Homework 2

Marios Marinos (5353106)
M.Marinos@student.tudelft.nl

October 15, 2020

Exercise 1

(1.) For the calculation of $E[T]$ we need to first calculate the arrival rates r_1 and r_2. Also we need to calculate the utilization ρ_1, ρ_2 and finally calculate the Average number of jobs in the system $E[N]$. By applying Little's Law we know that $E[T] = E[N]/\lambda$, *where* $\lambda = arrivalrate$.

$$r_1 = r + pr$$
$$r_2 = r_1(1-p) + r_2 q$$

$$Replace\ in\ r_2 = \ r_1(1-p) + r_2 q,\ where\ r_1 = r + pr$$
$$By\ solving\ we\ get\ r_2 = \frac{r - p^2 r}{1-q}$$

$$\rho_1 = \frac{r_1}{\mu_1} \Rightarrow \frac{r + pr}{\mu_1}$$
$$\rho_2 = \frac{r_2}{\mu_2} \Rightarrow \frac{r - p^2 r}{\mu_2(1-q)} \Rightarrow \frac{r - p^2 r}{\mu_2 - \mu_2 q}$$

So we have:

$$E[N] = E[N_1] + E[N_2] = \frac{\rho_1}{1-\rho_1} + \frac{\rho_2}{1-\rho_2} \Rightarrow \frac{\frac{r+pr}{\mu_1}}{1 - \frac{r+pr}{\mu_1}} + \frac{\frac{r-p^2 r}{\mu_2 - \mu_2 q}}{1 - \frac{r-p^2 r}{\mu_2 - \mu_2 q}}$$

$$\Rightarrow \frac{r+pr}{\mu_1 - pr - r} + \frac{r - p^2 r}{\mu_2 - \mu_2 q + p^2 r - r}$$

So we have by Little's Law $E[T] = \frac{E[N]}{r} \Rightarrow \frac{\frac{r+pr}{\mu_1 - pr - r} + \frac{r - p^2 r}{\mu_2 - \mu_2 q + p^2 r - r}}{r}$

(2.) If we swap the order of the queue we will have :

$$r_1 = r + qr$$
$$r_2 = r_1(1-q) + r_2 p$$

And by doing the same with 1. question we will have :

$$r_2 = \frac{r - q^2 r}{1-p}$$

Doing also the same with till $E[T_2]$ we will get : $E[T_2] = \frac{E[N]}{r} \Rightarrow \frac{\frac{r+qr}{\mu_1 - qr - r} + \frac{r - q^2 r}{\mu_2 - \mu_2 p + q^2 r - r}}{r}$

So by seeing the 2 equations with the mean response time we can observe that the only thing that change are the **respectively probabilities** and getting into account that the fact the r is chosen so as to not overload the network we can conclude that the $E[T_2]$ of the swapped queue will be the same.

Exercise 2.

Based on the code Simulation.ipynb I created a simulator for G/G/1 queue. The approach I used is that we generate the jobs based on the input file (time.csv) and picking random samples from the data set. After running the Simulation 10 times and getting the average, the Simulation Waiting time in queue is approximately 6.8 seconds. Using the formulas also we get for M/M/1 queue E[T_Q] = 1.01 sec. and for G/G/1 queue W[E_Q] = 6.55 secs which is close to the simulation time. On figure 1 we can also see the results from the python code.

```
Waiting time in Queue in a M/M/1 system is E[T_Q] = 1.0117026459940368 seconds.
Waiting time in Queue in a G/G/1 system is W[T_Q] = 6.550551347672369 seconds.
Waiting time in Queue from the Simulation G/G/1 : 6.7658717418864445 seconds.
```

Figure 1: Queue Waiting Times based on formulas and simulation.

Exercise 3.

(1.) Below figure displays the CTMC of M/M/1/N queue.

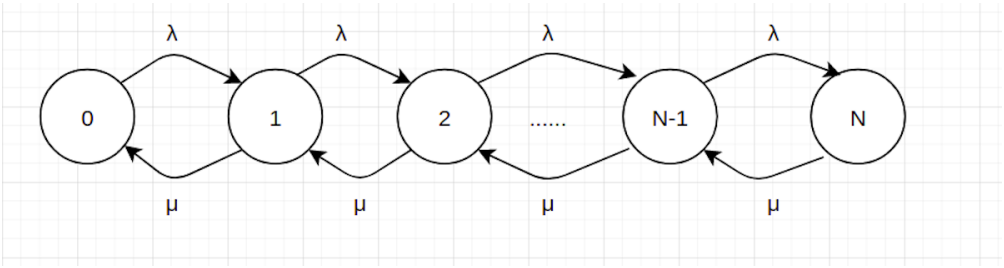

Figure 2: CTMC of M/M/1/N queue.

For the derivation of the limiting probabilities we have to use the balance equations.

$$\lambda\pi_0 = \mu\pi_1 \qquad \pi_1 = \frac{\lambda}{\mu}\pi_0 \qquad \sum_{i=0}^{N-1}\pi_i = 1$$

$$\lambda\pi_1 = \mu\pi_2 \qquad \pi_2 = \frac{\lambda}{\mu}\pi_1 = (\frac{\lambda}{\mu})^2\pi_0, \qquad \pi_0(1 + \frac{\lambda}{\mu} + (\frac{\lambda}{\mu})^2 + \cdots + (\frac{\lambda}{\mu})^N) = 1$$

$$\ldots\ldots \qquad \ldots\ldots \qquad \textit{Applying rule, } \sum_{k=0}^{N-1} ar^k = a\frac{1-r^n}{1-r}$$

$$\lambda\pi_{N-1} = \mu\pi_N \qquad \pi_N = (\frac{\lambda}{\mu})^N\pi_0 \qquad \pi_0 = \frac{1-\rho}{1-\rho^{N+1}}$$

$$\textit{And finally we have } \pi_N = \pi_0\rho^N \Rightarrow \frac{1-\rho}{1-\rho^{N+1}}\rho^N$$

To find the E[Number in system] = $E[N]$ we need again the below summation: .

$$E[N] = \sum_{i=1}^{N} i\pi_i \Rightarrow \sum_{i=1}^{N} i\rho^i \frac{1-\rho}{1-\rho^{i+1}}$$

By doing the calculations we get:

$$E[N] = \frac{\rho}{1-\rho} - \frac{(N+1)\rho^{N+1}}{1-\rho^{N+1}} \tag{1}$$

(2.) Firstly, the mean response time of the system is equal to $E[T] = \frac{E[N]}{\lambda}$ by applying Little's Law.

$$E[T] = \frac{\frac{\rho}{1-\rho} - \frac{(N+1)\rho^{N+1}}{1-\rho^{N+1}}}{\lambda} \Rightarrow E[T] = \frac{\rho + N\rho^{N+2} - N\rho^{N+1} - \rho^{N+1}}{(1-\rho)(1-\rho^{N+1})}$$

$$\textit{We know } E[T] = E[T_q] + E[T_s] \Rightarrow E[T_s] = E[T] - E[T_q], \textit{ Also } E[T_q] = E[T] - E[S]$$

$$\textit{So we have } E[T_s] = E[T] - (E[T] - E[S]) \Rightarrow E[T_s] = E[S] \Rightarrow \frac{1}{\mu}$$

(3.)

- Now we are gonna plug in, $N = 5$ *and* $\rho = \frac{\lambda}{\mu} = 0.8$ on equation 1.

  $$\pi_0 = \frac{1-0.8}{1-0.8^{5+1}} \Rightarrow \pi_0 = 0.2710...$$

  now we can calculate the π_5 using $\pi_N = \pi_0\rho^N$ to find the probability there are 5 customers in the system and thus, the loss probability.

  $$\pi_5 = 0.2710 * 0.8^5 \Rightarrow 0.0888013$$

- If we now double the buffer size N = 10 we have to calculate the π_{10}:

  $$\pi_{10} = 0.2710 * 0.8^{10} \Rightarrow 0.0290984$$

- If we double the CPU speed we will have $\frac{\lambda}{2\mu} = 0.8 \Rightarrow \frac{\lambda}{\mu} = 0.4 \Rightarrow \rho = 0.4$ so we have to re-caclulate the π_0.

  $$\pi_0 = \frac{1-0.4}{1-0.4^6} \Rightarrow \pi_0 = 0.60246...$$

  $$\pi_5 = 0.60246 * 0.4^5 \Rightarrow 0.00616926932$$

From the above results we can conclude that the best solution would be to double the CPU speed, as we can see that the probability of being 5 jobs in the system with double CPU speed is **much less** ($0.00616926932 < 0.0290984$) than of being in a 10-job-system with double buffer size.

Exercise 4.

To solve this exercise I used python and i will provide the copy of my code on the final zip. Below on Figure 1 the results are provided.

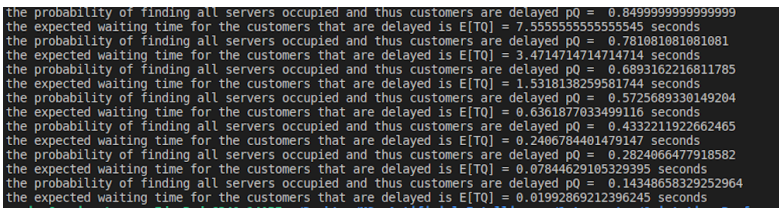

Figure 3: The results for M/M/k system with servers k = [1,2,4...,64].

From the results we can conclude that there is a "trend". We observe that the probability that a customer will come and wait in the queue starts very high (85% in case k = 1) and as the servers increase, this probability tends to decrease. The same phenomenon occurs for the expected waiting time for the delayed customers as the formula depends on the pQ.

Exercise 5.

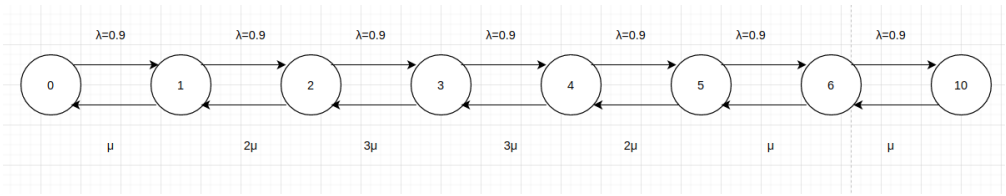

Figure 4: Markov Chain for exercise 5.

(1.) So, to calculate the π_0 we need to calculate it "manually". We know that the sum of π_i is equal to 1. So by that we have $\pi_0(1 + \frac{\lambda}{\mu} + (\frac{\lambda}{\mu})^2 + \cdots + (\frac{\lambda}{\mu})^N) = 1$. By solving this we get : $\pi_0(1 + \frac{0.9}{1} + 0.9 * \frac{0.9}{2} + 0.9 * \frac{0.9}{2} * \frac{0.9}{3} + 0.9 * \frac{0.9}{2} * \frac{0.9}{3} * \frac{0.9}{3} + 0.9 * \frac{0.9}{2} * \frac{0.9}{3} * \frac{0.9}{3} * \frac{0.9}{2} + 0.9 * \frac{0.9}{2} * \frac{0.9}{3} * \frac{0.9}{3} * \frac{0.9}{2} * 0.9 + ...) = 1$ After the last addition, we are gonna have repeatedly this plus the term of * 0.9 as the mean service time becomes 1 after n becomes 5. Approximately we get $\pi_0 = 0.4033$ Using the formula from the follow figure we have $\pi_0 = 1 - \rho \Rightarrow \rho = 0.5967$ So finally, from the below So by this formula we have $E[T] = \frac{E[N]}{\lambda} = \frac{\rho}{1-\rho}\frac{1}{\lambda} \Rightarrow \frac{0.5967}{0.4033}\frac{1}{0.9} =$ 1.64641705926 seconds.

(2.) When Bianca decides to have a multi-system queueing system with FCFS and PS, where the concurrently jobs are max 4 we will have different p_0. If we calculate it with the same approach as on the first question we get that : $\pi_0(1 + \frac{0.9}{1} + 0.9 * \frac{0.9}{2} + 0.9 * \frac{0.9}{2} * \frac{0.9}{3} + 0.9 * \frac{0.9}{2} * \frac{0.9}{3} * \frac{0.9}{3}) = 1$ we get that $\pi_0 = 0.4060$ By applying the same rules as the 5.1 question have $\pi_0 = 1 - \rho \Rightarrow \rho = 0.594$ Finally we have $E[T] = \frac{E[N]}{\lambda} = \frac{\rho}{1-\rho}\frac{1}{\lambda} \Rightarrow \frac{0.594}{0.406}\frac{1}{0.9} = 1.62561576355$ seconds.. The intuition behind Bianca's idea is when we have more than 4 concurrent jobs in the system the mean service time (μ) will start decreasing from 3 when it's 4 concurrent jobs to 1 when it goes 6+ concurrent jobs. So by adding a queue on the system she basically try to set a limit on the system jobs in order the maximum mean service time outcome that is on 3-4 jobs concurrently in the system.

Exercise 6.

(1.) So, firstly we need to find out if we are ever going to get back to state 0. Let T_0 be the number of steps to return to state 0 and $T_0 = \infty$ if we never do.

$$\begin{aligned} Pr[T_0 \geq 1] &= 1 && (T_0 \textit{ can never be } 0) \\ Pr[T_0 \geq 2] &= 0.4 && (\textit{Going } 0 \to 1) \\ Pr[T_0 \geq 3] &= 0.4 * 0.4 && (\textit{Going } 0 \to 1 \to 2) \\ Pr[T_0 \geq 4] &= 0.4 * 0.4 * 0.4 = 0.4^3 && (\textit{Going } 0 \to 1 \to 2 \to 3) \\ Pr[T_0 \geq k+1] &= 0.4^k && (\textit{Going } 0 \to 1 \to \cdots \to k) \end{aligned}$$

And because we know the $\lim_{k\to\infty} Pr[T_0 \geq k] = 0.4^{k-1} = 0$ we know that $Pr[T_0 = \infty] = 0$ with probability 1, we do return to 0: eventually. Using the formula $E[X] = \sum_{j=1}^{\infty} Pr[X \geq j]$ and if we know that we start on state 1 we have :
$E[T_0] = 0.4 + 0.4^2 + 0.4^3 + \ldots$ which is a geometric series which diverges. So $E[T_0] = \frac{3}{2}$ which means that we will return to state 0 with expected time $\frac{3}{2}$.

Exercise 7.

If we use the 5 algorithms without any change on the hyper-parameteres the best outcome will be of the MLP algorithm. The achieved accuracies is as follows :

- Logistic Regression accuracy : 85%
- Logistic Regression with PCA accuracy : 62.2%
- Support Vector Machine accuracy : 86.6%
- Decision tree (max_depth = 10) accuracy : 81.1 %
- Random Forest accuracy : 81.8%
- Multi-Layer Perceptron (MLP) accuracy : 90.9%

The first thing we can observe is that the accuracy drops by **22.1%** when we use Principal Component Analysis on Logistic Regression algorithm. That is probably because from the 115 features data, we downscale the data to 2 dimension space by combining the features and thus, we lose much information. Also, we can see that the Neural Network is outperforming the other algorithms as it was kinda supposed to.

- For Logistic Regression and SVM algorithms, there are not many parameters that can be changed. For both we can olny change the penalty as they only different from each other in terms of their loss function. If we set the penalty to none, we get a better training accuracy (85%) but it is more sensitive to have an overfitted model.
- For Decision tree tried different max_depth of the tree from 1 to 30 max_depth and the best was equal to 10.
- For Random Forest algorithm I decided to go with the way of sklearn tuning by using RandomizedSearchCV. This function takes as inputs the different parameters with their respective value range (e.g. num of trees on random forest from 100 to 200 with a step) you want to hyper-tune and then you set up the combinations that needs to be done and how many cross-validation will do. After done that with little combinations (see

code provided) the best parameters is the max_depth=None of the tree, n_estimators=124 which is the number of trees in the forest and min_samples_split=2 which is the minimum samples that are required to split up a node. The accruacy I get from that is **81.2%** and that's **less than** the default parameters. That occurs most likely because we use cross-validation and it lessens the accuracy but it makes the model insensitive to overfit.

– For Multi-Layer Perceptron (Neural Network) is more complicated and time-consuming to tune the hyper parameters, so I have used only little combinations and different things. Firstly, I tried to use multiple layers (2 and 3) but it didn't work out. The combination that had a slightly improvment of 0.4% was two hidden layers with 100 and 50 neurons respectively. So the accuracy we achieved is 91.28%

To conclude, the best performance was done by the Neural Network as it was most likely to. That is probably because the data are not linear seperable so neural networks work really well on this kind of data. To run code the .txt files should be on the same directory as the python script.

Exercise 8.

To begin with, I reduced the levels and replications of each factor because it was taking too long to run the experiments and instead I used 2 levels for each factor and 3 replications. The following table represents the values I used for my experiments which are 24 in total.

Factors	α	β	Q
Level 1	1	1	0.1
Level 2	2	2	0.2

Table 1: Levels for each each parameter.

By running the Exerc8.py script in Google Colaboratory I created the results.csv to use it afterwards on my 3-way ANOVA analysis with the script ANOVA.py. Also, I did change the function train of the main.py file to return the final_accuracy. By running this file the output we get is the ANOVA 3-way table where we calculate the SSE (Sum of Squared Error) from the column sum_sq, and then we calculate the Percentage Variation explained by each factor and their interactions to find which factor is the most significant one.

	sum_sq	df	F	PR(>F)	Percentage_Variation_Explained
C(a)	2.822203	1.0	0.449083	0.512328	2.254579
C(b)	2.105154	1.0	0.334982	0.570798	1.681748
C(Q_bar)	11.412606	1.0	1.816030	0.196560	9.117212
C(a):C(b)	1.392981	1.0	0.221658	0.644134	1.112814
C(a):C(Q_bar)	2.188897	1.0	0.348308	0.563316	1.748649
C(b):C(Q_bar)	0.997969	1.0	0.158802	0.695530	0.797249
C(a):C(b):C(Q_bar)	3.706776	1.0	0.589841	0.453667	2.961240
Residual	100.549909	16.0	NaN	NaN	80.326510

Figure 5: 3-way ANOVA table of factors α, β and Q bar.

By observing Figure 5 we can see that the most significant factor is the Q_bar (threshold) itself with 9% percentage. The second biggest, is the interaction of all factors α, β and Q_bar with 3% percentage. The experiments were done before the change on the exercise.

Exercise 9.

Paper Review : CROSSBOW: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers

1. Student Name : Marinos Marios, Student number : 5353106

2. The problem that the paper addresses, is the fact that modern frameworks(e.g. Tensorflow) which are used to train deep neural networks with multi-GPU systems, are using synchronous stohastic gradient descent which needs large batches to utilise the GPUs effectively and this results to reduction of the statistical efficiency of the training process. The users are trying to fix that by changing the hyper-parameters but not successfully. So, the authors goal is to design a system that effectively train with small batch sizes and meanwhile scaling to many GPUs. The 2 challenges they are going to confront are how to synchronise this potentially large number of model replicas without adversely affecting statistical efficiency and how to ensure that the hardware resources of each GPU are fully utilised, thus achieving high hardware efficiency. Therefore, the contribution the paper does is pretty huge, since recently years the deep neural networks training has become quite popular and the statistical efficiency of the training process is one of the significant factors to consider when you train a DNN as you want to get the optimal solution as fast as possible.

3,4. I strongly believe that the paper is very clear and well written as it constructs very smoothly and that makes it a very solid paper. Firstly, the authors provide and introduction about today's multi GPUs systems works (e.g. Tensorflow) and where the problem lies with these frameworks. After that, they introduce the problems that occurs when you scale the training by increasing the batch size. On 3rd section, authors present the SMA algorithm that is behind their system with their learners and show how to train them and also determine how many to use. Next section they basically describe how the CROSSBOW system will schedule and execute tasks, dynamically tune the learners etc. Finally, every paper that wants to be successful, have an experiments section on which they have to prove and line up with the assumptions they have made on the introduction and this paper does it great. The only downside I can find on the CROSSBOW design, is the fact that if the DNN model we need to train it's small size (e.g. ResNet-32, VGG, etc) and thus we have to use 1 GPU, since if we use more GPUs will lead to higher training time due to synchronisation overhead, the performance of the CROSSBOW is slightly worse than Tensorflow.

5. To conclude, there are a couple of ways to further extend this paper. Firstly, we could try to extend the CROSSBOW system to make it work on distributed clusters. Nowadays big clusters have got a very big increase in their usage, as they are used to train deep neural networks and not only. Therefore, if we manage to combine those two components the impact of that will be tremendous. In addition to that, it could be useful to know how sensitive will the CROSSBOW system be in heterogeneous environments (e.g. CPU, GPU) and how that will affect the performance of the system.