Prediction of Server Throughput via the Internet Using Neural Networks

Carl Anderson

5582311

EE5371

12/20/2020

I. Abstract

In the modern age servers are all around us, websites delivering pages, massive databases serving purchases for online shop fronts, and content delivery networks serving recorded or live video. One of the most frustrating aspects of using the internet is a laggy connection. If you can accurately predict slowdown and adapt the content being sent to reduce required bandwidth without stuttering you can prevent these annoying artifacts. I collected data using a simple Python3 socket server and client which I sent data to via a VPN which allowed me to emulate the server being located in many different locations around the world. I then used this data to train two main models: a multiple linear regression and a multilayer perceptron machine learning model. After training these models, I tested them using a separate set of testing data and concluded that neither of them are particularly good at predicting throughput ($R^2 \approx 0.65$), but are better than any of the inputs used alone.

II. Introduction

The exact problem I am attempting to solve is to determine whether multiple linear regression can be improved upon as a model for predicting server throughput over the internet. The primary motivation for this is that if a client can predict the throughput of a server then it takes less load off of the server in informing the client of throughput throttling that will occur. If the client recognizes that slowdown is imminent it can reduce the amount of data being sent to the server, hopefully preventing long delays from occurring if the server queue becomes too full. The reasoning behind testing a neural network model for this problem is because it is easy to imagine that the input variables for this kind of model could have relatively complex interactions, those between the variance of latency and number of clients connected, for example.

III. Summary of Previous Works

From my research into previous works it seems that most of the effort put into researching this topic has been put towards developing sufficiently advanced queueing models of this type of system. Generally speaking, servers follow queueing models quite well when in controlled environments. As long as you can sufficiently prove you have fulfilled the necessary requirements, the queueing model will be valid, however practically you cannot always do this. For instance, in a standard M/M/1 queueing model, you need to assume that jobs have some standard distribution of how they arrive into the system and are not all bunched up together and that the next state of the queueing model only relies on the previous state [Lilja].

A. Neural Networks

The first prediction model I looked into was neural networks, and I was able to find a good amount of research done into the use of recurrent neural networks for extremely similar problems to the one I am studying. The first is an article by Luo from 2005 which uses a RNN (recurrent neural network) to fit a curve to web server performance and time [Luo]. Experiments showed that their recurrent neural network model was more accurate and effective than quadratic exponential smoothing models and increased accuracy by as much as 5% [Luo]. The next paper which uses RNN's actually cited the Luo paper and builds on the previous work done. This paper, by Peng uses a recurrent neural network with long short-term memory units to predict web server performance through the temporal domain [Peng]. This paper diverges from the Luo paper because instead of fitting their curve to match server performance to time it actually matches server performance to the sequence of user URL requests, which varies over time. The Peng paper also cites another paper which actually matches my solution to the given problem extremely closely; they used a three layer MLP neural network to attempt to predict web server performance [Yu]. Another interesting part of the Yu paper is clustering which they

group or ‘cluster’ the workloads in their dataset using the K-medoid algorithm and then develop a neural network model for each of those clusters [Yu]. I find this extremely interesting but I don’t think it as necessary for my project as the dataset that they were making their models off of was much more diverse than the dataset I am using which was homogeneously gathered all from a single client/server program under different conditions. Another potential drawback is that when the server receives a new job you first need to cluster it to determine which neural network to use, which adds another step between a job added and the performance prediction. Another issue I can foresee is if you have a diverse web server that you expect to deal with many jobs from different clusters it could be quite difficult to incorporate multiple networks into one performance prediction. Another RNN paper I found was by Song in which a RNN was used for predicting performance on a cloud computing platform in order to improve resource allocation and utilization [Song]. I think at this point it is apparent that a recurrent neural network would be much better suited for this problem, however I did think that this paper’s methodology and procedure was good and if I wanted to attempt to use a recurrent neural network I would definitely reference it for a good experimental setup.

B. Queuing Analysis

The next analytical model I considered was a queueing-network analysis model which is a method by which measured and simulated results can be quickly validated. In short, queueing analysis reduces CPUs, processes and complex buffers down to basics: servers complete jobs, queues hold jobs after they have entered the system and before the server has completed the previous jobs, and jobs enter the system by some random distribution [Lilja]. The simplest and primary example of a queueing analytical model in our textbook is the M/M/1 queueing model which I mentioned above. M/M/1 means that we have a one-queue, one-server system which eliminates a lot of unknowns and, with a couple of assumptions about the distribution of the incoming jobs and how jobs are handled by the server, we can make some powerful inferences

about the performance of the system [Lilja]. I began my research into modern queueing models for the purpose of web server performance prediction with a paper by Van der Mei in which a web server is modeled using a tandem queueing model consisting of a submodel at each end of the TCP connection [Van der Mei]. One interesting component of this analysis is that they do not attempt to incorporate real world network slowdowns into their model. They do however incorporate the various nuances of network topologies, TCP algorithms, hardware/software configuration but don't measure over real internet connections [Van der Mei]. The paper by Cao uses a new queueing model, dubbed M/G/1/K*PS queue, which follows a Poisson-distributed birthrate and whose average service time is an arbitrary parameter in the context of the model [Cao]. They were able to obtain web server performance metrics in closed form expressions such as throughput, average response time, and blocking probability. This seems quite interesting because if I can get the required measurements to plug into the closed form equations I may be able to use this model to confirm my model predictions. Andersson was able to model the performance of an Apache web server with a bursty birth rate by representing the job arrivate rate using a Markov Modulated Poisson Process instead of the standard Poisson distributed process [Andersson]. This seems like it might fit my use case well given that most of my experiments had relatively short lifespans (10-20 seconds) with pretty high mean service rate so I am not sure it is totally worth developing an entire simulation around this relatively complex analytical method.

C. Linear Regression

I was able to find a single paper on using linear regression as a tool to predict server performance, although it was in the context of CPU usage which isn't a metric that particularly concerns my problem I decided it would be good to read through it and see if anything methods-wise stood out. Farahnakian uses groups of requests from users to virtual machines to predict host overloading which is interesting because I think this kind of mirrors how I am

attempting to predict throughput as a function of ‘client requests,’ or in my case incoming packets [Farahnakian].

D. Simulation

Simulation is tightly related to queueing analysis and are often intertwined, however I felt it necessary to separate them in the logical structure of my summary of previous works because generally in the papers I read they are either developing the queueing model portion or the simulation portion of the analysis. In his 2020 article, Waqas attempts to simulate the performance metrics of large-scale real-time telecommunication networks using a new simulator developed using XgBoost, Random Forest, and Decision Tree modeling [Waqas]. I think that this is pretty interesting because the internet at its core is really just a very complex and interconnected telecommunication network, though of course running on different hardware. For the purpose of this simulation that should not matter. Another analysis that would need to be done before implementing this simulation framework to be applied to networking systems is whether or not they are as real-time dependent as the examples provided in the paper (most of which being kinematic vehicle problems). Next, Wells uses a simulation technique called coloured Petri nets which is a powerful method for modeling and analyzing large systems [Wells]. Coloured Petri nets used to primarily be applied to logical validation and debugging but Wells is expanding it to be used for modelling & simulating the system at hand. For my purposes CP-nets is a bit intense of a testing methodology and there is not sufficient time to implement it into my evaluation but the paper provides interesting insight if I were to expand the methodology to include this tool.

IV. Theory

For this research project I decided to use the multiple linear regression model as well as the standard multilayer perceptron machine learning model because I believed that the problem

I am dealing with is too far outside the bounds of the queueing model requirements for the results to be valid. Multiple regression and neural network models are both much more flexible and have less requirements to be applicable to different research problems than queueing models, although when the requirements are met, queueing models can be extremely accurate.

A. Multiple Linear Regression

Multiple linear regression works by expanding a standard linear regression model in which the relationship between some input variable and some output variable is determined by attempting to fit a line to the relationship in the form of equation A.

$y = a + bx$ including the residual: $y = a + bx + e$
Equation A: Simple linear regression model form where a and b are regression parameters we wish to determine [Lilja].

We can write all of our input output pairs as (x_i, y_i) and find the value of the residual, e_i, which is the error for each datapoint from our expected trend line. We want to minimize the amount of error over all of our data points, which is referred to as the sum of squared residuals. The equation for our sum of squared residuals is shown in equation B.

$SSE = \sum_{i=1}^{n} e_i^2$ substituting eqn. A: $SSE = \sum_{i=1}^{n} (y_i - a - bx_i)^2$
Equation B: Sum of squared residuals [Lilja].

In order to get the minimum of a complex function we can take the partial derivative with respect to any number of input variables, set the result equal to zero to get the minimums and maximums, and pick the one with the smallest value. This will give us the parameters with the

minimum squared sum of residuals which will ensure that we have the most optimal solution for our model given the training dataset.

Multiple regression works extremely similarly to the simple linear regression I have outlined except that instead of just one input variable you have any of the input variables and you attempt to minimize the squared sum of residuals of the entire dataset with respect to the output variable. We start by expanding equation A to account for many input variables, as in equation C shown below.

$$y = a + b_1 x_1 + b_2 x_2 + b_3 x_3 + \ldots + b_n x_n$$

Equation C: Multiple regression equation [Lilja].

Using this model we do a very similar operation to the single linear regression, taking the partial derivative of the sum of squares of the residual over the entire training dataset, finding the minimum of the system over all of the parameter variables (of which there are only $n + 1$ where n is the number of input variables you have) and then you get your optimal parameters given the training dataset.

B. Multilayer Perceptron Neural Network

Multilayer perceptron neural networks are a relatively old machine learning framework and in my research I couldn't often find people using this sort of neural network for this type of problem. I have decided to attempt to use this style of neural network for the primary reason of it is easy to work with and compared to some neural network architectures such as convolutional or recurrent neural networks is less computationally intensive.

Multilayer perceptron neural networks are the simplest kind of neural network and work on the simple concept of perceptron. A perceptron, modeled off of how real neurons work (though in practice quite different) takes in some number of input values, scales them by a

constant weight, adds a bias, then takes that value and puts it through some activation function, and finally passes that value on to the next layer. Inputs can be input variables similar to a linear regression or outputs from previous layers. Taking many of these layers and stringing them together creates a deep neural network as shown in figure A. Generally, the more perceptrons you have over more layers, the more complex the function you can model. Equation D shows the output at a particular perceptron in a neural network.

$$y = b + w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_n x_n$$

Equation D: Perceptron output equation where b is the perceptron bias, w is the value of the weight for a particular input x.

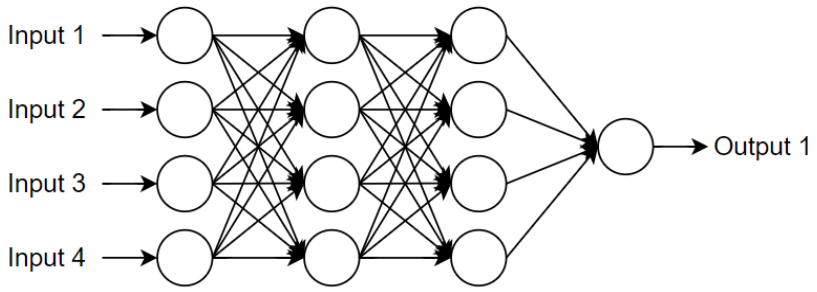

Figure A: An example multilayer perceptron deep neural network.

The parameters of the neural network are as follows: each perceptron has a certain amount of inputs, for the first layer, each will have one weight for each input variable (in the above example, 4) and each perceptron will have a bias. Because you can have different numbers of perceptrons in each layer and each layer will take the previous layers' outputs as inputs, the number of parameters can be modelled by the total number of perceptrons (for the biases) and the number of nets connecting all of the perceptrons (for the weights). This is far more parameters to optimize than the multiple regression which only had $n + 1$ with n being the number of input variables.

In order to get the minimal error for the neural network, a technique called gradient descent is paired with backpropagation. This is done by taking the output of your neural network, finding the mean squared error or the average square of the difference between the neural network output and the real training data output and taking the gradient of it. The equation for mean square error is shown by equation E. This mean squared error is found in terms of the various neural network parameters which allows us to use a bit of multivariable calculus called gradient descent which is when you take the partial derivative of a function of a scalar quantity with respect to different input variables, in our case the network parameters, and determine the direction which minimizes the scalar quantity. We are attempting to minimize the scalar quantity of the mean square error with respect to the input variables of the network parameters. Once we have solved this for the change in parameters which will minimize our mean squared error we can do backpropagation, in which we “propagate” the parameter changes which minimize the error of our neural network through the network.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Equation E: Mean squared error.

Doing this pattern of running training data through our network, performing gradient descent, getting our optimal parameter updates, and backpropagating them through our network over many times (called “epochs”) allows us to train the network to “learn” the mathematical relationship we are attempting to find, if there is one. One key part of neural network theory that I must touch on is generalization or overtraining which is a danger of neural networks in which you have enough parameters and a small enough set of training data such that your network can act as a memory bank, completely remembering your exact dataset, preventing it from being generalizable to solve your larger problem. I believe that I haven’t fallen into this trap by taking two major precautions. 1. I split my dataset according to industry standard bounds,

ensuring that my training and testing datasets were of similar sizes and randomly chosen and 2. I kept my number of perceptrons low with respect to the number of input variables I had, only having 8, 6, and 4 perceptrons over 3 hidden layers.

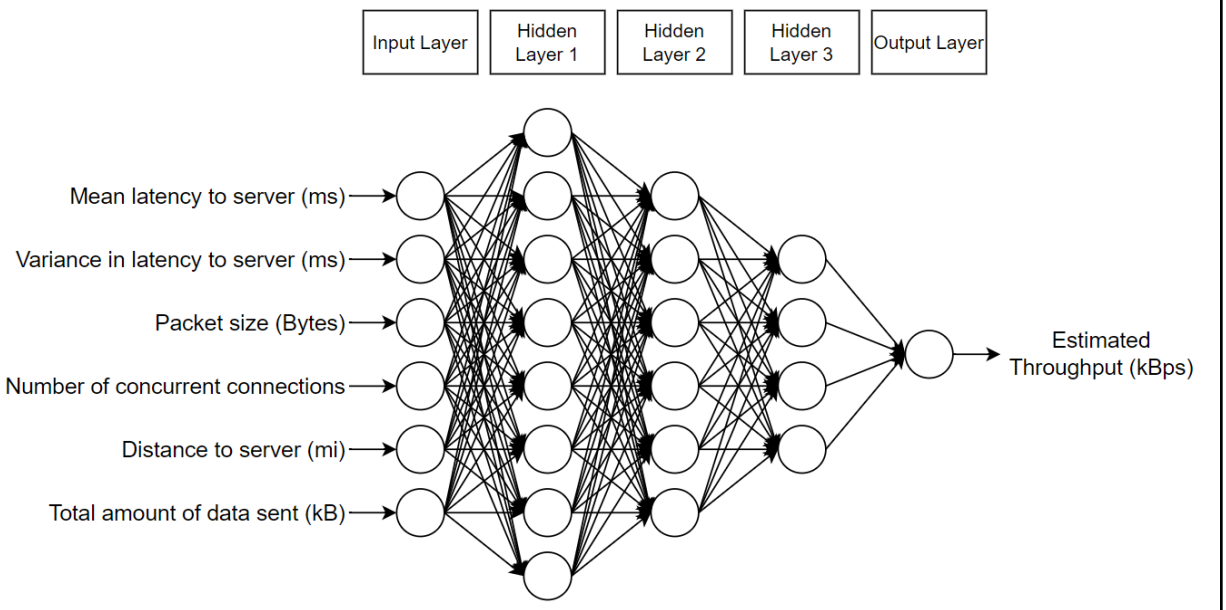

Figure B: The implemented neural network architecture.

Figure B shows the neural network architecture that was implemented with labelled input and output variables along with units. As you can see, there are three hidden layers along with one input and one output layer. The hidden layers have eight, six, and four perceptrons each.

V. Methodology

The experimental methodology behind my project is relatively simple. In order to gather data for a server's throughput I built a custom Python script to act as a simple echo socket server and another Python script to act as a client or number of clients using threads. Something I wanted to be able to incorporate into my model is having all of the traffic run through the internet instead of a private closed off network, to attempt to see if the model can take into account real world networking issues such as real latency, or routing delays.

The server program simply opens a socket and listens for incoming traffic to the server, when the server receives a new connection it will echo the received packets back to the client and then close the connection. It is important to note that the server does not take advantage of threading or multiprocessing in any way, which means that in a theoretical queueing model it would be a one server/one queue system. The client program is a bit more complicated. It starts by measuring the latency to the server by opening a connection to the server, sending a very small amount of data, and measuring the delay. I do this thirty times to get a good feeling for the variance of this value and then save the mean and standard deviation of the latency to the server. Next, the client performs throughput tests by spinning a varying number of threads, sending a set number of packets (1000) over various packet sizes, and measuring the time for all of the data to return to the clients. I made sure to iterate through all combinations of concurrent clients connected and packet sizes to ensure that the training data covers the full variable domains. Throughput was calculated using equation F, which is roughly the amount of data each client was able to push through the server divided by the time it took to push the data through the server.

$$\mu = 1000\frac{S_{byte}}{t_{end} - t_{start}}$$
Equation F: Throughput, S_{byte} is the packet size in bytes.

In order for my model to be able to take into account real world networking delays, I needed some way of quantifying the parameters of the network between the client and server. I decided to use latency as it is very easy to measure and relatively accurate as a measure of the quality of a network connection to a server, or at least as good as you can get without directly measuring bandwidth. However, in order to get a range of different qualities of the network over which I am transferring data I needed some way of moving the server around the world to simulate connecting to different server locations. This could have been done with a virtual server

host such as Amazon Web Services however these services generally cost money and are complex to set up if you haven't done any previous work with them before and generally have limited options for locations which you can have your server hosted. I opted for an alternative strategy which I believe is roughly comparable in practice, leveraging a VPN connection to introduce realistic artificial networking delays. If I wanted to simulate my server being in Brazil, instead of praying that AWS hosts servers in Brazil and then buying a virtual server in Brazil and loading my script onto it to run benchmarks, I would just connect the computer with the client running on it to a VPN which would route all of my outgoing and incoming traffic through a server located in Brazil which practically gives the exact same effect, that being that all of my packets go through the same inter-tubes as if they were going to the server in Brazil. This also has the added benefit that VPN services often have an extremely large range of options of where you can have your traffic routed through. For instance, the service I used, Private Internet Access, has servers in 78 different countries.

My neural network model was developed using one hundred epochs, a batch size of ten, mean squared error as the loss metric, and ReLU (rectified linear unit) as the activation function. For both the neural network and the multiple regression a testing to training data split of 30% / 70% was used.

VI. Results

The primary result gathered was the R^2 or mean square error for each of the models trained. This was determined by subtracting the ratio of the sum of squares of the residuals to the total sum of squares from one and is a standard method of determining the coefficient of determination. In addition to the multiple regression and MLP models, I did a single linear regression for each of the input variables and calculated the R^2 of that model.

Table A: R^2 of each model developed	
Model Name	R^2
Neural Network	0.687
Multiple Regression	0.647
Mean latency	0.122
Variance in latency	0.0158
Packet size	0.181
Number of connections	0.192
Distance to server	0.0653
Total amount of data sent	0.000591

Table A contains the R^2 values for each of the models which was obtained using equation G as shown below. R^2 score or coefficient of determination is an industry standard in machine learning for evaluating the performance of a certain machine learning model which is why I am confident in it as a model result metric.

$$R^2 = 1 - \frac{RSS}{TSS}$$

Equation G: R^2 or coefficient of determination, RSS is the residual sum of squares and TSS is the total sum of squares.

Table B contains the resulting parameters for each of the resulting linear regression models including the multiple linear regression model.

Table B: Regression models with parameters	
Model	Regression Equation
Multiple Regression	$Y = -166000A - 315000B + 163C - 845D - 2.77E - 0.0194F + 90500$
(A) Mean latency	$Y = 0.00122A + 45100$
(B) Variance in latency	$Y = 52.5B + 34700$

(C) Packet size	$Y = -5150C + 74600$
(D) Number of connections	$Y = -214000D + 51700$
(E) Distance to server	$Y = -193000E + 87900$
(F) Total data sent	$Y = -6.41F + 67100$

It should be noted that in the above table the different models were created independent of each other from scratch, though they were tested using the same partition of training/testing data.

VII. Analysis

As you can see from the table of R^2 values, neither of the models performed exceptionally well. The neural network did perform marginally better than the multiple linear regression, though the difference in R^2 values is minor if not completely negligible. Something of note is that a 'good' R^2 values for a lot of machine learning applications is generally 0.99+ and anything below 0.90 is considered weak, after all if your model is only correct nine out of ten times it is not amazing at classifying that relationship compared to say a human who will never mistake a fire hydrant for a dog. However, this is not a 'standard' machine learning application and I was not going into it knowing whether or not I was even using good input variables.

Another interesting result is how weak all of the single linear regression models were. None of them had an R2 above ~0.19 which even in linear regression terms is considered to be weak to no correlation. This means despite the multiple regression and neural network having comparatively low coefficients of determination, there is still definitely some underlying mathematical relationship between the various input variables which they are modelling better than if you just took any one of those variables.

The strongest of the single linear regression models were the number of concurrent connections, packet size, and then mean latency. All of the other parameters had R^2 scores far

below these three which were all within ~0.06 of each other. I think that these parameters being the most important definitely makes sense, the number of clients sending data to the server all at once will obviously make it take longer to server each of those connections. As for the packet size I think that this is a result of me doing my experiments via the internet including all of the associated end-to-end delays, as realistically there will always be some set time cost to per packet sent and, we all know that it is more efficient to send fewer large packets than many smaller packets (as long as you are not experiencing any packet loss). The mean latency also makes sense as an important parameter, given that the throughput was measured end-to-end the network latency was included in the calculation of the throughput and therefore if a server is further away from a client it will of course take longer to get a given amount of data processed by the server. I was slightly surprised that the variance of the latency didn't play a larger role, as I figured that a larger variance in latency would indicate a less stable connection which could cause all kinds of problems in terms of end-to-end server throughput.

VIII. Conclusion

This project has presented an overview of the numerous ways in which models are developed for the purpose of predicting web server performance. I have chosen two of these methods which are less commonly used, linear regression and multilayer perceptron neural networks, and compared them as tools for developing a way to best solve this problem. In the broader context, much work has been put into more accurate models for this problem however most of them are extremely resource intensive to construct and complex to use such as simulation and advanced queueing analysis. The tools that I used are simple and chosen on purpose to see if you can get a decent prediction, at the least better than guessing, using these relatively simple statistical tools. While the models I developed are not cutting edge, I think that given the ease of access they are more than impressive in their ability to predict server performance while taking into account real work internet networking delays.

Annotated Bibliography

Lilja, David J. Measuring Computer Performance: A Practitioner's Guide. Cambridge University Press, 2005, Cambridge Core, https://doi-org.ezp2.lib.umn.edu/10.1017/CBO9780511612398, Accessed 19 Dec. 2021.

Class textbook which was an invaluable resource on the topics of multiple linear regression, queueing analysis, as well as general design of experiment and analysis of data.

Yu, Yongjia, et al. "Integrating clustering and learning for improved workload prediction in the cloud." 2016 IEEE 9th International Conference on Cloud Computing (CLOUD). IEEE, 2016.

This article used a standard MLP deep learning model to predict workload on a cloud computing server serving jobs for many clients that may have varying usage needs. They used 3 hidden layers with 9, 3, and 1 perceptrons each.

Peng, Jiajun, Zheng Huang, and Jie Cheng. "A deep recurrent network for web server performance prediction." 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC). IEEE, 2017.

Extremely interesting article which uses a recurrent neural network with long short-term memory perceptrons to predict the performance of a web server to extreme accuracy.

Luo, Bin, and Shi-wei Ye. "Server performance prediction using recurrent neural network." Computer Engineering and Design 8 (2005): 57.

Similar to the other one about deep recurrent neural networks but slightly less in depth. Side note: difficult to access.

Song, Binbin, et al. "Host load prediction with long short-term memory in cloud computing." The Journal of Supercomputing 74.12 (2018): 6554-6568.

Another SLTM RNN for server throughput prediction.

Van der Mei, Robert D., Rema Hariharan, and P. K. Reeser. "Web server performance modeling." Telecommunication Systems 16.3 (2001): 361-378.

Good resource on the queueing model aspect. Gives good insight into how one would perform a real simulation to model the performance of this problem.

Cao, Jianhua, et al. "Web server performance modeling using an M/G/1/K* PS queue." 10th International Conference on Telecommunications, 2003. ICT 2003.. Vol. 2. IEEE, 2003.

Another more advanced queueing model which more accurately describes the webserver than a standard M/M/1 queueing model. Seems significantly more complicated, however.

Andersson, Mikael, et al. "Performance modeling of an apache web server with bursty arrival traffic." 4th International Conference on Internet Computing. CSREA Press, 2003.
The queueing model in this paper describes how often times standard queueing models such as M/M/1 are not sufficient when the axioms break down, such as service distribution not being exponentially distributed or deterministic.

Farahnakian, Fahimeh, Pasi Liljeberg, and Juha Plosila. "LiRCUP: Linear regression based CPU usage prediction algorithm for live migration of virtual machines in data centers." 2013 39th Euromicro conference on software engineering and advanced applications. IEEE, 2013.
Very interesting paper, the problem that they are looking at is different than mine but they apply a similar technique to predict the performance of CPUs in data centers. Good resource for regression-based models and analysis.

Waqas, Muhammad. "A simulation-based approach to test the performance of large-scale real time software systems." (2020).
Another simulation method which could be useful for modelling large systems such a web server with a large interconnection in between the client and server, if you are able to accurately simulate internet network variance.

Wells, Lisa, et al. "Simulation based performance analysis of web servers." Proceedings 9th International Workshop on Petri Nets and Performance Models. IEEE, 2001.
Introduces another method of simulation which I am unable to reproduce in the given time for my project however it was very interesting to read and gave general insight into the performance metric space.

Appendix - Commented Python3 Code

```
import socket

# Create server socket.
serv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, proto=0)

# Bind server socket to loopback network interface.
serv_sock.bind(('127.0.0.1', 6543))

# Turn server socket into listening mode.
serv_sock.listen(10)

while True:
    # Accept new connections in an infinite loop
    client_sock, client_addr = serv_sock.accept()
    print('New connection from', client_addr)

    chunks = []
    while True:
        # Keep reading while the client is writing.
        data = client_sock.recv(2048)
        if not data:
            # Client is done with sending.
            break
        chunks.append(data)
    client_sock.sendall(b''.join(chunks))
    client_sock.close()
```

server.py

```
import socket, time, numpy as np, string, random
from multiprocessing import Process
import sys

random.seed(time.time())
times = []

for i in range(30):
    # Create client socket.
    client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect to server (replace 127.0.0.1 with the real server IP).
    client_sock.connect(("75.72.170.157", 6543))

    start = time.time()

    # Send some data to server.
    sent_data = b'Hello, world'
    client_sock.sendall(sent_data)
    client_sock.shutdown(socket.SHUT_WR)

    # Receive some data back.
    chunks = []
    while True:
        data = client_sock.recv(2048)
        if not data:
            break
        chunks.append(data)
```

```
    diff = time.time() - start

    # Measure the latency
    print(result, 'took', 1000*diff, 'ms')
    times.append(diff)

    # Check validity of received data
    result = {True: "Success", False:"Failure"}[str(sent_data) ==
str(repr(b''.join(chunks)))]

    # Disconnect from server.
    client_sock.close()

# Print latency mean and standard deviation
print(np.mean(times), np.std(times))

def spin_thread(index, n_bytes):
    # Thread dumps a bunch of data to the server
    # and measures the time to receive it back

    client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Connect to server (replace 127.0.0.1 with the real server IP).
    client_sock.connect(("75.72.170.157", 6543))

    all_data = b''

    # Send some data to server.
    for i in range(1000):
        sent_data = "".join(random.choices(string.printable, k=n_bytes)).encode()
        client_sock.sendall(sent_data)
        all_data += sent_data
    client_sock.shutdown(socket.SHUT_WR)

    # Receive some data back.
    chunks = []
    while True:
        data = client_sock.recv(2048)
        if not data:
            break
        chunks.append(data)

    # Disconnect from server.
    client_sock.close()

# Input values for number of concurrent connections
n_conns = [i for i in range(1,11)]
# Input values for packet size
bytes_list = [1,5,10,25,50,75,100,250,500,750,1000]

# Iterate over variables
for conns in n_conns:
    for n_bytes in bytes_list:
        threads = []
        # Spin one thread per concurrent connection
        for i in range(conns):
            x = Process(target=spin_thread, args=(i,n_bytes,))
            threads.append(x)

        # Start all threads
        for i in threads:
            i.start()
```

```
        # Measure the time for all threads to complete
        start = time.time()
        for i in threads:
            i.join()
        diff = time.time() - start

        # Print some statistics
        print('Transferred', conns*1000*n_bytes, 'bytes')
        print('Took', 1000*diff, 'ms')
        print(conns*1000*n_bytes / diff / 1000, 'kBps')

        # Save important data to csv file
        with open("data.csv", "a") as outfile:
            outfile.write(str(conns*1000*n_bytes) + ",")
            outfile.write(str(n_bytes) + ",")
            outfile.write(str(conns) + ",")
            outfile.write(str(np.std(times)) + ",")
            outfile.write(str(np.mean(times)) + ",")
            outfile.write(str(sys.argv[1]) + ",")
            outfile.write(str(1000*n_bytes / diff) + "\n")
```

client.py

```
import numpy as np
import random, time
from sklearn import linear_model

random.seed(time.time())

# New multiple regression
clf = linear_model.LinearRegression(fit_intercept=True)

# Import and trim all data
data = []
for line in open("data.csv", "r"):
    data.append(line.replace("\n", "").split(","))
data = data[1:]

for i in range(len(data)):
    tmp = []
    for j in data[i]:
        tmp.append(float(j))
    data[i] = tmp

# Allocate testing data
test_data = []
while len(test_data) / (len(test_data) + len(data)) < 0.5:
    test_data.append(data.pop(random.randint(0, len(data) - 1)))

# Train multiple regression
clf.fit([i[:-1] for i in data], [i[-1] for i in data])

# Print regression parameters
print(clf.coef_)
print(clf.intercept_)

# Print R2 score for testing data in the model
print(clf.score([i[:-1] for i in test_data], [i[-1] for i in test_data]))
```

```
print("R2 for each input variable:")

# Run a single linear regression for each of the input variables
# And print the R2 score for each of those models
for i in range(6):
    clf = linear_model.LinearRegression(fit_intercept=True)
    clf.fit([[x[i]] for x in data], [x[-1] for x in data])
    print(clf.score([[x[i]] for x in test_data], [x[-1] for x in test_data]))
```

regression.py

```
from sklearn.metrics import r2_score
import numpy as np
from keras.layers import Dense, Activation
from keras.models import Sequential
from sklearn.model_selection import train_test_split
# Importing the dataset
dataset = np.genfromtxt("data.csv", delimiter=',',skip_header=True)
X = dataset[:, :-1]
y = dataset[:, -1]

# Splitting the dataset into the Training set and Test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.08,
random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Initialising the ANN
model = Sequential()

# Adding the input layer and the first hidden layer
model.add(Dense(8, activation = 'relu', input_dim = 6))

# Adding the second hidden layer
model.add(Dense(units = 6, activation = 'relu'))

# Adding the third hidden layer
model.add(Dense(units = 4, activation = 'relu'))

# Adding the output layer
model.add(Dense(units = 1))

# Compiling the ANN
model.compile(optimizer = 'adam', loss = 'mean_squared_error')

# Fitting the ANN to the Training set
model.fit(X_train, y_train, batch_size = 10, epochs = 100)

# Print R2 score for the testing dataset
y_pred = model.predict(X_test)
print(r2_score(y_test, y_pred))
```

nn.py