SOLVING MARKOV DECISION PROCESSES WITH NEURAL NETWORKS

< RESEARCH PAPER BUSINESS ANALYTICS >

Job Ammerlaan
2178729 – jan640

TABLE OF CONTENTS

Chapter 1 – Introduction 3
Chapter 2 – Markov Decision Processes 4
The Poisson Process 4
Kendall's notation for queuing models 4
Queuing model parameters 5
Markov Chains 5
Modelling the M/M/1 as a DTMC 6
Extending the Markov Chain to a Markov Decision Process 7
The M/M/1 queue with Admission Control 8
The Curse of Dimensionality 9
One-step policy improvement 10
Neural Networks 10
Chapter 3 – Model 12
Main Loop 12
The M/M/1 queue 13
Split server model 13
Chapter 4 – Results and discussion 15
The M/M/1 queue 15
Split server model 16
Complexity 17
Conclusion 17
References 19

CHAPTER 1 – INTRODUCTION

Many of today's problems can (in adapted or simplified form) be modeled as a Markov Decision Process. Markov Decision Processes (MDPs) were created to model decision making and optimization problems where outcomes are (at least in part) stochastic in nature. The Markov in the name refers to Andrey Markov, a Russian mathematician who was best known for his work on stochastic processes. MDPs are a subclass of Markov Chains, with the distinct difference that MDPs add the possibility of taking actions and introduce rewards for the decision maker. A Markov Decision Process is commonly used to model discrete time stochastic control processes, such as queueing systems or planning problems.

In short, the process moves through time one time step Δt at a time, resulting in the process residing in some state $\boldsymbol{s}$. The decision maker can then choose any valid action $\boldsymbol{a}$ that is currently available in $\boldsymbol{s}$, and the process moves stochastically to a new state $\boldsymbol{s}'$ in the next time step. The decision maker is rewarded based on the current state and action taken, and the next time step can be modelled.

The largest obstacle in solving a Markov Decision Problem is the explosion of the state space. In order to solve an MDP, the transition matrix needs to be computed. The transition matrix consists of a row and column for each (state, action) pair. This means that the number of matrix elements is of $O(A * N)$, where A is the number of distinct actions the decision maker can take, and N is the number of states in the model. To illustrate the sheer size of the transition matrix, even for small systems, we will show the ramifications of the state space explosion in a simple system throughout this paper.

Let us consider a queuing system where F arrival flows combine into a system that can only serve one customer at a time. Time is modeled as $\Delta t = 1\ second$ per time step to discretize the system. We consider a cyclic policy, where each queue is served in a set order. The policy is determined as when to switch over from one queue to the next. The complexity of this very basic system is mostly determined by the number of places in a queue; if we bound the system at Q customers per flow, the transition matrix contains $(1 + Q)^F$ queue states. Setting aside additional transition states, this means for a small system with 12 flows and 9 customers per queue line, we have 10^{12} queue states. The actual number of states will be larger still when considering additional modelling requirements, which will be discussed later. In any case, it is far too high to solve the corresponding MDP in any reasonable amount of time. This is known as the curse of dimensionality.

A number of methods have been devised that aim to solve an MDP without computing the full transition matrix. This paper will explore a method of solving MDPs by means of an artificial neural network, and compare its findings to traditional solution methods.

CHAPTER 2 – MARKOV DECISION PROCESSES

In order to understand how real-life problems can be modelled as Markov Decision Processes, we first need to model simpler problems. The first such technique used to model these types of stochastic problems is the Poisson Process.

THE POISSON PROCESS

A Poisson Process is a mathematical random process in which points are randomly located on a certain space [1]. In other words, a Poisson Process is a process that defines events happening randomly over time, with a number of useful mathematical properties. These properties make it exceptionally useful to model seemingly random processes in a variety of fields; biology, geology, physics, and most importantly for us: queuing theory.

The most important property of a Poisson Process is that each point that is drawn (i.e. each event that occurs) is stochastically independent to all the other points in the process. Stochastic independence is a term in probability theory that defines the independence of event B occurring regardless of the occurrence of event A.; i.e. the probability of both events occurring is equal to the product of the single occurrence probabilities. In formulae:

$$P(A \cap B) = P(A)P(B),$$

or

$$P(A) = \frac{P(A)P(B)}{P(B)} = \frac{P(A \cap B)}{P(B)} = P(A|B).$$

Similarly,

$$P(B) = P(B|A).$$

What this effectively means is that the occurrence of B does not affect the probability of A occurring, and vice versa: the events are independent. This independence is heavily used in the application of Poisson Processes. The next event occurs at a time randomly drawn from the exponential distribution, which has a memorylessness property:

$$P(T > s + t|T > s) = P(T > t)$$

We denote T as the time until the next event occurs. The memorylessness of the exponential distribution states that the time until T is independent of the time we have waited thus far; the probability of T occurring after $t + s$ given that it has not occurred before s is equal to T occurring after t with no such conditional information. Note that this is exactly the stochastic independence discussed above. A Poisson Process is characterized by its *rate*: the Poisson Process with rate λ has interarrival times that are exponentially distributed with parameter λ.

KENDALL'S NOTATION FOR QUEUING MODELS

Before we can apply this theory to a queuing model, we need to define the queue itself. To do this, we will be using Kendall's notation [2]. This notation describes queuing models with three factors: $A/S/c$. Here, A denotes the time between arrivals in the queue, S denotes the service time of a job and c the number of servers. A number of different symbols are possible for each factor; we will be using the simplest model:

- A: M – a memoryless (or Markovian) arrival process: a Poisson Process for arrivals.

- S: M – a memoryless (or Markovian) service time distribution; an exponential service time for each job.
- c: 1 – a single server to process jobs.

Combining the factors above results in the most basic queuing model: the M/M/1 queue.

QUEUING MODEL PARAMETERS

As stated above, the arrival process is a Poisson Process. This Poisson Process depends on one parameter, the intensity of the process, or the density of arrivals in the process. We will be looking at a homogeneous Poisson Process, that is to say the intensity is a fixed value λ over time. The service duration also takes one parameter, the rate of the exponential distribution μ. Note that this process is well-defined if and only if the queue is stable [3]. The queue is stable when $\lambda < \mu$, i.e., when arrivals happen slower on average than the time it takes for a job to complete. When they happen faster, queuing times grow indefinitely and no limiting distribution can be found for the number of people in the queue. We denote the load $\rho = \frac{\lambda}{\mu}$ as the average fraction of time the server is occupied. If $\rho < 1$, the system is stable.

MARKOV CHAINS

So far, we have been looking at the mathematical properties of the queue itself. Let us now examine a modelling technique to analyze the system's properties and performance, the Markov Chain. We divide time into small discrete time steps Δt, and model transitions at each time step. We denote π_i as the number of people in the system at time i. The evolution of the number of people in the system is then a sequence of random variables or states $\pi_1, \pi_2, \pi_3, \ldots$. For this system to be a Markov Chain, these variables need to satisfy the Markov property: the probability of transitioning to the next state depends only on the current state [1]. Since arrivals are being drawn from the exponential distribution and are thus independent, the transition probabilities do not depend on the history of the system, but only on the current state. We want to compute the stationary distribution; the time-limiting distribution that the DTMC is in a certain state. In order to be able to compute such a stationary distribution, there are a number of assumptions that we need to satisfy [1]:

- Let all the states $\pi_1, \pi_2, \pi_3, \ldots$ be called the state space X. We need to restrict $|X| < \infty$. Of course, since the number of possible customers in a queue is not bounded by any real number, we need to create an artificial bound on the queue length to model it as a DTMC. We will call this bound N.
- There exists at least one recurrent state $x \in X$. A recurrent state is a state that can be reached from any other state $y \in X$. The sequence of states visited between y and x is called the path from y to x.
- The greatest common divisor of all possible paths from x to x is 1 for at least one recurrent state $x \in X$. This means the shortest path from x to x is not a fixed length cycle, as that would imply an unstable stationary distribution.

If these assumptions are satisfied, we can use a Markov Chain to model the $M/M/1$ queue and analyze it [3]. We can draw a transition graph to show the modelling of the M/M/1 queue as a Discrete Time Markov Chain (DTMC). The states $\pi(x)$ correspond to the number of people x in the system. With rate λ, another customer enters the system, and with rate μ a customer leaves the system. The transition graph then looks as follows:

The big advantage of modelling the $M/M/1$ as a DTMC is that we can use the following equations, assuming we have a stationary distribution π:

$$\pi(x) = \sum_{y \in X} \pi(y) p(x, y)$$

Here, $\pi(x)$ is the probability of the process residing in state x, or x people currently being in the system. $p(x, y)$ is the probability of moving from state y, given it is in state x. Recall that both the arrival process and the departures from the system follow an exponential distribution with rates λ and μ. For such a simple queue, the above equation translates to 'rate in = rate out': the sum of all rates into each state is equal to the sum of all rate out of that state. Intuitively, this makes sense: we must make a transition at each time step (even if it is to the same state), so the total probability of transitioning is always 1. It easily follows that whatever rate goes into a state must also go out of that state again during the next time step. For the $M/M/1$ queue, this means the following for any state n between 0 and N customers:

$$\lambda * \pi(n-1) + \mu * \pi(n+1) = (\mu + \lambda) * \pi(n) \qquad \forall\, n \in \{1, \dots, N-1\}$$

Another important property of the stationary distribution in a Markov Chain that can be shown intuitively is the following equation. The intuition here is that all states in the stationary distribution (since it is a probability distribution) must sum to 1:

$$\sum_{x \in X} \pi(x) = 1$$

MODELLING THE M/M/1 AS A DTMC

Let us show the significance of the properties listed above by modelling an $M/M/1$ queue, bounded at an arbitrary N customers. This gives us the following equations for the stationary distribution:

$$\begin{aligned}
&\mu * \pi(1) = \lambda * \pi(0) \\
&\lambda * \pi(0) + \mu * \pi(2) = (\mu + \lambda) * \pi(1) \\
&\qquad\qquad \vdots \\
&\lambda * \pi(n-1) + \mu * \pi(n+1) = (\mu + \lambda) * \pi(n) \\
&\qquad\qquad \vdots \\
&\lambda * \pi(N-1) = \mu * \pi(N) \\
&\sum_{i=0}^{N} \pi(i) = 1
\end{aligned}$$

These equations can be rewritten in matrix form, creating the transition matrix P shown below:

$$\begin{bmatrix}
-\lambda & \mu & & & & & \\
\lambda & -(\lambda+\mu) & \mu & & & & \\
 & \lambda & -(\lambda+\mu) & \mu & & & \\
 & & \ddots & \ddots & \ddots & & \\
 & & & \lambda & -(\lambda+\mu) & \mu & \\
 & & & & \lambda & -\mu & \\
1 & \dots & \dots & \dots & \dots & 1
\end{bmatrix} = \begin{matrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 0 \\ 1 \end{matrix}$$

We can now use a simple solving tool to find the stationary distribution π:

$$\pi = \begin{matrix} 1-\frac{\lambda}{\mu} \\ \left(1-\frac{\lambda}{\mu}\right)*\frac{\lambda}{\mu} \\ \vdots \\ \left(1-\frac{\lambda}{\mu}\right)*\left(\frac{\lambda}{\mu}\right)^N \end{matrix} = \begin{matrix} \left(1-\frac{\lambda}{\mu}\right)*\left(\frac{\lambda}{\mu}\right)^0 \\ \left(1-\frac{\lambda}{\mu}\right)*\left(\frac{\lambda}{\mu}\right)^1 \\ \vdots \\ \left(1-\frac{\lambda}{\mu}\right)*\left(\frac{\lambda}{\mu}\right)^N \end{matrix} = \begin{matrix} (1-\rho)\rho^0 \\ (1-\rho)\rho^1 \\ \vdots \\ (1-\rho)\rho^N \end{matrix}$$

It can also be empirically shown through simulation that the average number of people follows the following formula:

$$\pi(x) = (1-\rho)\rho^x$$

for the $M/M/1$ queue. We will come back to the transition matrix P and stationary distribution π of the $M/M/1$ queue in the next chapters.

EXTENDING THE MARKOV CHAIN TO A MARKOV DECISION PROCESS

Since Markov Decision Processes (MDPs) are a subclass of Markovian problems that have to do with decision making, we need to extend the DTMC model with decisions. This is done by introducing a policy: a vector with a length equal to the number of states in the Markov Chain, where each element in the vector indicates the decision that will be made in that particular state of the Markov Chain. Naturally, if there are decisions involved, we need to define a measure that rates the quality of a decision. Therefore, we also introduce rewards in each state with the aim of determining the quality of a given policy. Hence, we now have the following properties of the problem:

- X, the set of states.
- A, the set of actions available in each $x \in X$.
- P, the transition matrix that defines the transition probabilities between states.
- r, the vector of rewards obtained in each state.
- R, the policy vector defining which action to take in which state.

We aim to maximize the sum of rewards we obtain over time by using a certain policy. That is, we call the total reward that we obtain by using a policy R for each $t \in \{0, \dots, T\}$ when starting in state x, $V_T^R(x)$. We are interested in two things specifically:

- The long term average reward, or the expected value of the rewards for the stationary distribution:

$$g = E\big(r(\pi)\big) = \sum\nolimits_{x \in X} \pi(x)V(x) \text{ or } g = lim_{T\to\infty}\left(\frac{V_T(x)}{T}\right)$$

- The total expected difference between starting in a state x and starting in stationarity:

$$V^R(x) = lim_{T\to\infty}(V_T^R(x) - gT)$$

The following relationship exists between these two measures. This equation is commonly known as the Poisson Equation, and is used heavily in the computations behind Markov Decision Processes [3]:

$$V(x) + g = r(x) + \sum\nolimits_{y \in X} p(x,y)V(y)$$

$V(x)$ is commonly known as the value function of a problem, $V^R(x)$ is the value function of a decision problem given a policy R. The Poisson Equation is of particular interest to us because it provides us with a method to solve for g once we have a solution for $V(x)$. Note, however, that $V(x)$

need not be unique because of this equation alone. If $V(x)$ is a solution to this equation, $V'(x) = V(x) + c$ is also a solution, where c is any constant factor. Therefore, we always set $V(0) = 0$ so policies can be compared to each other. Extending the Poisson Equation to include actions gives

$$V(x) + g = r(x, a) + \sum_{y \in X} p(x, a, y)V(y).$$

This equation can be used to model an MDP and evaluate a given policy [3].

THE M/M/1 QUEUE WITH ADMISSION CONTROL

We will now show the application of the theory outlined in the previous sections by modelling the M/M/1 queue with admission control. We model an M/M/1 queue with N places for customers. For each customer that arrives at the queue, we decide whether we accept that customer and add it to the queue, or if we reject the customer. If we reject it, we incur a one-time penalty cost C. We aim to minimize the number of customers in the queue, as well as to minimize the penalty costs. We model this by incurring a 'penalty' of 1 for each customer in the queue at every time step, on top of the rejection costs. Customers arrive at the queue according to a Poisson Process with λ and the server handles customers according to an exponential distribution with μ. This process with parameters $N = 1, C = 2, \lambda = 0.1, \mu = 0.15$ has the following properties:

- This system is stable, as $\rho = \frac{\lambda}{\mu} = \frac{0.1}{0.15} < 1$.
- $X = \{0, 1, 2\}$. There can be $N = 1$ customers in the queue, one customer can be served at a time, and the system can be empty, so there are 3 different states.
- $A = \{1, 2\}$. Accepting a customer translates to $a = 1$, rejecting it gives $a = 2$.
- The actual transition matrix P naturally depends on the policy R chosen. For completeness, we will list the transition matrices for both actions in every state:

$P(x, 1)$:

$$\begin{bmatrix} 0.90 & 0.10 & \\ 0.15 & 0.75 & 0.10 \\ & 0.15 & 0.85 \end{bmatrix}$$

$P(x, 2)$:

$$\begin{bmatrix} 1 & & \\ 0.15 & 0.85 & \\ & 0.15 & 0.85 \end{bmatrix}$$

- The reward vector r is again dependent on the policy R. Since we do not get any rewards, but rather incur penalties, the reward vector is mostly negative:

$$r(x, a) = \begin{bmatrix} 0 & -2 \\ 0 & -2 \\ -1 & -3 \end{bmatrix}$$

- The policy vector R will be initialized as $R(x) = [1 \quad 1 \quad 1]$: always accept new customers.

The Poisson Equations that correspond to this problem are then:

$$V(0) + g = 0 + 0.90V(0) + 0.10V(1)$$
$$V(1) + g = 0 + 0.15V(0) + 0.75V(1) + 0.10V(2)$$
$$V(2) + g = -1 + 0.15V(1) + 0.85V(2)$$
$$V(0) = 0$$

Recall that we explicitly state $V(0) = 0$ to obtain a unique solution to this problem. We can solve these equations using matrix algebra and find the best policy. In this case, we find

$$V(x) = [0 \quad -2.074 \quad -7.296], \qquad \pi(x) = [0.474 \quad 0.316 \quad 0.211],$$

giving us:

$$g = \sum_{x \in X} \pi(x)V(x) = 0.474 * 0 + 0.316 * -2.074 + 0.211 * -7.296 = -2.192$$

Overall, we expect to incur a cost of roughly 2.192 per time step following this policy. For reference, if we were to always block customers, this would be exactly 2 per time step. We can determine the optimal policy by simply evaluating all policies and selecting the one with the highest g. For this example, the optimal policy would be $R(x) = [1 \quad 2 \quad 2]$: if the server is currently occupied, we reject all incoming customers, so the queue length will never exceed 0. Such a policy is an example of a *threshold policy*: we take a certain action in all states up until a threshold, and for all states after this threshold we take another action.

THE CURSE OF DIMENSIONALITY

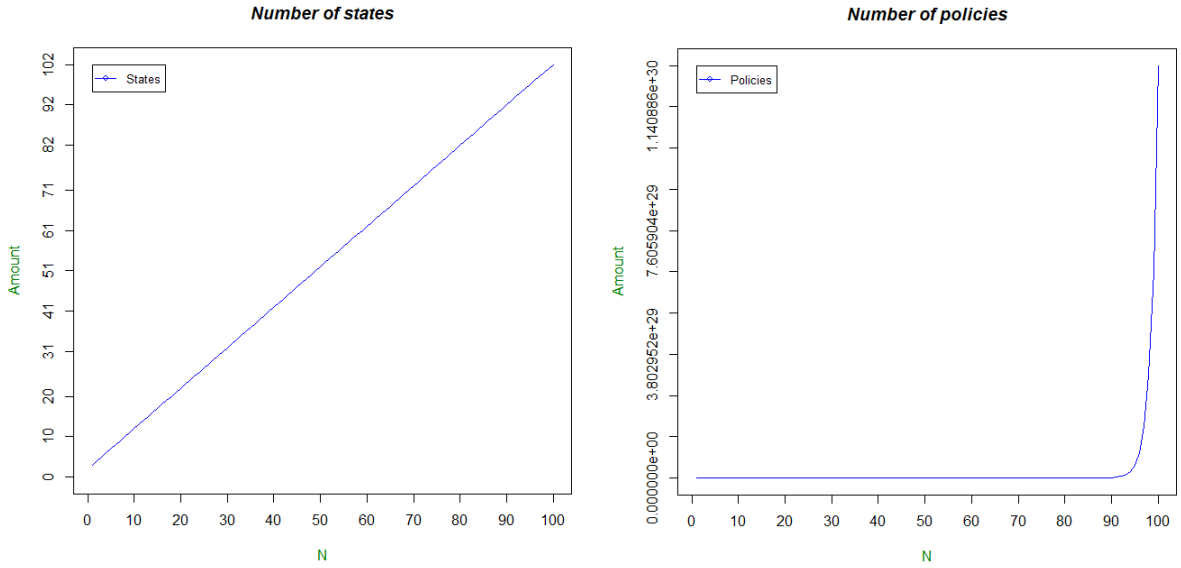
The example model we just discussed is hardly a realistic model. Real-life applications of Markov Decision Processes often encompass vast numbers of possible states. In the M/M/1 model, the number of distinct states $|S|$ is given simply by the queue size N, plus one state for the server, and one for the empty system:

$$|X| = N + 2$$

The number of states grows linearly with the queue size. The number of policies, however, grows significantly faster, as this is given by $|A|$ to the power of $|X|$:

$$\#\ policies = |A|^{|X|}$$

In the simple example, this means that there are $2^3 = 8$ unique policies, easily computable by hand. The graph below shows how the number of policies and states grow as we increase $|X|$, keeping $|A| = 2$ fixed:



This is the largest problem currently facing the customary solution methods for MDPs at this time: the so-called 'curse of dimensionality' [3]. Imagine, for instance, a queuing model where customers arrive and are assigned to one of 8 queues, each with a maximum capacity of 10 customers. Then, $|X| = 8^{10}$ and the number of distinct policies is $8^{8^{10}}$, a number with $9.6985 * 10^8$ decimal digits (!). This number is so large, it becomes wholly impossible to compute the optimal policy by brute force: evaluating all possible policies is no longer feasible. As a result, a number of algorithms and solution

methods have been devised to determine the optimal policy. We will be looking at the *one-step policy improvement* in the next section.

ONE-STEP POLICY IMPROVEMENT

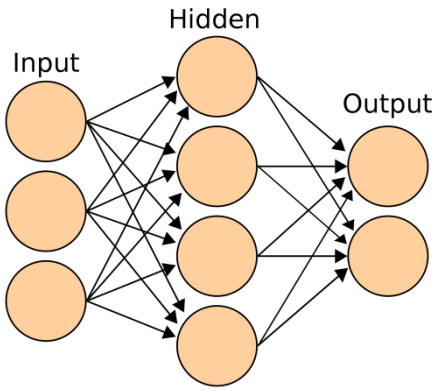
Since we are unable to evaluate all policies, we will improve upon a given policy (usually picked at random) and evaluate the process with the improved policy iteratively. We aim to optimize the actions taken in each state, i.e. maximize the reward. The corresponding equation is known as the Bellman Equation [4]:

$$V(x) + g = max_{a \in A}\left(r(x,a) + \sum_{y \in X} p(x,a,y)V(y)\right)$$

We can iteratively apply this equation to obtain the optimal policy $R^*(x)$ with corresponding $V^*(x)$ and $g^*(x)$. Computationally, the most challenging step in this equation is computing $V(x)$ for a given policy R, this needs to be done a large number of times while searching for the best action. Therefore, this action is usually done online: in the improvement step we simply compute $V(y)$ only if $p(x,a,y) > 1$. For most MDPs of a reasonable size, this is (though still computationally expensive) a feasible solution method. However, as we increase the state and action spaces, we will again run into performance issues. The next chapter will cover a different method of finding the optimal policy for MDPs that will reduce or even completely eliminate the issues brought forth by the curse of dimensionality. First, however, we will take a closer look at neural networks as a modelling technique.

NEURAL NETWORKS

Artificial Neural Networks (ANNs) are machine learning systems where a large collection of simple units work together to model complex problems by simulating the way the human brain handles inputs using neurons. The units in the model are heavily interconnected, allowing for the modeling of precise patterns without specifying the exact pattern itself [4]. The main advantage of ANNs is that very little information is required about the structure of the problem beforehand, as the model learns the structure of the problem on-line during training. The main disadvantage of using neural networks in this setting is that the model behaves as a black box; that is, it is nearly impossible to determine the root cause of the success of the model. Since very little can be said about the cause of a model performing exceptionally well or badly, it is relatively hard to make improvements to said model. An example of an ANN can be found to the right. Each node in the network processes input through its activation function ϕ to produce its output: if the inputs of the ANN are given by $x = x_1, x_2, x_3, ...$, the next node can be computed as $\phi(w^T x)$, with $w^T = w_1, w_2, w_3, ...^T$ the weights of each input. A common activation function for classification problems is the sigmoid activation function [6]:



$$\phi(w^T x) = \frac{1}{1 + \exp(-w^T x)}$$

Credit image: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Artificial_neural_network.svg/2000px-Artificial_neural_network.svg.png

Output can be classified as one of two classes, each belonging to an output of either 0 or 1. Since in this paper an ANN was used to model regression, a linear activation function was more appropriate:

$$\phi(w^T x) = cw^T x, \qquad c \in R$$

The model learns by adjusting the weights of nodes using a method called backpropagation for a set number of repetitions or until a (near) optimal set of weights has been found [5]. The exact process of backpropagation lies outside the scope of this paper; as such, it will not be further discussed here.

CHAPTER 3 – MODEL

Neural networks are a good choice for this type of problem for several reasons.

First, neural networks are fairly robust, in that they can deal with a variety of different problem structures and sizes while preserving their prediction accuracy. Thus, we can develop a single solution method that makes use of a neural network which can solve specific MDPs, and generalize this method to solve any MDP in a similar fashion.

Moreover, neural networks require no prior knowledge about the problem or solution structure. For example, we know that the optimal policy for the M/M/1 with admission control is a threshold policy. In theory, the neural network should 'learn' this property of the optimal policy by itself, so by simply feeding random policies (and their evaluation) into the neural network; the network should return a threshold policy as its optimal policy.

Finally, neural networks may be faster than the traditional solution methods for MDPs. Although training a neural network is relatively time-consuming, testing policies using a neural network should be orders of magnitude faster than simulating for a high number of time steps or using a different method of policy evaluation.

MAIN LOOP

The model was initialized by creating a single hidden layer neural network with $|X|$ input nodes, $L = 5$ hidden nodes and 1 output node. The transition matrix P, a 3-dimensional matrix with the transition probabilities for each action a, and the reward matrix R, a 2-dimensional matrix with the rewards for each station/action pair were created. For I iterations, the following actions were performed.

A random policy R_i was generated by sampling a random action for each state. V^R and g^R for this policy were computed as follows:

1. Simulate V_T^R for $T = 10000$.
2. Obtain π^R by solving the following equation, where P^R is the transition matrix specific to this policy:
$$P^R = \pi^R * P^R$$
3. Obtain g by computing:
$$g = \frac{\sum_{x \in X} \pi^R(x) V_T^R(x)}{T}$$
4. Train the neural network with this example, using $R_i(x)$ at the input nodes and g^R at the target output.

After these iterations, for another J iterations the following actions were performed:

1. Another random policy R_j was generated by sampling a random action for each state, and V^R and g^R were computed for this policy as before.
2. Subsequently, the g found by the neural network, $g^{R\prime}$, from the trained neural network was obtained.
3. If the policy R_j provided the best $g^{R\prime}$ at that time, this policy was saved as the best found policy R'.

After these J iterations, the current best policy was contained in R'. R' then had the following algorithm applied to it:

1. We start with the policy R'.
2. We loop over all the states $x \in X$:

   We set the action in state x to:

$$R^*(x) = \text{argmax}_{a \in A}(R'(x) = a)$$

3. If any action has been changed, start at step one with $R*$ as the new R'.

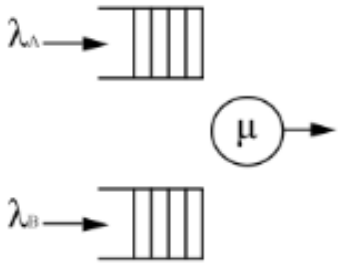
In words: the action of R' performed in state x was set to the action that results in the maximum g found by the neural network. Since we started in the empty system, this algorithm should result in a better policy than the one we started with. If after visiting all states an action in any state was changed, the algorithm was applied again to ensure we take the optimal actions starting from the empty system. The resulting policy R^* was the optimal policy found.

THE M/M/1 QUEUE

The M/M/1 queue has an optimal policy that is a threshold policy for all (stable) model parameters. In all states before the threshold state x_s customers are always accepted. From the threshold state onwards, customers are always rejected. For the M/M/1 the intuition behind this is simple: if we have a service duration expectation of μ time units, pay 1 unit of costs per time step for a customer in the queue, and have $x - 1$ customers currently in the system, we expect to pay $x * \mu$ queueing costs for the x'th customer. Thus, we start rejecting customers when $x * \mu > C$, and we can directly compute the threshold state from this formula. The goal for the neural network model is to determine this threshold state for several different parameters as accurately as possible.

SPLIT SERVER MODEL

Aside from the M/M/1, we will also be analyzing a slightly more complex queuing model with one server and two queues A and B. Each queue has its own arrival stream, with possibly different rates λ_A and λ_B. The queue costs, i.e. the reward in each state, $r(x, a)$ can be different for each arrival stream as well. To simplify the model somewhat, the server rate μ will be fixed and equal for both queues. For this model, we aim to optimize the server position, i.e. which queue to serve for a given pair of queue lengths. The optimal policy for this model is no longer a simple threshold state. An optimal policy matrix for this queuing model might look like so:



$$R(a, b) = \begin{bmatrix} 0 & A & A & A \\ B & A & A & A \\ B & B & A & A \\ B & B & B & A \end{bmatrix},$$

where a and b are the respective queue lengths for queue A and B, and each matrix item represents the queue to be served. In this example with $N = 3$ available spots in each queue, we serve queue B if and only if there are more customers in this queue than in queue A. This is of course a threshold policy of sorts – in fact it is simply a two-dimensional threshold – but encoded to an input vector for the neural network this policy is:

$$R(x) = [0 \quad A \quad A \quad A \quad B \quad A \quad A \quad A \quad B \quad B \quad A \quad A \quad B \quad B \quad B \quad A]$$

This type of pattern is more complex than a simple threshold state the neural network needs to find, which might lead to a lower accuracy for the policy quality predictions. Note that the action taken in

the first state is 0; for a completely empty system the currently served queue is obviously irrelevant, since we do not model switchover time or costs. The action taken in that state is entirely arbitrary.

CHAPTER 4 – RESULTS AND DISCUSSION

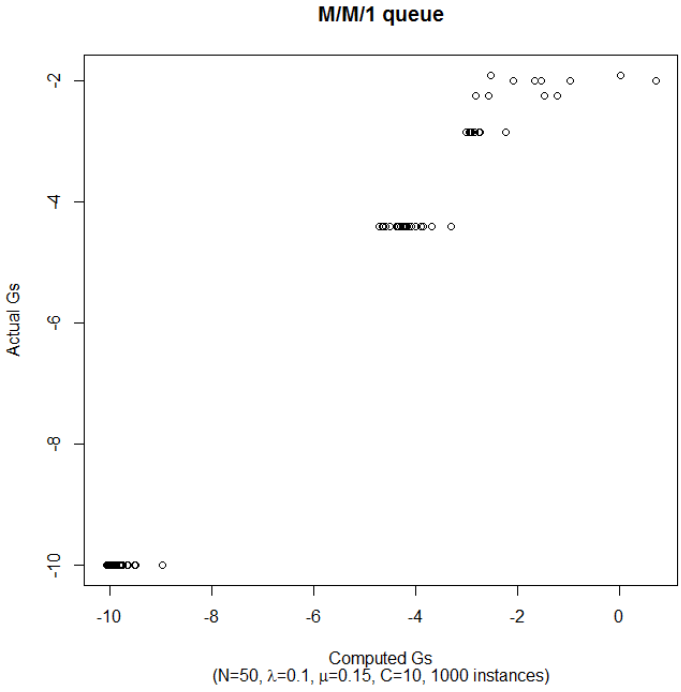
Two different queueing models will be discussed here, as discussed throughout the paper: the M/M/1 queue with admission control, and a split server model with two queues and one server. In the M/M/1 model, the decision is whether to accept or reject an incoming customer. In the split server model, we aim to optimize the server position: which queue to serve at which times.

THE M/M/1 QUEUE

The simplest model, the M/M/1 model, yielded largely positive results. Results for different instances of the problem solved using of the algorithm can be found in Table 1. For all runs, the following parameters were used:

$$\lambda = 0.1, \mu = 0.15, I = 1000, J = 100$$

The average long term reward g for the found policy R^* is listed both as computed by the neural network and the value found using traditional solution methods. The overall optimal g is also listed, as well as the threshold state – above which we start rejecting all customers – and the Mean Squared Error (MSE) of all $g's$ computed by the neural network. The MSE over 5 runs was computed as well.



From the graph to the right, we can see that for the instance with the system bounded at $N = 50$ customers, the neural network predicts policy qualities extremely accurately. This is confirmed by the high accuracy shown in Table 1 below. The MSE over all policies during 5 runs remains stable throughout the testing. When the rejection cost rises, lower-quality policies incur severely higher costs, and thus the MSE rises sharply to reflect this. While the policy that the algorithm declared optimal still attains a long-term average reward g that is close to the theoretical maximum, the optimal policy produced by the neural network shows no clear threshold: after the threshold, customers are still accepted in some (seemingly arbitrary) states. Therefore, the first alternative action is recorded as the threshold state.

	$N = 10, C = 0$	$N = 10, C = 5$	$N = 50, C = 10$	$N = 100, C = 10$	$N = 100, C = 50$
Computed g for R^*	0,0209	-1,1041	1,3209	2,0894	4,2645
Actual g for R^*	0	-1,8264	-1,9038	-2,2462	-2,2647
Optimal g overall	0	-1,6305	-1,8856	-1,8854	-0,0120
Computed threshold state	0	8	5	3	9
Actual threshold state	0	4	6	6	27
MSE over all policies	0,0134	0,0997	0,2769	1,2128	823,5715

SPLIT SERVER MODEL

When not explicitly mentioned, the following parameters were used:

$$\lambda = 0.1, \mu = 0.15, r = -1, I = 1000, J = 100$$

Table 2 shows the results for a variety of different parameters and models, using the same metrics as for the M/M/1 queue. As there is no longer a definitive threshold state, this metric was no longer recorded.

	$\boldsymbol{N = 3, \lambda_A = 0.1, \lambda_B = 0.15}$	$\boldsymbol{N = 5, \lambda_A = 0.1, \lambda_B = 0.15}$	$\boldsymbol{N = 10, \lambda_A = 0.1, \lambda_B = 0.15}$	$\boldsymbol{N = 5, r(x,a) = [-3,-1]}$	$\boldsymbol{N = 10, r(x,a) = [-3,-1]}$
Computed $\boldsymbol{g}$ for $\boldsymbol{R}^*$	-1,2602	1,3571	-0,4128	0,9440	5,8627
Actual $\boldsymbol{g}$ for $\boldsymbol{R}^*$	-1,2998	-3,5994	-6,3856	-5,7763	-7,0217
Optimal $\boldsymbol{g}$ overall	-1,2520	-2,0857	-3,3848	-2,4900	-2,9202
MSE over all policies	0,0159	0,2268	1,4301	2,7746	27,9962

Again, as expected, the MSE over all policies rises sharply as the number of states grows. Judging solely be the MSE, the algorithm performs reasonably well. However, as the state space grows, the computed optimal policy moves farther and farther away from the actual optimal policy. Compare the optimal computed policy with the actual optimal policy for the last test case, with $N = 10$ and a distinct reward structure $\boldsymbol{r(x,a)} = [\boldsymbol{-3,-1}]$ for the queue.

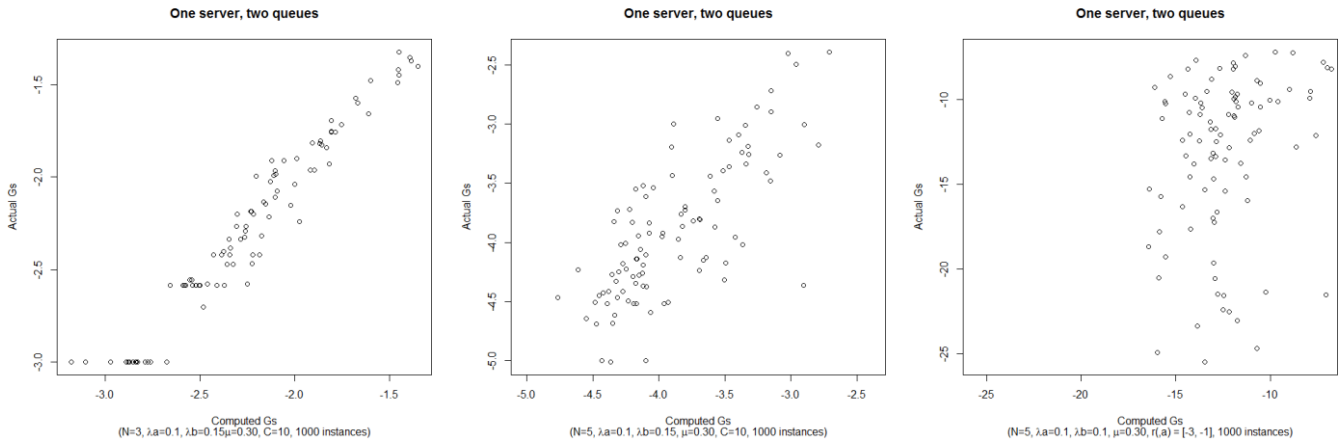
Computed optimal policy:

$$\begin{bmatrix} 0 & A & A & A & A & A & A & B & B & B \\ B & B & B & B & A & B & A & A & A & A \\ A & A & B & A & A & A & B & A & A & B \\ A & A & B & A & A & A & A & A & A & A \\ B & A & B & B & B & B & B & B & B & B \\ A & B & A & B & A & B & B & B & B & B \\ A & A & B & A & A & B & B & A & B & A \\ A & A & A & B & A & B & B & A & A & A \\ A & B & A & B & B & B & B & B & B & A \\ B & B & B & A & A & A & A & B & A & B \end{bmatrix}$$

Actual optimal policy:

$$\begin{bmatrix} 0 & B & B & B & B & B & B & B & B & B \\ A & A & A & A & A & A & A & A & A & A \\ A & A & A & A & A & A & A & A & A & A \\ A & A & A & A & A & A & A & A & A & A \\ A & A & A & A & A & A & A & A & A & A \\ A & A & A & A & A & A & A & A & A & A \\ A & A & A & A & A & A & A & A & A & A \\ A & A & A & A & A & A & A & A & A & A \\ A & A & A & A & A & A & A & A & A & A \\ A & A & A & A & A & A & A & A & A & A \end{bmatrix}$$

Intuitively, the actual optimal policy seems like a very good policy: we only serve queue B if there are no customers in queue A. Since queue A is 3 times as expensive, it is always more profitable to serve this queue first. However, the neural network retains nothing of this optimal strategy and creates a (seemingly) random pattern as its optimal policy. We observe the same problem in other cases. The bigger the state space gets, the more effort it takes the neural network to distuinguish the positive aspects of policies from the negative.



When plotting the g's computed by the neural network against the actual g's for those policies, the problem becomes even more apparent. We expect to see a figure like the first graph for each instance, where the actual g is highly correlated with the computed g. The higher the correlation, the more accurate the prediction is. However, as we increase the state space – which is still wholly reasonable, even for the largest instances – the correlation becomes weaker and weaker, until the algorithm degrades to the quality of trial and error.

COMPLEXITY

Finally, a note on complexity: note that to train the neural network, we compute the performance of a given policy at least $I = 1000$ times, which in itself can be problematic. For large instances of each problem, training the neural network was several times slower than computing the optimal policy through one-step policy improvement. Since speed is one of our main concerns, having to compute $V(x)$ a large number of times is suboptimal at best.

CONCLUSION

Small instances of the M/M/1 problem with admission control are analyzed with relatively small errors. We obtain a near-optimal policy in most small instances of the problem. For a larger state space, we obtain policies that behave decently well, but do not show the desired threshold property: none of the obtained optimal policies show a definitive threshold, so the first action that serves the alternate queue is selected as the threshold.

The results for both models overall show the same trend. The optimal policy for the model with one server for two queues has a significantly more complicated structure due to its two-dimensional state space (as opposed to the one-dimensional state space of the M/M/1 model). This is reflected in the model's outcomes: we see that the computed policies diverge from the actual optimal policy more quickly than in the M/M/1 model.

When observing graphs of the g´s computed by the model plotted against the true g of that policy, we see that the bigger the state space, the higher the spread is for these points. This is reflected in the MSE of those policies: an increasing MSE (even relative to the average g) shows that the results become less reliable for larger problems. Indeed, the graphs clearly show the correlation between the computed long-term average reward g and the actual g of a given policy decreasing.

In conclusion, solving Markov Decision Processes with Artifical Neural Networks works seemingly well for extremely small instances. However, the more the state space grows, the less accurately the neural network predicts how well a given policy performs. Moreover, the computation time required to

train the neural network rises far above the time required for any conventional solution method. As such, training a neural network to find the optimal policy for an MDP is not a viable solution technique for any reasonably sized instance.

REFERENCES

[1] H. Tijms, A First Course in Stochastic Models, John Wiley & Sons, 2003.

[2] D. G. Kendall, "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain," *The Annals of Mathematical Statistics*, pp. 338-354, 1953.

[3] G. K. S. Bhulai, Lecture Notes Stochastic Optimization, Amsterdam, 2014.

[4] R. Bellman, Dynamic Programming, Princeton, NJ: Princeton University Press, 1957.

[5] M. Nielsen, Neural Networks and Deep Learning, 2015.

[6] B. Wilson, The Machine Learning Dictionary, 2012.