An Architecture for Accelerated Large-Scale Inference of Transformer-Based Language Models

Amir Ganiev* and Colt Chapin and Anderson de Andrade and Chen Liu* Wattpad

Toronto, ON, Canada

Abstract

This work demonstrates the development process of a machine learning architecture for inference that can scale to a large volume of requests. In our experiments, we used a BERT model that was fine-tuned for emotion analysis, returning a probability distribution of emotions given a paragraph. The model was deployed as a gRPC service on Kubernetes. Apache Spark was used to perform inference in batches by calling the service. We encountered some performance and concurrency challenges and created solutions to achieve faster running time. Starting with 3.3 successful inference requests per second, we were able to achieve as high as 300 successful requests per second with the same batch job resource allocation. As a result, we successfully stored emotion probabilities for 95 million paragraphs within 96 hours.

1 Introduction

As data in organizations becomes more available for analysis, it is crucial to develop efficient machine learning pipelines. Previous work (Al-Jarrah et al., 2015) has highlighted the growing number of data centers and their energy and pollution repercussions. Machine learning models that require less computational resources to generate accurate results reduce these externalities. On the other hand, many machine learning applications also require results in nearly real-time in order to be viable and may also require results from as many data samples as possible in order to produce accurate insights. Hence, there are also opportunity costs associated with missed service-level objectives.

Attention-based language models such as BERT (Devlin et al., 2019) are often chosen for their relative efficiency, and empirical power. Compared to recurrent neural networks (Hochreiter and Schmidhuber, 1997), each step in a transformer layer (Vaswani et al., 2017) has direct access to all

other steps and can be computed in parallel, which can make both training and inference faster. BERT also easily accommodates different applications by allowing the fine-tuning of its parameters on different tasks. Despite these benefits, exposing these models and communicating with them efficiently possesses some challenges.

Machine learning frameworks are often used to train, evaluate, and perform inference on predictive models. TensorFlow (Abadi et al., 2016) has been shown to be a reliable system that can operate at a large scale. A sub-component called Tensor-Flow Serving allows loading models as services that handle inference requests concurrently.

System architectures for inference have changed over time. Initial approaches favored offline settings where batch jobs make use of distributed platforms to load models and data within the same process and perform inference. For example, Ijari, 2017 suggested an architecture that uses Apache Hadoop (Hadoop, 2006) and Apache Pig for largescale data processing, where results are written to a Hadoop Distributed File System (HDFS) for later consumption. Newer distributed platforms such as Apache Spark (Zaharia et al., 2016) have gained prominence because of their memory optimizations and more versatile APIs, compared to Apache Hadoop (Zaharia et al., 2012).

As part of this architecture, inference services would often be reserved for applications that require faster responses. The batch-based and service-based platforms have different use cases and often run in isolation. Collocating data and models in a batch job has some disadvantages. Loading models in the same process as the data forces them both to scale the same way. Moreover, models are forced to be implemented using the programming languages supported by the distributed data platform. Their APIs often place some limitations on what can be done.

With the evolution of machine learning frame-

^{*}Work done while the author was working at Wattpad.

works and container-orchestration systems such as Kubernetes,¹ it is now simpler to efficiently build, deploy, and scale models as services. A scalable architecture was presented in (Gómez et al., 2014) that proposes the use of RESTful API calls executed by batch jobs in Hadoop to reach online services that provide real-time inference. Approaches like this simplify the architecture and address the issues discussed previously.

In this work, we present an architecture for batch inference where a data processing task relies on external services to perform the computation. The components of the architecture will be discussed in detail along with the technical challenges and solutions we developed to accelerate this process. Our application is a model for emotion analysis that produces a probability distribution over a closed set of emotions given a paragraph of text (Liu et al., 2019). We present benchmarks to justify our architecture decisions and settings. The proposed architecture is able to generate results for 95 million paragraphs within 96 hours.

2 Architecture design

We deployed our model as a TensorFlow service in a Kubernetes cluster. A sidecar service preprocessed and vectorized paragraphs and forwarded requests to this service. We used gRPC to communicate with the services,² which is an efficient communication protocol on HTTP/2. Both nearly realtime and offline use cases made calls to these services. We used Apache Spark for batch processing, which we ran on Amazon's AWS EMR service.³ Our batch job was developed using Apache Spark's Python API (PySpark). The batch job fetched a dataset of relevant paragraphs, called the inference service, and stored the results. The job had two modes: a backfill mode and a daily mode, which ran on a subset of mutated and new paragraphs. This batch job was part of a data pipeline, scheduled using Apache AirFlow⁴ and Luigi.⁵ Figure 1 shows the main components of this architecture.

2.1 Kubernetes vs. Apache Spark

One of the key issues we faced in scaling up our inference services was the growing size of the memory footprint of an instance. A standard practice when conducting model inference at scale in a MapReduce program such as Apache Spark is to broadcast an instance of the model to each distributed worker process to allow for parallel processing. However, when the footprint of these instances becomes too large, they begin to compete with the dataset being processed for the limited memory resources of the underlying cluster and, in many cases, exceeding the capacity of the underlying hardware.

While this issue does not preclude the use of Apache Spark for running inferences on large models at scale, it does complicate the process of implementing the job in a cost-efficient manner. It is possible to allocate more resources, but because the clusters are static in size, a lot of work has to go into properly calculating resource allocation to avoid over or under-provisioning. This is where the idea of offloading the model to Kubernetes comes into play.

While our MapReduce clusters struggled to scale and accommodate the larger models being broadcasted, by leveraging Kubernetes we were able to monitor and optimize resource usage as well as define autoscaling behaviors independently of this cluster. That said, while there are clear benefits to isolating your model from your MapReduce job we must now consider the added overhead of the network calls and the effort to build and maintain containerized services.

2.2 Kubernetes node pool

To ensure optimal resource usage, we provisioned a segregated node pool dedicated to hosting instances of our models. A node pool is a collection of similar resources with predefined autoscaling behaviors. We leveraged Kubernetes' builtin taint/toleration functionality to establish the required behavior. In Kubernetes, *Taints* designate resources as non-viable for allocation, unless deployments are specifically annotated as having a *Toleration* for said *Taint*. For this node pool, we selected instance types that offer faster CPUs, but provide an adequate amount of memory to load our models.

2.3 REST vs. gRPC

Once we made the decision to deploy our model as a service, we had to determine which network protocol to use. While representational state transfer (REST) (Pautasso et al., 2013) is a well-known standard, there were two aspects of our use case

¹https://kubernetes.io

²https://grpc.github.io

³https://aws.amazon.com/emr

⁴https://airflow.apache.org

https://github.com/spotify/luigi

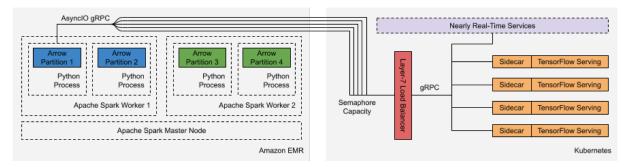


Figure 1: Architecture overview.

that made us consider alternatives. The first is that architecturally, our use case was far more functional in nature than REST. Second, the nature of our data means that request messages can be large. It was for this reason that we found the efficiency offered by the Protobul protocol a natural fit for our use case.⁶

Having decided to use gRPC and Protobuf, we encountered two issues. First, gRPC uses the HTTP/2 protocol which multiplexes requests over a single persistent TCP connection. Because of this persistent connection, Layer-4 load balancers that can only route connections are not able to recognize requests within them that could be balanced across multiple replicas of a service. To enable this, we rely on a Layer-7 load balancer which is able to maintain persistent connections with all devices, and identify and route requests within these channels accordingly.

The second issue was organizational in nature. REST is a widely accepted standard, but more importantly, it is a protocol that developers are familiar with. The introduction of a different API design has led to significant friction of adoption.

2.4 AWS EMR cluster configuration

The AWS EMR cluster needed to be configured to run a Apache Spark job that makes 95 million inference calls to our micro-service. Due to the unbounded nature of these paragraphs, which can become quite large in our use case, these 95 million records require a significant amount of disk space (in the order of terabytes).

Taking into account the cost constraints of this project, we chose an AWS *r5.xlarge* instance with 200 GiB of disk space as a master node, and 5 AWS *r5.4xlarge* instances with 1,000 GiB of disk space each, as worker nodes. This configuration ensures

that there is enough disk capacity to process the data and the number of cores is as high as possible without exceeding the cost constraints. Additionaly, we selected these to be memory-optimized to ensure we provide the job with enough RAM to efficiently process our joins.

The EMR cluster configuration is kept constant as a controlled variable throughout the project and in all of our experiments. This ensures that only the implementation changes affect the performance of the inference job.

2.5 Monitoring

There were two different solutions that monitor different aspects of the proposed architecture: Apache Spark console and DataDog. AWS EMR provided access to the Apache Spark console for its running tasks and a history server for completed tasks. The console displays the execution plan, the running stage, the number of partitions completed in that stage, the number of stages left to execute, as well as statistics and message logs of our inference job. Success or failure of this job and its pipeline was reported using DataDog.⁷ DataDog is a cloud based monitoring service that provides helpful visualization tools to monitor applications.

Additionally, our services were instrumented to report the number, latency, and status code of all calls received. We made use of DataDog to aggregate and monitor these metrics. In our implementation, the instrumentation was handled by functional wrappers around our endpoint handlers, as well as a synchronous gRPC Interceptor for the client on our sidecar service. Figure 2 shows an example of our request count on our daily job.

⁶https://developers.google.com/
protocol-buffers

⁷https://www.datadoghq.com

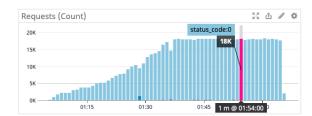


Figure 2: DataDog visualization of a daily job, consisting of 600,000 paragraphs calls to the service. The *X-axis* is the local time starting at 1 a.m. The *Y-axis* is the total number of calls executed within 1 minute. The highlighted vertical bar shows that 18,000 calls were executed within 1 minute (300 per second) at 1:54 a.m. The calls started at around 1 a.m. and reached a peak speed at around 1:40 a.m.

3 Architecture optimization

Our initial approach, which used the configuration in the previous section, was resilient to failures but performed slowly at around 4 requests per second during the inference step. With a backfill target of 95 million paragraphs, running this job was intractable. Our investigations concluded that the issues were rooted in a low request pressure on the backend services. Thus, the sections below describe the steps taken to address these issues and speed up the inference process.

3.1 Scaling model service

Our autoscaling group consisted of instances with Intel's Xeon Platinum 8175M CPUs and 64 GB of RAM. The use of GPUs is not cost-effective without a proper batching mechanism, which is considered to be outside of the scope of this work. Each instance had 8 physical cores and 16 logical cores. To reduce the memory footprint but also allow a fine-grained resource allocation, Kubernetes pods had a limit of 2 physical cores. In our experiments, pods did not consume more than 4 GB of memory under heavy load. Network utilization remained well under 10 Gb/s. We set up an autoscaling policy with a target CPU utilization of 70%.

With a maximum number of 100 pods (25 instances), we achieved a maximum of 300 requests per second, each request being a paragraph with at least 15 characters. Our daily job usually finished within 60 minutes.

3.2 Tuning TensorFlow Serving parameters

We evaluated the performance of TensorFlow Serving with multiple parameter configurations. The

MKL	OpenMP	Intra-Op	Req/Sec
Yes	2	2	5.207
Yes	2	4	5.931
Yes	4	2	4.786
Yes	4	4	5.714
No	-	2	5.464
No	-	4	6.452

Table 1: Average requests per second of the service under different TensorFlow Serving settings: MKL, number of OpenMP threads, and number of intra-operation threads. OpenMP is only used by MKL. Other configurations do not match the number of physical or logical cores available.

only settings that tangibly impacted performance included: enabling Intel's Math Kernel Library (MKL), the OpenMP number of threads for MKL, and the thread pool size for TensorFlow intraoperations. We used TensorFlow Serving version 2.3.0, which uses MKL-DNN version 0.21. Table 1 illustrates performance under different configurations for pods with 2 CPU physical cores and 4 logical cores. In particular, we note that disabling MKL and allocating a thread pool the size of the number of logical cores gave us the best performance for this model.

3.3 Spark job tuning

Configuring parameters of TensorFlow serving and successfully scaling up BERT micro-service allowed for a faster inference speed. However, adjusting the service alone did not yield better results as the speed remained relatively similar (3.3 complete calls per second). Therefore, a PySpark job reached its limits in the proposed configuration. The microservice was not receiving enough requests to trigger its autoscaling condition and capped out at 7 pods (far short of our max off 100). To address this, we sought to introduce more load by increasing the rate at which the client makes calls to the microservice.

Using synchronous calls, the number of requests the batch job can make is bounded by the number of cores assigned to it. Since the computation is done by the service, these cores will be mostly waiting for the service responses.

To address Python's synchronous nature limiting the rate at which a single core can make calls to the service, we leveraged the AsyncIO library⁸ within a PySpark User Defined Function (UDF),

⁸https://www.python.org

which allowed a single core to implement quasiconcurrent calls and leverage the idle thread awaiting a response. Since AsyncIO was utilized, the gRPC AsyncIO API⁹ was imported instead of default gRPC. The async gRPC is compatible with AsyncIO and can create asynchronous channels. Exceptions or errors returned by the call were accessed with *grpc.aio.AioRpcError* method.

Even with everything above implemented within the PySpark UDF, it was not possible to take advantage of AsyncIO yet. By default, a vanilla PySpark UDF receives only one tabular row at a time containing one paragraph. That means that the AsyncIO loop within the UDF was not be able to execute concurrent calls if only one paragraph was available. Apache Spark's Vectorized UDFs (Pandas UDFs) allows us to process partitions in batches and achieve the desired level of concurrency. Each batch is represented in memory using the Apache Arrow format and accessible with the Pandas API.

Apache Arrow is an in-memory columnar data format that facilitates the transference of data between the JVM (Java virtual machine), which runs the Apache Spark job, and Python processes (i.e. Pandas UDFs). It offers zero-copy reads between processes for data access without serialization overhead. In our work, a scalar Pandas UDF was defined to receive paragraphs as a Pandas Series and return a probability distribution of the emotion classes for each paragraph, as a new Pandas Series.¹⁰

With Python AsyncIO, gRPC Async, and Pandas UDFs using Apache Arrow, the load created by the client (PySpark job) substantially increased. AsyncIO ensured that extra paragraphs were sent to the server while waiting to receive emotion probabilities for outstanding calls. However, as soon as the first one thousand calls were sent to the server (in a matter of seconds), the PySpark job failed. The errors received by the client were canceled, unavailable or the deadline was exceeded (more about gRPC errors in section 3.4). That indicated that the client actually created too much load on the server causing it to respond with errors. The Kubernetes Deployment was overwhelmed and did not have enough time to scale up the micro-service. This led to unavailable and deadline errors.

To limit the maximum number of concur-

Semaphore Value	Responses Per Second
10	170
25	256.7
50	298.3
75	Service upscaling fails

Table 2: Achieved number of successful gRPC calls per second vs Semaphore value. EMR configuration, table partitions, and paragraphs are kept constant.

rent calls to the service we utilized Semaphore. Semaphore is a class in the AsyncIO library¹¹ that implements an internal counter (set by user) to limit the number of concurrent requests as described by Dijkstra, 1968. Number of concurrent requests running in each core can never exceed the maximum Semaphore counter value.

To identify the maximum Semaphore value that successfully scales up the number of Kubernetes pods without errors, we conducted tests. Results of the experiments are shown in Table 2.

With the Semaphore value set to 50, PySpark ran successfully and significantly increased the load set by the client to the server. Table 3 summarizes all libraries and tools used to increase the number of calls per second.

3.4 Errors during gRPC calls

As gRPC async calls were made to the service, errors were returned. The most common gRPC response status code exceptions¹² encountered were: *cancelled*, *unavailable*, and *deadline exceeded*. We expected to receive errors when the service was scaling to process the received requests. When an error was received by a running PySpark client, the running job would terminate. Thus, we were unable to produce inference results without a solution that handles errors and keeps running the Spark job.

A Circuit Breaker (Nygard, 2018) was implemented to prevent clients from overwhelming services and a gRPC Interceptor was implemented to wait for services to be available and retry failing calls. Table 4 shows the total number of requests made and the number of errors that were handled by the circuit breaker.

⁹https://grpc.github.io/grpc/python/ grpc_asyncio.html ¹⁰https://spark.apache.org/docs/latest/

[&]quot;https://spark.apache.org/docs/latest/ api/python/user_guide/arrow_pandas.html

¹¹https://docs.python.org/3/library

¹²https://grpc.github.io/grpc/core/md_ doc_statuscodes.html

Tool	Definition	Description
Async IO	Python Library	Write concurrent requests with coroutines
gRPC AsyncIO	Python Library	GRPC client that works asynchronously
PyArrow with Pandas	Data Format and Python	PySpark's tabular data format to pass to
	Library	UDF as a Pandas table
Semaphore	Class in Async IO	Limits number of running requests
Async Circuit Breaker	Asynchronous Design Pat-	Resends client calls on failure
	tern	

Table 3: All libraries, design patterns, and data formats imported to PySpark job to accelerate inference speed.

Code	Status	Notes	Amount
0	ОК	Returned on success	97.84M
1	CANCELLED	The operation was cancelled, typically by the caller	666
4	DEADLINE_EXCEEDED	The time expired before the operation was complete	469.58K
14	UNAVAILABLE	The service is currently unavailable	27.90K

Table 4: Number of status codes returned in gRPC responses for the entire batch job.

3.5 Asynchronous circuit breaker

A circuit breaker is a software design pattern that was implemented to detect and act upon response failures received by the PySpark client. As discussed in Nygard, 2018, in a closed state the circuit passes through and all gRPC calls are being made. If a number of consecutive failures are received, the circuit opens and subsequent request attempts return a failure immediately. After a time period, the circuit switches to a half-open state to test if the underlying problem still exists. If a call fails in this half-open state, the breaker is once again tripped. When a call finally succeeds, the circuit breaker resets back to the default closed state.

The circuit breaker implementation was taken from an open-source library.¹³ Modifications were made to support AsyncIO, so calls running through it are sent concurrently. The state of the circuit breaker is shared across requests that use the the same gRPC client. To open or close the circuit, the circuit breaker only considers the deadline exceeded, unavailable, and cancelled gRPC status codes. Other errors are directly returned to the client.

Finally, a gRPC Interceptor uses this circuit breaker to block requests until the circuit breaker is closed again and retry each request up to 4 times, after which the data point is skipped and the batch job continues. The interceptor gets attached to the gRPC channel on creation. This design pat-

¹³https://github.com/fabfuel/

tern allows clients to not overwhelm services with requests and halts our batch job as the service deployment scales up.

4 Results

All steps in Section 3 improve the batch job speed and results in satisfactory performance. The data pipeline is able to produce inference results for more than 95 million paragraphs in around 96 hours with an inference speed of around 300 requests per second. The semaphore value is set to 50.

4.1 Daily runs of the batch job

Once the backfill data is stored, the data pipeline runs daily to find new and updated paragraphs from our S3 datasets. Everyday, around 600,000 (varies daily) paragraphs need to have their inference values stored. The graph in Figure 2 illustrates the typical daily run for the pipeline. It shows that it takes about 40 minutes for the Kubernetes micro-service pods to fully scale up. We limited the maximum number of pods for daily jobs to 100.

4.2 Analytics platform

Inference results were stored in an AWS S3 bucket. This dataset was registered in a AWS Glue Data Catalog.¹⁴ Amazon Athena¹⁵ is a query service that made it possible to run SQL queries on this dataset. Redash¹⁶ is a cloud-based analytics dashboard that we used to visualize insights from the inference

circuitbreaker

¹⁴https://aws.amazon.com/glue

¹⁵https://aws.amazon.com/athena

¹⁶https://redash.io

results. In includes a SQL client that makes calls to Amazon Athena and displays the query results. Redash was connected to Amazon Athena as a data source, which enabled us to perform queries to all tables registered in AWS Glue.

5 Conclusion

This paper discussed a successful machine learning architecture for both online and offline inference that centralizes models as services. We present solutions that use concurrency to increase the inference speed of offline batch jobs in Apache Spark. Because of this, the majority of resources are still assigned to these services, and the batch job resources grow at a much smaller rate in comparison.

We used a resource-intensive language model for emotion classification, where we demonstrated how proper tuning of TensorFlow Serving and Kubernetes can improve the service's performance. We also showed that by parallelizing the calls made to the service in PySpark, we can significantly improve inference speed.

Finally, results were presented that provide useful insights into the inference performance. Together, all these components resulted in a satisfactory architecture, which resulted in the emotion probabilities of 95 million paragraphs to be stored within 96 hours. We hope the architecture can be applied to other language tasks or machine learning models.

References

- Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265– 283, Savannah, GA. USENIX Association.
- Omar Y. Al-Jarrah, Paul D. Yoo, Sami Muhaidat, George K. Karagiannidis, and Kamal Taha. 2015. Efficient machine learning for big data: A review. *Big Data Research*, 2(3):87–93.
- J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*.

- Edsger W Dijkstra. 1968. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer.
- A. Gómez, Esperanza Albacete, Y. Sáez, and P. I. Viñuela. 2014. A scalable machine learning online service for big data real-time analysis. 2014 IEEE Symposium on Computational Intelligence in Big Data (CIBD), pages 1–8.
- Apache Hadoop. 2006. Apache hadoop. *http://hadoop.apache.org*.
- Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.
- Abhish Ijari. 2017. The study of the large scale twitter on machine learning. *International Research Journal* of Engineering and Technology (IRJET), 4:247–251.
- Chen Liu, Muhammad Osama, and Anderson De Andrade. 2019. Dens: a dataset for multi-class emotion analysis. *Proceedings of the EMNLP Conference*.
- Michael T Nygard. 2018. *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf.
- Cesare Pautasso, Erik Wilde, and Rosa Alarcon. 2013. *REST: advanced research topics and practical applications*. Springer.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.
- Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. 2012. Fast and interactive analytics over hadoop data with spark. *Usenix Login*, 37(4):45–51.
- Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.