Optimized computational modeling of adaptive activation-based concept learning

Velislav Slavov
author
velislav.slavov@student.uni-tuebingen.de

Dr. Detmar Meurers
advisor
dm@sfs.uni-tuebingen.de

September 2018

Abstract

When faced with the challenges of vocabulary learning, technology has been proven to provide increased results compared to conventional learning methods by implementing explicit formal models of human cognition. One such model was developed by Philip Pavlik and John Anderson and was based on the ACT-R cognitive architecture. Applying algorithm optimization techniques to their algorithm and increasing its computational efficiency promotes its application in real-life learning systems. This thesis develops an implementation of the Pavlik and Anderson model which should improve its efficiency without sacrificing effectiveness in language learning tasks.

Declaration of Authorship

Chapter 1

Introduction

Factual learning is an important part of human development and can be found in every aspect of everyday life. One of the increasingly popular fields, where this process takes place is language learning and in particular - vocabulary learning. While an individual gains knowledge of their mother tongue through a process referred to as ”language acquisition” (due to the fact that learning happens mostly passively), learning further languages requires a conscious effort from the learner, in which case it is referred to as ”language learning”.

Cognitive psychology has been able to identify a number of patterns, called cognitive biases, which try to explain how humans process information, such as the modality effect, the testing effect and the spacing effect. The increasing use of technology to facilitate learning and the development of cognitive architectures such as the Adaptive Control of Thought—Rational (known as ACT-R) provides a real-life application for such biases and allows educators to develop learning systems, which reliably produce improved results. One of the challenges of applying such a system in real life is developing an algorithm complex enough to accurately model human cognition as well as one that performs efficiently enough to not hinder the learner’s interaction with the system.

The goal of this thesis is to analyze an adaptive activation-based learning model, based on the ACT-R architecture and try to develop a more computationally efficient implementation of that model. The performance as well as the accuracy of the resulting implementation will be compared to the original implementation of the learning model by conducting a series of experiments, which simulate real learning conditions.

// TODO: add what to expect in chapter 1 and 2

Chapter 2

Background

This section introduces the two aspects of the optimization process. First it describes research done in the field of human cognition and introduces the learning algorithm, which serves as the base for optimization, and then describes techniques commonly used to optimize the algorithm given stated challenges.

2.1 The spacing and testing effects

When it comes to factual learning tasks, massed learning (colloquially known as cramming) has become the go-to approach. It consists of exposing the learner to a big amount of information for a short amount of time in an attempt to prepare the learner for an upcoming examination on the studied material. Research, however, has shown (Ebbinghaus, 1885) that massed learning only produces short-term results and therefore is unsuitable for achieving long-term information retention. The proposed alternative for effective long-term results is the so-called distributed (or spaced) learning, which is based on Ebbinghaus' theory that spacing the individual item interactions [1] further apart actually produces consistent long-term retention. This discovery is known as the "spacing effect". Furthermore, spaced learning attempts to optimize both intra-session intervals (how often each word gets presented within a study session) and inter-session intervals (how far apart individual study sessions should take place).

2.2 The basics of ACT-R

Since the model, which is being analyzed in this thesis is based on the ACT-R cognitive architecture, we will provide a very superficial introduction to it in order to introduce some important terminology and concepts. ACT-R stands for "Adaptive Character of Thought - Rational" and its aim is to explain and formally define how humans perceive and express both declarative and procedural knowledge. Declarative knowledge refers to the ability of the human brain to retrieve **factual information**, whereas procedural knowledge is connected to the subconscious ability to accomplish somewhat **complex procedures** (riding a bicycle, going up the stairs, writing a text etc.). Since vocabulary learning can be compared to learning facts, the model described in this thesis will make use of the implementation of declarative memory in the ACT-R architecture. The model is also referred to as an activation-based one, as it relies on the concept of "activations", which ACT-R defines as the strength of each vocabulary item in the learner's memory. Van Woudenberg (2008)

2.3 The Pavlik and Anderson model

Pavlik and Anderson have managed to formalize this research into their model by modifying the base ACT-R formula for calculating activations. Pavlik and Anderson (2005). An item's

[1]The pairs "vocabulary item" - "word" and "item interaction" - "word encounter" will be used interchangeably from now on.

activation is expressed as a power function of the sum of the time deltas of its previous encounters (how long ago each encounter occurred), scaled by the rate of decay at the given encounter (how fast the activation decays with the passage of time). This results in the latest encounters having the highest impact on the item's current activation [2]. Formula 2.1 shows the equation developed by Pavlik and Anderson for calculating the activation of an item at a given time t.

$$m_i(t) = \ln\left(\sum_{j=1}^{n;t_j<t} (t - t_j)^{-d_{i,j}}\right) \tag{2.1}$$

The variables in this formula are defined as follows:

- m_i = the current item's activation
- n = the number of previous encounters with this item
- t = a given timestamp
- t_j = the timestamp of a previous encounter with the item (note that t_j is always smaller than t)
- $d_{i,j}$ = the rate of activation decay of item i at encounter j

The rate of decay for the **current encounter** is calculated by taking into account the activation of the item at the previous encounter:

$$d_{i,j} = ce^{m_i(t_j)} + \alpha_i \tag{2.2}$$

If a previous encounter happened when the item's activation was high, this would also produce a high rate of decay, which when plugged into the activation formula would further scale down the time difference, resulting in a smaller contribution of that encounter to the current activation of the item (meaning that being presented with a recently encountered item has little effect when it comes to strengthening that item in memory). Decay can further be modified using the other two variables seen in the formula. The c parameter represents a scaling factor [3], which governs how strong the spacing effect is and the α parameter represents the current lowest decay value of an item. The lowest decay value usually starts at 0.3 since that is the default decay value in ACT-R, however, as the user encounters the item more often, that value gets adjusted in order to adapt to their performance. In order to calculate an item's activation at a previous encounter, the formula creates a mutual recursion with the activation calculation formula and the base case occurs when activation is being calculated for a time t when there are no previous encounters. In the base case the result is an activation of negative infinity (-INF), since the item has never been encountered before and therefore has no strength whatsoever in the learner's memory. If decay is being calculated for an activation of -INF, this results in the default decay value.

Having calculated the activations of all items in the current vocabulary list given the current time, the model decides which item to present next based on the following priority list: items below the forgetting threshold > new items > item with lowest activation.

The model achieves adaptability by tweaking the α parameter based on the learner's performance. Correctly guessing a word during one of the encounters means that the learner is making progress in strengthening that word's activation in their memory. In that case the α gets reduced, which results in a lower decay and therefore the item is shown less often during the study session. If the reverse is the case (the learner gets the word wrong), then they need to practice it more in order to learn it better. To make sure this happens, the model increases the α, which in turn increases the decay and results in more frequent presentations. Furthermore, the model tries to present words before their activations fall below a forgetting threshold τ, at which point they would be forgotten by the learner and previous learning will have been in vain. After a state is reached, where the decay is so minimal that the item's strength in memory is considered constant, the item stops being presented (and is considered as "learned") and a new item from the vocabulary list enters the system.

[2]The definition of "word encounter" varies depending on the learning system. It could consist of presenting word-pairs, sentences with missing word, etc.

[3]A list of all predefined constants can be found in Appendix C.

Figure 2.1 below shows the change in Activation and Decay (α) for a given item during the course of learning. After each encounter, the item's activation peaks, meaning that it is fresh in the learner's memory and it gradually decays based on its α value, which is either increased or decreased depending on the outcome of the encounter.

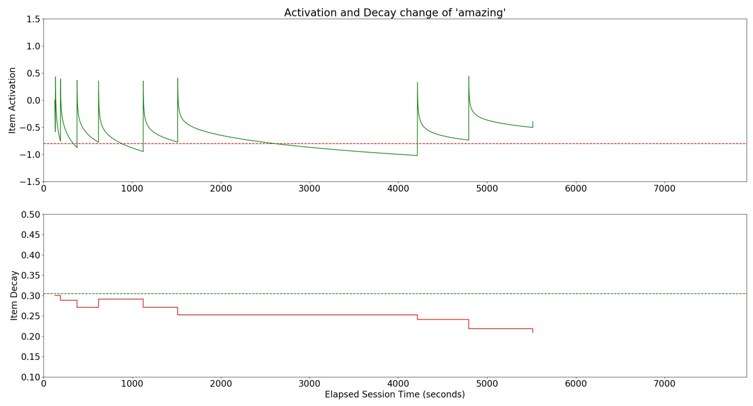

Figure 2.1: Activation and decay change of the item "amazing" from a simulation of 2 study sessions, taking place 24 hours apart and lasting 30 minutes each

2.4 Enhancements of the Pavlik and Anderson Model

The Pavlik and Anderson model has been used as a basis for a lot of the research which targets vocabulary learning (and factual learning in general). Extensions to the model have been made by Pavlik (2007); Van Rijn et al. (2009); Nijboer (2011) and others. All of the proposed extensions keep the general idea of the model, but aim to improve the activation calculation to either better represent the learner or better show human learning and cognitive patterns.

In 2007 Pavlik suggested an improvement to the activation formula by adding 3 additional β parameters (Formula 2.3). Each of those parameters would contain either item-specific or learner-specific information, which would better represent the learner's connection to the particular item. Since different people learn at different rates, the β_s parameter was added in order to introduce the learner's learning ability to the activation calculation. In addition to that, certain words are more difficult to learn than others (either because of their morphology or their semantics). To account for that, Pavlik added the β_i parameter to the formula as well. The first two parameters are then combined in the $\beta_{s,i}$ parameter in order to represent the relative difficulty of a given item for the particular learner. To add an additional dimension to the calculation, the b_j parameter was added, which allows the system to apply scaling to different word presentations. This means that it can now evaluate word-pair presentations differently than presentations of the word in context, which allows for much more control over the learning process. Pavlik (2007)

$$m_i(t) = \beta_s + \beta_i + \beta_{s,i} + \ln\left(\sum_{j=1}^{n;t_j<t} b_j(t-t_j)^{-d_{i,j}}\right) \tag{2.3}$$

Different approaches were taken by Van Rijn et al. (2009) and Nijboer (2011). They proposed that the learner's reaction time be recorded and used when calculating the activation of each item. This suggestion is based on the idea that if the learner takes longer to finish the word encounter, then it is more difficult for them to recall the item from memory. Nijboer further expanded upon the idea of using reaction time by providing a way to predict the expected time needed to process the item presentation exercise (for example the time needed to read a sentence, before the learner has to attempt to retrieve a missing word from their memory).

2.5 The problem

Even though most of the proposed extensions have been shown to provide improved effectiveness of the model, none of them tackle the problem of computational efficiency. When integrated in a real-life system, an inefficient algorithm could inconvenience the user by making them wait while the activations of all items are calculated before each encounter, wasting precious learning time during the study sessions. Furthermore, since the activation calculation takes into account all previous encounters with the item, it becomes increasingly expensive when it comes to learning more difficult words (since they would require more encounters to learn). All of these factors have inspired this attempt at improving the computation time of the algorithm.

2.6 Program optimization methods

The process of algorithm optimization usually aims to improve an algorithm by reducing the usage of a certain resource (whether it is execution time, hard drive space or another available resource). However, sometimes optimization with regards to one resource requires increased usage of another one. In the case of the Pavlik and Anderson learning algorithm, the valued resource is execution time (since the algorithm is executed before each item encounter and needs to quickly determine which word should be presented next)

Program optimization is usually applied using established and proven methods. Those methods differ in the level at which the optimization occurs, as well as in the type of optimization problem they are trying to solve. The different levels of program optimization include:

1. Optimal choice of technologies
2. Algorithm design with attention to resource efficiency
3. Writing source code with attention to efficient command execution
4. Compiler optimizations

The higher levels of optimization (where most of the work is usually done by the developer instead of the computer) tend to have a greater impact on program efficiency than the lower ones. This means that an inefficient algorithm, which was run on a highly optimized system will execute slower than an efficient algorithm, which was run on a sub-optimal system. Therefore the higher levels are usually targeted first when an optimization problem occurs.

Some of the most famous optimization methods make use of recursion in order to simplify the initial problem into smaller, more manageable sub-problems. One such method is the well known *Divide and conquer*, which splits the initial problem recursively until the sub-problems are simple enough to be easily solved. After the base problems are solved, their solutions are combined in a way that yields the solution to the initial problem. Divide and conquer algorithms have found a wide usage from sorting problems to syntactic analysis problems. Thomas H. Cormen and Rivest (2000)

2.6.1 Dynamic programming

Another optimization technique, which tries to optimize recursively solvable problems is *Dynamic programming*. While Divide and conquer is only suitable for problems, where the merging of the solutions to the recursively created sub-problems yields the result of the original problem, Dynamic programming is used in cases, where those solutions can be directly used to solve the original problem. The applicability of Dynamic programming techniques to the problem at hand depends on 2 properties: **optimal substructure** and **overlapping sub-problems**. Optimal substructure is fulfilled when an optimal solution to the problem contains optimal solutions to its sub-problems [4]. The other property of Dynamic programming refers to the fact that problems, to which it is applied do not constantly generate new sub-problems at each step (which is what Divide and conquer algorithms tend to do), but instead repeatedly solve the same problems. Since

[4]notice that when talking about optimization, there is no single optimal solution, there are multiple possible ones

this repeated solving is very inefficient, Dynamic programming promotes the use of *Memoization* in order to store the results of sub-problems in a map so that the algorithm doesn't need to re-solve them every time they come up, but instead just fetch the solution from memory. Thomas H. Cormen and Rivest (2000)

An example of a problem, to which Dynamic programming could be applied is generating a number from the Fibonacci sequence. This problem exhibits both optimal substructure (since each number is a result of summing all previous numbers) and overlapping sub-problems (calculating each number requires the calculating of all previous ones, which inevitably makes some of the calculations overlap). Other uses can be found in problems such as finding the shortest path, Levenstein distance calculation as well as parsing (The Earley chart parser). The activation calculation of Pavlik and Anderson's learning model, however, behaves in the same way that the Fibonacci sequence is calculated: in order to get the activation of an item at an encounter with a given time t, we need to calculate the activations of that item at all encounters, which occurred before the given one. Once the calculation recurses down, it causes a lot of the values to be re-calculated, therefore using Dynamic programming techniques (in particular Memoization), could speed up the calculation and make it less demanding. Of course storing each word's activation for each encounter requires the usage of more hard drive space, but the size of this information should be small enough to justify the undeniable improvement in performance.

2.6.2 Approximate computing

Another approach to optimization can be observed in Approximate computing, which uses result accuracy as an available resource. This method is based on the idea that solving a problem, the accurate calculation of which proves to be quite resource-intensive, by instead approximating the result (within a limited range of error), provides an unexpected improvement in performance. Since result approximation can have catastrophic consequences if applied improperly, Approximate computing follows certain rules about which data can be approximated and which - not. The general rule states that only data which is not critical to the execution of the program can be approximated. Therefore it is up to the developer to carefully apply this method in order to achieve its potentially significant results. Mittal (2016)

Chapter 3

The Optimization Process

3.1 Naive implementation

In its most inefficient form, Pavlik and Anderson's activation calculation algorithm is a mutual recursion between the activation function 2.3 and the decay function 2.2. The activation function depends on two sets of values: the timestamps of all encounters with a particular item and the item's decay rate at each of those encounters. While encounter timestamps are most likely already stored in each item's encounter history, calculating the decay proves a little bit more intricate. The item's decay intercept (α) is easily accessible, but calculating the activation of the previous encounter requires multiple recursive calls between the two functions until no more previous encounters exist. Although this is inefficient, it should still function when the number of encounters isn't that big, but once an item has been encountered a certain number of times, the continuous mutually recursive calls overflow the stack [1].

An obvious solution would be to simply store the activation value for each encounter (since a history of encounters is already being maintained) and then simply retrieve those activations whenever the decay rate needs to be calculated for a certain encounter. This, however, is further complicated due to the fact that the model adjusts based on the user's performance. As already mentioned, the model adjusts its representation of the user and their interaction with each item by tweaking the item's α value, which directly affects the item's decay rate and hence how frequently it is chosen for the next encounter. This results in the α value indirectly influencing the very activation of an item after each encounter. Since the activation value is constantly changing, retrieving it from memory at different times could produce invalid results.

3.2 Base implementation

In order to avoid the above scenario, it is more sensible to implement the algorithm in such a way, that recursion only occurs in the activation function. The purpose of recursing then is to calculate the activation of the previous encounter and once that is achieved, the value can be used for directly calculating the encounter's rate of decay. Although this solves the stack overflow problem, it is still a very inefficient solution with an exponential complexity. Since the activation function is recursively called 2^n times, where n is the number of encounters in the item's encounter history, the algorithm will perform worse as the encounter histories of all items grow.

A closer inspection of the learning algorithm shows that the bottleneck occurs because of the number of function calls, which are executed with the same parameters. The reason for this is that calculating the activation of an item at the current encounter requires the activations of all previous encounters with the item to also be calculated. Recursing down, this results in the activations of earlier encounters being recalculated multiple times, because they are all (even though sometimes indirectly) needed for calculating the activations of latter encounters. For example if an item has only been encountered 3 times, calculating its activation at encounter 4

[1] A stack overflow occurs when the capacity of a program's call stack is exceeded (i.e. there are too many function calls)

requires recursive calls to activations 3, 2 and 1, each of which also requires recursive calls to their previous encounters. In the end the activation of encounter 1 ends up being calculated 4 times and the whole process makes a total of 8 recursive function calls.

This behavior can also be observed when trying to generate a number from the Fibonacci sequence, since each number is the sum of all previous ones. Given that a more efficient implementation of the Fibonacci algorithm can be achieved by applying Dynamic programming techniques and that the activation algorithm fulfills both requirements for being able to use dynamic programming (the identical recursive calls prove that the activation algorithm includes overlapping sub-problems and the fact that each activation is dependent on all previous ones shows the optimal substructure of the algorithm), it is reasonable to attempt to apply dynamic programming to the activation calculation as well. This is achieved by caching the results of successfully executed recursive calls so that whenever the activation function needs to be executed with the same parameters, the result is simply retrieved from memory. Making use of this technique drastically decreases the time complexity of the algorithm from exponential to linear and speeds up the calculation. Instead of 2^n recursive calls (where n is the number of times the item has been encountered), the algorithm now only makes $n + 1$ calls (one for each previous encounter and one for the current one).

3.3 Heuristic approach

Despite the implementation described above significantly speeding up the calculation, the fact that this calculation occurs for each word before each item encounter and that it becomes more taxing as the number of encounters increases could require that it be further optimized in order to not hinder user interaction when faced with more difficult words or longer word lists. It is impossible to make further significant improvements to the time complexity of the actual activation calculation algorithm, but it is feasible to attempt to solve the problem with a heuristic approach. This would be based on reducing the amount of calculations done during the study session by making use of the time between sessions. There are two reasons why the recursive calculation needs to take place before each item encounter:

1. The time, elapsed since the last encounter of each item affects the item's current activation
2. The α value of each item is adjusted after each encounter in order for the model to adapt to each learner

Although the elapsed time cannot be neglected, since it represents the spacing effect, it might be possible to make use of the changes in the items' α values when further optimizing the algorithm. In his PhD Thesis, Florian Sense shows that an individual's rate of forgetting (α) stays relatively stable over time. Sense (2017). This information, combined with the experience of Approximate computing, can be utilized to develop a heuristic solution for faster activation calculations.

As mentioned above, the values around which the heuristic solution is built are the items' rates of forgetting (α). Since those values are adjusted based on the outcomes of each user interaction with each item and since it is impossible to predict the outcomes in advance, no true approximation of these values can be made. Instead, this solution is based on caching the activation values of all previous encounters and then using the cached values whenever the current activation needs to be calculated, which results in each activation calculation requiring a single function call. Even though this means that the previous activations don't get recalculated with the newest α, its stable rate of change should ensure that no substantial differences occur between the actual updated activation values and the cached ones. That being said, if the previous activations are never updated, they run the risk of becoming outdated once the difference between the current rate of decay and the one used for calculating a previous encounter's activation becomes significant enough. To prevent this from happening, the model should perform a complete update of the cached activations, which consists of recalculating the activations of all previous item encounters using the item's current α value. Since the goal is to avoid wasting time during study sessions, this update should take place between them, which preserves the user's interaction with the system during the actual process of studying.

Chapter 4

Evaluation of Performance (Setup and Results)

All experiments conducted in order to research the spacing effect in vocabulary learning systems use varying parameters for their learning model (study session duration, time between study sessions, vocabulary size, etc.). The reason for this is because learner performance largely differs based on internal factors, such as the learner's language learning capabilities, their familiarity with the domain, in which particular words are used and others. This makes coming up with the perfect set of parameters a challenge. Van Woudenberg (2008). In order to provide a stable environment, which allows for comparison between the results of the different experiments, we have chosen to conduct simulations based on those in Christopher Dilley's Paper on the "Application of Personalized Adaptive Learner Modeling to Vocabulary Learning Software". Dilley (2017)

4.1 Simulation setup

The experiments simulate a learning process, which consists of 2 study sessions, each lasting 30 minutes and taking place 24 hours apart. These values should allow for testing of the spacing effect not only during study sessions, but also in the time between them. The goal of the simulations is to replicate the process of a human learner being presented with a list of 37 words during the specified studying time.

The simulation system keeps track of two sets of α values for each word:

- Real α: randomly assigned to each word before the simulation in order to show how difficult that word is to learn
- Predicted α: adjusted based on word interaction outcomes as a way of simulating the model's adaptability (starts at a default value of 0.3)

At each word encounter, a learner's response is simulated by calculating the probability of them being correct, which is based on the item's activation with respect to its **real** α. This calculation can be seen in formula 4.1, where:

- p_r is the probability that the given item will be recalled
- m_i is the activation of the item at the current encounter
- τ is the forgetting threshold
- s is a parameter used to smooth noise in the activation

$$p_r(m_i) = \frac{1}{1 + e^{\frac{\tau - m_i}{s}}} \tag{4.1}$$

The probability is then used to simulate a loaded coin flip, which produces either a correct or an incorrect response. The formula is shown in 4.2, where x is a randomly generated real number between 0.0 and 1.0. Depending on the simulated response, the model adjusts the **predicted** α in order to reflect whether the word is being learned. If the response was correct, the item's **predicted** α gets reduced by a certain amount, which results in a lower rate of decay for the item's activation and therefore causes the item to occur less often during study sessions. On the other hand, if the simulated response was incorrect, the **predicted** α is increased by a certain amount, increasing the item's activation decay and causing the item to be shown more often during studying.

$$f(x) = \begin{cases} false, & \text{if } x < p_r \\ true, & \text{otherwise} \end{cases} \tag{4.2}$$

In certain cases the simulated item interaction produces a result, which only requires a slight adjustment of the predicted α, which is reflected in formula 4.3. In this formula:

- α(pred) is the α predicted by the model
- $_r$(pred) is the recall probability, resulting from the predicted α
- 0.02 is the value chosen to adjust the α

Producing expected results (a correct response when the recall probability is very high and an incorrect one when it is very low) means that the model already represents the learner's knowledge of the word correctly and therefore its α only needs a slight adjustment in order to include the outcome of the interaction. Producing the opposite outcomes (a correct answer when the recall probability is very low and an incorrect one when it is very high) means that the model's prediction of the item's alpha is incorrect and therefore needs to be adjusted more significantly in order to account for the user's progress or regress with respect to learning the particular item.

$$\alpha(pred) = \begin{cases} \alpha(pred) - (1 - (p_r(pred)) \times (2 \times 0.02))), & \text{if } success \\ \alpha(pred) + (1 - (p_r(pred)) \times (2 \times 0.02))), & \text{otherwise} \end{cases} \tag{4.3}$$

4.2 Evaluation of performance

In order to evaluate the progress of the algorithm optimization process, we tracked the duration of the simulations as a measure of how fast the activation computations took place and the average α errors as a measure of how accurately the model adapts to the user's performance.

Since the simulations do not require actual user interaction with the words, they track the duration of word encounters and study sessions by using an internal clock. This means that the most time during a simulation is spent on the process of calculating item activations when choosing the next item to be presented, which makes the duration a suitable measure of the calculation's performance. However, due to the use of random number generation when assigning the real αs of each item, as well as when simulating the learner's response, we have taken the average duration of 50 simulations.

Keeping in mind that the model's goal is to adjust each item's predicted α based no the simulated learner interactions, the average α error (the average difference between each item's predicted α at the end of the simulation and its real one) can be considered a good measure of tracking the model's effectiveness in teaching words. Following the same logic as above, the average α error of 50 simulation was taken into account. This information not only allows comparison between the computational efficiency of the different algorithm implementations, but also reveals whether improvements in efficiency come at a cost of the algorithm's effectiveness.

4.3 Result comparison (Cached recursion)

The first performance comparison results can be observed below, in Table 4.1. The displayed results are based on simulating learning using a vocabulary list of the size, specified in the first column and tracking the duration of the simulation. While the total number of encounters remains relatively stable regardless of the vocabulary size, we can see a positive correlation between the average number of encounters per word and the duration of the simulation. This occurs due to the fact that the increasing number of encounters in a word's encounter history slows down the activation computation, since the algorithm needs to recurse for every single encounter. The real performance comparison, however, is evident in the last two rows of the table, which both use a vocabulary size of 37 for the simulation. The difference between them is whether values were cached during the recursion of the algorithm in order to avoid recalculating of already computed data (described in Chapter 2, Section 3.2). As seen by the duration of the simulation, caching the values considerably speeds up the calculation and reduces the number of recursive calls that would be made during the 15^th encounter from 32768 to 16.

Item count	Encounter count (total, average)	Encounter count (average per word)	Function call count (last encounter)	Duration (simulation)	Cached values
5	554	110.8	111	89.42s	TRUE
10	559	55.9	57	45.35s	TRUE
20	547	27.35	28	20.69s	TRUE
37	560	15.135	16	12.61s	TRUE
37	551	14.892	32768	2530.67s	FALSE

Table 4.1: Cached Recursion Test Results

4.4 Result comparison (Encounter duration)

In order to accurately represent learner interaction and record the spacing effect, the simulation needs to advance its internal clock after each word encounter by an amount, resembling how much time an actual learner would take in order to complete the encounter. To achieve this, Formula 4.4, which predicts the reaction time of the learner based on an item's activation, was used to calculate the amount of time it would take the learner to start interacting with the system after being exposed to the cue. Van Rijn et al. (2009). In this formula F is a scaling factor for the reaction time, f is the base reaction time and $m_{i,j}$ is the activation of item i at a given encounter j.

$$RT_{i,j} = \begin{cases} 3.788, & \text{if } m_{i,j} = -INF \\ Fe^{-m_{i,j}} + f, & \text{otherwise} \end{cases} \tag{4.4}$$

Since the learning process increases the reaction time, some padding was added in order to adjust the encounter duration to resemble that of a real life learner. The adjustment was based on the results, shown in Table 4.2, which show that a too long or too short encounter duration do not accurately reflect the benefit of the spacing effect in the learning process. A padding of 3 seconds on top of the predicted reaction time was chosen, as it produced the most impressive combination of α error (difference between real α and predicted α at the end of the simulation), α bias (which of the two α values tends to be higher when calculating the error) and simulation duration compared to the rest of the tests.

Padding (seconds)	Simulation duration (seconds, average)	Encounter count (total, average)	Alpha error (average)	Alpha error bias (average)
0.5	35.34021312	1860.54	0.03527160	-0.00513339
1.0	17.28412206	1373.08	0.03351216	-0.00488028
1.5	11.35507244	1097.92	0.03301070	-0.00329461
2.0	6.381862320	918.52	0.03140569	-0.00241167
2.5	4.796245000	792.76	0.03173698	-0.00071015
3.0	4.051432420	697.50	0.03106919	-0.00097124
3.5	3.133872360	624.24	0.03159645	-0.00054740
4.0	2.568388580	566.64	0.03229276	0.00077986
4.5	2.121258580	518.34	0.03333832	0.00166486
5.0	1.874361840	478.92	0.03299457	0.00270733
5.5	1.764593420	444.54	0.03491841	0.00289049
6.0	1.561120180	416.44	0.03496761	0.00265287
6.5	1.360384920	392.02	0.03649705	0.00458655

Table 4.2: Encounter Duration Test Results

4.5 Result comparison (α adjustment value)

Another important aspect of the model is how it adjusts the item αs after each encounter. While some implementations of a spacing effect system use the learner's reaction time in order to calculate an appropriate new value (Van Rijn et al. (2009), Koelewijn (2010), Nijboer (2011), Van Thiel (2010)), our implementation uses the item's recall probability as a way of scaling the adjustment value and applying it to the item's α depending on the outcome of the encounter. The α adjustment value is doubled, because of the fact that the model attempts to present words when they are at around 50% recall probability (defined by the forgetting threshold τ, which means that most of the time, the original adjustment value will be applied. The calculation can be seen in the formula 4.3.

Tables 4.3 and 4.4 show that an α adjustment value of 0.02 not only leads to a lower average α error, but also provides stable results due to the fact that the α error is not skewed in favor of either the predicted α or the real one (expressed as Alpha error bias).

Adjustment (value)	Simulation duration (seconds, average)	Encounter count (total, average)	Alpha error (average)	Alpha error bias (average)
0.02	3.92598758	698.68	0.035044720	-0.00472192
0.04	4.45247410	702.02	0.039973121	0.00621806
0.06	4.86823896	708.12	0.048540735	0.01590292
0.08	4.88345254	714.44	0.059730479	0.02860231
0.10	4.68681396	720.22	0.073491184	0.04398006

Table 4.3: Main Alpha Adjustment Value Test Results

Adjustment (value)	Simulation duration (seconds, average)	Encounter count (total, average)	Alpha error (average)	Alpha error bias (average)
0.01	3.54518940	699.08	0.02890271	-0.00337387
0.02	4.43280080	696.82	0.02996629	0.00067759
0.03	3.89115106	698.04	0.03399767	0.00552154

Table 4.4: Secondary Alpha Adjustment Value Test Results

4.6 Result comparison (Cached encounter histories)

The results of the implementation of the heuristic approach described in Chapter 3, Section 3.3 can be seen in Table 4.5. The displayed data can be used to compare the performance of the algorithm when the activations of previous item encounters are cached and then used for calculating the activation of the current encounter and when they are not. The results additionally show how the algorithm performs when the cache is updated at various intervals. The first thing to note is that caching the item activations not only provides very similar results in terms of learning effectiveness (as evident by the similar α error values and biases), but also significantly improves the computational performance of the algorithm (proved by the impressively shorter simulation duration). The second important aspect of the results is the cache update interval. As already mentioned, even though caching all previous activations improves the algorithm's computational efficiency, if the cache is never updated, there will come a point where the items' αs have been updated enough times for the data in the cache to become outdated and therefore lead to disastrous results. The fact that this claim is not supported by the data could be a result of each word in the given simulation having an average of 18 encounters and that possibly not being enough time for the cache to get outdated.

Keeping in mind that only one item's α changes after each item interaction, it would be reasonable to update the cache after each encounter (since only one item's activation would need to be recalculated). Table 4.5, however, also shows that updating the cache after every encounter provides higher α error and is not as effective at improving the algorithm's performance. The most effective implementation is the one where the cache only gets updated after the study session is over. This helps avoid any unwanted fluctuations in the αs during the sessions and does not allow for the cache to become out of date. The added benefit of this implementation is that since the cache update happens between study sessions, this does not influence the system's performance while it is in use and does not hinder user interaction.

Encounter history (cached?)	Cache update (when)	Simulation duration (seconds, average)	Encounter count (total, average)	Alpha error (average)	Alpha bias (average)
FALSE	-	22.5245806	697.94	0.03816317	-0.01081656
TRUE	-	4.10533710	700.34	0.03788089	-0.01028201
TRUE	immediately	5.28907236	697.54	0.03825728	-0.01085931
TRUE	post-session	4.40695388	700.12	0.03764329	-0.00930992

Table 4.5: Cached Encounter History Test Results

A more detailed overview of the simulation results with the finally optimized version of the Pavlik and Anderson learning algorithm (last row from Table 4.5) can be seen in Appendix A.

4.7 Result comparison (Learning duration)

As already mentioned, finding the most fitting combination of model parameters is highly dependent on various factors, among which is user performance. Since simulations provide unbiased results, they are suitable for a comparison between different model settings. The results of this comparison can be seen in Appendix B, sorted according to their α error.

Based on the data, the different parameters hierarchically influence the model's effectiveness. Since the model is based on the spacing effect, the most influential parameter is the inter-session time. Simulations with study sessions, which took place 24 hours apart are shown to consistently perform significantly better than all other simulations. An explanation for this could be the fact that less inter-session time means more time spent learning words, which introduces competitors in the learner's mind and can influence the items' activations in memory. Furthermore, the inter-session time is scaled down in order to accommodate the spacing effect, therefore it being too short could neglect the effect and therefore influence learning. The second pattern, which is observed is that simulations with a higher session count and medium session duration tend to show better results. This signifies that study sessions, which last less than 15 minutes are generally not suitable for learning, possibly due to the fact that the activations of the items being learned have not been strengthened enough in the learner's memory, which results in slower learning speed overall. On the other hand, having a single session which lasts around 60 minutes (an example of ”cramming”) also performs poorly due to the lack of spacing in the item encounters.

Chapter 5

Outlook

This thesis has provided an implementation which significantly improves the computational efficiency of the Pavlik and Anderson activation calculation algorithm by taking a heuristic approach based on the stable rate of forgetting of human memory. The algorithm can further be optimized by taking into account other of its properties or by implementing it in a system, where real learner input can be received and then analyzed in an attempt to discover patterns in the way actual humans process words.

5.1 Shortening the encounter history

Activation calculation becomes more expensive as the encounter history of an item grows. While there is a finite number of encounters, which a learner has to undergo in order to learn a particular word, the growing number will eventually influence the calculation to a point when its duration will impact the learner's interaction with the system. The solution to this problem is outside the scope of this thesis, but a possible approach would be to take advantage of the fact that the activation algorithm is designed in such a way as to consider the last encounter as the most impactful one for an item's current activation. This implies that all encounters before the last one provide progressively less impact to the item's current activation. In this case it is possible to attempt to further optimize the algorithm by empirically determining at what point in an item's encounter history can the activations of previous encounters be disregarded as they bring an insignificant contribution to the current activation. Doing this would allow a system to only calculate each item's activation based on a subset of its encounter history, which would naturally relieve the computation.

5.2 Analyzing real learner data

Simulations like the ones described in this thesis allow for optimization and testing of the algorithm itself, but an alternative approach is to base algorithm improvements on results, gathered from real learners. If the algorithm is implemented into a real system, the analysis of the gathered data could lead to useful information regarding the nature of human learning. Analysis could focus on finding patterns in the changes of item activations or α values after each word encounter. If such patterns are discovered, a possible optimization would be to cluster items according to the identified patters, which would allow for recalculation of whole clusters instead of individual items. Another analysis could attempt to cluster items according to the domain, in which they are used and attempt to directly link this to the domains, in which the language learner has previous knowledge in. If it is proven that a learner's previous knowledge in a domain affects how fast they learn other words from that domain, a system could take advantage of this information by predicting the learner's performance based on their previous knowledge and therefore use the predicted data to make even more efficient decisions as to the order in which the items should be presented.

Chapter 6

Conclusion

Researchers are faced not only with the difficulty of developing a conceptually complex enough theory to model human memory, but also one computationally efficient enough to provide a reasonable user experience and thus applicable in real life vocabulary learning systems. This thesis has proven that using estimated values is an adequate solution for achieving that. The resulting model still takes into account the impact of the spacing effect, which is evident by the results of the learning duration test in Appendix B (i.e. simulations with less spacing between study sessions generally performed poorly). The proposed implementation also achieved its goal of improving the computational efficiency of the original algorithm without compromising its effectiveness in vocabulary learning, which is supported by the fact that the α error values of the final implementation are closely comparable to those of the original algorithm implementation.

Technology enables such simulations and attempts at developing a model based on expected results, but its limits regarding the inability to fully represent how humans process language are still evident. Implementing the model into a real vocabulary learning system and collecting authentic language learner data not only makes further optimization based on patterns in the data possible, but also allows for more effective and adaptable learning. Whether it is the learner's familiarity with the domain in which a word is used or even orthographic, semantic or syntactic similarities between items presented shortly after one another, determining what makes learning more difficult allows for more accurate adaptations in a real system.

Employing the above suggestions should result in a highly efficient, adaptable and effective vocabulary learning system, which can be used as a basis for further implementations in the domain of discrete fact learning.

Appendix A

Final implementation results

This appendix contains the item-specific data, gathered by calculating the average of 50 simulations of the final optimized version of the Pavlik and Anderson learning algorithm. The implementation builds on the base formula for calculating item activations (Formula 2.1) by caching values during recursion (explained in Chapter 3, Section 3.2) and caching encounter activations during study sessions (explain in Chapter 3, Section 3.3).

Word	Encounter count (average)	Incorrect answers (percentage, average)	Alpha (real)	Alpha (predicted)
noodles	17.54	33.47983264	0.29524848	0.28785772
where	22.96	36.26889073	0.35149911	0.33767255
many	15.44	29.21323734	0.24795644	0.26511870
way	17.82	32.56175455	0.28869968	0.29017707
market	33.12	37.18038195	0.42841451	0.40526655
castle	19.04	34.08035733	0.32131938	0.30306387
group	21.4	35.35670417	0.33082526	0.32554260
restaurant	17.96	32.67135221	0.29574122	0.29328144
amazing	19.52	34.97322791	0.31363545	0.30958762
to visit	50.3	38.24197179	0.51870467	0.48669001
each	19.18	34.35055572	0.31023359	0.30852093
tree	18.22	33.90481349	0.30720531	0.29492939
British	26.12	37.23884195	0.38105765	0.36367329
adult	23.82	36.50340957	0.36267934	0.34791179
a day	14.88	29.16932621	0.24498433	0.25996955
open(from...to...)	16.94	31.84299664	0.28801516	0.28339746
furniture	12.64	23.95922688	0.18546626	0.23111772
a year	10.08	14.02387612	0.03424036	0.19294444
open	15.58	29.20736362	0.25710831	0.26877816
free time	12.22	22.34968919	0.17272323	0.22695964
canal	17.4	32.16698476	0.28683849	0.28735417
Chinese	28.74	36.48165684	0.40444202	0.38434795
stall	12.56	23.52746893	0.18401725	0.23226656
playing field	13.38	25.22488841	0.22296496	0.24180841
fancy	19.24	33.89328576	0.33149602	0.30983942
a week	11.86	20.81879231	0.17367400	0.22061395
to enjoy	24.46	35.89117006	0.36953320	0.35528614
best	14.84	27.72794243	0.25163889	0.26202301
wonderful	14.68	28.20075039	0.25321099	0.26166396
expensive	21.16	34.21212479	0.34806222	0.32847172
to add	27.96	36.08667367	0.40421330	0.37882536
boat	19.78	33.38146256	0.32701793	0.31475231
to join in	13.64	25.63385620	0.22608814	0.24977690
view	15.82	30.02128520	0.27546900	0.27597488
canoeing	11.04	16.69089799	0.08890224	0.21017056
flower	15.52	28.48940741	0.26727894	0.27134638
area	13.26	24.31336016	0.21747901	0.24556921

Table A.1: Final Implementation Per-Word Results

Appendix B

Session spacing test results

This appendix shows the top and bottom 10 results of running a simulation with the specified parameters.

Inter-session time (hours)	Study session (count)	Study session length (seconds)	Simulation duration (seconds, average)	Alpha error (average)	Alpha error bias (average)
24	4	1800	20.32074736	0.03436126	-0.00907207
24	3	1800	9.931795420	0.03447829	-0.00937304
24	4	2700	64.18474976	0.03485828	-0.00964994
24	3	3600	65.12516570	0.03491870	-0.01075798
24	4	3600	158.1656642	0.03492370	-0.00973170
24	3	2700	29.77002738	0.03508249	-0.01027251
2	4	3600	154.5556253	0.03573720	-0.01166020
4	4	2700	66.22802800	0.03574672	-0.01053655
24	2	3600	24.25220504	0.03580125	-0.01164732
24	4	900	3.157394600	0.03613755	-0.00557596
			. . .		
24	2	1800	4.025952640	0.03866311	-0.01145634
			. . .		
4	1	2700	1.860178920	0.04535570	-0.01242237
24	2	900	0.700354800	0.04807315	-0.00750375
4	2	900	0.703479280	0.04852638	-0.00962511
2	2	900	0.838126240	0.04892341	-0.01206129
4	1	1800	0.725312080	0.04956804	-0.01062272
2	1	1800	0.848957300	0.04974269	-0.01239915
24	1	1800	0.790586820	0.05019760	-0.01193468
24	1	900	0.182783100	0.06301916	-0.00923081
4	1	900	0.167927080	0.06326311	-0.01127533
2	1	900	0.165350120	0.06338653	-0.00920847

Table B.1: Learning Duration Test Results

Appendix C

Predefined constants

This appendix contains all predefined constants mentioned in this thesis.

Constant	Value	Meaning
α_d	0.3	Default starting α for all items
c	0.21	Rate of decay scale factor
τ	-0.8	Forgetting threshold
s	0.255	Recall probability noise reduction factor
δ	0.025	Inter-session time scale factor
F	1	Reaction time scale factor
f	0.3	Base reaction time

Table C.1: Learn Duration

References

C. Dilley. Application of personalized adaptive learner modeling to vocabulary learning software. *University of Tübingen*, 2017.

L. Koelewijn. Optimizing fact learning gains: Using personal parameter settings to improve the learning schedule. Master's thesis, University of Groningen, 2010.

S. Mittal. A survey of techniques for approximate computing. *Article in ACM Computing Surveys*, 2016.

M. Nijboer. Optimal fact learning: Applying presentation scheduling to realistic conditions. *Groningen, The Netherlands: Unpublished master's thesis, University of Groningen*, 2011.

P. I. Pavlik. Understanding and applying the dynamics of test practice and study practice. *Instructional Science*, 2007.

P. I. Pavlik and J. R. Anderson. Practice and forgetting effects on vocabulary memory: An activation-based model of the spacing effect. *Cognitive Science*, 2005.

F. Sense. Making the most of human memory: Studies on personalized fact-learning and visual working memory. *University of Groningen*, 2017.

C. E. L. Thomas H. Cormen and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 2000.

H. Van Rijn, L. Van Maanen, and M. Van Woudenberg. Passing the test: Improving learning gains by balancing spacing and testing effects. In *Proceedings of the 9th International Conference of Cognitive Modeling*, 2009.

W. Van Thiel. Optimize learning with reaction time based spacing, 2010.

M. Van Woudenberg. Optimal word pair learning in the short term: Using an activation based spacing model. *Unpublished master's thesis, University of Groningen*, 2008.