Human Factors in Design, Engineering, and Computing, Vol. 159, 2024, 172–181
https://doi.org/10.54941/ahfe1005580

Comparison of AI Model Serving Efficiency: Response Time and Memory Usage Analysis

Ji-Yeon Kim[1], Seong-Hyeon Jo[1], Sang-Hyun Ha[2], Ki-Hwan Kim[2], Young-Jin Kang[2], and Seok Chan Jeong[1,2,3]

[1]Department of Artificial Intelligence, Dong-Eui University, Busan, 47340, South Korea
[2]AI Grand ICT Research Center, Dong-Eui University, Busan, 47340, South Korea
[3]Department of e-Business, Dong-Eui University, Busan, 47340, South Korea

ABSTRACT

NLP (Natural Language Processing) models are in increasing demand, making the research into effective serving methods crucial. Particularly, cost efficiency and rapid response times are key factors in the serving process of NLP models. This paper compares various methods for optimizing the serving of NLP models. Three serving methods were applied using REST API, TensorFlow Serving, and TensorFlow.js, and each method's response speed and memory usage were evaluated. This research is thought to provide foundational guidelines for enhancing the efficiency of serving NLP models, aiming to minimize potential issues in the serving process and improve user experience through such studies.

Keywords: Tensorflow serving, Tensorflow.js, Nodejs, LSTM, GRU

INTRODUCTION

Since 2019, large language models (LLMs) like GPT have gained significant attention in the field of natural language processing (NLP), with services utilizing these models experiencing rapid growth. Language model services simplify the process of querying and receiving responses through text, providing convenience and being applied across various applications, with some services reaching millions of daily users. Alongside this rapid growth, the importance of serving large language models quickly and reliably is also increasing.

Large language models tend to perform better as they increase in size, but they also require significant computational resources and memory. This has become a major factor in slowing response times and increasing costs during the serving process, especially in environments that require real-time responses.

Wang et al. (2024) found that inference within web browsers is considerably slower compared to native environments. In-browser inference demands significant memory, sometimes requiring up to 334.6 times more memory than the deep learning models themselves, increasing the rendering time of GUI components by an average of 67.2% and significantly degrading the overall user experience.

Jia et al. (2024) proposed an in-browser inference system called nnJIT for enhancing deep learning inference on edge devices through Just-in-Time (JIT) kernel optimization. nnJIT, which was co-designed with TensorFlow and web compilation, significantly reduced compilation and tuning costs, delivering performance 8.2 times faster than the baseline model. They also proposed an innovative Hybrid Bayesian Evolution Strategy (HBES) algorithm for adapting resource usage models to dynamic and heterogeneous environments in edge computing, using GRU (Gated Recurrent Unit) neural networks. The performance improvements with GRU-RNN models and HBES were notable, and nnJIT's maximum memory usage was on average 55.7% and 49.6% lower than TF.js and ORT-Web, respectively, with only a 2.2% increase in maximum memory.

In this study, as a first step to addressing these issues, we experimented with various serving methodologies using small-scale text classification models based on LSTM and GRU. The reasons for using small-scale models are as follows: Firstly, small models require relatively fewer resources compared to large-scale models, allowing for the quick and efficient evaluation of different serving methods. Secondly, results from experiments with smaller models can provide foundational data that can be used to identify and optimize potential problems when scaling up to more complex models, as well as offering direction for such expansions.

Therefore, this research aims to provide a foundational guide to improve the efficiency of serving natural language processing models by comparing the serving performance of small models using REST API (Bansal and Ouda, 2022), TensorFlow Serving (Olston et al., 2017), and TensorFlow.js (Smilkov et al., 2019), and evaluating the advantages and disadvantages of each method.

Web-Based AI Model Serving Overview

Models like ChatGPT are accessed via web browsers or dedicated applications on the internet, which optimizes artificial intelligence (AI) model services to be web-based rather than requiring the model to run directly on personal computers or mobile phones. This approach leverages server support, eliminating reliance on low-performance endpoint devices. AI model serving refers to the process of deploying trained AI models for real-world use, allowing them to function online for real-time predictions or offline for batch processing tasks (Kwon et al., 2023).

There are three main types of AI model serving: online serving (real-time serving), batch serving, and the microservices architecture. Online serving provides immediate prediction results in response to user requests. This system is commonly implemented through web servers or Application Programming Interfaces (APIs), handling various user requests and returning appropriate responses. Online serving processes HTTP requests from web browsers or other clients, provides responses such as HTML documents or necessary data, and performs data preprocessing and predictions using the model. Additionally, it serves as an interface between the server and client, exposing necessary functionalities for programs or applications. Common methods of implementing online serving include building APIs with web

frameworks such as Python's FastAPI, utilizing cloud services like Amazon Web Services (AWS), or employing serving libraries. Key considerations for online serving include real-time performance (minimizing latency for quick response times), stability, scalability, and maintainability. Online serving is exemplified by large language models (LLMs) like ChatGPT.

Batch serving processes data in bulk at specific intervals rather than handling individual requests in real-time. It is ideal for tasks where immediate results are not required. For example, data processing commands can be executed at 10 AM and 11 AM daily. Data is grouped into batches (such as every 30 minutes) and processed together, often using workflow automation tools such as Apache Airflow or Cron Job. The main advantages of batch serving include ease of implementation and high data throughput, as large datasets can be processed simultaneously without concerns about latency. However, the disadvantages include the inability to provide real-time processing and issues such as the "cold start" problem, which means newly introduced content cannot be recommended immediately.

The microservices architecture is an approach in which independent services communicate through lightweight APIs. This architecture enables different components of an application to be developed and managed separately, allowing development and operations teams to collaborate efficiently without interference. As a result, more developers can work on the same application simultaneously, reducing overall development time.

Understanding NLP Overview

NLP (Natural Language Processing) is a field of technology that enables computers to interpret and understand human language. This field involves analyzing and processing text and speech data for use in various applications such as machine translation, sentiment analysis, dialogue systems, and information extraction. NLP is situated at the intersection of computer science, artificial intelligence, and linguistics, and through these technologies, it allows computers to 'understand' human language and respond in natural language or extract information in a useful way. LSTM (Long Short-Term Memory) (Hochreiter and Schmidhuber, 1997) and GRU (Gated Recurrent Units) (Cho et al., 2014) are variations of recurrent neural networks (RNNs) commonly used in NLP, specialized in learning the features of sequence data, particularly the temporal sequence and context of language data.

LSTM networks are a type of RNN specifically designed to overcome the limitations of traditional RNNs, such as the vanishing gradient problem. This problem occurs when gradients, calculated during the training process through backpropagation, become exceedingly small, effectively preventing the model from learning long-range dependencies in the data. LSTM networks address this issue with a series of gates.

These gates allow LSTMs to selectively remember or forget patterns, which is especially useful in tasks where understanding the context spread across significant parts of the text is crucial, such as sentiment analysis or topic categorization in tweets regarding disasters.

GRU is a simpler variant of LSTM, combines the forget and input gates into a single "update gate" and merges the cell state and hidden state, resulting

in a model that is easier to compute and often faster to train than its LSTM counterpart. Despite its simplicity, GRUs have shown to perform on par with LSTMs on various sequence modeling tasks. The architecture of a GRU is designed to capture dependencies of different ranges without relying on a memory unit separate from the hidden state. It effectively captures the short-term dependencies with fewer parameters and less computational overhead.

Suggest Experiments

The model used in this study is a text classification model based on LSTM, a type of recurrent neural network. The data utilized is 'Disaster Tweets', and the model processes text data composed of up to 15 words, performing binary classification in the final layer to predict whether the text belongs to a specific class. The total number of parameters is 3,988,421, of which 1,329,473 are trainable. Assuming each parameter requires 4 bytes (32 bits), the total memory requirement of the model is approximately 15.21MB. To assess the response speed and memory usage of such an NLP model, serving methods such as REST API, TensorFlow Serving, and TensorFlow.js were applied. The REST API is an architecture that lies between the client and server, facilitating communication between them. It is commonly used in web service development and exchanges data based on the HTTP protocol. REST has a resource-centric structure and allows the server and client to operate independently, enhancing scalability and making it suitable for various services. TensorFlow Serving is a tool designed to smoothly deploy and operate models, capable of handling multiple client requests simultaneously due to its high performance. It also utilizes GPUs to enhance inference speed, showing strengths in real-time predictions. These features provide high performance, scalability, and real-time prediction capabilities, efficiently serving models and reducing the complexity and time involved in model deployment and management. TensorFlow.js consists of Core API and Layers API and is a library that enables the execution of TensorFlow models in a JavaScript environment. It allows for running models locally without server communication and performing inference on the client side.

Figure 1 shows the process where a web client sends a RestAPI request to a web server to retrieve the result. The server is based on FastAPI and uses Tensorflow (Python) to directly load and utilize an AI model (.h5) in the backend environment. The client sends input values to the server, enabling the server to produce output values.

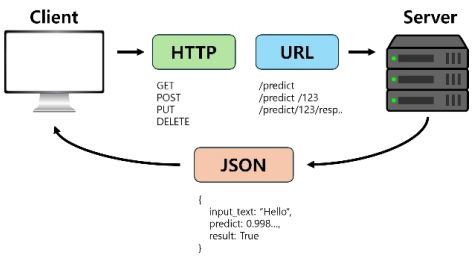

Figure 1: Interaction between web client and server using RestAPI with FastAPI and TensorFlow.

Figure 2 shows the AI model was saved in the SavedModel format, and the TensorFlow Serving image was downloaded in Docker. The prepared model was served via Docker. For the deployed model, requests can be sent in API format to receive the model's output values. Input data is sent through the client, and the server transmits the input values to the TensorFlow Serving API address to obtain the results, which are then sent back to the client.

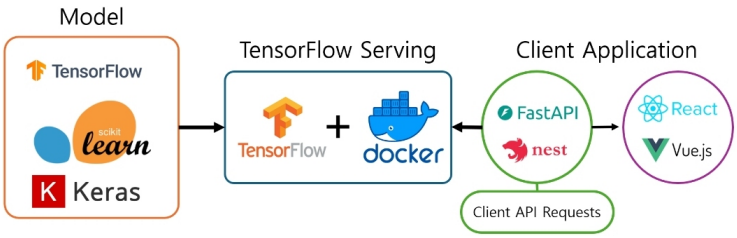

Figure 2: Workflow of AI model deployment and interaction using TensorFlow serving in docker.

Figure 3 shows TensorFlow.js utilizes WebGL for GPU acceleration, which significantly enhances the speed of large matrix operations. In most cases, WebGL is automatically employed in environments where GPUs are available, optimizing computational performance.

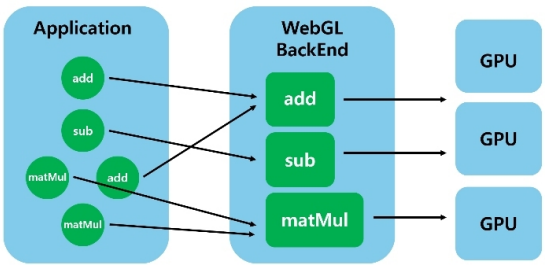

Figure 3: Optimizing computational performance with GPU acceleration in TensorFlow.js using WebGL.

Figure 4 show the process of converting and utilizing a TensorFlow model into TensorFlow.js involves several key steps. Firstly, the tf.data API is used to set up a data input pipeline designed for efficiently handling large-scale data. This pipeline may perform feature engineering tasks using tf.feature_column to enhance model performance. Feature engineering involves transforming or creating features to make the data more understandable for the model, such as categorizing, scaling, or converting categorical variables into numerical values, thereby optimizing the data for better model performance. In the model construction and training phase, tf.keras or tf.estimator APIs are used to build and train the model. If transfer learning is needed, pre-trained modules from TensorFlow Hub can be integrated. Once training is complete, the Distribution Strategy API enables the model to be trained in a distributed manner across various hardware environments (CPU, GPU, TPU), facilitating

large-scale data processing and optimizing training speed. The trained model is then exported in the standard SavedModel format, which can be utilized across different deployment environments and converted for use in TensorFlow.js. The conversion is done using the tensorflowjs_converter tool, which generates a model.json file and several.bin files. The model.json file contains the structure and metadata of the model, while the.bin files store the split weights data the model has learned. These two file sets serve to load and run the model in web browsers and Node.js environments, and the converted TensorFlow.js model can be deployed and utilized in web browser or Node.js server environments. Table 1 presents the list of libraries used in the experimental environment of this paper, along with their version information. The key libraries include Express.js and TensorFlow.js for NodeJS, and FastAPI, Keras, Numpy, Pandas, and scikit-learn for Python.

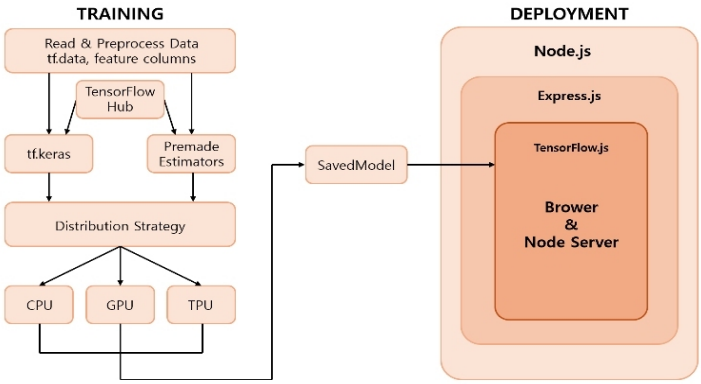

Figure 4: Interaction between web client and server using RestAPI with FastAPI and TensorFlow.

Table 1. Library information for experimental environment setup.

Librarys
NodeJS 16.20.2
Express.js: 4.19.2
csv-parse: 4.19.2
@tensorflow/tfjs-node: 4.21.0
Python 3.19.9
FastAPI: 0.113.0
uvicorn: 0.30.6
joblib: 1.4.2
keras: 3.5.0
pydantic: 2.9.0
numpy: 1.26.4
pandas: 2.2.2
tensorflow: 2.17.0
scikit-learn: 1.5.1

EXPERIMENTS RESULTS

The Table 2 and Table 3 represent the speed of processing and memory usage by inputting data into the LSTM model ten times and applying REST API (Fast API), TensorFlow Serving, and TensorFlow.js techniques. It can be seen that the REST API takes about 0.22 seconds for the first request but maintains a consistent speed of under 0.05 seconds from the second request onwards. The slower processing speed for the first request appears to be due to the need for model loading, and the subsequent requests show improved speed as the model resides in memory. TensorFlow Serving consistently maintains the fastest processing speed of less than 0.005 seconds. Compared to the REST API (Fast API), it maintains a fast speed from the first request, likely because TensorFlow Serving preloads the model into memory. TensorFlow.js has a processing speed similar to TensorFlow Serving on the first request but shows around 0.025 seconds thereafter, and from the seventh request, it shows about 0.019 seconds, which is slower than TensorFlow Serving but faster than REST API (Fast API). This performance is due to the method of TensorFlow.js, which allows direct model execution in the browser, eliminating communication delays as the model runs directly on the client side.

The Table 2 and Table 3 display data on memory usage across ten inputs into the LSTM model, comparing three different model serving techniques. The REST API (Fast API) had the highest memory usage at approximately 450MB, followed by TensorFlow Serving and TensorFlow.js, in that order. TensorFlow Serving and TensorFlow.js showed similar patterns of memory usage, but from the fourth request onwards, TensorFlow.js exhibited lower memory consumption. This is likely because TensorFlow.js runs the model on the client side, thus utilizing minimal server resources.

Table 2. LSTM model processing speed (sec).

Iteration	1	2	3	4	5	6	7	8	9	10
RestAPI (FastAPI)	0.221	0.046	0.046	0.046	0.046	0.048	0.045	0.046	0.046	0.047
TensorFlow Serving	0.044	0.005	0.006	0.005	0.005	0.004	0.004	0.005	0.005	0.005
TensorFlow.js	0.044	0.024	0.025	0.023	0.026	0.024	0.019	0.03	0.02	0.017

Table 3. LSTM model memory usage (Mb).

Iteration	1	2	3	4	5	6	7	8	9	10
RestAPI (FastAPI)	500	450	451	451	451	451	452	452	452	452
TensorFlow Serving	127	127	127	127	114	113	113	113	113	113
TensorFlow.js	129	130	134	99	99	99	98	99	99	99

Additionally, we tested a GRU-based text classification model, which is also an NLP model, using the same three serving methods to verify if similar results could be obtained as with other models. This model was trained using the same data as the LSTM-based model and has a total of 3,951,941

parameters, with 1,317,313 being trainable. The total memory requirement for the model is 15.08MB, indicating that it has similar specifications to the LSTM-based model.

According to the Table 4 and Table 5, which analyze processing speeds, using the REST API (Fast API) with the GRU model resulted in about 0.25 seconds for the first request, similar to the LSTM model, and subsequently maintained a speed of around 0.05 seconds. TensorFlow Serving shows a very fast response, maintaining a consistent speed of less than 0.0045 seconds from the second request onwards. TensorFlow.js takes about 0.03 seconds for the first request and stabilizes at around 0.02 seconds for subsequent requests.

As for memory usage, as shown in the Table 4 and Table 5, serving with the REST API (Fast API) results in the highest memory usage. Following this, similar to when serving the LSTM model, TensorFlow Serving and TensorFlow.js show lower memory consumption, in that order.

When analyzing the processing speed and memory usage for serving methods using LSTM and GRU-based models, both models exhibited similar patterns in their results. Both the LSTM and GRU models demonstrated the fastest processing speeds and the most efficient usage when the TensorFlow Serving method was applied. When using TensorFlow.js, the models maintained relatively fast speeds with the least memory usage. Lastly, the REST API (Fast API) method recorded comparatively higher values in both processing speed and memory usage, particularly showing a tendency for reduced performance on the first request.

Table 4. GRU model processing speed (sec).

Iteration	1	2	3	4	5	6	7	8	9	10
RestAPI (FastAPI)	0.229	0.0454	0.0452	0.0602	0.0456	0.0461	0.0466	0.0443	0.0452	0.0449
TensorFlow Serving	0.0239	0.0045	0.0048	0.0045	0.0049	0.0049	0.0042	0.0045	0.0048	0.0046
TensorFlow.js	0.0505	0.0285	0.028	0.029	0.0257	0.0249	0.0243	0.0202	0.0195	0.0187

Table 5. GRU model memory usage (Mb).

Iteration	1	2	3	4	5	6	7	8	9	10
RestAPI (FastAPI)	451.37	452.37	453.14	453.67	454.12	454.27	454.15	454.24	454.5	454.69
TensorFlow Serving	126.9	127.1	127.24	127.35	112.85	112.81	112.95	113.06	113.22	113.34
TensorFlow.js	128.56	129.93	132.83	99.61	99.75	99.85	100	98.47	98.99	99.89

CONCLUSION

In this paper, we analyzed the processing speed and memory usage for serving NLP models using REST API, TensorFlow Serving, and TensorFlow.js.

Experiments conducted with an LSTM-based model showed that TensorFlow Serving offered the best performance in terms of processing speed and memory usage. When applying TensorFlow.js, it processed at a relatively fast speed and exhibited the lowest memory usage. The REST API had comparatively higher processing speeds and memory usage than the other two methods. This pattern was also observed with GRU-based models, which are similarly used in NLP applications like LSTM. The results suggest that TensorFlow Serving is most suitable for large-scale real-time serving environments. Moreover, the minimal memory usage demonstrated by TensorFlow.js could be particularly beneficial for lightweight web applications or mobile environments.

Future research will involve conducting additional experiments across various server and network environments. We aim to analyze performance changes through scenario-based experiments that reflect real-world situations such as increases in concurrent user numbers and varying complexities of prediction requests. Beyond the currently used performance metrics of speed and memory usage, we also plan to evaluate additional factors such as CPU/GPU utilization and network bandwidth to further expand the performance metrics. Through this expansion, we intend to provide a more comprehensive evaluation of system performance, thereby enhancing the accuracy of system efficiency and stability analysis and offering better directions for model serving.

ACKNOWLEDGMENT

This work was supported by Innovative Human Resource Development for Local Intellectualization program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government (MSIT) (IITP-2024-RS-2020-II201791).

REFERENCES

Bansal, P., & Ouda, A. (2022, July). Study on integration of FastAPI and machine learning for continuous authentication of behavioral biometrics. In 2022 International Symposium on Networks, Computers and Communications (ISNCC) (pp. 1–6). IEEE.

Cho, K., Van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2020). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv 2014.arXiv preprint arXiv:1406.1078.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-term Memory. Neural Computation

Jia, F., Jiang, S., Cao, T., Cui, W., Xia, T., Cao, X.,... & Yang, M. (2024). Empowering In-Browser Deep Learning Inference on Edge Devices with Just-in-Time Kernel Optimizations.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H.,... & Stoica, I. (2023, October). Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th Symposium on Operating Systems Principles (pp. 611–626).

Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F.,... & Soyke, J. (2017). Tensorflow-serving: Flexible, high-performance ml serving.arXiv preprint arXiv:1712.06139.

Smilkov, D., Thorat, N., Assogba, Y., Nicholson, C., Kreeger, N., Yu, P.,... & Wattenberg, M. M. (2019). Tensorflow. js: Machine learning for the web and beyond. Proceedings of Machine Learning and Systems, 1, 309–321.

Wang, Q., Jiang, S., Chen, Z., Cao, X., Li, Y., Li, A.,... & Liu, X. (2024). Anatomizing Deep Learning Inference in Web Browsers. ACM Transactions on Software Engineering and Methodology.