Using Machine Learning to Predict Future Load Testing Patterns based on Website Load Data

Chaitanya Deepti Gadireddi
Software Engineering
Arizona State University
Tempe, Arizona
cgadire1@asu.edu

Divya Sri Sai Bojanki
Software Engineering
Arizona State University
Tempe, Arizona
dbojanki@asu.edu

Shreya Thummalapalli
Software Engineering
Arizona State University
Tempe, Arizona
sthumm14@asu.edu

***Abstract*—In order to guarantee the performance and stability of software systems under varied circumstances, load testing is essential. Conventional load testing methods are frequently reactive; they are unable to anticipate future load patterns or proactively address possible problems. In this work, we suggest a unique method for forecasting future load patterns based on historical data: the use of convolutional neural networks (CNNs). By reliably predicting load trends, our CNN-based methodology goes beyond reactive load testing and allows for preemptive resource modifications to avoid bottlenecks. Constantly comparing results to expectations makes it easier to identify abnormalities early on, allowing for quick inquiry and correction to protect user experience. Additionally, by forecasting future requirements, the model optimizes resource allocation and guarantees economical provisioning in line with expected load levels. By using a transformative approach, load testing becomes a proactive activity instead of a reactive one, improving system performance and stability.**

***Index Terms*—Load Testing, CNN, Machine Learning, Apache JMeter, Website performance**

I. Introduction

A crucial part of developing and maintaining software is load testing, which makes sure that programs can manage the anticipated demand without experiencing system failures or performance deterioration. Reactive tactics are frequently used in traditional load testing methods, where testers examine system behavior under present loads without the capacity to forecast future trends. Reactive techniques might not be enough to guarantee optimal performance, nevertheless, given the growing volume and complexity of contemporary software systems. A unique method that uses Convolutional Neural Networks (CNNs) to forecast future load patterns based on historical data in order to get around this restriction is used. Our CNN-based algorithm can precisely predict load trends and spot possible bottlenecks before they arise by examining previous load data. With the help of this prescriptive methodology, testers can proactively modify resources to guarantee seamless system functioning even during peak loads.

II. Related Work

A thorough method for projecting future resource loads in Internet-based systems is presented in the paper, with a special emphasis on scenarios involving heavy-tailed workloads [1]. It presents a two-step process that combines prediction and load tracking with the goal of assisting in real-time decision-making in such systems. The study first addresses the difficulties associated with heavy-tailed workloads and the shortcomings of conventional time series models in terms of effectively capturing load trends. The design and assessment of efficient load trackers, such as cubic splines and linear trackers like Simple Moving Average (SMA) and Exponential Moving Average (EMA), are then suggested in order to precisely record the load trend of resources. To ascertain the efficacy of these load trackers, their accuracy is thoroughly assessed in a range of workload conditions.In order to anticipate future load trends using historical data, the research expands on the load tracking phase and presents a linear load predictor model based on the linear regression of load tracker results. Prediction error metrics are used to evaluate the load predictor's accuracy. EMA and cubic spline predictors are compared to see which is more accurate in predicting future load patterns. To contextualize the suggested strategy, the study also offers a thorough analysis of the body of research on load prediction, load balancing, admission control, and system scalability in web-based systems. The research concludes by recommending the use of the suggested two-step approach and highlighting how crucial precise load tracking and prediction are to efficient resource management in Internet-based systems with heavy-tailed workloads.

A. CNN Model:

Convolutional Neural Networks can recognize relationships and trends within images because they are inspired by the structure of the human visual brain. For applications like picture recognition, object detection, and facial analysis, this makes them extremely helpful.

Layers such as convolutional, pooling, and fully connected layers are used in the construction of CNNs. Using filters,

the convolutional layers extract features and patterns from the picture data. See the image as a checkerboard sliding over a chessboard as you move the filter across it. By highlighting the existence of the pattern it is detecting in the image, the filter output builds a feature map. To control complexity and avoid overfitting, pooling layers minimize the dimensionality of the data. Following the processing of the image by the convolutional and pooling layers, the image is classified by fully connected layers.

CNNs use the optimization of filters, commonly referred to as kernels, to teach themselves feature engineering. Transformed feature maps are created by combining the input features that these filters slide over. CNNs employ regularized weights over fewer connections to avoid the vanishing or exploding gradients that plagued earlier neural networks during backpropagation. Convolutional, pooling and fully linked layers are among the layers that make up CNNs. Convolutional layers extract features from input data by applying filters. By downsampling the feature maps, pooling layers lower computing complexity.

B. Google Search Console:

Search Console tools and reports help you measure your site's Search traffic and performance, fix issues, and make your site shine in Google Search results [2]. Google offers webmasters, website owners, and SEO specialists a powerful tool called Google Search Console (previously known as Google Webmaster Tools) to help them monitor, maintain, and troubleshoot their website's appearance in Google search results. It gives you information on clicks, impressions, click-through rates, and average position for particular keywords and pages, among other metrics that indicate how well your website is performing in Google search results. It displays the pages on your website that Google has indexed as well as the ones that have indexing problems. Given the popularity of mobile search, it helps you make sure your website is mobile-friendly and optimized for mobile consumers by alerting you to any difficulties with mobile usability. It tells you about any rich results (including rich cards, featured snippets, and other enhanced search results) for which your website might be qualified as well as about any mistakes or problems that keep your material from showing up as rich results. Load data required for a website can be generated using this tool.

III. METHODOLOGY

A. Load Testing

Load testing, done with special software, puts pressure on your website to see if it stays steady. The software checks how fast your website responds to actions. If your website slows down or gets shaky under pressure, it probably can't handle more users. This means you need to fix the problem to keep your website running smoothly.

Load testing is a crucial test to determine whether your software development project is ready for distribution when it is almost finished. You can ascertain your web application's breaking point (should it occur below the peak load condition) and how it will operate under regular and peak load levels with this kind of performance testing. Fundamentally, load testing is done to verify that your online application satisfies the performance goals or objectives you have in mind, which are usually stated in a service level agreement (SLA).

Load testing is essential to verifying that your application can operate correctly under realistic load situations, since more users than ever before rely on online apps to obtain goods or services. Load testing not only reduces the chance that your software may malfunction, but it also reduces the chance that users will become irate with downtime and stop using it completely, which could negatively impact your business's profits [3].

Load testing helps developers check things like:

- How fast the system works, especially during busy times.
- How much of the computer's power is being used?
- How well the hardware, like the CPU and memory, is performing.
- How the load balancer, which distributes web traffic, is working.
- If there are problems when many people are using the software at once.
- If there are any mistakes in the software's design or how it works.
- How many users can use the app before it slows down or breaks?

For example, a tax company might use load testing to see if their website can handle lots of people during tax season.

Here, our objective is to predict future patterns in Load Testing. We are doing this by leveraging the powerful prediction abilities of Convolutional Neural Networks. These predictions can be helpful to improve Load Testing in the following ways:

- Forecasting Load Trends: By analyzing historical load testing data using CNNs, testers can develop models that accurately predict future load patterns [7]. These predictions enable organizations to anticipate potential spikes or changes in load, such as increased traffic during peak hours or seasonal variations. With this foresight, testers can proactively adjust resources or configurations to ensure system stability and prevent performance degradation.
- Identifying Anomalies: CNNs are adept at detecting anomalies or irregularities in load patterns that may signify performance issues or unexpected behavior [7]. By continuously monitoring load data and comparing it to predicted patterns, CNN models can identify deviations indicative of potential problems. Early detection of anomalies allows testers to investigate and address underlying issues before they impact users, thereby maintaining system reliability and user satisfaction.
- Optimizing Resource Allocation: Accurate predictions of future load patterns provided by CNNs enable testers to optimize resource allocation more effectively [8]. By aligning resource provisioning with predicted loads, testers can ensure optimal performance without unnec-

essary over-provisioning, which can be costly, or under-provisioning, which can lead to performance degradation under heavy loads. This optimization enhances resource utilization and cost-effectiveness while maintaining system performance.

- Enhancing Scalability Testing: CNN predictions offer valuable insights into how the system behaves under different load scenarios, facilitating scalability testing efforts. Testers can simulate various load levels based on predicted patterns to evaluate the system's scalability and determine its capacity to handle increased loads [9]. This approach enables organizations to proactively identify scalability limitations and plan for future growth or demand spikes, ensuring that the system can scale effectively to meet evolving requirements.
- Improving Test Planning: CNN-based load predictions inform test planning by helping testers identify critical periods of high load or potential stress points in the system [10]. Test scenarios can be prioritized based on predicted load patterns, ensuring that the most relevant and impactful tests are conducted. This targeted approach to test planning enhances test coverage and effectiveness, enabling testers to focus their efforts on areas of the system that are most likely to experience performance challenges.
- Continuous Performance Monitoring: Integrating CNN models into performance monitoring systems enables continuous monitoring of load patterns in real-time. Testers can receive alerts or notifications when deviations from predicted patterns occur, allowing for prompt action to maintain system performance. This proactive monitoring approach minimizes downtime and performance issues by enabling timely interventions and adjustments based on real-time insights provided by CNN predictions [10].

B. Data Set Creation:

The open-source Apache JMeterTM software is made entirely of Java and is intended to load test functional behavior and gauge performance. Although it was first created to test Web applications, it has recently been expanded to include other test features. It can be used to assess the resilience of a server, network, item, or set of servers by simulating a high load, or it can be used to examine overall performance under various load scenarios [11].

We used Apache JMeter to collect our dataset. We chose the Arizona State University website to perform Load Testing on. First, we added ThreadGroups. Since the ThreadGroups reflect the number of users, we manually changed the ThreadGroup with every Load Test result generation to sequential values from very low to very high [11]. Next, we added all the Arizona State University website sections as links in the HTTP Requests. HTTP Requests are Samplers.

JMeter is instructed to send requests to servers and wait for a response by samplers. They are handled in the tree's order of appearance. It is possible to change a sampler's repeat count

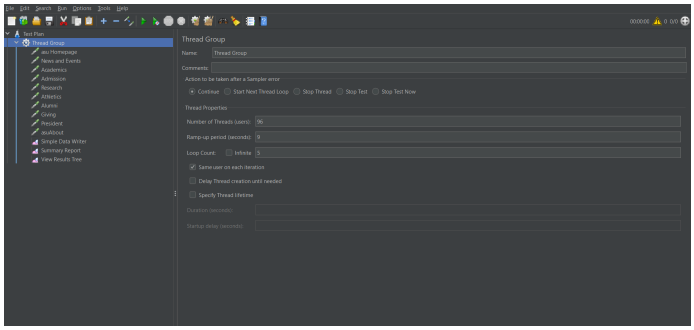

Fig. 1. JMeter Set Up

Fig. 2. JMeter Listeners and Simple Data Writer

by using controllers. Finally, a listener called Summary Report was added. Data was obtained from Simple Data Writer. This element saves all HTTP request results in a CSV file and displays a visual data model. We created a test plan in this way.

The basis of any test plan is its thread group elements. Every controller and sampler needs to belong to a thread group. If additional components, like listeners, are positioned directly under the test plan, they will apply to every thread group. As its name suggests, the thread group element regulates how many threads JMeter uses to run your test [4]. You can do the following using a thread group's controls:

- Decide how many threads there are.
- Decide on the ramp-up time.
- Decide how many times to run the test.

Every thread will carry out the test plan in its entirety, working in isolation from the other threads. To mimic concurrent connections to your server application, you can use several threads.

JMeter is informed by the ramp-up period about how long it will take to "ramp-up" to the total number of threads selected. JMeter will take 100 seconds to start all 10 threads if 10 threads are used and the ramp-up period is 100 seconds. Ten (100/10) seconds after the commencement of the previous thread, each new thread will begin. Every additional thread will be delayed by 4 seconds if there are 30 threads and a 120-second ramp-up period.

A test's ramp-up period should be both sufficiently lengthy to prevent an excessive amount of work at the beginning and sufficiently short to prevent the last threads from starting to run before the first ones do unless that is what is intended [11].

As you need to alter, start with Ramp-up = number of

timeStamp	elapsed	label	responseCode	responseMessage	threadNa	dataType	success	bytes	sentBytes	grpThreac	URL	Latency	Connect
1.71E+12	41	Admission-0	301	Moved Permanently	Thread Group 1-11		TRUE	335	123	30	http://www.asu.edu/	41	19
1.71E+12	24	Admission-1	429	Too Many Requests	Thread Gr	text	FALSE	539	123	30	https://www.asu.edu/	24	0
1.71E+12	70	Athletics	429	Too Many Requests	Thread Gr	text	FALSE	874	246	30	https://www.asu.edu/	44	23
1.71E+12	44	Athletics-0	301	Moved Permanently	Thread Group 1-7		TRUE	335	123	30	http://www.asu.edu/a	44	23
1.71E+12	26	Athletics-1	429	Too Many Requests	Thread Gr	text	FALSE	539	123	30	https://www.asu.edu/	26	0
1.71E+12	134	President	200	OK	Thread Gr	text	TRUE	156784	368	30	https://www.asu.edu/	60	21
1.71E+12	60	President-0	301	Moved Permanently	Thread Group 1-13		TRUE	335	123	30	http://www.asu.edu/p	60	21
1.71E+12	33	President-1	301	Moved Permanently	Thread Gr	text	TRUE	62265	123	30	https://www.asu.edu/	32	0
1.71E+12	41	President-2	200	OK	Thread Gr	text	TRUE	94184	122	30	https://www.asu.edu/	29	0
1.71E+12	51	Admission	429	Too Many Requests	Thread Gr	text	FALSE	874	246	30	https://www.asu.edu/	32	15
1.71E+12	32	Admission-0	301	Moved Permanently	Thread Group 1-29		TRUE	335	123	30	http://www.asu.edu/a	32	15
1.71E+12	19	Admission-1	429	Too Many Requests	Thread Gr	text	FALSE	539	123	30	https://www.asu.edu/	19	0
1.71E+12	57	News and Events	429	Too Many Requests	Thread Gr	text	FALSE	876	250	30	https://www.asu.edu/	41	19
1.71E+12	41	News and Events-0	301	Moved Permanently	Thread Group 1-15		TRUE	337	125	30	http://www.asu.edu/n	41	19
1.71E+12	16	News and Events-1	429	Too Many Requests	Thread Gr	text	FALSE	539	125	30	https://www.asu.edu/	16	0
1.71E+12	57	President	429	Too Many Requests	Thread Gr	text	FALSE	874	246	30	https://www.asu.edu/	42	19
1.71E+12	42	President-0	301	Moved Permanently	Thread Group 1-4		TRUE	335	123	30	http://www.asu.edu/p	42	19
1.71E+12	15	President-1	429	Too Many Requests	Thread Gr	text	FALSE	539	123	30	https://www.asu.edu/	15	0
1.71E+12	64	President	429	Too Many Requests	Thread Gr	text	FALSE	874	246	30	https://www.asu.edu/	47	25
1.71E+12	47	President-0	301	Moved Permanently	Thread Group 1-3		TRUE	335	123	30	http://www.asu.edu/p	47	25
1.71E+12	16	President-1	429	Too Many Requests	Thread Gr	text	FALSE	539	123	30	https://www.asu.edu/	16	0
1.71E+12	68	asu Homepage	429	Too Many Requests	Thread Gr	text	FALSE	864	226	30	https://www.asu.edu/	39	16
1.71E+12	39	asu Homepage-0	301	Moved Permanently	Thread Group 1-17		TRUE	325	113	30	http://www.asu.edu/	39	16
1.71E+12	27	asu Homepage-1	429	Too Many Requests	Thread Gr	text	FALSE	539	113	30	https://www.asu.edu/	27	0
1.71E+12	70	Alumni	429	Too Many Requests	Thread Gr	text	FALSE	871	240	30	https://www.asu.edu/	42	20
1.71E+12	42	Alumni-0	301	Moved Permanently	Thread Group 1-23		TRUE	332	120	30	http://www.asu.edu/a	42	20
1.71E+12	27	Alumni-1	429	Too Many Requests	Thread Gr	text	FALSE	539	120	30	https://www.asu.edu/	27	0
1.71E+12	67	President	429	Too Many Requests	Thread Gr	text	FALSE	874	246	30	https://www.asu.edu/	46	19
1.71E+12	46	President-0	301	Moved Permanently	Thread Group 1-22		TRUE	335	123	30	http://www.asu.edu/p	46	19

Fig. 3. Dataset

threads and go from there. The thread group is set up to loop through its elements once by default.

Thread lifetime can also be specified using Thread Group. To enable or disable additional fields where you can input the test length and startup delay, click the checkbox located at the bottom of the Thread Group panel. To adjust the length of each thread group and the number of seconds it starts, you can set the Duration (seconds) and Startup Delay (seconds) parameters. JMeter will wait for the Startup Delay (seconds) upon test launch before initiating the Threads of the Thread Group and running the test for the specified Duration (seconds).

C. An overview of the dataset:

- timeStamp: Indicates the timestamp of each sample, allowing us to track the progression of load over time. elapsed: The column labeled ”elapsed” in the JMeter data shows how long each sample took in total. A possible performance bottleneck may be indicated by an increase in response time as record size grows [5]. Response time and elapsed are frequently used simultaneously while discussing JMeter.
- label: It is the name we have given to each of our HTTP Request Samplers.
- responseCode: This is responseMessage in an encoded form.
- responseMessage: It contains values of the result of the HTTPRequest, like ’OK’, ’Moved Permanently’, ’responseMessage’, and ’Too Many Requests’.
- threadName: It is the particular thread’s name in the threadGroup.
- dataType: The”dataType” column in test results refers to the type of data being measured or recorded during the test. It specifies the nature of the data being collected for each request or sample. The dataType field categorizes the data into different types, such as text, binary, or other custom types, depending on what is being measured or analyzed in the test scenario. This information is valuable for understanding and interpreting the results of the performance test accurately [5].
- success: The success or failure of each sample is indicated in the ”success” column. The error rate is computed as the proportion of unsuccessful requests. A high mistake rate could be a sign that the system isn’t handling bigger data sets well.
- bytes and sentBytes: Consider the ”bytes” and ”sentBytes” columns to gauge the volume of data being sent. More data transfer will come from larger record sizes, therefore you’ll need to be sure the network and server can manage this increase [5].
- grpThreads: These are the number of users we are simulating.
- URL: These are the URLs we gave as input in the test plan that Jmeter used for the simulation.
- Latency: The time it takes to send a request and obtain a response is shown in the ”Latency” column of the results. A decrease in the speed of communication between the client and the server may be indicated by an increase in latency as record sizes increase [5].
- Connect: This is the amount of time needed to connect to the server. Larger record sizes may suggest problems with database connection handling if there is a noticeable increase in connection latency [5].

The columns responseCode, bytes, sentBytes, grpThreads, Latency, and Connect could be utilized well to predict future Load patterns.

D. Selecting the Machine Learning Model

The convolutional Neural Network Model was chosen to make a machine learning model that can predict future patterns in Load Testing Data. It was chosen because of the following reasons:

- Load testing data often exhibits spatial patterns, where certain combinations of features across time (such as response times, request sizes, etc.) can indicate specific load testing patterns or behaviors [6]. CNNs are designed to recognize spatial patterns in data through the use of convolutional layers, which are capable of detecting local patterns within the input data [12].
- In testing, data often represents a series of events happening one after another. This kind of data is called sequential data. Convolutional neural networks (CNNs) are a good tool for working with sequential data because they can find patterns and relationships within the sequence. They do this using special filters that analyze the data step-by-step, like reading a sentence one word at a time [13].
- CNN architectures consist of multiple layers that learn hierarchical representations of the input data [6]. Lower layers in the network learn low-level features, while higher layers learn increasingly abstract and complex features. This hierarchical feature learning capability allows CNNs to automatically extract relevant features from load testing data, potentially capturing both short-term and long-term patterns.
- CNNs leverage convolutional kernels to share parameters among several input data segments. Due to their ability to train from big datasets effectively and without requiring

an excessive amount of parameters, CNNs are a good choice for jobs involving high-dimensional input data, such as load assessment.
- Regularisation methods like dropout and batch normalization, which lessen the possibility of learning noise from the training set, are frequently used in CNN designs to prevent overfitting. Regularisation enhances the model's capacity to generalize, enabling it to more accurately forecast load testing trends from unobserved data in the future.
- All things considered, CNNs can be used to forecast future load testing patterns due to their efficient parameter sharing and regularisation capabilities, as well as their capacity to capture spatial and temporal patterns in sequential data.

E. Data Pre-processing

We removed some columns that did not contribute to the learning of the Convolutional Neural Network(known as CNN from here on) model's learning.

a). Converting dataset to numerical form:
We converted the object type columns to integer type, replaced the Nan values, and converted them to integer type by leveraging Python's NumPy. The use of Int64 allows for the presence of NaN values, making it suitable for columns where missing data is possible. This conversion ensures that the data is in the appropriate format for training the CNN model. We used the LabelEncoder from the sklearn.preprocessing module to encode categorical labels in our data frame. We encoded columns containing string values like label and URL into numerical values using Label encoding because the CNN model only takes numerical data as input.

b). Dataset Scaling:
MinMaxScaler from scikit-learn (sklearn.preprocessing) has been used to scale numerical features in our data frame. The numerical characteristics in the data frame are subjected to the fit_transform method of the MinMaxScaler. By scaling each feature to a predetermined range (by default, between 0 and 1), this method fits the scaler to the data and transforms the data depending on the fitted scaler. The data frame's associated columns hold the scaled numerical features once they have been saved. For some machine learning algorithms, especially those that are sensitive to the scale of the input characteristics, this procedure guarantees that numerical features are scaled to a common range.

c). Re-shaping the data into a 3D Tensor:
Because CNNs can record spatial hierarchies, they are frequently used for image data, but they may also be applied to other forms of data that have sequential or spatial structures.(12)Since we had non-image data, we reshaped our dataset to convert each row into a tensor. Reshaping the dataset into a 3D tensor allows you to represent each row of your non-image data as a tensor, which is suitable for processing by a CNN. By reshaping the data in this way, we can leverage the spatial hierarchies that CNNs are designed to capture, even though our original data may not have a spatial structure like images [14].

So, reshaping the dataset into a tensor format enables the application of CNNs to non-image data by treating the features as channels and potentially capturing relationships between features that are analogous to spatial relationships in images. This allows CNNs to potentially extract meaningful patterns and representations from the data. The data has been reshaped into a 3D tensor with dimensions using NumPy's expand_dims function. In the reshaped tensor, each row in the original data, which represented a sample, is now a 2D slice after reshaping [15]. Every feature is shown as a distinct channel in the tensor.

F. Implementing the Machine Learning Model

1. Importing Libraries:
We imported the necessary libraries for data manipulation, model construction, and visualization. NumPy is imported for numerical operations, scikit-learn for data preprocessing and splitting, tensorflow.keras for building and training neural networks, and Matplotlib for plotting.

2. Data Preparation:
We defined feature_columns and target_columns lists to represent feature and target variable names.

Feature_columns are elapsed, responseCode, bytes , sentBytes, grpThreads, Latency, Connect, label_encoded, threadName_encoded, dataType_encoded, success_encoded, and URL_encoded.

The model is supposed to learn from all the columns and predict all the columns.

Target_columns are elapsed, responseCode, bytes , sentBytes, grpThreads, Latency, Connect, label_encoded, threadName_encoded, dataType_encoded, success_encoded, and URL_encoded.

Then we checked for missing values in both features and target variables using .isnull().sum(). Missing values were filled with mean values using .fillna(). This method calculates the sum of missing values for each column in the data frame. It helps in identifying if there are any missing values in the dataset. Missing values are filled with mean values using the .fillna() method. This approach is a common strategy in data preprocessing, where missing values are replaced with the mean (or median) of the available data. It helps to ensure that the dataset remains complete and usable for subsequent analysis or modeling. By performing these steps, we ensured that the dataset was properly prepared for further analysis and modeling. It handles missing values by filling them with appropriate values, thereby ensuring that the dataset is complete and ready for use in training machine learning models.

3. Data Scaling:
Feature and target variables are scaled using MinMaxScaler. This scaling ensures that all features lie within the same range, which helps the neural network converge faster during training.

4. Data Splitting:
The dataset is split into training, validation, and test sets using train_test_split function. It splits the data into 80% training and

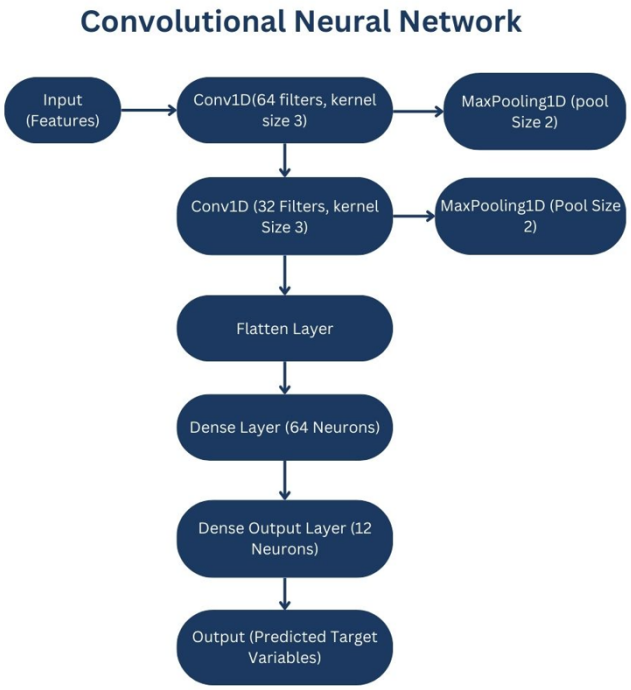

Fig. 4. Model Architecture

20% combined for validation and testing. The split is done in a stratified manner to maintain the distribution of classes in the target variables.

5. Reshaping Data for CNN:

Since the model being built is a Convolutional Neural Network (CNN), the input data needs to be reshaped into 3D arrays (samples, timesteps, features). This reshaping allows the CNN to interpret the sequential nature of the data.

6. Model Architecture:

The CNN model is defined using Sequential API from Keras. It consists of two convolutional layers followed by max-pooling layers for feature extraction and then flattened to be fed into fully connected layers. The output layer has as many neurons as the number of target variables.The functionalities of each layer are as follows:

- Input Layer: The input features are fed into the Convolutional Neural Network (CNN) model.
- Convolutional Layer 1: The first convolutional layer consists of 64 filters with a kernel size of 3. This layer performs convolution operations on the input features to extract various features.
- MaxPooling Layer 1: The output from the first convolutional layer is passed through a max-pooling layer with a pool size of 2. Max-pooling reduces the spatial dimensions of the feature maps, preserving the most important information.
- Convolutional Layer 2: The output from the first max-pooling layer is then passed through a second convolutional layer with 32 filters and a kernel size of 3. This layer further extracts higher-level features from the input.
- MaxPooling Layer 2: Similar to the first max-pooling layer, the output from the second convolutional layer is passed through another max-pooling layer with a pool size of 2.
- Flatten Layer: The output from the last max-pooling layer is flattened into a 1D vector. This prepares the data to be fed into the fully connected layers.
- Dense Layer (64 neurons): The flattened output is then passed through a fully connected dense layer with 64 neurons. This layer performs a linear transformation on the input data, followed by the application of a non-linear activation function (ReLU).
- Dense Output Layer (12 neurons): Finally, the output layer consists of 12 neurons, representing the number of target variables. This layer produces the predictions for the target variables.
- Output (Predicted Target Variables): The predicted target variables are the output of the model, which are compared with the actual target variables during training to calculate the loss and update the model parameters.

The CNN model architecture consists of multiple layers, including convolutional layers for feature extraction, max-pooling layers for dimensionality reduction, and fully connected layers for making predictions. This architecture allows the model to learn hierarchical representations of the input data and make accurate predictions for the target variables.

7. Compiling the Model:

The model is compiled with the Adam optimizer and mean squared error loss function. Adam optimizer is chosen for its adaptive learning rate capabilities, which help in efficient training.

8. Data Generator Function:

A custom data generator function is defined to yield batches of data during training. It creates a TensorFlow dataset from input features and target variables, shuffles it if it's for training, batches it, and repeats it indefinitely. This generator function is crucial for handling large datasets efficiently during training.The custom data generator function is designed to handle the efficient generation of batches of data during the training of a neural network model. It's particularly useful for handling large datasets that may not fit into memory all at once.

The function follows a specific workflow to generate batches of data:

a. Creating TensorFlow Dataset:

The function starts by creating a TensorFlow dataset using tf.data.Dataset.from_tensor_slices(). This method slices input features and target variables into individual elements along the first dimension, creating a dataset of tuples.

b. Shuffling :

If the generator is being used for training (validation_flag is False), the dataset is shuffled to ensure that the model does not learn from any inherent order in the data. Shuffling helps prevent the model from being biased towards the order of the data samples.

c. Batching:

Next, the dataset is batched using the .batch() method. Batching involves combining consecutive elements of the dataset into batches of a specified size. This step is crucial for efficient processing of data during training.

d. Repeating Indefinitely:

To ensure that the generator keeps providing data indefinitely during training, the dataset is set to repeat indefinitely using the .repeat() method. This allows the model to train on the data continuously without running out of samples.

e. Creating Iterator:

An iterator is created for the dataset using the iter() function. The iterator allows us to iterate over the dataset and extract batches of data during training. Finally, within a while loop that runs indefinitely, the function continually yields batches of features and target variables using the iterator's get_next() method. Each time the function is called, it returns the next batch of data.

This custom data generator function is used as an input to the fit() method of a Keras model during training. By passing this generator function, the model will receive batches of data continuously during training, allowing it to update its parameters iteratively.

9. Training the Model:

The model is trained using the fit method. It takes the training data generator, steps per epoch, number of epochs, validation data generator, and validation steps as input. During training, the model learned to minimize the defined loss function by adjusting its parameters using backpropagation.

10. Plotting Training History:

Finally, the training and validation loss over epochs was plotted using Matplotlib, which is shown in the results section. This visualization helps to understand the model's performance and check for overfitting or underfitting. By monitoring the loss curves, one can determine if the model is learning properly or if adjustments to the architecture or training procedure are necessary.

IV. RESULT AND DISCUSSION

This section delves into the performance evaluation of the employed Convolutional Neural Network (CNN) model for predicting future load testing patterns. The model was configured with a learning rate of 0.001, a common choice for optimizing gradient descent algorithms. The ReLU (Rectified Linear Unit) activation function was selected due to its efficiency and ability to mitigate the vanishing gradient problem. The training process encompassed 20 epochs, enabling CNN to iteratively extract and learn the underlying patterns resident within the JMeter load test data. This trained model subsequently possesses the capability to generalize its knowledge and predict future values of various JMeter load testing parameters based on the historical data it was trained on.

A. Data Collection for Load Testing

To assess the learning process and generalizability of the model, we analyzed the training and validation loss curves

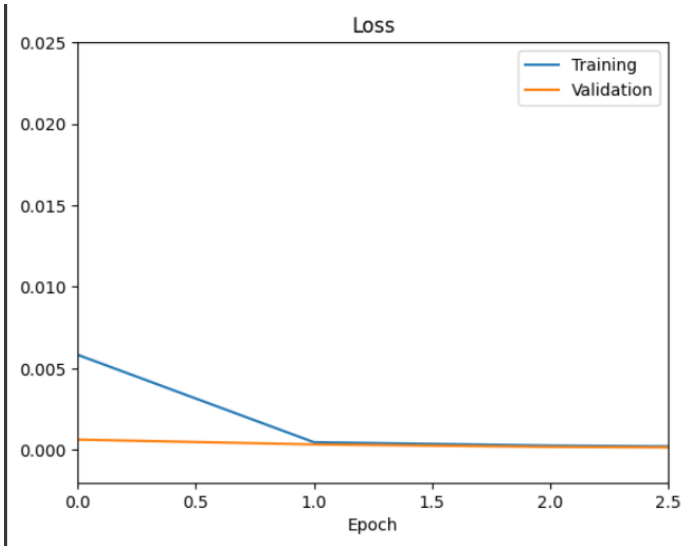

Fig. 5. Loss in Training and Validation

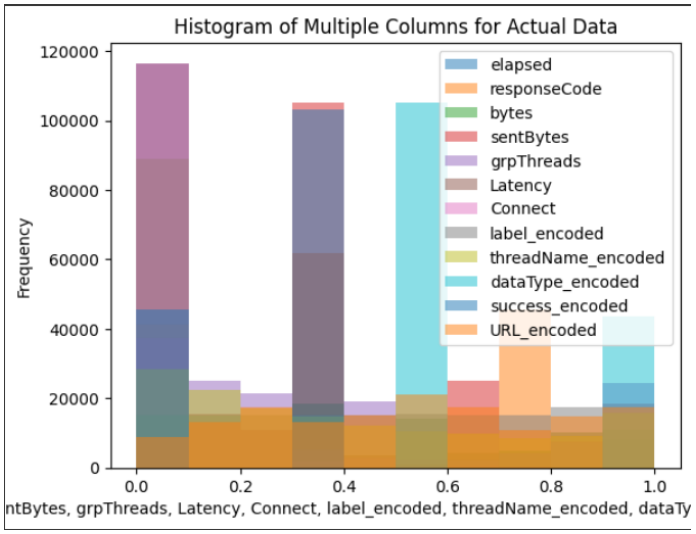

Fig. 6. Histogram of Actual Load Data

across the training epochs. The provided graph in 5 visualizes this behavior. The training loss curve demonstrates a consistent decrease as the number of epochs increases. This observation signifies the model's effectiveness in progressively learning the underlying patterns present within the JMeter load test data. In simpler terms, the model is adept at fitting the training data with a diminishing loss value as it iterates through training epochs.

However, to assess the model's generalizability beyond the training data, it's crucial to analyze the validation loss curve concurrently. Ideally, we would observe a similar decreasing trend in the validation loss alongside the training loss. This scenario suggests that the model is successfully capturing the general patterns within the data and avoiding overfitting to the specific training set. Conversely, if the validation loss stagnates or even increases while the training loss continues to decrease,

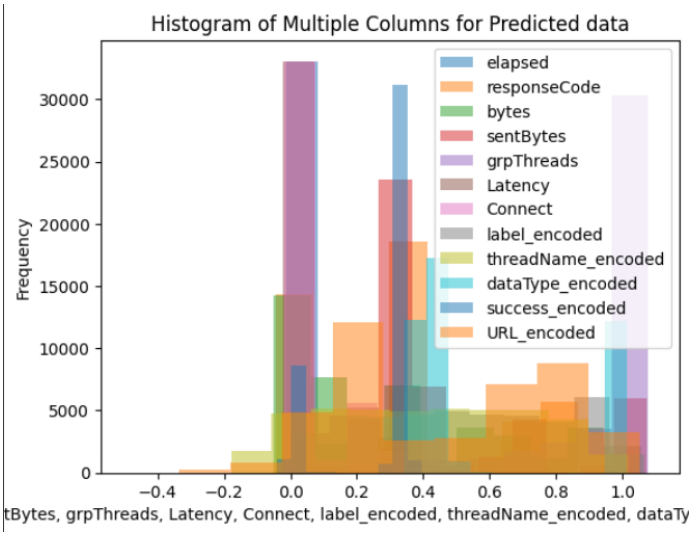

Fig. 7. Histogram of Predicted Load Data

it might indicate overfitting. Overfitting signifies the model's potential to have memorized specific features from the training data that might not generalize well to unseen data.

B. Visual Assessment of Model Generalizability Through Histograms

To gain further insights into the model's performance and potential biases, we employed histogram visualizations. Histograms effectively depict the distribution of data points within a specific numerical variable. In this analysis, we created two separate histograms. The first histogram(6) visualizes the distribution of features within the actual load test data (data). The second histogram(7)depicts the distribution of the same features within the predicted load test data. These features are represented by the columns By comparing these histograms, we can visually identify potential discrepancies between the actual and predicted data distributions.

Complementing the performance metrics discussed earlier, a visual comparison of the actual and predicted data histograms offers further insights into the model's generalizability. These histograms depict the distribution of various features extracted from the load test data, including elapsed time, data transfer volume (bytes sent/received), and encoded categorical variables (thread names, data types, and success). A significant observation is the similarity in distribution shapes for several key features between the actual and predicted data. This suggests the model effectively captures the underlying patterns within the data and preserves the essential characteristics during prediction. For instance, the model appears to accurately predict the range of elapsed times (response times) experienced in real-world load tests. Additionally, the histograms for features like data transfer volume ideally reflect the distribution patterns observed in the actual data, even after incorporating min-max scaling during pre-processing. This preservation of distribution characteristics highlights the model's ability to adapt to pre-processing steps commonly employed in real-world machine learning pipelines. While encoding categorical variables like response code, URL, and success is a common practice for model training, it can introduce slight deviations in the predicted distribution compared to the original data. A more detailed feature-wise comparison, which will be discussed in the next section, will delve deeper into these discrepancies to assess their potential impact on model performance. This combined analysis of distribution shapes and feature-wise comparisons will provide a comprehensive understanding of the model's generalizability and potential areas for further refinement.

C. Feature-Wise Comparison: Unveiling Insights Through Kernel Density Estimation

To delve deeper into the model's ability to capture the nuances of individual features, we turn our attention to a feature-wise comparison of actual and predicted load test data. Kernel density estimation (KDE) offers a powerful visualization tool to elucidate these comparisons. KDE constructs smooth, continuous probability density curves, allowing for visual contrasts of the underlying distributions of each feature.

Following the visual assessment of overall distribution through histograms (Section B), we conducted a feature-wise comparison using kernel density estimation (KDE) plots as shown in 8. Here, we present KDE plots for various features extracted from the load test data (elapsed time, latency, connect, bytes, response code, success). Each plot depicts two density curves: the actual data distribution (blue) and the model's predicted distribution (red). Visual inspection of these KDE plots reveals a high degree of overlap between the actual and predicted distributions for most features, suggesting the model effectively captures the underlying patterns within the data. This is particularly evident in features like elapsed time, data transfer volume, and sent bytes, where the curves closely resemble each other in terms of shape and peak locations. However, minor deviations between the curves are also observed in some plots, likely due to the inherent randomness associated with the number of threads utilized by JMeter during load tests. This randomness can introduce slight discrepancies in features influenced by thread count. Despite these minor variations, the overall alignment between the actual and predicted distributions across various features underscores the model's effectiveness in replicating the essential characteristics of the load test data.

V. CONCLUSION

Our CNN-based model transcends reactive load-testing by predicting future load patterns through analysis of historical data. This prescriptive approach empowers testers. By accurately forecasting load trends, the model allows proactive resource adjustments to prevent bottlenecks before they occur. Additionally, continuous monitoring of load data against predictions enables the identification of anomalies indicative of potential issues. This early detection facilitates prompt investigation and rectification, safeguarding user experience. Finally, the model optimizes resource allocation by predicting

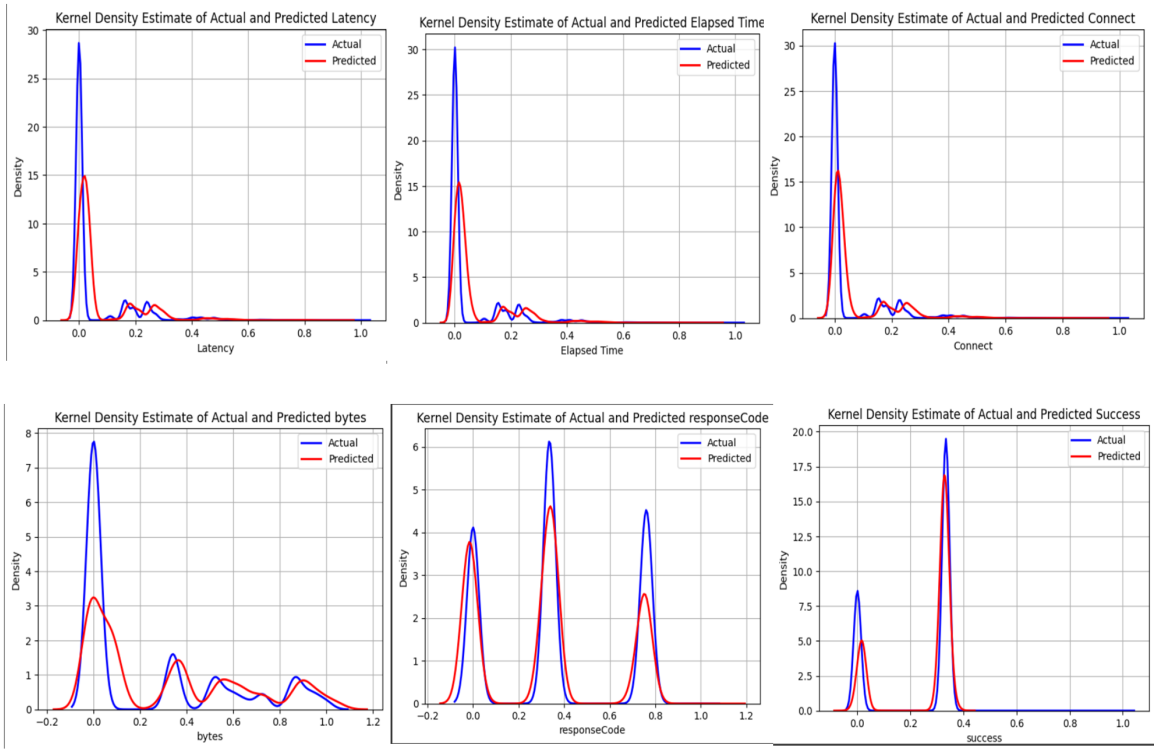

Fig. 8. Comparison of actual vs predicted values of load test parameters

future needs, ensuring cost-effective provisioning that aligns with anticipated load levels. Overall, the model transforms load testing from a reactive process to a proactive one, safeguarding system stability and performance through its predictive capabilities. Furthermore, the feature-wise analysis using KDE plots revealed a high degree of overlap between actual and predicted distributions, signifying the model's ability to capture the nuances of individual load test characteristics. While minor deviations were observed due to JMeter's thread randomization, the overall alignment underscores the model's effectiveness. This study demonstrates the potential of CNNs to significantly enhance load testing practices, enabling organizations to proactively manage system performance and scalability.

VI. FUTURE SCOPE

The current study establishes a foundation for incorporating Convolutional Neural Networks (CNNs) into load testing practices. The model's effectiveness in predicting future load patterns based on historical data has been demonstrably successful. However, one aspect that merits further exploration is the impact of seasonal variations on load patterns. The current data utilized for model training did not encompass a substantial time frame, potentially limiting the model's ability to capture these cyclical trends. Real-world system usage often exhibits seasonal fluctuations. Peak hours, holidays, and specific events can all trigger significant increases in load, while other periods may experience lower traffic volumes. By incorporating data spanning across seasons, the model could be enriched with a more comprehensive understanding of system behavior under varying load conditions. This enriched data would encompass historical patterns of both high load (e.g., peak hours, holidays) and low load, enabling the model to learn and predict these cyclical variations with greater accuracy. Building upon this foundation, future research can delve into the application of time series models like ARIMA (Autoregressive Integrated Moving Average) for load pattern prediction. Time series models excel at identifying and forecasting cyclical patterns within data. By incorporating seasonal data and leveraging the strengths of time series models, future studies can potentially achieve even more accurate predictions of future load trends. In conclusion, while the current CNN model demonstrates promising results, integrating seasonal data and exploring time series models presents an exciting opportunity to further enhance the accuracy of load pattern prediction. This advancement would provide load testers with an even greater level of foresight, empowering them to proactively ensure system stability and optimal performance throughout the year. This refined approach to load testing holds significant potential for organizations seeking to maintain a high level of user experience and system responsiveness in the face of ever-evolving load patterns

VII. ACKNOWLEDGMENT

We would like to express our sincere gratitude to Professor Nouh Alhindawi for his invaluable guidance and support throughout this research project. His insights and expertise were instrumental in shaping the direction of this work.

We are also grateful to the teaching assistants and graders who provided valuable feedback and helped us refine our understanding of the concepts involved.

Finally, we would like to thank Arizona State University for providing the resources and facilities that enabled us to conduct this research. Their commitment to fostering a stimulating academic environment is deeply appreciated.

REFERENCES

[1] Andreolini M, Casolari S. Load prediction models in web-based systems. Published online January 1, 2006. doi:https://doi.org/10.1145/1190095.1190129.

[2] Google. Google Search Console. Google.com. Published 2009. https://search.google.com/search-console/about

[3] What is Load Testing? How it works. OpenText. https://www.opentext.com/what-is/load-testing

[4] Apache JMeter User's Manual. Apache.org. Published 2019. https://jmeter.apache.org/usermanual/index.html

[5] Racheal Iperu. Interpreting JMeter Results for Performance testing. www.linkedin.com. Accessed April 7, 2024. https://www.linkedin.com/pulse/interpreting-jmeter-results-performance-testing-racheal-iperu-bnkcf/

[6] Kala S, Paul D, Jose BR, Mathew J, S Nalesh. Performance Analysis of Convolutional Neural Network Models. Published online November 1, 2019. doi:https://doi.org/10.1109/icacc48162.2019.8986201

[7] Nguyen D, Kim D, Lee J. Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs. Published online March 1, 2017. doi:https://doi.org/10.23919/date.2017.7927113

[8] Wibawa AP, Utama ABP, Elmunsyah H, Pujianto U, Dwiyanto FA, Hernandez L. Time-series analysis with smoothed Convolutional Neural Network. Journal of Big Data. 2022;9(1). doi:https://doi.org/10.1186/s40537-022-00599-y

[9] Patel N. Software Scalability Testing Guide. www.qable.io. Published April 3, 2024. Accessed April 7, 2024. https://www.qable.io/blog/software-scalability-testing

[10] Britto V. The Software Testing scenario. Medium. Published February 27, 2024. Accessed April 7, 2024. https://medium.com/@vitorbritto/the-software-testing-scenario-325ecb4f8a73

[11] Apache Software Foundation. Apache JMeterTM. Apache.org. Published 2019. https://jmeter.apache.org/

[12] Alzubaidi L, Zhang J, Humaidi AJ, et al. Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. Journal of Big Data. 2021;8(1). https://journalofbigdata.springeropen.com/articles/10.1186/s40537-021-00444-8

[13] Wikipedia Contributors. Convolutional neural network. Wikipedia. Published February 27, 2019. https://en.wikipedia.org/wiki/Convolutionalneuralnetwork.

[14] Shah A. Reshaping the Dataset For Neural Networks. Medium. Published December 11, 2022. https://medium.com/@jwbtmf/reshaping-the-dataset-for-neural-network-15ee7bcea25e

[15] Brownlee J. Manipulating Tensors in PyTorch. Machine Learning Mastery. Published April 8, 2023. Accessed March 6, 2024. https://machinelearningmastery.com/manipulating-tensors-in-pytorch/