Solving Classical Problems using Machine Learning

Manas Mulpuri[*]
mulpuri.m@iitgn.ac.in
IIT Gandhinagar

G. Manoj Taraka Ramarao[*]
manoj.gtr@iitgn.ac.in
IIT Gandhinagar

Krishna Mohan P[*]
krishna.mp@iitgn.ac.in
IIT Gandhinagar

Harsh Patel
harsh.patel@iitgn.ac.in
IIT Gandhinagar

Shivam Sahni
shivam.sahni@iitgn.ac.in
IIT Gandhinagar

ABSTRACT

Machine learning approaches are being used widely in many fields ranging from the healthcare industry to the manufacturing industry. One of the non-intuitive applications of machine learning can be observed in the improvement of algorithms of classical problems where we improve the performance of an algorithm using some intuitive or clever predictions. These predictions often exempts the need for exhaustive searching, thereby optimizing the time complexity of the algorithm.

This project explores the usage of Machine Learning and Reinforcement Learning on classical problems. We worked on Binary Search [4], Bloom Filters [4] and the Secretary Problem [6].

KEYWORDS

Machine Learning, Neural Networks, Reinforcement Learning

Our work can be found at this Github Link

1 RELATED WORK

There are many problems in this field that have been shown to perform better using machine learning algorithms rather than their traditional counterparts. Some of the notable ones are:

- Caching with predictions is an optimization problem that deals with minimizing number of page faults over a sequence of page requests. In this paper [3] ML is used to design a predictor that can predict the next arrival time of a page.
- Scheduling with predictions is an optimization problem that deals with scheduling jobs in a processor in order to maximize the usage of the processor. In this paper [4] ML is used in this problem to design a predictor that can predict the job length.

2 BINARY SEARCH

Given a sorted array A of n elements and query element q, we need to find whether q is present in the array or not. The binary search algorithm starts searching from the middle of the array and reduces the search space to find the element, irrespective of the location of the query element. Rather than starting from the middle of the array every time, if we can start from the position nearer to the query element, then we can find the query element faster. We can use Machine Learning to find the nearer element.

[*]All the authors contributed equally to this research.

2.1 ML-Based Binary Search

Suppose we have a predictor h that could determine the position of an element that is nearer or equal to the query element. Let $h[q]$ be the predicted position of the query element. Suppose if the element at the predicted position is less than the query element, then we probe elements at the positions $h[q]+2$, $h[q]+4$, $h[q]+8$, and so on until we get an element that is greater or equal to the query element. If the element at the predicted position is greater to the query element, then we keep probing elements at the positions $h[q]-2$, $h[q]-4$, $h[q]-8$, ... till we get an element that is lesser or equal to the query element. We do a binary search in the reduced space interval to find the query element. Using this logic, we can beat classical binary search in terms of number of array accesses if our predictor is good. However, we are neglecting the time complexity of the predictor(h).

2.2 Implementation

In our case, the predictor(h) is a multi-layer perceptron model. We constructed different MLP models where the number of hidden layers are varied from zero to two. The input layer of all the earlier-mentioned neural networks contains $n+1$ neurons. We give a sorted array of size n where all of the elements in the array are between 0 and 1, and query element q as input to the neural network. The output layer of the contains one neuron returns a value between 0 and 1 that determines the $\frac{predicted\ index}{n}$ of the position of the query element. We used the Adam optimizer, RMSE as a loss function and the ReLU activation function in all of the neural networks. We ran these neural networks on samples from different distributions such as uniform, normal, exponential, and the mix of these three distributions with different array sizes.

2.3 Results

Tables 1 to 3, represents tables showing the performance of our neural network with normal binary search when the input data is a mix of uniform, normal and exponential distributions.

As seen in the tables, we can observe that as n increases, the predictors performance decreases for a fixed number of hidden

	Average Number of Steps			
	Uniform	**Normal**	**Exponential**	**Mixed**
BS	4.8676	4.8406	4.868	4.8692
HL=0	3.6516	4.2116	5.4792	4.7108
HL=1	3.666	3.508	3.8576	4.4896

Table 1: n(array size) = 50, BS: Binary Search, HL: Number of Hidden Layers in Predictor

	Average Number of Steps			
	Uniform	**Normal**	**Exponential**	**Mixed**
BS	5.791	5.8104	5.7902	5.792
HL=0	4.5604	5.2882	6.899	6.1286
HL=1	4.5604	5.1736	4.3016	5.1242
HL=2	4.1398	4.1992	3.4616	5.3862

Table 2: n(array size) = 100, BS: Binary Search, HL: Number of Hidden Layers in Predictor

	Average Number of Steps			
	Uniform	**Normal**	**Exponential**	**Mixed**
BS	8.9914	8.986	8.9918	9.0004
HL=0	7.0568	10.3204	12.8356	12.2916
HL=1	6.9124	9.7422	7.0298	9.878
HL=2	7.288	7.8852	6.6274	7.9124

Table 3: n(array size) = 1000, BS: Binary Search, HL: Number of Hidden Layers in Predictor

layers, and as the number of hidden layers increase for a fixed n, the predictors perform better. The predictors are also able to beat binary search as we increase the number of hidden layers.

3 BLOOM FILTER

Bloom filter is a special type of data structure used to answer set membership queries. It consists of an array of m bits initially set to zero and $(h_1, h_2, h_3..h_k)$ k distinct independent random hash functions that give values ranging from 0 to m-1. It is used to represent a set $S = s_1, s_2, s_3, s_4.., s_n$ that consists of n elements. Each element from the set is passed through the bloom filter. The values of the array at the positions that we obtain from the hash values corresponding to the k hash functions of each element are changed from 0 to 1. To check the membership of an element p in the set S, we need to check whether all $h_i[p] = 1, 1 <= i <= k$. The problem with the bloom filter is that it also classifies the elements that are actually absent as present, causing false positives. When more entries in the bloom array are set to 1, there might be a chance of getting more false positives.

3.1 Implementing Learned Bloom Filter and Sandwich Bloom Filter Models

We implemented two models where in one model we used a neural network as a prefilter cascaded with standard bloom filter as a backup filter and the second model, the sandwich model is an improved variation of a learned bloom filter with a learned oracle (neural network) between two standard bloom filters.

We train a neural network in both of the models on an input set to predict the probability of an element being in the set. We predict the probability that each element is in the set, and if the predicted probability is high(>threshold), we conclude that the element is in the set. The ones that are not accepted in the neural network are added to the bloom filter, which reduces the number of entries in the bloom filter, reducing the number of false positives.

In the second model, the initial bloom filter in the sandwich model does not pass true negatives to the learned oracle. The second bloom filter in the sandwich model receives fewer elements from the learned oracle than the learned bloom filter. As a result, It yields better performance than a learned filter with one standard bloom filter.

We constructed two neural networks that consist of two and four hidden layers. We used the ReLU activation function, binary cross entropy loss function and Adam optimiser in the two neural networks.

Input Data. We used data that consists of both good and bad URLs. Each URL in the dataset is converted into a sequence of integers, by mapping every character in the URL to an integer. We padded every converted URL with the required number of zeros to maintain the same input length while feeding to the neural network. The neural network's output consists of one neuron that determines whether the URL is good or bad. We send this preprocessed data into the above mentioned two models.

3.2 Results

Figure 1 and Figure 2 compare the accuracy of different models as the Bloom Filter Error rate increases. The Sandwich Bloom Filter performs better than the Learned Bloom Filter in both cases, that is, when the number of hidden layers are 2 and 4. It can be seen that both outperform a normal Bloom Filter. Also as the number of hidden layers is increasing we can see that models are almost classify all of the URLs properly(near 100% accuracy).

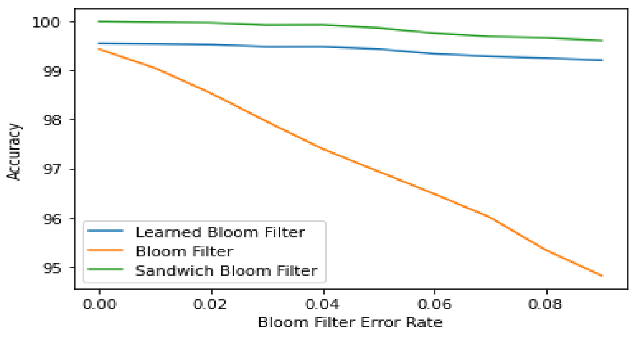

Figure 1: Performance of Models, 2 Hidden Layers in Neural Network

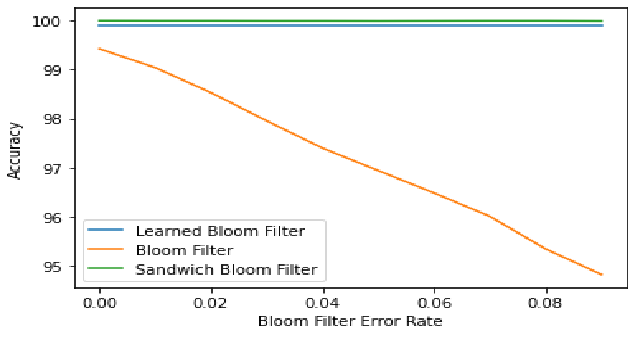

Figure 2: Performance of Models, 4 Hidden Layers in Neural Network

Figure 3 and Figure 4 represents the number of URLs accepted by the predictor and by the bloom filter in the Learned Bloom Filter model. Compared to 2 hidden layers, when there are 4 hidden layers in the neural network almost all of the "good" URLs are being accepted by the neural network with a high confidence, so the backup bloom filter is built on a very small number of "good" URLs decreasing the false positives.

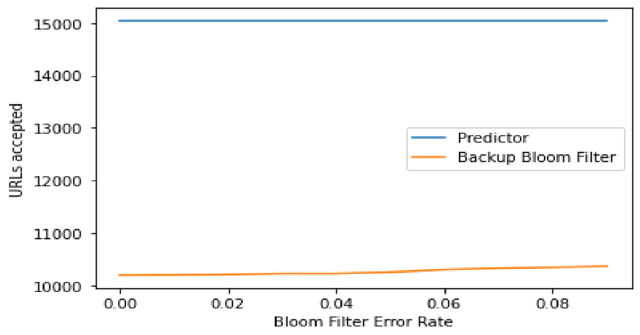

Figure 3: URLs accepted by Learned Bloom Filter, 2 Hidden Layers in Neural Network

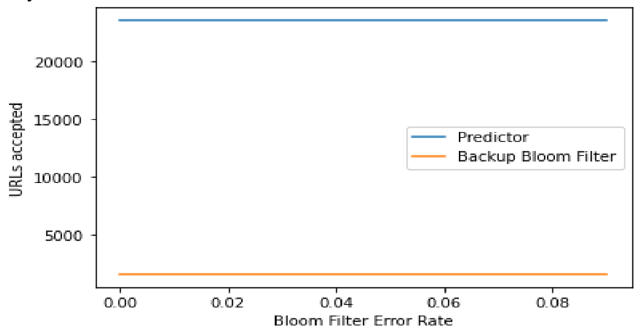

Figure 4: URLs accepted by Learned Bloom Filter, 4 Hidden Layers in Neural Network

4 SECRETARY PROBLEM

The problem consists of a recruiter who wants to hire a candidate for a particular job. Candidates give their interview one after another and each candidate has a score and if a candidate is accepted then the process ends otherwise the candidate is rejected. If there are no candidates left then the last candidate gets automatically accepted. The goal of the recruiter is to pick the best candidate for the job. There is no correlation between the candidate scores and the order in which they are interviewed.[6] [2]

The optimal strategy for the Secretary problem follows the 1/e - law of the best choice [1]. Suppose there are total of N candidates in the list. According to the optimal strategy, you reject the first N/e candidates by noting their maximum score. This step intuitively gives you a chance to know about the trend of the candidate scores. For the next candidates, you accept the candidate if the candidate's score is greater than the noted maximum score. Otherwise, reject the candidate.

We took the variation of the secretary problem where the goal is to maximize the average score of the hired candidates and used Q-learning and Deep Q-learning for the optimization.

4.1 Q-learning

Q-learning is a reinforcement learning algorithm used to find the optimal action-selection policy using a lookup table like structure called the Q-Table. The Q-Table stores the values of quantities called the Q-Function values. The Q-Function values are a function of the current state and the action taken in the current state. The Q-values are our current estimate of the sum of the agent's total future rewards if it performs a particular action in the current state. The agent tunes the Q-values based on current Q-values and the rewards obtained from moving through the states during the training process. After an optimal policy is learned, in any state the agent finds the best action to take in any state by choosing the action with the highest Q-value. The Q-value updates are done according to

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha.(r_t + \gamma.\max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Theoretically, given infinite time and space, Q-learning can learn the optimal action-selection policy for any given environment.

4.2 Deep Q-learning

Deep Q-Learning is a modified version of the Q-Learning algorithm in which the Q-Table is replaced by a neural network to compute the best action to take in any given state. In other words the neural network is used to approximate the Q-Function discussed in Q-Learning. The shift to neural networks is intuitive since the Q-Table is computationally feasible only for a small number of states and actions. We are no longer restricted to a small number of states and actions as the neural network predicts the best action to take based on the input observation(current state). The neural network is trained by comparing its value with the value obtained from the Bellman equation

$$Q(s, a) = r(s, a) + \gamma.\max_a Q(s', a)$$

by updating its parameters to minimize the loss function. In the above equation $Q(s, a)$ is the function that the neural network tries to predict.[5]

4.3 Implementation

In general, the agent makes a decision from a state in the environment, and the environment provides the resulting state for that decision and a reward to the agent in the RL solution. The agent tries to maximize the reward. The states represent the situation of the agent and the action space determines the possible actions that the agent can do. In our case, the state has three entities they are

- The fraction of candidates interviewed till then.
- The maximum score of the candidates till then that the agent has seen.
- The current score of the candidate.

The action space is either to accept the candidate and terminate the process or reject the candidate. The reward received by the agent for rejecting a candidate is zero and for accepting is the score of the candidate.

We made the environment needed using the Open AI Safety Gym. The candidate scores were between 0 and 1 and the total number of candidates were $n = 20$. For training DQN we used StableBaselines3.

4.4 Results

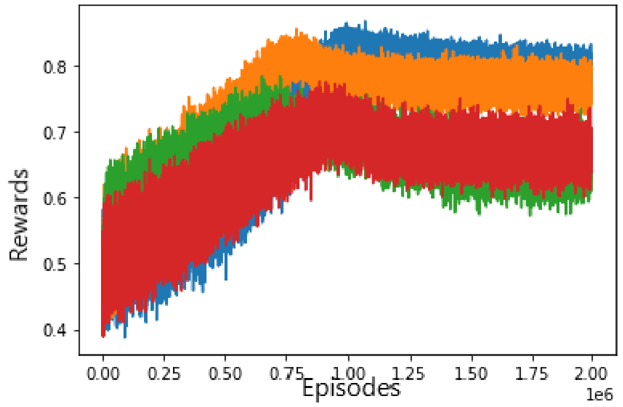

Figure 5: Q-learning Training Reward vs Episodes, averaged over last 100 episodes

Color	Gamma	Epsilon	A	B
Blue	0.99	0.9	0.158	3.072
Orange	0.99	0.7	0.184	2.581
Green	0.9	0.7	0.305	1.561
Red	0.9	0.9	0.287	1.643

Table 4: Q-learning, Test Values A : Mean of difference of best score and chosen score, B: Mean of number of candidates seen

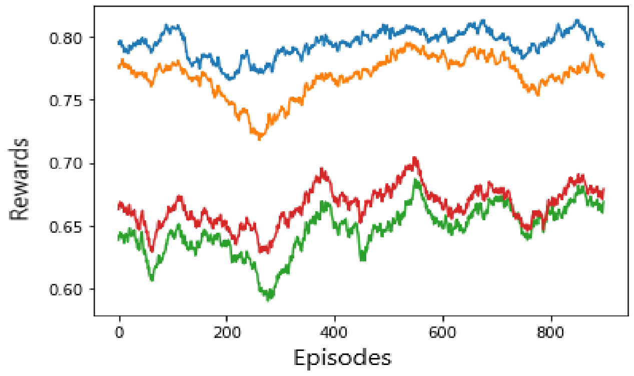

Figure 6: Q-learning Testing Reward vs Episodes, averaged over last 100 episodes

Color	Gamma	Epsilon	C
Dark Blue	0.99	1	0.900
Red	0.9	1	0.823
Light Blue	0.9	0.9	0.797
Pink	0.99	0.9	0.906

Table 5: DQN, Test Value C : Average Reward

We trained Q-learning and DQN models with varying models. Here we can observe that for high γ(discount factor) and epsilon both models are performing well.

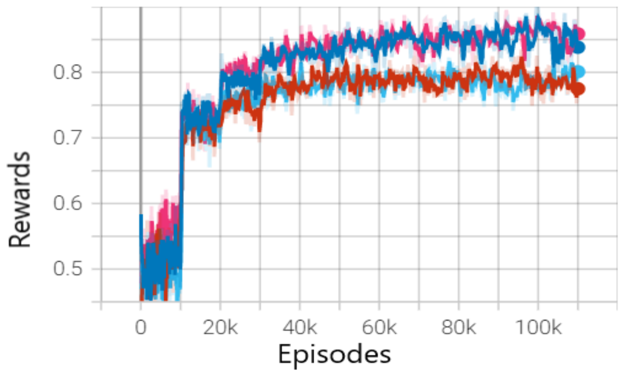

Figure 7: DQN Training Reward vs Episodes

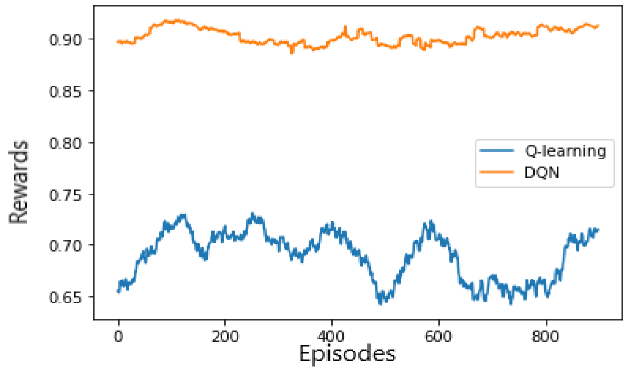

Figure 8: Q-learning vs DQN, Reward vs Episodes

In figure 8, we compared the DQN and Q-learning models to the same inputs and observed that DQN model is able to select candidates of scores greater than 0.9 consistently.

We also ran a learned DQN model on new instances where n(number of candidates) were 50 and 100(the DQN model was trained on instances where $n = 20$) and observed that the mean selected candidate score on $n = 20, 50, 100$ are 0.903, 0.898, 0.906. So the learned DQN model was able to perform well on instances on which it hadn't seen until then.

5 CONCLUSION

Applying ML on classical problems can improve the performance of classical algorithms. After applying ML, we observed that the time complexity is improved in case of binary search, false positives are reduced in case of bloom filters and we received a score of a candidate around 0.9 in case of the secretary problem. ML can be used to improve the performance of many classical problems in future.

REFERENCES

[1] L. Bayon, P. Fortuny Ayuso, J. M. Grau, A. M. Oller-Marcen, and M. M. Ruiz. 2017. The Best-or-Worst and the Postdoc problems. https://doi.org/10.48550/ARXIV.1706.07185

[2] Weiwei Kong, Christopher Liaw, Aranyak Mehta, and D. Sivakumar. 2019. A new dog learns old tricks: RL finds classic optimization algorithms. https://openreview.net/pdf?id=rkluJ2R9KQ

[3] Thodoris Lykouris and Sergei Vassilvitskii. 2018. Competitive caching with machine learned advice. *CoRR* abs/1802.05399 (2018). arXiv:1802.05399 http://arxiv.org/abs/1802.05399

[4] Michael Mitzenmacher and Sergei Vassilvitskii. 2020. Algorithms with Predictions. *CoRR* abs/2006.09123 (2020). arXiv:2006.09123 https://arxiv.org/abs/2006.09123

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[6] Nima H. Siboni. 2020. *Deep Reinforcement Learning for Optimal Stopping Problems.* https://medium.com/deepmetis/deep-reinforcement-learning-for-optimal-stopping-problems-9750a245b8cc