Deep neural networks models offer us huge support in artificial intelligence tasks such as image recognition, and captioning, video classification and captioning, speech recognition and text processing. Such DNN models need to be launched on the cloud instantly in serving mode to process the requests of the users. DNN serving platforms should satisfy two requirements: First, they should offer short response time to the user’s requests. Second, DNN should support fast deployment of applications. Once DNN models are trained and ready for deployment, the serving system should be able to accept online requests within a few minutes. Since DNN models are large containing billions of neural connections, processing them in sequential fashion on a single server is less preferred compared to handling them parallel. We can employ parallelism in three phases: First, Parallel hardware threads (Cores) on a machine can be used to handle a request (intra-node parallelism). Second, the same threads (cores) can be used to process multiple requests at the same time (service-level parallelism). Third, the model could be partitioned among multiple machines to leverage the aggregate compute cycles and memory bandwidth for faster processing of each requests.

Finding good parallelism configurations can minimize the latency by large degree. Also, latency values are a function of load being processed.

To find the best parallel configuration among all, we can exhaustively search for the best performance by running all kinds of expected loads under all available parallelism combinations. However, this could be expensive and may take hours or day to find out. Another way to do the task is, to use analytical modelling to find out the best performance. However, since performance depends on a lot of systems parameters, interplay of network characteristics, hardware and combined impact of memory contention and communication overhead, developing accurate analytical model would be very hard. In such a scenario, SERF tries to profile certain information using actual simulation and uses analytical modeling to find out the best configuration. Hence, it can obtain the best configuration in small time and use it to deploy the configuration online very fast.

However, SERF is limited by the fact that they are able to perform very well in presence of Exponential arrival rates only. Because, online systems may receive requests in any random arrival rates with various distributions, SERF cannot handle such situations. In such situation, we cannot use SERF, or any other methods that it had replaced.

We need to use machine learning algorithms to learn the best system configuration itself, and change the system parameters itself while the requests are coming to our system. If we are able to do this, this also saves us the time required to do some profiling. This can be accomplished by using Reinforcement Learning, a machine learning algorithm that interacts with the incoming requests and the system parameters to find out the best configuration for incoming requests by itself. The goal of reinforcement learning is to collect as much reward as possible ( in our case reward being a function of latency). And, based on the reward different configuration are selected. At the end of learning period, the RL algorithm learns the configuration that produces the least average latency and thus we know the best configuration. RL algorithm converges to the optimal activity with high probability. In worst case, it converges to sub-optimal configuration which is very close to optimal configuration, and the performance is always bounded.

We randomly select a configuration to operate the requests. The processing of the requests is divided into time frames. At the end of each time frame, the average latency of that time frame is used to find out reward function and the probability of that configuration getting selected,.

Reward Function:

= otherwise ( a(t) )

Probability of being selected,

And

At the end of each time frame, average latency for that time frame is used as input to calculate these function and a configuration is selected randomly based on the probability function. Then, the requests are processed using the newly selected parameters and average latency is collected for that time frame, and the functions are updated. This goes on for long time at the end of which the algorithm converges to the optimal configuration by itself. This algorithm avoid the profiling hassle, and can be used directly online. Also, it will be able to deal with any distribution of inter-arrival rates.

We ran simulation for various combination of inter-arrival rates and service distribution.

Inter-arrival rate is the time duration between consecutive requests. Although online requests may not follow any distribution, we assume some distribution to simplify our analysis.

Service Distribution is the distribution of the time required to process each requests. We assume, it has some distribution. In SERF’s simulation, we used parameters (collected from prior experiments and profiling) to find out the service time, but in RL algorithm we assume that the service time is distributed in some distribution of the same parameters.

For Uniform Inter-arrival rates, distributed about g\_arrival mean, and x number of request in a time frame, we observe the following results:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G\_arrivalmean | Number of requests in a time frame(x) | Average Latency for SC | Average latency for RL | Optimal Latency | Remarks |
| 200 | 5 | 170.83 | 18.91 | 29.64 |  |
| 200 | 10 | 170.83 | 31.94 | 29.68 |  |
| 200 | 100 | 170.83 | 36.91 | 29.61 |  |
| 100 | 100 | 142.93 | 39.54 | 29.61 |  |
| 100 | 50 | 142.93 | 33.4 | 29.63 |  |
| 100 | 10 | 142.93 | 31.53 | 29.69 |  |
| 100 | 8 | 142.93 | 29.8 | 29.65 |  |
| 100 | 9 | 142.93 | 30.84 | 29.65 |  |

WE can see that RL si better than SC in all the cases, and closely approximates the optimal latency at all the times.

For Uniform inter-arrival rate about G\_arrivalmean and Exponential service time distribution, we can see that RL performs better than SC and closely approximates optimal average latency.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G\_arrivalmean | Number of requests in a time frame | Average Latency for SC | Average latency for RL | Optimal Average Latency | Remarks |
| 200 | 9 | 170.83 | 28.73 | 29.66 |  |
|  |  |  |  |  |  |
| 70,80,90,100,150,200, 400,500,1000,2000 | 50 | 109.514 | 53.23 | 43.94 |  |
| 70,80,90,100,150,200, 400,500,1000,2000 | 35 | 109.514 | 86.81 | 43.81 |  |
| 70,80,90,100,150,200, 400,500,1000,2000 | 100 | 109.514 |  |  | Didn’t converge, (unstable region) |
| 100,150,200,250,260,270,290,295,300,500 | 100 | 96.8 | 76.6 | 38.88 |  |
| 100,150,200,250,260,270,290,295,300,500 | 20 | 96.8 | 46.08 | 39.0225 | 1000 time frame |
| 100,150,200,250,260,270,290,295,300,500 | 10 | 96.8 | 61.4434 | 38.549 | 1000 time frame |
| 100,150,200,250,260,270,290,295,300,500 | 5 | 96.8 | 31.85 | 38.07 |  |
| 100,150,200,250,260,270,290,295,300,500 | 20 | 96.8 | 37.62 | 38.88 | 5000 time frames |
| 100,150,200,250,260,270,290,295,300,500 | 50 | 96.8 | 44.99 | 38.99 | 5000 time frames |

Conclusion:

For the above cases, we can see that RL is better than SC by huge margin, and RL approximates the optimal case closely. Hence, we can say that RL could be used for implementing DNN models faster and for better performance than existing approaches.