Neural Hash Functions

Anders Museth, CS undergrad at the University of California Berkeley

Extended Abstract

In this paper we investigate the use of Reinforcement Learning (RL) for optimization of hash functions. Generally the performance of hash functions is characterized by two metrics, namely its computational efficiency and the hash functions ability to avoid or minimize collisions. As indicated by its name, hash functions must satisfy the fundamental definition of mathematical functions, which is to say a given input key must be guaranteed to always map to the same output index or hash value. In other words, the hash function must be both deterministic and stateless, i.e. it cannot depend on history in any way or be truly random. In this paper we limit the scope of our investigations to exclusively focus on the collision-based performance metric. Specifically, we have chosen to narrow the scope of our study to hash functions applied to words in a dictionary, e.g. mappings from variable length character strings to indices into a fixed-size hash table. For reference we chose to compare the collision performance of the neural hash functions to that of the default string hash function available in Python. Our experiments demonstrated that we are strictly not able to improve the collision performance of the hash function by traditional means of Reinforcement Learning. This includes numerical experiments based on classic RL models like Policy Gradient, Actor Critic, and Soft Actor Critic. As a sanity check we added extra information to the state during RL learning, in which case the neural hash functions did indeed achieve fewer collisions. However, by adding this extra information we also violated the fundamental assumption that the key-index mapping of the hash function must be independent of its calling history, thereby rendering it useless for all but a few special applications where the calling sequence is known and fixed. However, if we change the desired goal slightly we were able to create a perfect hash function using reinforcement learning. Specifically, when we trained a Policy Gradient model on the same relatively small batch of words over many iterations we could train the actor to learn, or in a sense memorize, the optimal placement of words into the bins of a fixed-size hash table. In fact, this technique produced a perfect neural hash function, which is generally not achievable by our reference (expert) hash function implemented in Python3. More importantly, this (arguably small and overfitted) neural hash function is still capable of decent generalization when hashing words that were not part of the training set. In fact, we observed a collision performance of this neural hash function, on arbitrary words, that was comparable to Python's hash function. As such, we conclude that based on our explorations it does not appear to be feasible (or practical) for a deep reinforcement model to learn string hash functions in a way that consistently minimizes collisions for arbitrary inputs while still satisfying the fundamental requirements of being a true function. However, if we narrow the problem to allow the function to either use state information or optimize for a small subset of inputs then it is indeed possible to achieve fewer collisions and even a perfect hash function in some cases.

Introduction

Hash tables are ubiquitous in computer science, since on average they offer what at first glance might sound magical, namely constant-time random lookup into large data sets stored as a key-value pair. It achieves this desirable performance by using a (typically fixed-size) hash table with buckets that can be quickly accessed with a simple lookup into a linear array, which is itself constant-time. The promise of constant-time access into the hash table is therefore only strictly guaranteed if two or more key-value pairs are not mapped to the same bucket in the hash table. This phenomenon is called a collision of the hash function, and this is the source of optimization in this paper. However, it is also important to emphasize another desirable property that good hash functions need to satisfy, which is fast (ideally constant-time) computation. While the property of computational efficiency is indeed very important, we will focus our attention on collisions since it is arguably a prerequisite of any high-performance hash function. In other words, if a given hash function cannot minimize collisions, it is basically irrelevant how fast it is to evaluate since an excessive amount of collisions will introduce other computational bottlenecks that impairs the overall computational performance of accessing key-value pairs in the hash table.

In general a hash is a function that takes a variable sized input and outputs a seemingly random but fixed length bit string which is easily converted to an index in a given range, e.g. using the bitwise AND operator, e.g. n & m where n is a random integer and m is the size-1 of the hash table. However, the application of these hash functions requires them to be deterministic in the sense that the same input (key) is guaranteed to map to the same output every time the hash is evaluated - regardless of its calling history. As mentioned above hash functions are a fundamental component in the implementation of hash tables, and one of their most sought after properties is the hash function’s ability to minimize (or ideally avoid) collisions between input keys. If a hash function completely avoids collisions it is called a *perfect hash function*. So, in the language of mathematics a perfect hash function is simply an injective function, i.e. one-to-one. Unfortunately perfect hash functions are often not achievable in practice since the number of input keys typically exceeds the number of allowed output indices, which is defined by the size of the hash table. In other words, for most applications of hash functions collisions are simply unavoidable, so the problem becomes to minimize them as opposed to eliminating them.

The overarching theme of this paper is to explore the possibility of using Reinforcement Learning for developing, i.e. learning, neural hash functions that satisfy the desirable property of producing few collisions, while simultaneously being deterministic. Given the fact that the optimal output for a given input is in general ill defined, the use of supervised machine learning is challenging. Since our metric of success is based on collisions amongst specific input keys, and the set of possible input keys are application specific, so are hash functions. Consequently we need to limit our scope of exploration to a specific application of hash functions. For the sake of simplicity we have chosen hashing of words found in an English dictionary. That is, the input keys are string characters of variable length and the outputs integers in a fixed range. Also, note that for our choice of application the value in the key-value pair of the hash table is just the key,

i.e. dictionary word, itself, which is just an implementation detail and more to the point inconsequential to the conclusions that we draw for the experiments.

Lets conclude this introduction by summarizing the problem statement of our paper as follows: Explore the use of Reinforcement Learning to derive a neural hash function that minimizes the number of collisions when applied to words in a given dictionary. Our metric of success is judged relative to the number of collisions produced by Python's default string hash function, and specifically we will ignore their computational efficiencies. With this (admittedly narrow) goal [1] in mind we will spend the remainder of this paper explaining how we successfully achieved it.

Previous work

It should come as no surprise that there is a large body of previous work on both optimizations of hash functions [1] and Reinforcement Learning [2]. However, we were indeed surprised to discover that there appears to be little existing work that explores the intersection of these two subject matters. Admittedly this was initially a motivating factor for the author, but in hindsight it is clearly also an indication of the challenges posed when attempting to improve hash function by means of RL models. In fact, the only examples we could find are [3] that explored the application of deep RL on hashing of 2D images. For a survey on Deep Hashing Methods please see [3].

It appears that most previous work on the application of neural networks to hashing primarily focuses on images and aims to preserve similarities between images for hash-based retrieval. In other words, these applications are based on the core idea of mapping images to "hash codes" in such a way that similar images produce similar though not identical hashes [4, 5, 6, 11]. However, in our case we want to optimize for reducing collisions meaning that the similarities between the data would be disregarded.

So, to the best of our knowledge the exact problem as stated in the previous section has not been published before, which satisfies a requirement for this research paper.

Methods

We used words from "Moby Word Lists" which is a list of 466550 words [7] as the data to optimally hash. Then we created a hash table state, which keeps track of which bin the actor has placed a word, and returns a corresponding reward. Specifically the agent receives a positive reward (10 units) when it hashes a word into an empty bin and a negative reward when placing the word into an occupied bin (-10 units). We experimented with a few other reward functions, but found this model to be the most successful.

To evaluate the performance of our neural hash function we decided to use Python3's string hash function as our benchmark. This hash function achieves an average of 369 collisions when placing 1000 words into 1000 bins.

The first test uses a simple supervised learning model to try and replicate Python3's default hash function. The next test was to use Policy Gradient to examine whether or not the model could improve. Policy Gradient is a relatively simple method in which we optimize for a policy that the agent follows to maximize the expected reward. Here we will update our MLP using the negative value of the following equation as our loss (this is because we want to maximize the reward) [8].

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

Next we used a more advanced algorithm, namely Actor Critic. This is where we train two separate neural networks, dubbed the actor and critic, to learn an optimal policy. The actor learns a policy while the critic learns a so-called value function that approximates the value of a given state. This is accomplished by updating our actor MLP using the following [9]:

$$\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \hat{V}_\phi^\pi(\mathbf{s}_i') - \hat{V}_\phi^\pi(\mathbf{s}_i)$$

$$\nabla_\theta J(\theta) \approx \textstyle\sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i | \mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$$

The final reinforcement learning model that we tested was Soft Actor Critic. This is a variation of the Actor Critic model that maximizes expected entropy of the policy along with the policy. This encourages exploration and helps create stability when learning. This is done by adding an entropy regulation term to our objective function as shown below [10]:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[r(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathcal{H}(\pi(\,\cdot\,|\mathbf{s}_t)) \right]$$

For a full description of Soft Actor Critic we refer the reader to [10].

Note that these reinforcement models needed to be altered slightly since in the traditional setting they have the models predict a distribution of actions to take. However, in our application to hash functions the actions need to be discrete and deterministic to allow for data retrieval into the hash table. To achieve this for the Policy Gradient, Actor Critic and Soft Actor Critic models the actor simply uses the action with the highest probability for a given state. This can be done since the action space is discrete rather than continuous. The action space being discrete also

allows for us to give the neural network actions that we know will produce good rewards in hopes that it will learn what actions are beneficial.

Results

In this section we will discuss the results and thought processes behind the various experiments we conducted in an attempt to achieve the goal outlined in the introduction. However, as a prelude we want to emphasize that these experiments were initially guided by a more ambitious goal, namely to find a replacement for Python’s string hash function. That is, rather than developing a neural hash function for a specific set of words, as defined by a given dictionary, we were attempting to find a general hash function that would outperform Python's on a general (vs specific) set of words. As we shall see, this turned out to be challenging and we therefore decided to narrow the scope to a fixed input. While this is arguably a much simpler goal, since we’re allowing the neural network to memorize, it is still a problem with substantial practical value, e.g. to allow for collision-free random access into a fixed data-set, like a dictionary.

Our first experiment was to investigate how applicable supervised machine learning is to our problem space. Since supervised ML is fundamentally relying on labeled inputs this experiment attempted to train a MLP on our choice of “expert”. That is to say, train the MLP to emulate the behavior of the default string-hash function provided by Python 3. Specifically we used a MLP with 2 hidden layers of size 200 and 300 respectively. Then we trained it on all 466550 words [8] in an attempt to classify these words into 1000 bins computed by Python’s hash function. As shown in Figure 1 we found that this supervised neural network was incapable of learning much of anything, which is evident by the fact that the loss is oscillating randomly as a function of the epochs. This is almost certainly because the neural network is trying to replicate the pseudo random behavior for a classification type problem. More precisely, the large set of inputs (466550 words) is mapped to a few indices (1000) in a process that mimics a random number generator, which is understandably challenging given the limited capacity of the MLP.

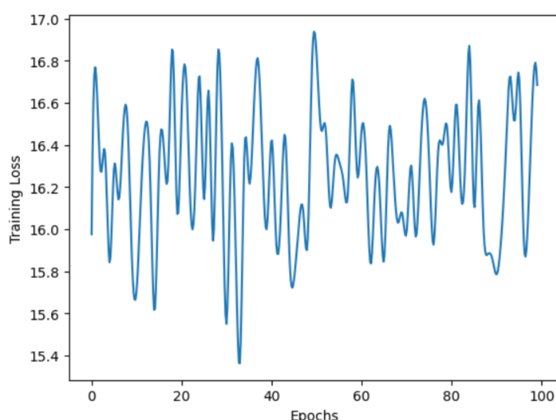

Figure 1: Training loss as a function of epochs when using supervised machine learning to replicate the classification of Python’s string hash function.

Since supervised (or imitation) learning was unsuccessful we decided as the next experiment to use Reinforcement Learning in the form of Policy Gradient. Thus, we implemented a basic

form of policy gradient where each state is defined as the number of words in each bin of a hash table, i.e. the number of collisions in the hash table. The observation given to the agent was the characters of the word, and the action returned by the neural network corresponded to the choice of bin in which to place this word. Next, we created a reward function that gave high rewards for placing a word into an empty bin and a negative reward when placing a word into a bin that already contains words. The MLP used has two hidden layers both of size 300.

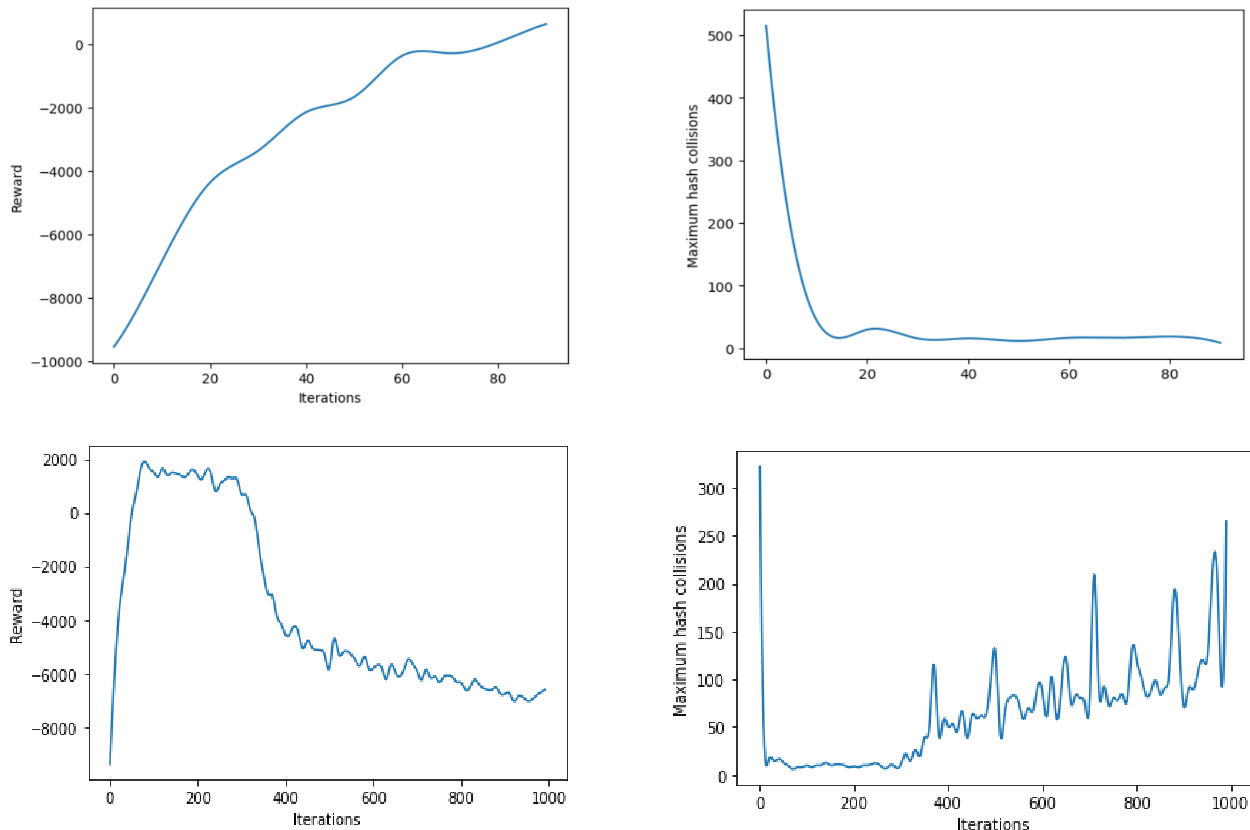

Figure 2: Results from using Policy Gradient model hashing 1000 words into 1000 bins after 100 iterations (top row) and 1000 iterations (bottom row). The figures to the left show the reward the agent receives at each iteration. The figures to the right depict the maximum collisions in one bin achieved by the agent at each iteration.

As shown in figure 2 (top), it initially appears that the policy gradient is actually learning. Figure 2 (top left) shows the reward achieved by the agent when hashing 1000 words into 1000 bins of a hash table and figure 2 (top right) shows the corresponding maximum number of collisions in the bins after the insertion of 1000 words. However, as shown in Figure 2 (bottom) we discovered that as the neural network continues to train it eventually stops improving and actually gets worse. A plausible explanation for this behavior is that the initial iterations produce seemingly random rewards causing the actor to put the words into random bins which in turn leads to higher rewards. However, as the model attempts to optimize it fails to understand what causes high rewards and tries to avoid actions that produce low rewards. As a consequence the model ends up clustering words into a few bins.

In an effort to address the clustering problem described above, we next experimented with an Actor Critic model, which is expected to have more expressive power than the Policy Gradient model since it introduces a second neural network (dubbed the critic) that models a value function which criticizes the actions made by the actor. In other words, we hope that the addition of the critic neural network may help the actor to choose better actions and prevent the actor from getting stuck. However, the results, shown in Figure 3, are similar (or worse) than those obtained with the policy gradient model. This suggests that the actor critic model is not capable of learning the value function (or optimal policy) due to the seemingly random nature of the problem. We speculate that this is a consequence of the fact that the state (i.e. the word) has little information about the current state of the hash table.

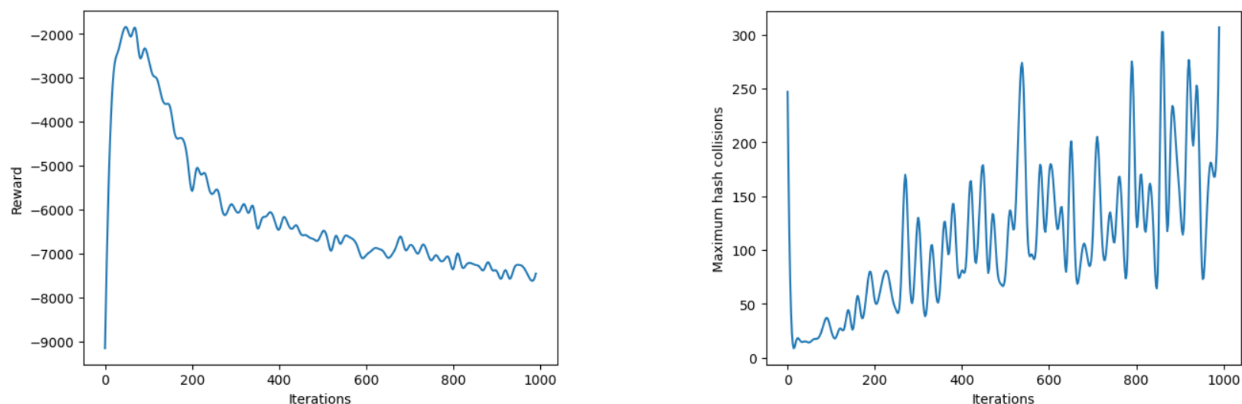

Figure 3: Results using the Actor Critic model hashing 1000 words into 1000 bins. The left graphs depict the reward the agent receives (left) and the maximum collisions in one bin achieved by the agent (right) at each iteration.

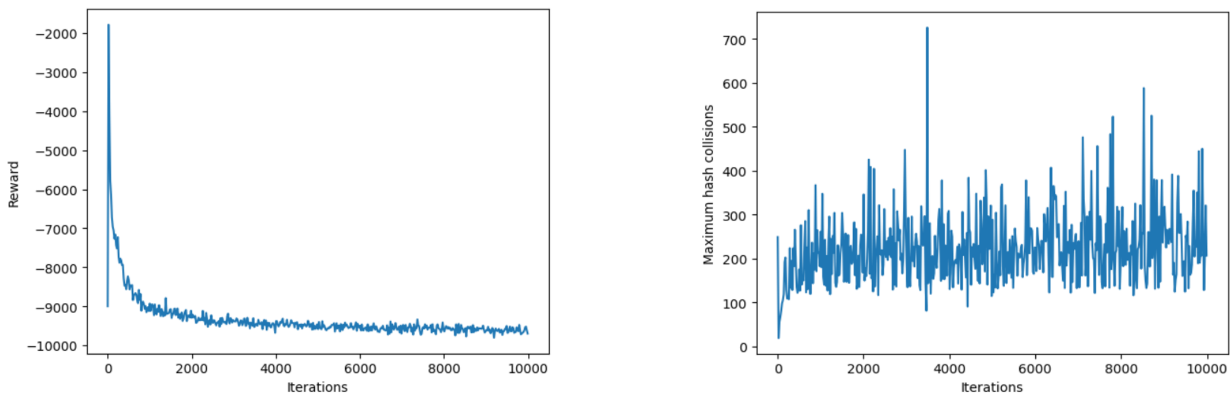

Figure 4: Soft Actor Critic (SAC) of placing 1000 words into 1000 bins. The graphs visualize the reward the agent receives (left) and the maximum collisions in one bin achieved by the agent (right) at each iteration.

The next experiment was to investigate if Soft Actor Critic could perform better due to its entropy regularization. However, as shown in figure 4 SAC also failed to produce a uniform distribution of words in the hash table. In fact, the SAC model performs very similarly to the Actor Critic model, which suggests that the entropy regulation of the model has little effect.

The next experiment was to revisit the Policy Gradient model, but this time feed the model additional information about the state. Specifically we added information about the current state of the hash table (what bins were occupied) to the observations that were fed to the neural network. In other words, this allows the network to “see” both the word and the bins. While this approach clearly violates the fundamental requirement that the neural hash function should always proceed the same hash for a given word, regardless of its history, we think of this experiment as a sanity-check. As predicted, the additional state information does indeed improve the performance significantly. The top row of Figure 5 shows the increasing reward and decreasing maximum hash collisions for an experiment where only 100 words are hashed into 100 bins, and the bottom row shows a similar experiment with 1000 bins with 1000 words. As expected these Figures also show that it takes much longer to learn the hash function for the larger problem size due to the correspondingly larger feature space for the input, which in turn requires many more parameters to train.

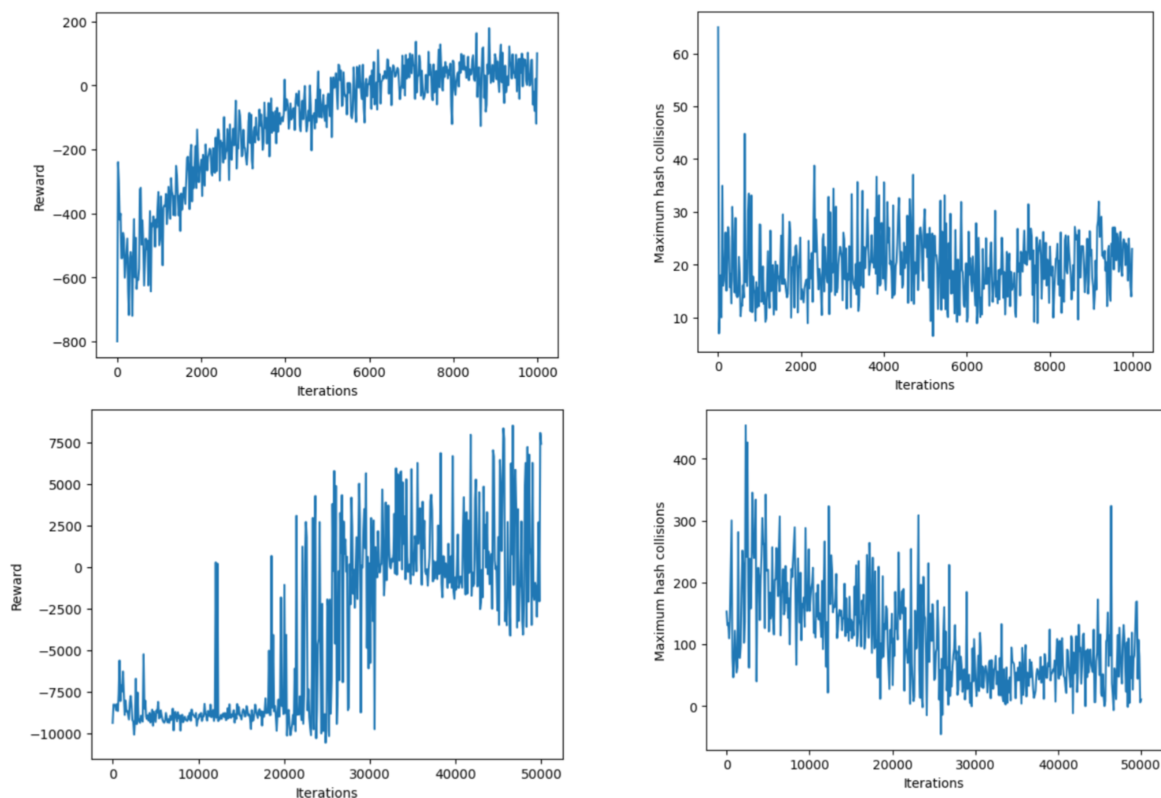

Figure 5: Policy Gradient model, where the model receives information of the current state of the hash table. The top graphs show the results of placing 100 words in 100 bins and the bottom graphs the results of placing 1000 words into 1000 bins. The reward the actor received at each iteration is depicted to the left and the maximum collisions in one bin achieved by the actor model at each iteration to the right.

So far our experiments have struggled to produce neural hash functions that minimize collisions while simultaneously generalizing to arbitrary words and still be independent of its calling history. Consequently, we decided to narrow the scope of our problem statement and investigate the possibility of solving a problem with a fixed known set of inputs. That is, produce a neural hash function that is deterministic and minimizes collision when hashing words from a given dictionary. Interestingly this has been our setup all along but so far we have to make use of the fact that the input is fixed. So, while this of course changes the game significantly in the sense that we will benefit from situations when the neural network is memorizing (i.e overfitting), we will still argue that the new narrower goal has real practical value. Thus, our final experiment was to see if the Policy Gradient model could memorize a hash for a given set of words. Specifically, during training we repeatedly feed the network the same batch of words. We were excited to observe that the model does indeed manage to learn (or more precisely memorize) the classification of the words resulting in a perfect neural hash function with no collisions. This can be seen in Figure 6 and 7 for two different batch sizes. Another interesting property of the resulting neural hash function, is that it actually generalizes quite well when hashing words that were not in the training set. In fact, the performance in terms of minimizing collisions on random (unseen) words is remarkably similar to that of the Python3 hash! This is most likely because the actor places the words into seemingly random bins, which is exactly the desired behavior for a hash function.

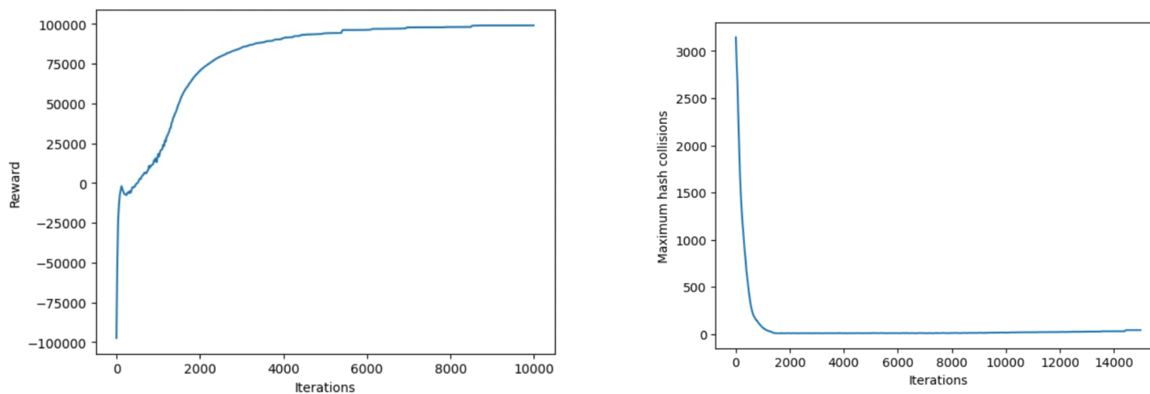

Figure 6: Policy Gradient model training on a given set of 10,000 words.The left figure shows the reward achieved by the policy gradient actor after each iteration and the right figure the maximum collisions in one bin achieved by the policy gradient actor at each iteration.

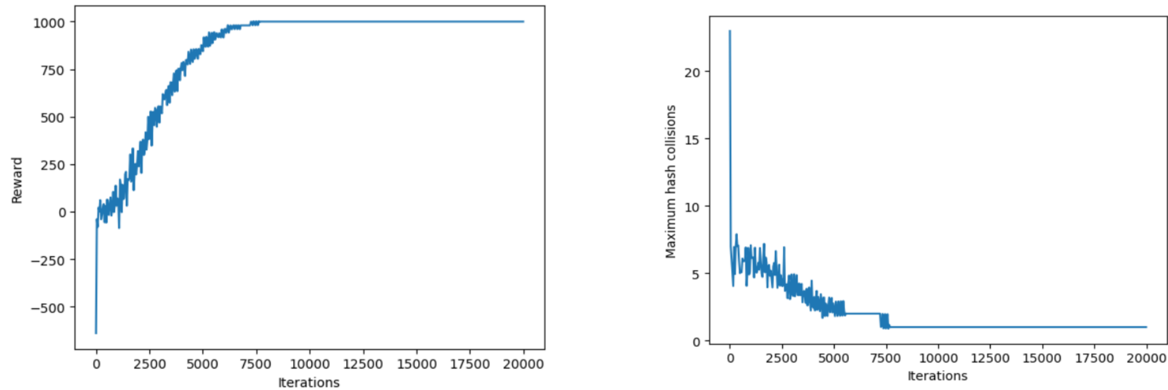

Figure 7: The left figure shows the reward achieved by the policy gradient actor after each iteration when trained on the same 3,000. The right figure shows what the maximum collisions in one bin achieved by the policy gradient actor at each iteration of placing the same 3,000 words into 1,000 bins (1000 words at each iteration).

While we are pleased with the encouraging results from our last experiment there are of course some limitations that need to be highlighted. The most significant limitation is arguably the fact that the training scales unfavorably with the problem size. More to the point, it took 5 hours and 40 minutes to train neural hash function on a batch size of 10,000 words. In fact, it was so slow that we gave up on attempting to train on the entire dictionary with 466550 words. However, training times are not the only concern, since inferencing also turned out to be slow. Our benchmark tests showed that Python's hash function takes only 79 ms to hash 300,000 words while our novel neural hash function takes 4988 ms to hash the same number of words. That's a staggering performance difference of 63X in favor of Python's hash function. While we are pretty confident we can improve the performance of the neural hash function, we doubt it can surpass or even close the gap rendering Python's hash much faster. Nevertheless, for small sets of input data the neural hash function does offer a significant advantage since, unlike the Python hash, it is perfect, i.e. completely collison free. Thus, we speculate that the neural hash function only has practical value for very specific use cases, e.g. where collisions are computationally expensive to resolve.

Future work

While we have conducted many experiments in an attempt to develop a neural hash function, our exploration is by no means exhaustive. For instance, as future work we propose to train a neural network to learn embeddings of the words, which might allow the agent to learn more about the similarities and relationship between the different words/datapoints. In general, embeddings map data from a high-dimensional space to a low-dimensional space in such a way that the relationships between the data points are preserved.

Another interesting angle might be to employ meta-reinforcement learning in an attempt to reduce the training time for each specific use case while hopefully also lower the number of parameters needed (i.e. network capacity). This might significantly improve the practicality of our neural hash functions since fewer parameters will speed up both training and inference times. Along the same lines we would like to optimize our models by testing new combinations of hyper parameters and improve the action selection process.

Conclusion

We have presented results from several numerical experiments that aimed at using Reinforcement Learning to teach an agent how to minimize collision in a hash function. Specifically we experimented with RL models like Policy Gradient, Actor Critic, and SAC. While some models arguably performed better than others we conclude that these traditional models of Reinforcement Learning were in general not able to produce neural hash functions that consistently outperform Python's hash function, when the performance is measured on collisions from hashing of arbitrary words. This result is not surprising since it essentially amounts to asking a neural network to derive optimal policies that are known to be pseudo random uniform distributions. Put differently, since we are asking the RL model to perform seemingly random actions, it fails to learn any meaningful relationship between the words and their optimal placement in a hash table. This explanation is strengthened by the improvements we observed after including additional state information, at the cost of violating the fundamental assumptions of a true hash function, rendering this approach all but impractical. However, if we limit the scope to that of hashing a relatively small set of fixed words we did indeed observe some promising results. This scenario allowed the neural network to effectively memorize the optimal hashing resulting in perfect hash functions for up to 10000 words. Equally exciting is the fact that these neural hash functions seemed to perform approximately as well as the reference hash function in Python. So, while these neural hash functions are not (on average) better in terms of minimizing collisions for arbitrary inputs, they were perfect for a fixed input and comparable for random inputs. This is an encouraging result and as such we conclude that if (and this is a big if) we can optimize both the training and inference times of the neural hash functions they might indeed have practical value for certain hashing applications.

Acknowledgement

We would like to thank Professor Sergey Levine and the GSIs of CS285 for guidance and feedback on early versions of this report, as well as my parents for general support and proofreading of the final version.

References

1. Anonymous feedback on milestone report: “Initial results seem insightful. If it seems unlikely to be able to optimize for both collisions and computational speed, maybe focus on only one of these properties for the RL-based algorithm.”
2. L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research*, Vol. 4, pp. 237–285, 1996.
3. Luo, Xiao and Wang, Haixin and Wu, Daqing and Chen, Chong and Deng, Minghua and Huang, Jianqiang and Hua, Xian-Sheng, “A Survey on Deep Hashing Methods”, *ACM Transactions on Knowledge Discovery from Data,* Accepted on April 2022.
4. Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen, A Survey on Learning to Hash., *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. 13, No. 9, April 2017.
5. Singh, Avantika and Gupta, Shaifu, “Learning to Hash: A Comprehensive Survey of Deep Learning-Based Hashing Methods”, *Knowledge and Information Systems*, page 2565–2597, Vol. 64, Oct 2022.
6. X. Yao, M. Wang, W. Zhou and H. Li, "Hash Bit Selection With Reinforcement Learning for Image Retrieval," in *IEEE Transactions on Multimedia*, 2022.
7. https://github.com/dwyl/english-words Git hub to text file with all words
8. Sergey Levine, CS285, lecture 5, slide 17, 2022.
9. Sergey Levine, CS285, lecture 6, slide 12, 2022.
10. Haarnoja T., Zhou A., Abbeel P., and Levine S. , Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, *arXiv:*1801.01290, 2018.
11. Jinhui Y, Huan L, Shengyong D, Wei L, Supervised Hashing for Image Retrieval via Image Representation, aaai.v28i1.8952, 2018.