Intelflash: AI for flashcards

Jaemin Cheun and Joon Yang
CS 182: Intelligent Machines Final Project

Abstract

Flashcards have traditionally been used to help us memorize words or any other associations of question and answer pairs. For many years, however, the exercise has rarely been approached with a novel method--most flashcards are “unintelligent,” simply showing the user the cards in the same order, over and over again. Intelflash uses Markov Decision Process (MDP) to gauge the user’s understanding of the deck and chooses which flashcard to test by using the Expectimax algorithm. Instead of implementing a completely offline expansion of nodes, which would take exponential time, Intelflash uses several methods such as Star1, random classifier, dynamic horizon control to help a fast and efficient runtime expansion of the game tree.

Table of Contents

The Problem

A user’s understanding of each card (question and answer pair) cannot be mapped with absolute certainty. Every moment, the user learns and forgets--a very natural human process--and this means our numbers our always slightly outdated. Furthermore, even if the user’s knowledge of each card is retained, our quantification of the user’s understanding is dependent upon merely proxies that reflect understanding; namely (a) the correctness of the user’s response and (2) how long it took the user to answer. Therefore, we face an inherent problem where we cannot represent the user’s true knowledge of the deck at any given point.

Ultimately, we decided that each card will have a number between 0 and 1 associated with it, where 1 represents the agent being 100% certain that the user knows the word, 0 representing 100% certain the user does not know, and 0.5 meaning the chances of the user knowing the word is 50-50.

The next challenge we had to surmount in modeling was figuring out, on a high-level, which questions we should ask. In other words, given a list of understanding levels (i.e. ‘apple’ has 0.7 and ‘banana’ has 0.4), which flashcard should the agent test the user? The naive approach is to simply ask the word the the user knows least. In this case, it would be ‘banana’. However, that is not an “intelligent” way to ask a question. Intelflash looks at the the understanding levels, and depending on the likelihood that the user will get the question right, branches out nodes to the game tree and calculates the optimal question based on the rewards associated with each state-action. This Expectimax-like structure is what provides the backbone to solving this problem. We will delve into the specifics of this in the Algorithms section under Implementation.

Finally, in implementing MDP to find the optimal question to ask at every state, we realized that the problem, no matter how we model it, ultimately carries an exponential space complexity. This problem persisted in the various POMDP approaches, which we will discuss in the Knowledge Representation section, as we had to trace a list of observation vectors that each branched out exponentially as the number of cards increased. This meant one of two things: we sacrifice depth (since expanding the branches beyond one or two levels would be impractical), or we come up with a novel solution. As we will further discuss, we chose the latter, and went beyond the traditional offline implementation of MDP.

Definition of the Goal

Our goals for Intelflash were both simple and clear. After implementing our algorithm, if a user learns more when using Intelflash compared to (1) a typical analogue flashcard that repeats a cycle of showing cards in a predictable order, or (2) a “dummy” algorithm that chooses a card at random, we can positively assert that we succeeded in creating a meaningful program.

Another layer to our goal is that the implementation should be seamless. In other words, the computation involved in choosing the next question to ask should not take a significant amount of time[1]. As it is intuitively apparent, there is a fine line between choosing speed (as we are expanding our game tree on runtime after each card) and meaningful results[2].

Other miscellaneous goals include intuitive User Interface (UI), a clean, readable, and appropriately commented code base that follows best practices, and being able to identify aspects of the final product that we know how to improve.

Implementation

I. Knowledge Representation

a. Experimenting with POMDP

Initially, we tried to tackle the problem by using Partially Observable MDP (POMDP). We chose this method because we cannot fully observe the user's understanding of the deck and can only partially observe it by assessing the user's performance. This meant that we had to model the problem with a 6-tuple representation in POMDP:

{S: set of states, *A:* set of actions, *O:* set of observations, *T:* conditional probability, *Ω:* conditional observation probabilities, *R:* reward function}.

Using POMDP, we attempted to model the problem in two different ways:

i. Simple POMDP

We initially tried to use POMDP to keep a belief state of the user's understanding. While modeling our problem with a 6-tuple representation in POMDP, we realized that there were major complexity issues. Because we had to keep track of progress on each word, we decided to represent each state as a list of progress the user makes on each word, but the complexity was exponential in state space.

[1] Significant, here, defined as taking a time both long and frequently enough that the user feels like the program is slowing her down. Albeit subjective, after testing on 5 users who are blinded from which algorithm of the aforementioned three is being used, we were able to tell whether the complaints were valid (true positives), or invalid (false positives). This will be further discussed in Testing Results section.

[2] Although we did not implement Value Iteration (VI) or Policy Iteration (PI), we concluded that it was more likely that a policy reached after exploring depth 5 was more meaningful than one reached from depth 3 or 4.

ii. Multiple POMDPs

To address the issue of complexity, we then considered constructing a POMDP for each word. This seemed reasonable because each word was independent to one another with respect to the transition function for a given action, and the complexity will then be linear in the number of words. We created N POMDPs, where N is the number of words, and each POMDPs had two states that represented either full understanding of the word or no understanding. Each POMDP had one possible action that asked a question about the word. For a given action, we modeled transitional probabilities in terms of the belief state of our understanding of the word conditional on the observations we see. We also visioned a centralized controller that determined which of the POMDPs to run at every step with the goal of maximizing the utility. However, it was unclear how we can solve this model because the model as a whole is not a POMDP anymore.

b. MDP

After considering POMDP, we adjust to MDP, but approached it in a novel way:

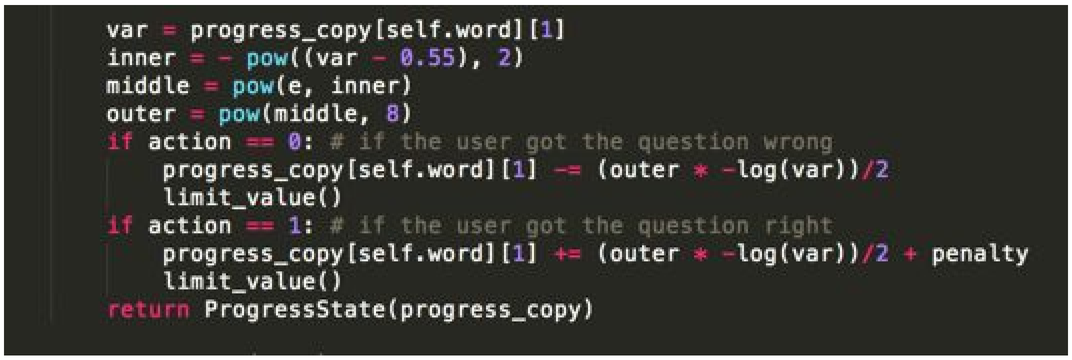
i. *S* is usually defined as a finite state space. However, in our case, *S* resembles a list with *n* number of elements, where *n* is the number of cards stored, and each element can take any value between 0 and 1. This yields an infinite number of different possibilities for *S*, which is why we chose runtime expansion of the game tree.

ii. *A* is defined as a finite number of actions. In line with the traditional definition, the total number of actions possible (branching factor) from each state is finite--in particular, equal to the number of elements *n*.

iii. *T* defines the probability function. In our case, T_{ij}^{a} = P(S_j | <S_i , *a*>), where the transition probability is conveniently equal to the value of the element of the word that we ask the user (or 1 subtracted by that value when branching to a node where the user answered incorrectly). If ‘apple’ has 0.3 *understanding*, our two branches from asking the word will lead to a correct response branch with 30% probability, and the incorrect one with 70%. Furthermore, in deciding what our S_j value should be, we decided to map the user’s understanding levels as a modified normal curve. Since we valued asking questions that the agent was not sure if the user really knew or not most, and valued asking a card that has *understanding* 0.3 over one that has 0.7, an equation that reflected both of these had the modified normal curve:

$$\left(e^{-(x-0.55)^2}\right)^8$$

After much trial and error, we realized that we needed another factor that could boost up the score of low *understanding* cards compared to high ones, and decided to reflect that using natural log. This gave rise to the following equation:



```
var = progress_copy[self.word][1]
inner = - pow((var - 0.55), 2)
middle = pow(e, inner)
outer = pow(middle, 8)
if action == 0: # if the user got the question wrong
    progress_copy[self.word][1] -= (outer * -log(var))/2
    limit_value()
if action == 1: # if the user got the question right
    progress_copy[self.word][1] += (outer * -log(var))/2 + penalty
    limit_value()
return ProgressState(progress_copy)
```

Notice the penalty is applied depending on how long it took the user to answer the question correctly.

iv. *R* is the reward. We have defined it as the average of progress on each word.

v. γ is the discount factor. We do not need γ because we are working with a finite horizon.

II. Managing Data

Along with modeling the problem choosing which format to load, write, and manage our data was one of the first issues we tackled. After considering several different formats (many of which were similar), we debated between .csv and .tsv files and chose .csv for its prevalence and ease to parse.

After writing and storing each card, we use a dictionary during the user's study session to efficiently query data. The *question* is stored as the key, and the value associated with it is recorded as a list with two elements: the first is the *answer*, and the second is a numeric representation of the agent's expectation of the user's *understanding* of each card. Both the key and the values are stored as strings, but when we use expectimax, calculate the reward, or any other computation with *understanding*, we convert it into a float before doing so.

Every time a new deck of cards are created, the .csv file is stored under a folder named flashcards that lives under the intelflash folder. The user can always edit the files directly by accessing them through Microsoft Excel or a text editor.

III. Algorithms & Performance Enhancing Utilities

a. Expectimax Search Algorithm

For a given MDP, our goal is to find a policy that maximizes the agent's utility. One approach is to view an MDP as a game against nature, and we can solve this by building a game tree. There are two kinds of nodes in the tree: the nodes where one gets to move (i), and the nodes where nature moves (<*i, a*>: *i* is the current state and *a* is the action user has taken). The game tree grows from the initial state S_0 , and when an action *a* is taken at node S_i , it transitions to the node <S_i, *a*>. On node <S_i, *a*> the game can transition to any state T_{ij}^a.

After constructing the game tree, we can solve the tree using an algorithm called expectimax search. We assign every node of the tree a value according to the following three equations:

$$\pi^*(i) = \arg\max_a Q(i,a)$$

$$V(i) = Q(i, \pi^*(i))$$

$$Q(i,a) = R(i,a) + \sum_j T_{ij}^a V(j)$$

$\pi^*(i)$ represents the optimal action that can be taken at node *i*, *V(i)* represents the value of node *i* assuming optimal policy, and *Q(i, a)* represents the value of nature's node <*i,a*>, also assuming optimal policy. Using these three equations, optimal actions at every node can be calculated, and thus optimal policy can be found.

For our project, node i represents the nodes where the computer asks the user a question, and <i,a> represents the nodes where the user answers the question a.

Now we are going to look at the complexity of the algorithm. Let the horizon be *N*, the number of actions *A*, the maximum number of transitions for any state and action *M*. Then the cost of the algorithm is $O((AM)^N)$. For our project, even though M is only two (the user can either get the question right or wrong), but the number of actions equal to the number of words being quizzed, so as the horizon becomes bigger, the computation gets slower exponentially. With 30 or more words, a horizon of 4 or more can interfere with our goal of making the program seamless.

Therefore, we needed a way that could effectively reduce the number of nodes explored without hindering the original results of the algorithm.

b. ExpectiPrune: Analogue to *αβ*-Pruning for Expectimax

For the Minimax algorithm, Alpha-beta pruning algorithm decreases the number of nodes evaluated by the Minimax algorithm in its search tree. This is done by stopping evaluating a move when at least one possibility was found to prove that the move is worse than a previously examined move. This algorithm cannot be applied to the expectimax search algorithm because Alpha-beta pruning does not take in consideration the uncertainty involved in chance nodes.

There are other algorithms similar to Alpha-beta pruning that can be applied to traditional expectimax algorithm such as Star1 and Star2, but they are also not applicable to our algorithm because our expectimax does not have any min nodes, and unlike traditional expectimax algorithm where rewards are only assigned to the leaves, nature nodes also have rewards assigned to them. Therefore, we have come with our own pruning algorithm for expectimax search for solving MDPs.

According to the three equations that were introduced in the expectimax search section, the value of node i will equal to the highest Q value among its children. Rather than computing Q values for each child and finding the maximum, we can significantly cut down the computation by not evaluating a QValue when we discover that it is worse than a previously examined Q value. We do this by keeping track of the highest Q value among the children that had been already explored. We call this value alpha. When we compute the Q value of the next child node, we assess the possibility of whether it will be smaller than alpha. At any step of the computation, we calculate the upper bound of the Q value assuming that its child nodes take the maximum possible values. If the resulting Q value is smaller than alpha, we stop its computation and prune the remaining branches.

We will look at some concrete examples. If alpha is 10 and the reward for the current Q value is 5, and if we know the maximum value the children of the nature's node can take is 3, then the upper bound for Q is 8. Because the upper bound is smaller than alpha, we can confidently say that this Q value will be smaller than some other Q value. This time, if the maximum value the children of the nature's node can take is 6, because there is a chance that the Q value can be bigger than alpha, we can continue with our computation. We then look at its first child, which has a value of 4 with a probability of 0.7. Then the computed Q value up to this point is 5 + 0.7*4 = 7.8. If we assume that the remaining branches will take the highest possible values, the upper bound for the Q value is 7.8 + (1-0.7) * 6 = 9.6, which is smaller than the alpha value. Therefore, we can prune the remaining branches without having to expand them.

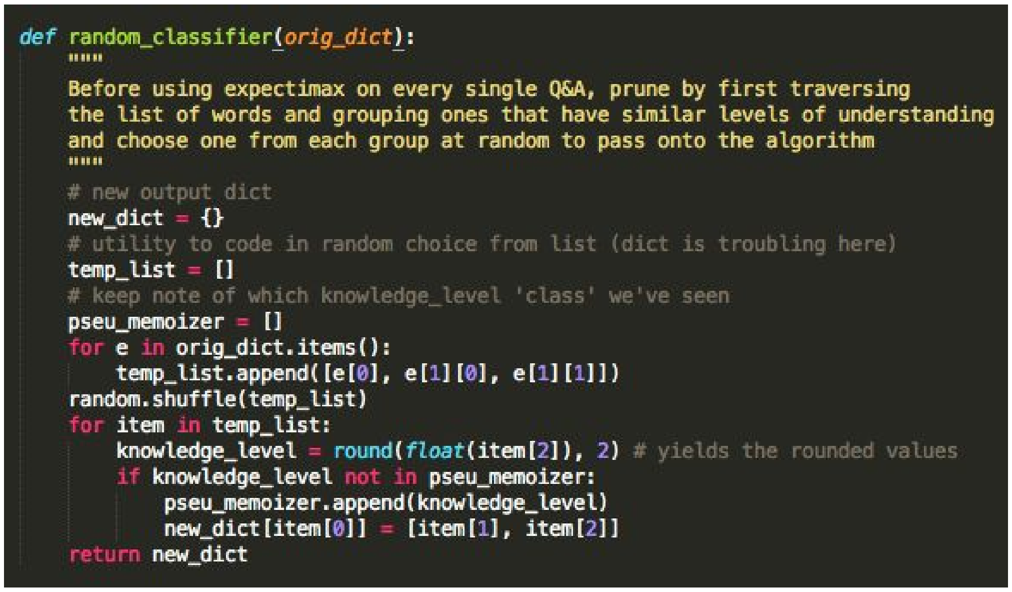
We have tested the efficiency of Expectiprune against Expectimax, and Expectiprune consistently outperformed Expectimax. For example, testing on 25 words with depth 4, Expectimax took 44 seconds to find the optimal policy, while Expectiprune took 38 seconds to find the same result.

c. Why Not VI and PI

We have decided not to use value iteration and policy iteration for our model because they compute values for all states at every iteration, and for our model that has a very big state space, both algorithms would have been computationally inefficient. On the other hand, expectimax only computes values for states that are actually reached.

d. Random Classifier

Despite our efforts to reduce the number of nodes, we realized that we cannot be too careful of cases when the user decides to run a deck with 50 or even 100 words. In the case of 100 words, setting the horizon, or depth to merely 2 will result in $2^2 * 100^2$ (2 comes from answering correctly or incorrectly at each node, and 100 from the possible words the agent can ask). This is why we chose to implement what we call a random classifier. The following code is how we implemented it:



```
def random_classifier(orig_dict):
    """
    Before using expectimax on every single Q&A, prune by first traversing
    the list of words and grouping ones that have similar levels of understanding
    and choose one from each group at random to pass onto the algorithm
    """
    # new output dict
    new_dict = {}
    # utility to code in random choice from list (dict is troubling here)
    temp_list = []
    # keep note of which knowledge_level 'class' we've seen
    pseu_memoizer = []
    for e in orig_dict.items():
        temp_list.append([e[0], e[1][0], e[1][1]])
    random.shuffle(temp_list)
    for item in temp_list:
        knowledge_level = round(float(item[2]), 2) # yields the rounded values
        if knowledge_level not in pseu_memoizer:
            pseu_memoizer.append(knowledge_level)
            new_dict[item[0]] = [item[1], item[2]]
    return new_dict
```

Notice that we convert the data structure into a list for ease of operation, shuffle the list, so that the first one we choose is random, and add it to the new output dictionary only if we have never seen another card that has a similar level of understanding.

We devised this method after noticing that cards with the same understanding levels have identical branching patterns. If the card 'apple' and 'banana' both have 0.5 in *understanding*, the agent always selects an arbitrary[3] one that was stored in the dictionary. We ultimately decided we would give a looser error margin for similarity of *understanding*--rounding up to 2 digits of the float.

d. Dynamic Horizon Control

Although our random classifier was able to reduce our initial node to go from 50 words to 30, 20, and depending on the progress, under 10 nodes, we were dissatisfied with how our upper bound for the horizon was almost certainly restricted by the number of cards the user chose to

[3] Technically, Python dictionaries are stored based on a hash table, but for our intents and purposes, the storing process is not based on a unique algorithm that would affect the agent's decision. In this case, the agent would select 'apple' no matter what depth you select, so we decide it to be unnecessary to branch out 'banana'.

store (i.e. it would be impractical to expand more than 1 or 2 nodes for a deck of cards that has over 200 question and answer pairs). Keeping the horizon static meant that we could not take full advantage of our computational capacity of our machine even when our random classifier produced 10 or 20 words as our starting node. This is why we used a simple yet elegant system called dynamic horizon control.

The idea, as well as the implementation, is very simple. We set boundaries based on the number of cards we are still left to explore after random classifier, and depending on how many we still have, we decide on the horizon we want to use. In other words, if we only have 7 or 8 words left to explore, we boost up our computation by exploring a lot more nodes, compared to when we have 70 or 80 words.

Testing Results

To assess whether we achieved our goal, we tested our algorithm on 5 different users. 3 of them were also tested on similar cards but with different information using both random and static-ordering flashcards we intended to outperform. We ask them to use the flashcards for 2 minutes, then 1 minute, then another 2 minutes. The results were promising. All 5 users' understanding level (or score) increased from start to finish. Furthermore, every round, their score increased. 3 of the 5 demonstrated relatively poor performance (negative scores) on the first round, but in the second round, received 80 to 120 points, and in the third round scored over 200 to 250 consistently.

In 2 of the 3 cases, the random flashcard actually outperformed our agent in the first round, but the scores did not improve as much or as drastically in the second and third round as Intelflash. The static-ordering flashcard performed the most poorly on all 3 rounds, scoring below 70 to 130 on all three rounds.

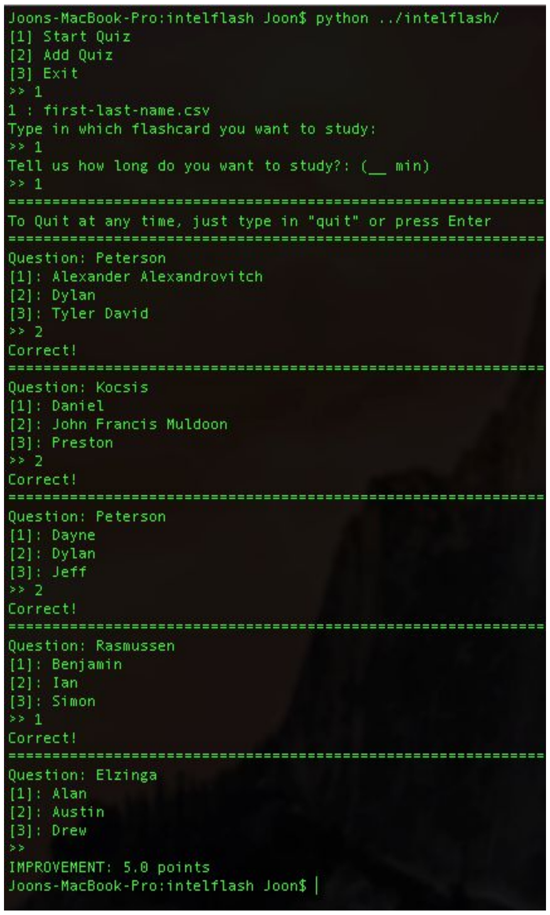
The 3 users who were blinded from which flashcards were being used mentioned Intelflash, compared to random and static-order flashcards, had minor but noticeable lags at some point[4]. This minor (and expected) setback was offput by the fact that users commenting if Intelflash had a slightly faster computing power, then they would have gone through just as many flash cards as the other two, and would have scored even higher. Finally, our code-base, with relatively minor exceptions, followed best practices for coding in Python and is very readable.

[4] A further examination revealed this invariably occurred when there were 8 or 9 initial words after random classifier, and when we set the depth to our maximum 4 (5 previously), the system slowed down. 10 and above nodes has a depth 1 smaller, and 7 initial nodes and below seem to have no problems in computing the optimal question almost instantly.

Closing Thoughts

Intelflash successfully used made use of MDP, but did so in a novel setting where we dealt with an infinite state space, and therefore ran the algorithm on runtime. Through methods like Expectiprune, Random Classifier, and Dynamic Horizon Control, we were able to optimize performance and produce promising results that consistently outperformed traditional flashcards.

With a simple GUI and some more performance optimization for speed and acquiring more depth, Intelflash may be used for use by the mass in production.



```
Joons-MacBook-Pro:intelflash Joon$ python ../intelflash/
[1] Start Quiz
[2] Add Quiz
[3] Exit
>> 1
1 : first-last-name.csv
Type in which flashcard you want to study:
>> 1
Tell us how long do you want to study?: (__ min)
>> 1
=============================================================
To Quit at any time, just type in "quit" or press Enter
=============================================================
Question: Peterson
[1]: Alexander Alexandrovitch
[2]: Dylan
[3]: Tyler David
>> 2
Correct!
=============================================================
Question: Kocsis
[1]: Daniel
[2]: John Francis Muldoon
[3]: Preston
>> 2
Correct!
=============================================================
Question: Peterson
[1]: Dayne
[2]: Dylan
[3]: Jeff
>> 2
Correct!
=============================================================
Question: Rasmussen
[1]: Benjamin
[2]: Ian
[3]: Simon
>> 1
Correct!
=============================================================
Question: Elzinga
[1]: Alan
[2]: Austin
[3]: Drew
>>
IMPROVEMENT: 5.0 points
Joons-MacBook-Pro:intelflash Joon$ |
```

Appendix 1

On the left, we demonstrate a new round after having performed poorly on Dylan Peterson's name. Instead of asking two times in a row, we ask the card with a buffer question, and once the understanding seems up to par, we proceed to another card, all of which is modeled by our algorithm.

Appendix 2

Please refer to README.md in intelflash.

Appendix 3

Joon Yang - Developed front end, modeled MDP, implemented Random Classifier and Dynamic Horizon Control
Jaemin Cheun - Implemented Expectimax search and Expectiprune

References

[1] N. Jacques: Emotionally Adaptive Intelligent Tutoring Systems using POMDPs
[2] D. Braziunas: POMDP solution methods
[3] J. Veness: Expectimax Enhancements for Stochastic Game Players

[4] T. Hauk, M. Buro, J. Schaeffer: Rediscovering *-Minimax Search
[5] POMDPs for Dummies
http://cs.brown.edu/research/ai/pomdp/tutorial/index.html
[6] https://github.com/sydneyweidman/pycards