Pay-Per-Request Deployment of Neural Network Models Using Serverless Architectures

Zhucheng Tu, Mengping Li, and Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo
{michael.tu, m282li, jimmylin}@uwaterloo.ca

Abstract

We demonstrate the serverless deployment of neural networks for model inferencing in NLP applications using Amazon's Lambda service for feedforward evaluation and DynamoDB for storing word embeddings. Our architecture realizes a pay-per-request pricing model, requiring zero ongoing costs for maintaining server instances. All virtual machine management is handled behind the scenes by the cloud provider without any direct developer intervention. We describe a number of techniques that allow efficient use of serverless resources, and evaluations confirm that our design is both scalable and inexpensive.

1 Introduction

Client–server architectures are currently the dominant approach to deploying neural network models for inferencing, both in industry and for academic research. Once a model has been trained, deployment generally involves "wrapping" the model in an RPC mechanism (such as Thrift) or a REST interface (e.g., Flask in Python), or alternatively using a dedicated framework such as TensorFlow-Serving (Olston et al., 2017). This approach necessitates provisioning machines (whether physical or virtual) to serve the model.

Load management is an important aspect of running an inference service. As query load increases, new server instances must be brought online, typically behind a load-balancing frontend. While these issues are generally well understood and industry has evolved best practices and standard toolsets, the developer still shoulders the burden of managing these tasks. A server-based architecture, moreover, involves a minimum commitment of resources, since *some* server must be running all the time, even if it is the smallest and cheapest virtual instance provided by a cloud service. Particularly relevant for academic researchers, this means that a service costs money to run even if no one is actually using it.

As an alternative, we demonstrate a pay-per-request serverless architecture for deploying neural network models for NLP applications. We applied our approach to two real-world CNNs: the model of Kim (2014) for sentence classification and the CNN of Severyn and Moschitti (2015) for answer selection in question answering (SM CNN for short). On Amazon Web Services, the models cost less than a thousandth of a cent per invocation. Model inference does not require the developer to explicitly manage machines, and there are zero ongoing costs for service deployment. We show that our design can transparently scale to moderate query loads without requiring any systems engineering expertise.

2 Serverless Architectures

A serverless architecture does not literally mean that we can magically perform model inference without requiring servers; the computation must happen somewhere! A serverless design simply means that the developer does not need to explicitly manage servers—the cloud provider shoulders this burden behind the scenes.

Serverless architectures make use of what is known as function as a service (FaaS), where developers specify blocks of code with well-defined entry and exit points and the cloud provider handles the invocation. Typically, function invocation involves spinning up virtual machine instances and bootstrapping the execution environment, but all of these tasks are handled by the cloud provider. The developer pays per function invocation no matter the query load; scalability and elasticity are the responsibility of the cloud provider. In most cases, these invoked functions are stateless, with state usually offloaded to another cloud ser-

Proceedings of NAACL-HLT 2018: Demonstrations, pages 6–10
New Orleans, Louisiana, June 2 - 4, 2018. ©2018 Association for Computational Linguistics

vice. The standard design pattern begins with the invoked function reading from a persistent store and writing results back to the same or a different store. The serverless paradigm meshes well with microservice architectures that are in fashion today, and multiple cloud providers (Amazon, Google, Microsoft) have FaaS offerings.

Recently, Crane and Lin (2017) proposed a novel search engine built using a serverless architecture on Amazon Web Services whereby postings lists are stored in DynamoDB and query execution is encapsulated in Lambda functions. We explore how similar techniques can be applied to neural network models for NLP applications. We are aware of a recent blog post describing the deployment of NN models using Lambda (Dietz, 2017). However, that work focuses on vision applications; the additional technical challenge we overcome is the need to access word embeddings for NLP applications, which requires more than just Lambda deployment.

3 Serverless Neural Network Inference

In this work, we selected Amazon Web Services (AWS) as our deployment platform due to its market-dominant position, although other cloud providers have similar offerings.

To enable serverless neural network inference for NLP, the trained models are packaged together with the function to be invoked and dependent software libraries. The cloud provider is responsible for creating the environment for execution. During inference, the Lambda function takes input text, which is supplied externally via an API gateway. Sentences need to be first transformed into an embedding matrix constructed using word vectors, in our case from word2vec (Mikolov et al., 2013). These are fetched from DynamoDB. Finally, the Lambda function applies feedforward evaluation on the embedding matrix according to the supplied model, yielding a final prediction. Figure 1 illustrates this architecture, described in detail below.

3.1 Lambda Deployment Package

A complete Lambda deployment package comprises the code of the function as well as its dependencies. In this work, we use PyTorch v0.3.1 for inference. Thus, our deployment package requires PyTorch as well as its dependencies, the model definition, the model weights, and a handler that specifies how the function should be executed. Note that although an emerging deployment pattern is to use PyTorch for training models and Caffe2 for running inference in a production environment through the use of ONNX,[1] support for this approach remains immature and thus we rely on PyTorch for inference as well.

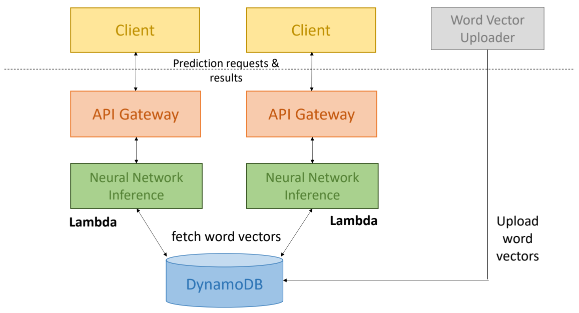

Figure 1: Serverless architecture for deploying NNs.

There is a 250 MB limit on the size of the Lambda deployment package. To stay within this limit, we had to build PyTorch from source on an AWS EC2 machine to exclude CUDA and other unnecessary dependencies. The machines that execute Lambda functions do not have GPU capabilities, so this does not incur any performance penalties. The deployment package is compressed and uploaded to S3, Amazon's object storage service. In the Lambda execution model, an Execution Context is initialized upon invocation of the Lambda function, which is a runtime environment that bootstraps the dependencies in the deployment package. The Execution Context may be costly to set up (as in our case) and therefore the Lambda API provides hooks that facilitate reuse for subsequent invocations.

3.2 DynamoDB Storage of Word Vectors

Most neural networks for NLP use pre-trained word vectors to build an input representation as the first step in inference. In a serverless architecture, these word vectors need to be stored somewhere. Due to the size restrictions described above, the word vectors cannot be stored in the deployment package itself.

To overcome this issue, we adopt the solution of storing the word embedding vectors in DynamoDB, much like how Crane and Lin (2017) store postings lists. DynamoDB (DeCandia et al., 2007) is a hosted "NoSQL" database service that offers low latency access to arbitrary amounts of data, as Amazon scales up and down capacity au-

[1]https://onnx.ai/

tomatically. We use DynamoDB as a key–value store for holding the word vectors, where each word is the key and its word vector is the value stored as a List type. Kim CNN uses 300 dimensional word vectors from word2vec trained on the Google News corpus and SM CNN uses 50 dimensional word vectors from word2vec trained on English Wikipedia. Thus, we created separate DynamoDB tables for these 50 and 300 dimensional word vectors. For expediency in running experiments, we only load the word vectors for words in the vocabulary of the datasets we use.

3.3 Neural Network Inference

Our API for invoking the NN models comprises a JSON request sent to the AWS API Gateway via HTTP, which is a proxy that then forwards the request to Lambda. A request for feedforward inference using Kim CNN consists of a single sentence in the request body, whereas for SM CNN, the request body holds a pair of sentences.

Upon receiving a request, the Lambda handler first tokenizes and downcases the input. It then issues `BatchGetItem` requests to DynamoDB to fetch the word vectors for unique words in the input. These queries retrieve the word vectors in parallel to reduce latency. The function then blocks until all word vectors are retrieved, after which they are concatenated together to construct a sentence embedding matrix.

In our implementation, the model is initialized outside of the Lambda handler function scope so that if an existing Event Context is available, a previously-loaded model can be reused. If the model has not been initialized, it will be loaded from the deployment package. Note that context reuse is completely opaque: unbeknownst to us, AWS performs caching to support efficient invocations as query load ramps up, but we have no explicit control over the exact mechanisms for eviction, warmup, etc. The sentence embeddings are fed into the model for feedforward evaluation (handled by PyTorch) and the result is returned as JSON from the handler.

4 Experiments

We first provide some implementation details: Kim CNN is a sentence classification model that consists of convolutions over a single sentence input matrix and pooling followed by a fully-connected layer with dropout and softmax output. We used the variant where the word embeddings are not fine-tuned via backpropagation during training (called the "static" variant). SM CNN is a model for ranking short text pairs that consists of convolutions using shared filters over both inputs, pooling, and a fully-connected layer with one hidden layer in between. We used the variant described by Rao et al. (2017), which excludes the similarity matrix (found to increase accuracy) as well as the additional features that involve inverse document frequency. In our experiments, we are focused only on execution performance, which is not affected by these minor tweaks, primarily for expediency. All of our code and experiment utilities are open-sourced on GitHub.[2]

Before detailing our experimental procedure and results, we need to explain Amazon's cost model. Lambda costs are very straightforward, billed simply by how long each function executes in increments of 100ms, for a particular amount of allocated memory that the developer specifies. DynamoDB's cost model is more complex: it supports two modes of operation, termed manual provisioning and auto scaling. In the first mode, the developer must explicitly allocate read and write capacity. Amazon provides the capacity, but the downside is a fixed cost, even if the capacity is not fully utilized (and over-utilization will result in timeouts). Thus, this mode is not truly "pay as you go". The alternative is what Amazon calls auto scaling, where the service continuously monitors and adjusts capacity on the fly.

For our experiments, we opted to manually provision 500 Read Capacity Units (RCUs), which translates into supporting a DynamoDB query load of 1000 queries per second (fetching the word vector for each word constitutes a query). This choice makes our experimental results easier to interpret, since we have little insight into how Amazon handles auto scaling behind the scenes. Note however, that we adopted this configuration for experimental clarity, because otherwise we would be conflating unknown "backend knobs" in our performance measurements. In production, auto scaling would be the preferred solution.

To evaluate performance, we built a test harness that dispatches requests in parallel, with a single parameter to control the number of outstanding requests allowed when issuing queries. We call this

[2]`https://github.com/castorini/serverless-inference`

C	tput	Latency (ms)			Cost
	(QPS)	mean	p50	p99	($/10^6$ Q)
5	7.0	700	678	1285	\$1.46
10	13.0	740	722	1283	\$1.66
20	23.7	802	779	1357	\$1.87
30	32.3	845	817	1447	\$1.87

Table 1: Latency, throughput, and cost of serverless Kim CNN under different loads (C).

C	tput	Latency (ms)			Cost
	(QPS)	mean	p50	p99	($/10^6$ Q)
5	12.1	410	381	657	\$1.04
10	21.1	468	443	780	\$1.04
15	30.8	467	439	827	\$1.04
20	38.5	496	486	785	\$1.04
25	44.4	530	519	814	\$1.25

Table 2: Latency, throughput, and cost of serverless SM CNN under different loads (C).

the concurrency parameter, which we vary to simulate different amounts of query load. With different concurrency settings (ramping down from maximum load), we measured latency (mean, 50th and 99th percentile) and throughput. For Kim CNN, we used input sentences from the validation set of the Stanford Sentiment Treebank (1101 sentences). For SM CNN, we used input sentence pairs from the validation set of the TrecQA dataset (1148 sentences). We conducted each experimental trial multiple times before taking measurements to "warm up" the backend.

Results are shown in Table 1 for Kim CNN and Table 2 for SM CNN. Our deployment package is bundled with OpenBLAS to take advantage of optimized linear algebra routines. In both cases, we see that latency increases slightly as throughput ramps up. This suggests that we are not achieving perfect scale up. In theory, AWS should be proportionally increasing backend resources to maintain constant latency. It is not clear if this behavior is due to some nuance in Lambda usage that we are not aware of, or if there are actual bottlenecks in our design. Note that Kim CNN is slower because it is manipulating much larger word vectors (300 vs. 50 dimensions).

The final column in Tables 1 and 2 report Lambda charges in US dollars per million queries based on the mean latency. As of February 2018, for functions allocated 128 MB of memory, the cost is \$0.000000208 for every 100ms of running time (rounded up). Note that these costs do not include provisioning DynamoDB, which costs 0.013 cents per Read Capacity Unit per hour. We have not probed the scalability limits of our current architecture, but it is likely that our design can handle even larger query loads without additional modification.

We performed additional analyses to understand the latency breakdown: logs show that approximately 60–70% of time inside each function invocation is spent building the embedding matrix, which requires fetching word vectors from DynamoDB. In other words, inference latency is dominated by data fetching. This is no surprise since these queries involve cross-machine requests. The rest of the time is spent primarily on feedforward evaluation. The amortized cost of loading the model is negligible since it can be reused in subsequent invocations.

5 Future Work and Conclusions

We describe a novel serverless architecture for the deployment of neural networks for NLP tasks. Our design appears to be feasible, and experiments show that it scales up to moderate query loads inexpensively. For reference, a sustained query throughput of 20 queries per second translates into 1.7 million queries per day. While there are certainly many web-scale services that handle larger query loads, our serverless design is able to achieve this scale with *zero* engineering effort, since the cloud provider handles all aspect of load management without any developer intervention.

In terms of design improvements within our control, tackling the latency of DynamoDB queries would yield the biggest impact, since fetching the word vectors accounts for most of the request latency. One simple idea would be to retain a cache of the most frequent words in the Lambda itself. This would not improve "cold" startup latency, but would speed up requests once the cache has been populated. Beyond elements in our control, further advances in cloud infrastructure "behind the scenes" will improve usability, performance, and cost, making serverless architectures increasingly attractive.

Acknowledgments. This research was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

Matt Crane and Jimmy Lin. 2017. An exploration of serverless architectures for information retrieval. In *Proceedings of the 3rd ACM International Conference on the Theory of Information Retrieval (ICTIR 2017)*, pages 241–244, Amsterdam, The Netherlands.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 205–220, Stevenson, Washington.

Michael Dietz. 2017. Serverless deep/machine learning in production—the pythonic way. https://blog.waya.ai/deploy-deep-machine-learning-in-production-the-pythonic-way-a17105f1540e.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, pages 1746–1751, Doha, Qatar.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv:1301.3781*.

Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, high-performance ML serving. In *Workshop on ML Systems at NIPS 2017*.

Jinfeng Rao, Hua He, and Jimmy Lin. 2017. Experiments with convolutional neural network models for answer selection. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1217–1220. ACM.

Aliaksei Severyn and Alessandro Moschitti. 2015. Learning to rank short text pairs with convolutional deep neural networks. In *Proceedings of the 38th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2015)*, pages 373–382, Santiago, Chile.