Auto Scaling Online Learning

Marco Tulio Ribeiro, Shrainik Jain

Mar 17, 2014

We propose a framework and algorithms for scaling online machine learning up or down, according to demand, and the priorities of the system. Different systems have different needs in terms of the cost assigned to machines, the cost of a bad user experience, etc. In this project, we focus most on the part that is general to auto scaling any application (and not only machine learning algorithms), but propose a framework that takes some ML particularities into account.

1 Introduction

Online machine learning algorithms operate on a single instance at a time. They have become particularly popular in natural language processing and applications with streaming data, including classification, ranking, etc [1, 2, 3]. Online learning is particularly interesting in the scenarios where data keeps streaming in, such as a web search engine doing advertisement placement. It is also interesting for scenarios where the whole dataset is too large to fit in main memory, as online learning only operates on a single example at a time.

A lot of tasks that use online learning have a particular structure that can be broken down into 2 major components: 1 - Learning a model from data, and 2 - making predictions according to the model. Going back to the web search engine scenario as an example: the system needs to make predictions for every user doing a query - and must also learn from the feedback given by those users. We call machines that make predictions **predictors**, and machines that learn from the feedback **learners**.

This structure comes with multiple challenges. First, the the amount of data is always growing, so archiving it comes at a cost, both because of storage constraints and computation constraints. A solution to this is to just keep the current model in memory, and archive the rest in the background. A second challenge is the variable speed at which data streams in. Imagine a learning problem where in the learning dataset is a live twitter stream for a hashtag. In this scenario the rate at which the data comes in is a function of the popularity of the hashtag. Finally, different applications have different costs for learning, and different requirements for the latency of predictions.

The problem we tackled in this project is the problem of automatically handling the resources needed for online learning. The ideal system would allocate the resources necessary to keep the prediction latency acceptable, while at the same time learning appropriately. Finally, the system would be able to handle bursts (such as increase in demand) and different learning requirements for different systems. Since online learning in a distributed system is a research problem on its [4, 5], we abstracted this part from our work, and focused on some of the systems challenges.

Current works in auto-scaling[6, 8] address many of the issues we address in this report (such as load prediction, framing the problem as a cost minimization problem, etc). However, one aspect that we thought was lacking in current work (at least from the papers we read) is dealing with node failures and uncertainty. We reformulate the cost function in terms of expected cost, in order to account for uncertainty pertaining future load predictions and node failures. Finally, we evaluate some baselines and our proposed approach with simulated loads.

2 Architecture

We assume an architecture similar to the one presented in Figure 1. There is a set of nodes acting as a Load Balancer (whose addresses are known to the clients of the machine learning service). When a client needs a predictor or a learner, it first requests an a node from the Load Balancer, and then it proceeds to make the request. In Figure 1, the client got assigned the node in red.

The Load Balancer reads the state of the current system from some distributed, reliable storage. The state is comprised of which nodes are up and acting as learners or predictors, and a measure of the **power** of each node - that is, how many requests each node can effectively handle in a time interval. This allows for load balancing even when heterogeneous nodes are present.

The state of the current system is modified every so often by the set of nodes denoted **Autoscaler** in Figure 1. The Autoscaler periodically sends heartbeat messages to nodes and to the load balancer, and tests the prediction / learning time. It also gets the status - i.e., the number of requests served, eventual node failures, etc. Finally, it then updates

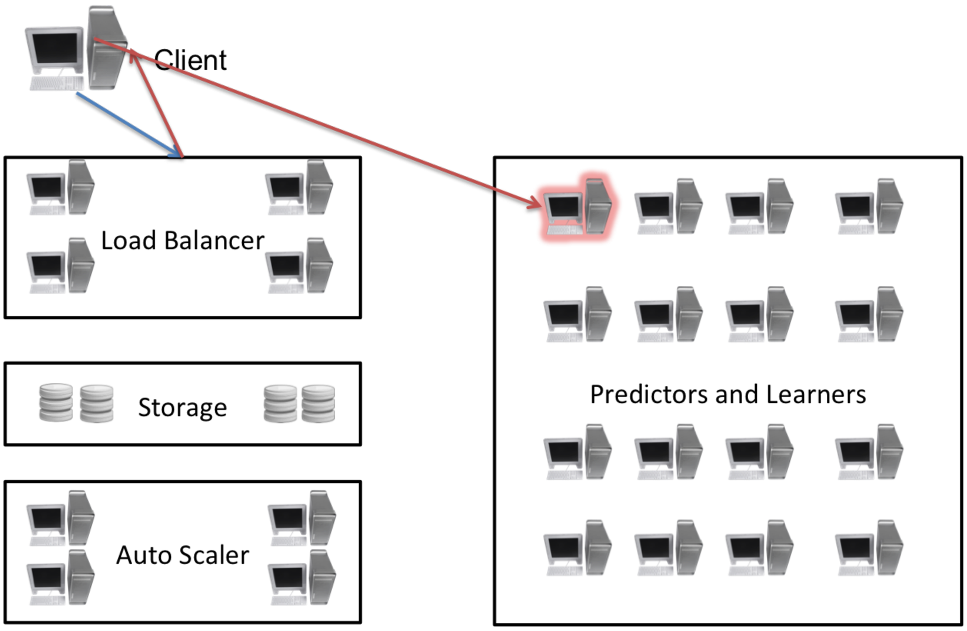


Figure 1: Architecture: Node asks the load balancer for a predictor, and then asks the predictor for a prediction.

the state in the distributed storage according to some policy. This architecture is very straightforward, and can be used with commodity machines for handling a lot of requests, while still being resilient to failures (depending of course on the Autoscaler policy).

3 Auto Scaling

3.1 Objective

The problem of auto scaling online learning can be formulated as minimizing a cost function over a sequence of time intervals. This function for each time interval is presented in Equation 1.

$$cost = \alpha SLA + \beta P + \gamma LR + \delta L \tag{1}$$

SLA is an indicator variable that is positive when a Service License Agreement (SLA) is violated in the time interval in question. This SLA is related to predictions, and can be defined as a threshold on the average time per prediction, on the maximum time per prediction, etc. There is a cost α associated with the SLA violation, that depends on the specific application that depends on the system. A bad user experience due to slow page loading on a search engine can be fatal to the success of the business, while a slower prediction for some internal testing service can be less costly. **P** denotes the number of predictors active in the time interval, while β is a measure of the cost of each predictor. **LR** is a measure of the learning rate achieved by the machine learning algorithm. Some of the most popular online learning algorithms, such as Stochastic Gradient Descent, have a property that error ϵ decreases linearly with the number of training examples [7]: that is, the relation is $O(1/\epsilon)$, meaning that if training examples are available, adding more learners would offset the training error, and thus increase whatever measure of Learning Rate is being used - although probably not with linear speedup [4, 5]. **L** denotes the number of learners active in the time interval.

This objective formulation decouples learning from predicting, which brings advantages and disadvantages. Usually, the processor / memory requirements for learning and predicting are different - one would want a more powerful machine for learning than for predicting. Decoupling also ensures that failures in learners do not affect predictors, and vice versa. However, having learners and predictors together could be interesting in that learning involves predicting, and thus separating them creates the need to do predictions twice for the same examples: once in the predictor and once in the learner. In this project, we abstracted out the learning part, and focused on minimizing the first two terms in Equation 1. This choice was made assuming that a fixed, small number of learners would be sufficient for many applications, and that this number would not need to be changed very often - in contrast to predictors, who would need to be changed according to load.

3.2 Load Prediction

Ideally, we would employ a machine learning algorithm in order to predict the load in each future time interval. Features that we would consider include:

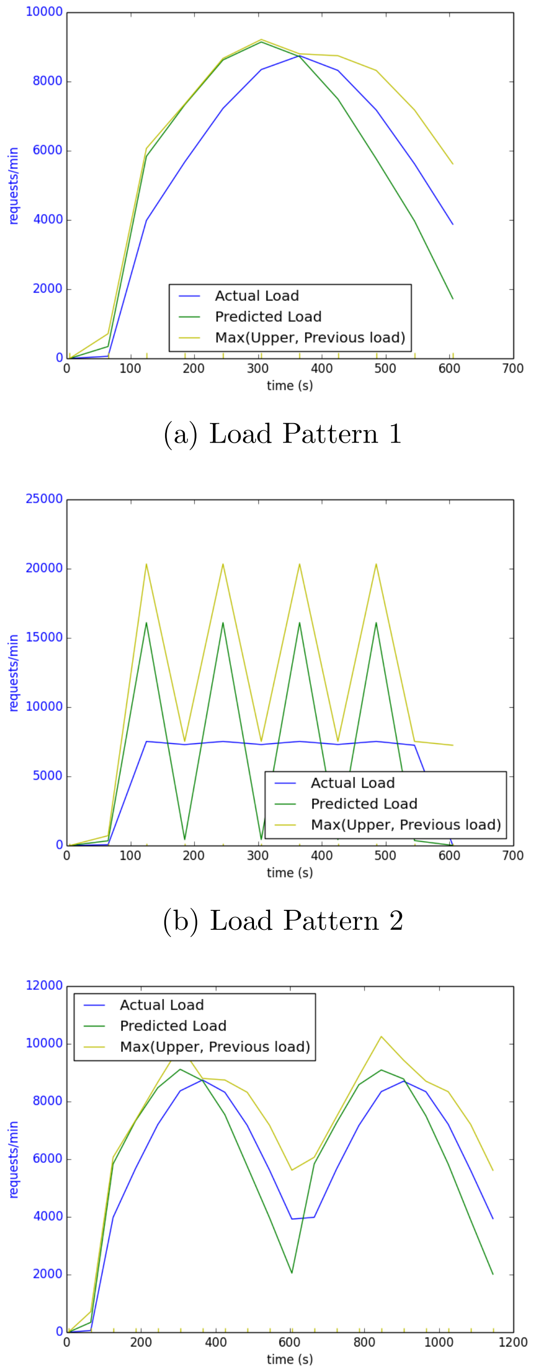
1. Time of day: traffic in most applications have load patterns that are very time-specific. Most people are sleeping at night, have a lunch break around the middle of the day, etc.

2. Trends in other services by the same company: if Facebook sees an increase in the number of users in the website at a particular time, it is more likely that internal services that use machine learning will also see an increase.

3. Usual load pattern: certain applications have very predictable load patterns. An internal service that always gets called in exponentially growing bursts would be an example.

4. Trend in the last time interval(s): by measuring the load at smaller intervals in the last time interval, one can build a “load trend” for the next time interval.

Since we used simulated data (we do not have access to such a rich dataset), we employed the trend in the last time interval in order to predict future load, by using traditional linear regression. In order to ensure a smoother decay (and thus prevent SLA violations due to wrong predictions), we took the max between the upper limit on a 95% confidence interval on the linear regression and the number of requests in the previous time interval as a prediction. Figure 2 illustrates the behaviour of our predictions on simulated data. We assume that we must predict the load at every minute, and collect the current load at every 5 seconds in order to build the trend line. The blue lines are the actual load, while the green and yellow lines indicate the linear regression prediction and the max we cited before, respectively. It is clear that the yellow line is more “conservative”, and always stays on top of the actual load. This is what we used for predicting loads in our experiments.



(a) Load Pattern 1

(b) Load Pattern 2

(c) Load Pattern 3

Figure 2: Predicted vs Actual load, different load patterns.

3.3 Strategies

3.3.1 Naive Strategy

The Naive strategy, presented in Algorithm 1, is probably the first solution anyone would come up with when faced with the auto scaling problem. It is a reactive strategy that adds more nodes when the SLA is being violated, and removes nodes when the SLA is far from being violated. In our experiments, we assumed that the SLA was based on average prediction time being smaller then some constant T, and set $C = T/2$.

```
if There is an SLA violation in the current time interval then
    Add more nodes to the system
end
if The mean prediction time is smaller than some constant $C$
then
    Remove nodes from the system
end
```

Algorithm 1: Naive Strategy

3.3.2 Power Strategy

The Power strategy, depicted in Algorithm 2, takes into account the power (number of requests the machine can handle) of each node, and compares the sum of the power of all the current nodes against the number of predicted requests for the next time interval. If the current power of the system is smaller than the predicted requests, it adds more nodes. If removing nodes does not cause the power of the system to be smaller than the number of predicted requests, it removes nodes.

```
while Current Power < predicted requests do
    Add more nodes to the system
end
while Current Power - power($node_i$) > predicted requests do
    Remove $node_i$
end
```

Algorithm 2: Power Strategy

3.3.3 Smart Strategy

The two strategies presented before do not take into account the costs α and β. Furthermore, they do not take into account the probability of nodes failing, or the uncertainty associated with the load prediction. A reformulated objective function is presented in Equation 2. Here, we are selecting the configuration of machines P and L that minimizes the expected cost in the next iteration.

$$min_{P,L} : E[cost] = E[\alpha SLA + \beta P + \gamma LR + \delta L] \tag{2}$$

Keeping the learners fixed, as we did before, the reformulated objective becomes:

$$\begin{aligned} min_P : E[cost] &= E[\alpha SLA + \beta P] \\ &= \alpha E[SLA] + \beta P \end{aligned}$$

One way of defining a SLA violation is when we have less power in a time interval than needed for the number of requests. Ideally, we would integrate out the uncertainty over our future prediction, and calculate $E[SLA]$ as follows, given that ρ is the probability that a node fails in the time interval, and each r denotes a possible value for the future load.

$$E[SLA] = P(SLA|P, \rho) = \int P(power < r|P, \rho, r)dr$$

This would require a more reliable estimator of future load, one that could output a probability distribution over the possible values of r. There are multiple machine learning algorithms that can do this, but since we are only using the trend in the last interval as a feature, we did not think it worthwhile to pursue this route for this project. Therefore, we assumed that the prediction for the future load was correct. $E[SLA]$ then becomes as following, noting that k is the minimum number of nodes required for the system's power to be greater than the number of requests:

$$\begin{aligned} P(SLA|P, \rho, requests) &= P(power < requests|P, \rho) \\ &= P(|\text{non failing nodes}| < k) \\ &\approx CDFBinomial(P, k-1, 1-\rho) \end{aligned}$$

Note that the last approximation depends on the nodes' power being close to uniform. This is a reasonable assumption on most cases. The smart strategy, therefore, amounts to picking P such that the expected cost function presented in Equation 2 is minimized.

References

[1] Antoine Bordes and Léon Bottou. The huller: A simple and efficient online svm. In *Proceedings of the 16th European Conference on*

Machine Learning, ECML'05, pages 505–512, Berlin, Heidelberg, 2005. Springer-Verlag.

[2] Vitor R. Carvalho and William W. Cohen. Single-pass online learning: Performance, voting schemes and online feature selection. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 548–553, New York, NY, USA, 2006. ACM.

[3] Mark Dredze, Koby Crammer, and Fernando Pereira. Confidence-weighted linear classification. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 264–271, New York, NY, USA, 2008. ACM.

[4] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS workshop on BigLearning*, pages 1223–1231, 2013.

[5] Mu Li, Li Zhou, Z Yang, A Li, Fei Xia, D Andersen, and A Smola. Parameter server for distributed machine learning. In *NIPS workshop on BigLearning*, 2013.

[6] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 49:1–49:12, New York, NY, USA, 2011. ACM.

[7] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM J. on Optimization*, 19(4):1574–1609, January 2009.

[8] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, July 2011.