there isn't a specific word chosen at the start but instead at each turn, when the player chooses a letter, the gamemaster can choose what pattern corresponding to this letter he chooses to reveal (he can also choose to reveal nothing as to say that the player is wrong). The gamemaster needs to choose a pattern such that the set of remaining possible words isn't empty. The goal of this section will be the find an algorithm/agent that chooses the best pattern sequence possible.

Greedy Strategy A first idea that comes to mind as to design an algorithm that plays the evil hangman is to choose, at each turn, the pattern that leaves the most remaining words possible ie :

$$\arg\max_{m \in M_t} |\theta_m(D_t)| \tag{11}$$

with D_t, the set of remaining possible words after t turns,M_t the set of admissible masks (ie masks m such that $\theta_m(D_t) \neq \varnothing$) after t turns. This strategy is efficient but we can't prove that this strategy would even end up forcing the player to make an arbitrary t amount of questions before winning. Moreover we can prove that for a given dictionary D of words of length l, and integer d, deciding if:
There exists a strategy (function of $\mathcal{P}(D) \times \mathcal{A} \to \{0,1\}^l$) that forces the guesser to make at least d guesses,
is CoNP-hard. [6]. This statement enables us to say, under the assumption that P $\neq$ coNP, that we can't find an algorithm that gives such a strategy in polynomial time.

Deep Learning Solution We showed that we can't find a deterministic polynomial algorithm that forces the player to make at least an arbitrary number of turns, however we can re-use the ideas (using the same network architecture and just modifying the reward function) proposed in this section to make an agent that learns, for a given dictionary, what patterns seem to work the best against a certain strategy used by the player (for instance, we used entropy maximisation as a strategy). We got the following results when training over a randomly generated dictionary of 200 words with an alphabet of size 11:

Figure 3: Diagram of Qnet construction

We see that our argent didn't manage to surpass the greedy algorithm, however just proving that our agent can learn to force the player to take a certain amount of takes is promising as with dictionaries where the greedy strategy doesn't work (see mentioned article), our neural network approach might yield better results.

[6] See *The Computational Complexity of Evil Hangman* by Jérémy Barbay, Bernardo Subercaseaux for proofs