# Predicting Request Service Times

**Isaac Wolverton** [1]

## Abstract

Interactive web services require meeting strict latency requirements. Some examples of these services are: web search, financial trading, video games, and social networks. The compute requirements behind these services are growing at an exponential rate. Therefore the more effectively these resources can be utilized the better. One particularly important area optimized for this request times, specifically meeting strict tail latency requirements. Until recently optimizations were mainly based on heuristics but had a limited insight into the nature of the request and the state of the compute resources. Recently methods involving machine learning have demonstrated an unrivaled ability to generalize over unknown data and make accurate predictions. The application of machine learning to the prediction of request service times would have a broad positive impact for the optimization of tail latency. I approach this by using an image processing server and recording data about test workloads I generated. A machine learning model is then fitted to this data to demonstrate the ability of the model to predict request service times. After the model has been fitted and its ability to make predictions has been demonstrated, the performance of the model is investigated. Several ways of reducing the models runtime are investigated in order to make it more practical in a real world environment, namely running as a part of the application and dynamically informing important scheduling decisions.

## 1. Introduction

Compute requirements are growing at an exponential rate, and optimizing these systems for specific properties often requires dealing with complex high-dimensional problems. New methods of optimization are moving away from heuristics and towards machine learning which can vastly outperform these traditional methods. Machine learning has been applied to many areas including: scheduling (Mirhoseini et al., 2017), data structure design (Kraska et al., 2017), memory management (Maas et al., 2020), compilers (Cummins et al., 2017), and more.

Similarly there has been work into optimizing request service times for tail latency. For instance Few-to-Many (Haque et al., 2015) dynamically increased the parallelism of long running requests and Sparrow (Ousterhout et al., 2013) used a distributed scheduler to assign tasks to the least busy machines. However there is significant room for improvement when optimizing for tail latency. One of the limitations of both Few-to-Many and Sparrow was that they are purely reactionary. Few-to-Many waits to see if the request runs long before assigning it additional parallelism. It would be far more beneficial if Few-to-Many could assign requests additional parallelism to long running requests as soon as they arrived. Similarly, Sparrow takes a batch sample of the worker machines' queues and uses their length to make scheduling decisions. Unfortunately queue length is not always a good indicator of workload and thus the authors had to use mitigations like late binding and proactive cancellation. However if Sparrow knew the amount of work associated with each request when it arrived, it could more accurately assess the enqueued workload of each worker and make more intelligent scheduling decisions such as queuing the longest requests first to reduce the overall latency of an entire task. This would also eliminate the need for overhead like late binding and proactive cancellation.

In both the Few-to-Many and the Sparrow case, large improvements could be made if they knew the request service time when it arrived. Unfortunately this kind of information about each request is unrealistic in real world systems where the space of possible requests can be practically infinite (consider an image processing server that receives a user provided image and performs a series of tens or hundreds of transforms). Fortunately machine learning can help address this. By training a machine learning model on data sampled from an application about requests and their duration, one can use the model to predict request service times as they arrive. Allowing for more efficient scheduling of resources and an overall reduction in tail latency.

[1]Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts. Correspondence to: Isaac Wolverton <isaacw@mit.edu>.

## 2. Design

The first question in any work that involves supervised machine learning is what the dataset will be. Since the problem formulation is to take application requests and predict service times, the dataset will have to include information about the requests and their corresponding duration at a minimum. Additional metadata could also be useful as features to help the model make accurate predictions. The question then became where to get this dataset. I could either find a request trace with the information required of a real world workload or generate one from a suitable application. After searching for quite some time no request traces that met these requirements presented themselves. Therefore I decided to generate the data instead.

This approach has a number of advantages and one significant disadvantage. The disadvantage is that the data is not from a real workload, and therefore may not be representative of a real data center situation. However, by generating the data myself I can gather as much metadata as I want that would not normally be included in the request races, and I can also vary the data to see its effect on the model's prediction accuracy.

The next question was which application to use to generate the data. I first investigated Redis As it is a very common industry standard key value store. Unfortunately Redis does not suit the task well as the request service times are mostly uniform and all quite small. Instead I decided to take inspiration from the Llama paper and use an image processing server. This kind of application has several distinct advantages, the service times vary widely, the metadata associated with each request is large, and it is easy to simulate a wide variety of workloads. After some research I ended up using imageflow `https://github.com/imazen/imageflow` an open source image processing server.

Imageflow supports a wide variety of requests. Some of the most common are transforms, filters, and encodings. A single request can contain any sequence of these operations. Transforms are comprised of operations like: rotating, cropping, scaling, structuring, flipping, etc. Filters are comprised of operations like: sharpening, downsampling, upsampling, converting to grayscale, inverting colors, and adjusting brightness or contrast or saturation. Imageflow Is able to encode into many different file types, with variable amounts of either lossless or lossy compression.

After choosing a variety of arbitrary requests (where a request is comprised of one or more transforms, filters, or encodings) I fed these requests to the image server and logged the contents of each request as well as the corresponding service time. This formed the foundation of my dataset. The data was then cleaned in a similar fashion to Llama. Strings were split on delimiters and each token was embedded. However, unlike Llama, I added an additional embedding vector for each token that represented the position of the token in the string in an approach similar to BERT (Devlin et al., 2018). This encoding of the position allows for a reduction in depth of the neural network as the hidden state does not have to be passed along for each token. Since this was demonstrated to perform well in BERT and it reduces the runtime of the model I added this optimization preemptively.

Since the evaluation of the model is not complete, I do not have a final structure for it yet. However it currently consists of two fully connected hidden layers and an output layer with softmax for classification. The reason for this structure is because of the desired production. The model can be thought of as a function approximator which gets trained on input output pairs $(x, y)$. Then for any unknown new $x$, it makes a prediction about its corresponding $y$. In this case the input $x$ is the tokenized and embedded string from the request. Since it is not necessary to have an exact answer about how long each request will take I opted to follow Llama's design and formulate the problem as classification rather than regression. So $y$ in the classification case is simply a label corresponding to the class. I am still experimenting with what the class sizes should be. Ultimately then the input layer is as large as the tokenized string, there are two hidden layers, and the output layer is equal in size to the number of classes.

When it comes time to evaluate I plan on performing standard k-fold cross-validation. K-fold cross-validation is the process where the available data set is split into k groups, and then the machine learning model is repeatedly trained on all but one of the groups and is evaluated on the group that was left out. This technique helps establish confidence that the model will generalize well to an independent unseen data set.

There are a couple of limitations to my project currently. So far I have only worked on training a model for single application, instead of a bevy of applications. However I think this is a reasonable first demonstration of the ability of machine learning models to predict request service times, as the model still has to generalize significantly within the scope of what Imageflow can do. Additionally, many of the web's most latency sensitive applications run on hardware that is specifically dedicated to them, so it does not seem like so much of a stretch to collect data about these particular applications and train models for them. The usefulness of predicting request service times could certainly outweigh this inconvenience. And further work could be done to make models that generalize across many applications. One other limitation stands out as a possible hindrance to real world applicability. All of the models training and learning is

done offline, on data collected/sampled from the application. As presented in the Llama paper is not unreasonable to assume that application workload changes over time and thus request service times may also change. This would involve retraining a new model with the newly sample data. Further work in this area could address this limitation by modifying the model to learn online as it experienced the data coming in. This would help the model adapt to shifting workloads over time and eliminate the need for retraining offline.

## 3. Generating The Data

Imageflow comes as a command line tool and webserver. Both variants accept what is called a query string. A query string is a sequence of commands that are concatenated by the ampersand symbol. Imageflow has an internal order operations so it does not matter which order these commands are written in. They will always follow the order: trim whitespace, srotate, sflip, crop, scale, filter, pad, rotate, flip. One example command line execution is:

```
imageflow_tool v1/querystring
--in source.jpg --out thumbnail.jpg
--command "width=50&height=50&
mode=crop&format=jpg"
```

The web server equivalent is:

```
my/url/source.jpg?width=50&
height=50&mode=crop&format=jpg
```

Both execute the exact same sequence of commands in the same order.


*Figure 1.* Jpeg compression quality 0 to 100 (in increments of 10) with runtime in seconds below in black and turbo runtime in red.

Part of the reason to choose Imageflow was the wide variety in runtimes between operations (as this could help better embody a real world application). We demonstrate this in two ways, with the jpec encoder and by timing various operations. It takes about 0.2 seconds for a simple image resize compared to 1-3 seconds to apply a sepia filter which is still much less than the 2 minutes to encode a large png as a high quality jpeg (no turbo). Although this is a source of variety between operations there is also variety inside of operations. As shown in Figure 1, the jpeg encoder has two main parameters, a quality integer and a turbo boolean. Encoding time is positively correlated with quality but in the case of turbo the encoding time is relatively consistent. This kind of complexity more closely resembles real word applications and would require the agent to learn that the runtime of:

```
jpeg.quality=100 >
jpeg.quality=50 >
jpeg.quality=0 >
jpeg.quality=100&jpeg.turbo=true
```

We are confident that the variety inside of commands and between them helps Imageflow better represent a real world use.

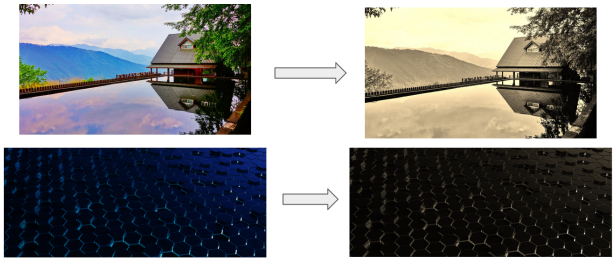

*Figure 2.* The sepia filter applied to two images. On top is waterhouse.jpg a 821.0 kB file with a resolution of 2048x1153. On the bottom is tiles.png a 7.8 MB file with a resolution of 3440x1440.

One area of investigation that has to be explored before generating the data is the consistency of the runtime. If the runtime is not consistent, prediction is essentially meaningless. We investigate this by performing the sepia filter 200 times on two images, see Figure 2. The results and corresponding histograms are detailed in Figure 3 and 4. Notice that the runtimes are relatively consistent and the deviations appear to follow a normal distribution. This is likely due to processor frequency, background tasks, etc.

| | House - sepia |
|---|---|
| **count** | 200 |
| **mean** | 0.72453 |
| **std** | 0.038446 |
| **min** | 0.534895 |
| **25%** | 0.716927 |
| **50%** | 0.732435 |
| **75%** | 0.744033 |
| **max** | 0.852589 |

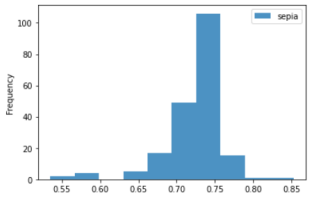


*Figure 3.* The results of the sepia filter applied to waterhouse.jpg.

The data was generated by first acquiring 50 random input images whose file type and size are varied. Then I wrote an expression tree of sorts that can generate random query

| | Tiles - sepia |
|---|---|
| count | 200 |
| mean | 3.129406 |
| std | 0.180405 |
| min | 2.625766 |
| 25% | 3.00936 |
| 50% | 3.119181 |
| 75% | 3.248654 |
| max | 3.59193 |

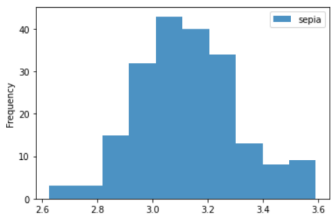


*Figure 4.* The results of the sepia filter applied to tiles.png.

strings with random parameters for each individual command. I tried to incorporate as many Imageflow features as possible to achieve the variety spoken about in previous paragraphs. Ultimately, I generated 10,000 random query strings with lengths from 3 to 5 and paired with random input files. I then ran image flow on each of these query strings and recorded the runtime in microseconds. An example of the data that was generated is in Figure 5, and the complete summary is in Figure 6.

| | query | time |
|---|---|---|
| 0 | imageflow_tool v1/querystring --in in/37.webp --out out/water9999.jpg --command "h=1000&h=600&s.saturation=0.8&s.contrast=0.6" | 0.062320 |
| 1 | imageflow_tool v1/querystring --in in/17.jpeg --out out/water9998.jpg --command "s.grayscale=sepia&s.grayscale=sepia&format=webp&webp.lossless=true&h=200" | 0.044926 |
| 2 | imageflow_tool v1/querystring --in in/8.jpeg --out out/water9997.jpg --command "flip=y&s.grayscale=sepia&s.grayscale=sepia" | 0.023656 |
| 3 | imageflow_tool v1/querystring --in in/waterhouse.jpg --out out/water9995.jpg --command "format=jpeg&jpeg.turbo=true&format=jpeg&jpeg.turbo=true&jpeg.quality=50&format=webp&webp.quality=100&format=webp&webp.lossless=true&s.brightness=-0.7" | 0.118022 |
| 4 | imageflow_tool v1/querystring --in in/7.webp --out out/water9993.jpg --command "rotate=90&srotate=90&rotate=270" | 0.040955 |
| 5 | imageflow_tool v1/querystring --in in/8.jpeg --out out/water9996.jpg --command "format=jpeg&jpeg.turbo=true&jpeg.quality=0&h=600&s.alpha=0.6&flip=x&s.alpha=0.7" | 0.280188 |
| 6 | imageflow_tool v1/querystring --in in/31.jpeg --out out/water9992.jpg --command "w=800&format=webp&webp.quality=30&s.grayscale=sepia&format=webp&webp.quality=50&format=jpeg&jpeg.quality=20" | 0.031077 |
| 7 | imageflow_tool v1/querystring --in in/32.jpg --out out/water9991.jpg --command "format=png&f.sharpenwhen=always&f.sharpen=20&s.grayscale=true&s.contrast=1.0" | 0.095999 |
| 8 | imageflow_tool v1/querystring --in in/tiles.jpg --out out/water9994.jpg --command "s.brightness=-0.0&f.sharpenwhen=always&f.sharpen=40&h=600&s.grayscale=sepia" | 0.307064 |
| 9 | imageflow_tool v1/querystring --in in/16.webp --out out/water9989.jpg --command "s.grayscale=sepia&w=600&s.grayscale=sepia&rotate=90" | 0.038704 |
| 10 | imageflow_tool v1/querystring --in in/31.jpeg --out out/water9990.jpg --command "format=png&png.quality=90&s.saturation=0.8&format=webp&webp.lossless=true&flip=xy" | 0.191514 |

*Figure 5.* Twenty example query strings and their associated runtime in microseconds.

### 3.1. Processing The Data

The command section of each query string is taken out and the name of the input file is appended to the beginning of it. This serves as the basis of what the model will eventually be fed. This superstring is then split on the ampersand delimiter, and each of these tokens that is produced is eventually considered part of the vocabulary. Ultimately, out of the 10000 query strings the total vocabulary length came to 197. That is the number of unique tokens. Each token is then given a single integer number, that serves as an index into the first layer, the embedding layer, of the model. Lastly, any sequences of commands that are not the maximum length (12 in this case) are padded with zeros to the maximum length. Now that the input to the model has been properly encoded, the run times are converted into either zero or one based upon whether or not they are considered long. A

| | Runtime |
|---|---|
| count | 10000 |
| mean | 2.946746 |
| std | 21.811589 |
| min | 0.001396 |
| 25% | 0.063841 |
| 50% | 0.139309 |
| 75% | 0.42037 |
| max | 707.50322 |

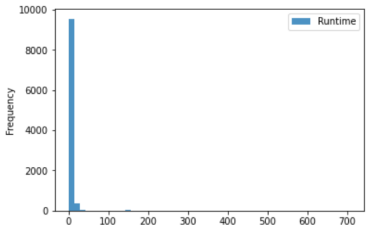


*Figure 6.* The stats from the runtime of all 10,000 randomly generated query strings. Note the large disparity between the median and the mean. Also note the shape of the histogram. There are a disproportionately small amount of long running queries but they dominate the average runtime.

long-running request is defined as a request whose runtime is greater than one second. The data pairs are then shuffled, and 80% of them are chosen for training, while 20% of them are held for validation.

## 4. Creating The Model

I used the open source library Pytorch and a popular wrapper for it, Pytorch Lightning to build and train the model. The model is most easily understood in terms of shapes. Since the encoded input to the model is always at the maximum sequence length, twelve, this is the first shaping we must consider. The first layer of the model is the embedding layer. This is where the integer encoding comes in. The embedding layer is basically a hash table whose index is the token's corresponding integer. The embedding layer stores 16 floating point numbers associated with each token in the vocabulary. These embeddings are learned over time to suit the models representational needs. Thus the output to the embedding layer is of size 192 since each of the 12 tokens gets a length 16 embedding. The next layer is a fully connected layer from 192 to 32 and with a ReLU activation. Since the relative order of the commands doesn't matter we don't need RNNs for this model. The last layer is a fully connected layer from 32 to 1 with a sigmoid activation. The sigmoid is key to the binary classification problem, since it smashes all network output between 0 and 1. If the output is less than 0.5, that input is classified as short, otherwise it is classified as long. I use Binary Cross-Entropy for the loss function and Adam for the optimizer. Specific hyperparameters like the learning rate, as well as $\alpha$ , and $\beta$ are available on the public github repository. The model is trained for 25 epochs where one epoch is a full iteration through the 8,000 training data points. Total training time is just over a minute on my desktop. This is due to the small dataset and even smaller model. The entire model has ¡10k parameters.

## 5. Evaluation

Since 16.5% of the data is in the long class the model must be able to beat 83.5% accuracy to have learned something. This is because the model could learn to just classify everything as short and it would be correct about 83.5% of the time. Fortunately as shown in Figure 7 and in Figure 8. Notice that the model converges on the training data. Unfortunately it has managed to completely memorize every data point in the generated dataset. However the validation accuracy does not reach 100% like the training accuracy. This is because the model is overfitting. This could be imporved in the future by adding more regularization during the training process (e.g. dropout) and by increasing the size of the dataset.

The model was able to obtain 94.52% validation accuracy after 25 epochs of training. It has certainly learned something about what differentiates short from long requests. With more data and better tuned network structure/hyperparameters performance could easily improve beyond this.

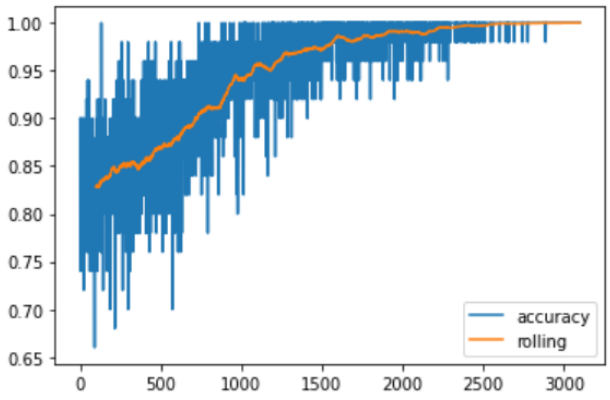


*Figure 7.* Training accuracy of the model plotted against the training step. The blue line is the raw data whereas the orange line is a 100 step exponential moving average. Notice that the model converges on the training data. It has managed to completely memorize every data point in the generated dataset.

### 5.1. Model Inference Performance

Figure 9 shows the time required for model inference. The model is running on standard PyTorch with no modifications like just in time compilation or exporting to torchscript. Inference is being performed on my desktop CPU and is being timed using Python's built in time.time_ns().

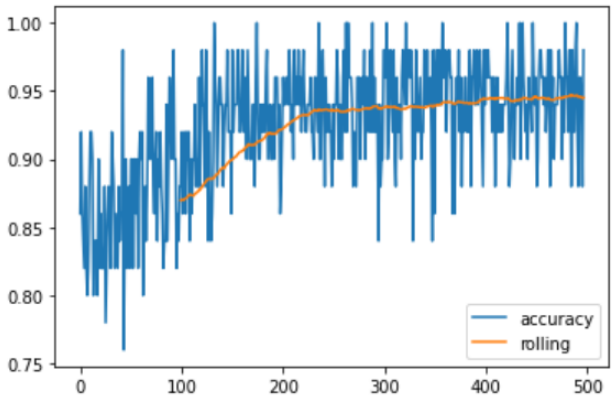


*Figure 8.* Validation accuracy of the model plotted against the training step. The blue line is the raw data whereas the orange line is a 100 step exponential moving average. Notice that the validation accuracy does not reach 100% like the training accuracy. This is because the model is overfitting.

## 6. Conclusion

Ultimately the model was able to successfully learn what queries tended to be long. However there are several limitations to this proof of concept. The space of possible iImageflow queries is not that large so the generalizeability of an approach like this to more complex applications, while likely given LLAMA's accomplishments, is not guaranteed. Additionally, the data for this experiment was entirely synthetic and potentially not representative of a real world trace. Since the model only tried to learn patterns for one application it was small enough to have impressive inference times for raw Python but it is certainly not fast enough for any work at the microsecond scale.

There are many areas for potential future improvement: adding more regularization and do hyperparameter tuning to reduce overfitting, rewriting the model in TensorFlow and compiling to JAX, pruning the model weights, training with mixed precision to take advantage of new hardware (Nvidia Tensor Cores), trying different model architectures trying on other tasks, and integrating with a running application to improve scheduling, to name a few.

After working on this project I am quite confident that we will continue to see a surge of ML applied to systems problems. I would not be surprised if it took many diverse forms as well. For example Google's LLAMA is a highly general data center wide upgrade to their memory management system. This kind of model however is only available to those with mountains of data. I hope to have shown that there can still be a place for small models that handle more specific tasks. It is not entirely unreasonable to imagine a future in which the two or three bottleneck applications in a

| | Model Inference Time (microseconds) |
|---|---|
| **count** | 160 |
| **mean** | 190.344825 |
| **std** | 52.601335 |
| **min** | 165.63 |
| **25%** | 178.12025 |
| **50%** | 181.0505 |
| **75%** | 185.096 |
| **max** | 672.033 |

*Figure 9.* Statistics computed from running the fully trained model 160 times on validation inputs.

data center have individual models built and trained just for them in a similar fashion to this proof of concept. After all, small models have several distinct advantages. They require less data to train, the models train faster, and they are faster for inference which makes them particularly appealing for applications like low latency scheduling. Ultimately, ML and systems will undoubtedly have a solid future together.

## References

Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 219–232, 2017. doi: 10.1109/PACT.2017.24.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Haque, M. E., Eom, Y. h., He, Y., Elnikety, S., Bianchini, R., and McKinley, K. S. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. *SIGPLAN Not.*, 50(4):161–175, March 2015. ISSN 0362-1340. doi: 10.1145/2775054.2694384. URL https://doi.org/10.1145/2775054.2694384.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. *CoRR*, abs/1712.01208, 2017. URL http://arxiv.org/abs/1712.01208.

Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., and Raffel, C. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pp. 541–556, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378525. URL https://doi.org/10.1145/3373376.3378525.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. *CoRR*, abs/1706.04972, 2017. URL http://arxiv.org/abs/1706.04972.

Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 69–84, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522716. URL https://doi.org/10.1145/2517349.2522716.