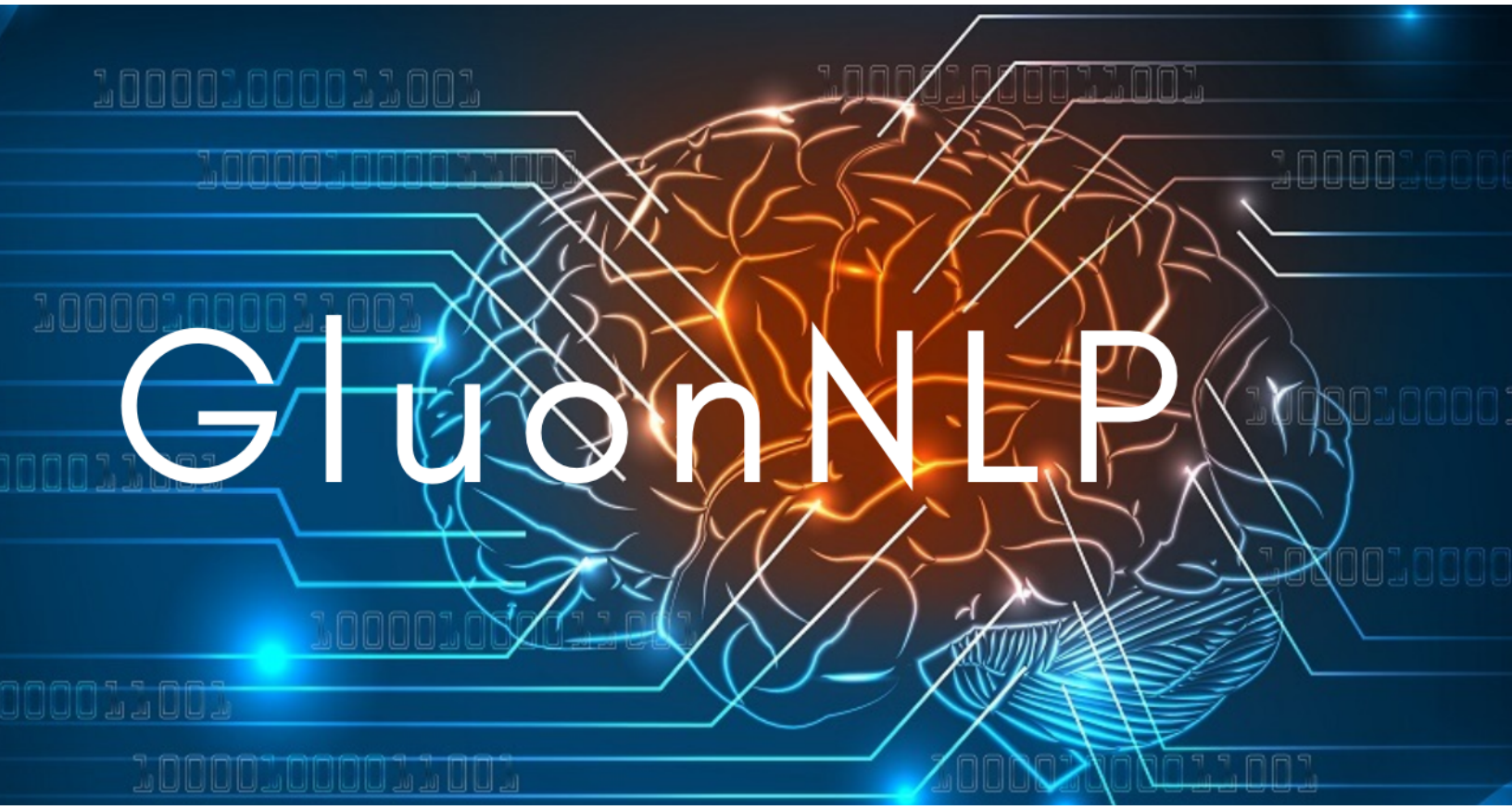
CS 410 Final Project

Gluon NLP with MXNet on AWS Sagemaker

Training for Yelp Sentiment Analysis



Text Information Systems

Fall 2020

Ryan Rickerts (ryanjr3)

https://github.com/theRocket/CourseProject

CS 410 Final Project

Gluon NLP with MXNet on AWS Sagemaker

Yelp Sentiment Analysis

Motivation

State-of-the-art deep learning models in natural language processing (NLP) are fascinating to read about and try to understand for a graduate-level computer science student. It is also possible for a hobbyist or student to obtain pre-trained models from these research breakthroughs and perform inference using the meager hardware at their disposal while working and studying from home (i.e. no access to lab clusters at an institution).

Perhaps a deeper and more satisfying approach for investigating this multi-layered architecture with broad areas of application is to train a dataset by one's self, much like raising your own puppy to learn preferred tricks. However, time and compute resources can be an intimidating constraint in these compute-intensive algorithms.

For this project, I aimed to find a relatively cutting edge approach in NLP where the training can be replicated by a hobbyist developer. The Gluon Project aims to meet this need. According to this blog post:

> ✘ Symptom: Natural language processing papers are difficult to reproduce. The quality of open source implementations available on Github varies a lot, and maintainers can stop supporting the projects.
> √ GluonNLP prescription: Reproduction of latest research results. Frequent updates of the reproduction code, which comes with training scripts, hyper-parameters, runtime logs etc.[1]

A broad description from the GluonNLP website is as follows:

[1] https://medium.com/apache-mxnet/gluonnlp-deep-learning-toolkit-for-natural-language-processing-98e684131c8a

> GluonNLP provides implementations of the state-of-the-art (SOTA) deep learning models in NLP, and build blocks for text data pipelines and models. It is designed for engineers, researchers, and students to fast prototype research ideas and products based on these models.[2]

From their model zoo, we selected the Text Classification example, which trains the FastText classification model on the Yelp review dataset. This is a binary classification dataset (positive vs. negative sentiment), and we aim to replicate their validation accuracy score of 94% in a reasonable time frame and cost.

For optimum training times, we turned to NVIDIA Tesla V100 GPUs available on Amazon Web Services (or AWS) Sagemaker instances. These are available on-demand and can be powered up just for training times to keep costs at a minimum. We investigate adapting the FastText scripts provided by the Gluon project to the AWS environment, such as loading required Jupyter kernels and dependent libraries for this algorithm, reading and writing data from S3 buckets, and of course keeping compute regions and permissions (or roles) in good order.

[2] https://nlp.gluon.ai/#about-gluonnlp

Fast-text Word N-gram

This model is a slight variation of the one published by the Facebook AI Research team in 2016 in a paper called "Bag of tricks for efficient text classification" by Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. The aim of this approach, which they call fastText or a "library for efficient text classification and representation learning," is also to increase training speed over the dominant deep learning algorithms of the day. The abstract states:

> Our experiments show that our fast text classifier fastText is often on par with deep learning classifiers in terms of accuracy, and many orders of magnitude faster for training and evaluation. We can train fastText on more than one billion words in less than ten minutes using a standard multicore CPU, and classify half a million sentences among ~312K classes in less than a minute.[3]

The Gluon-provided code used in this project, found at text_classification/fasttext_word_ngram.py in the repo, does not reference any code from the fastText project. It uses only gluonnlp and mxnet python libraries, so it appears to be a complete rewrite of the approach. It is modified here to meet the requirements of running on AWS infrastructure, which we cover in the next section, and becomes text_classification/fasttext_word_ngram_aws.py in the Github repo.

Apache MXNet is an open-source deep learning framework that is well supported on AWS and is rivaled by TensorFlow and PyTorch in popularity and performance on NLP. The advantage of MXNet (short for mix net) is the broad language support of Python, R, C++, Julia, and Scala. It also scales well:

> "Another advantage is that the models built using MXNet are portable such that they can fit in small amounts of memory. So, once your model is trained and tested, it can be easily deployed to mobile devices or connected systems. MXNets are scalable to be

[3] https://arxiv.org/abs/1607.01759

used on multiple machines and GPU simultaneously. This is why Amazon has chosen this framework for its deep learning web services."[4]

In this code, the Gluon authors create two classes:

- a MeanPoolingLayer — a block for mean pooling of encoder features — which takes a gluon.HybridBlock as a parameter.
- a FastTextClassificationModel—the trained embeddings layer—which takes a HybridBlock as a parameter.

The rest of the logic centers around argument parsing to the main function, the creation of n-grams (defaults to 1 and recommended <=2 for large datasets), data loading, preprocessing, labeling, and evaluating accuracy of the training against a test dataset.

Scripts to fetch the the training and test datasets from Google Drive are hosted at fastText's Github repo[5], but these proved problematic given the response time of the downloads. I manually fetched the Yelp Review Polarity set only, then modified the code to normalize the text. This modified bash script is at text_classification/data_fetch.sh in the project repo, and the CSV files are placed in the data/ sub-directory.

Once this was prepared, we can start running the python script locally with the recommended parameters as follows:

```
python fasttext_word_ngram.py --input data/yelp_review_polarity.train \
                              --output data/yelp_review_polarity.gluon \
                              --validation data/yelp_review_polarity.test \
                              --ngrams 1 --epochs 10 --lr 0.1 --emsize 100
```

This set of parameters is notably missing the '`–gpu=0`' flag for the local run only. This is because my laptop, a 2019 MacBook Pro with a 6-Core Intel i7 CPU, has a AMD Radeon Pro 5300M graphics card that does not implement CUDA architecture. This job will run on the CPU instead, and while threading does not appear to be implemented, according to MXNet documentation:

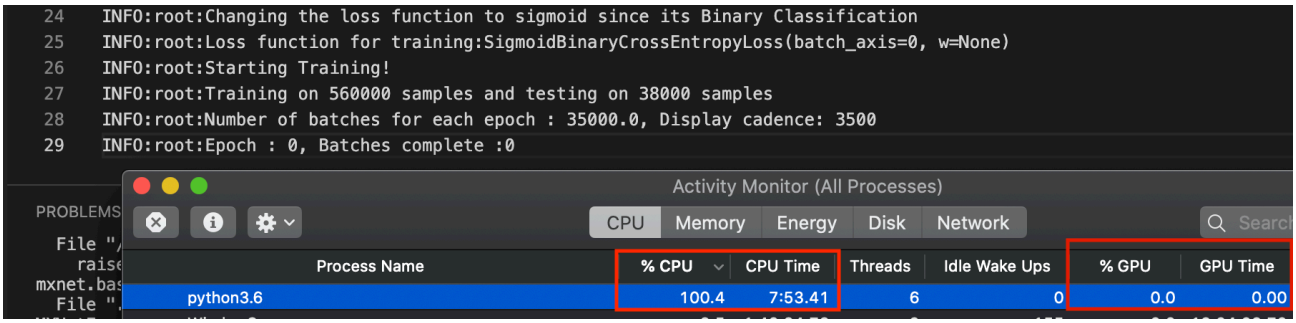
---

[4] https://analyticsindiamag.com/mxnet-tutorial-complete-guide-with-hands-on-implementation-of-deep-learning-framework/

[5] https://github.com/facebookresearch/fastText/blob/master/classification-example.sh

> "For a CPU, MXNet will allocate data on main memory, and try to use all CPU cores as possible, even if there is more than one CPU socket."[6]

The Activity Monitor on the laptop shows the CPU being pegged at 100% and no use of the GPU:



With 35000 batches configured to run on 464402 vocabulary words and 560000 sentences, the first attempt never even completed a single epoch. I quit the job after 8 hours of processing time and only 17000 batches completed. So now we move this workload to the GPUs hosted on the cloud!

AWS Sagemaker and NVIDIA Tesla V100 GPUs

I implement this training job using high-end P3 AWS Sagemaker instances.[7] According to the table of instance sizes listed at the bottom of the above linked page, the cheapest instance offered - `p3.2xlarge` - provides one Tesla V100 GPU with 16GB of GPU memory for $3.07/hr on demand. A helpful reference were these published benchmarks of NVIDIA GPUs provided in TensorFLOPs, which are units of floating-point arithmetic performance aimed at NVIDIA GPU hardware called Tensor Cores:

> "A new, specialized Tensor Core unit was introduced with "Volta" generation GPUs. It combines a multiply of two FP16 units (into a full precision product) with a FP32 accumulate operation—the exact operations used in Deep Learning Training

[6] https://mxnet.apache.org/versions/1.7.0/api/python/docs/tutorials/getting-started/crash-course/6-use_gpus.html

[7] https://aws.amazon.com/ec2/instance-types/p3/

> computation. NVIDIA is now measuring GPUs with Tensor Cores by a new deep learning performance metric: a new unit called TensorTFLOPS."[8]

According to that metric, the Tesla V100 GPU rates around 112-125 TensorTFLOPS (exact figure depending on the use of PCI-Express or SXM2 SKU interfaces). For comparison, the maximum known deep learning performance at any precision of the Tesla K80 is 5.6 TFLOPS for FP32. This GPU is provided on the P2 Sagemaker instances, and for 1 GPU on the `p2.xlarge` instance size, the cost is $0.90/hr. If we can attain a 20x performance increase on our training job for approx. 4x compute resource cost, that seems like a great win! The clock time used by the `p3.2xlarge` instance and costs incurred for several training runs is discussed in the Results sub-section below.

Data Workflow

The first step on AWS Services is to select a region for all our work, in this case US East (Ohio) or us-east-2. This is important for several reasons: one, the data transferred between Amazon S3 buckets and Amazon EC2 instances in the same Region and account is relatively straightforward and free, and our training data sizes are significant (where I live in a remote area, it took 30 minutes to upload due to low bandwidth). Two, the cost of the P3 instances (also significant) are determined by region, and we want to accurately estimate our costs for this workload to avoid a surprise bill. Furthermore, I had to request access to these highly specialized machines on my account via a support ticket, and their staff response time was not immediate. On the first attempt, they enabled a different region than I had uploaded the training data, so I had to sync it across to another S3 bucket, now called sagemaker-cs410-finalproj and pictured below.

Sagemaker has an S3 Uploader library available for outputting the model parameters (a local file save is first required first) and returning the training results —net.params— to the S3 bucket under a yelp_sentiment_polarity.gluon directory, as pictured above. These parameters will be needed for later inference use cases outside this JupyterNotebook instance, so they must offloaded for access.

---

[8] https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/

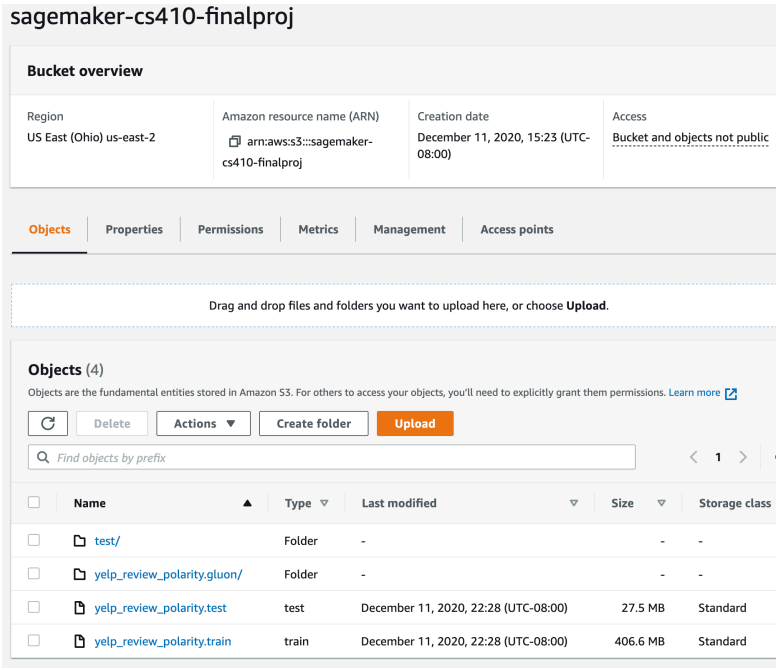
```
def save_model(net, output_file):
    file_name = "net.params" # local version
    net.save_parameters(file_name)
    S3Uploader.upload(file_name,output_file)
```

Since we established this bucket in the same region, the S3 Uploader utility only requires the bucket name to perform this work. That variable was configured in the first JupyterNotebook cell, with:

```
bucket = 'sagemaker-cs410-finalproj'
```

Although you can also use a default bucket for the session, which is initialized like this:

```
bucket = Session().default_bucket()
```



For streaming lines from the data is S3 to the data parsing method, we used a library called **smart_open**[9] which simulates reading in data from a local file in an iterable function such as a for-in loop, like:

```
for line in open(filename):
   tokens = line.split(',', 1)
   labels.append(tokens[0].strip())
   data.append(tokens[1].strip())
 return labels, data
```

[9] https://pypi.org/project/smart-open/

This proved very effective and the data was parsed quickly into labels and normalized text data for the training run.

Training Results

The one Tesla V100 GPU performing text classification on our `p3.2xlarge` instance exceeded expectations and churned out training and test results for 10 Epochs in under 20 mins. Furthermore, the same accuracy of 94% was achieved as mentioned on the Gluon page where we sourced this code. SGD performed slightly better than Adam as an optimizer, and this is discussed in the **Further Work** section below.

Jupyter notebook run 1 with `adam` **as optimizer, 10 Epochs**

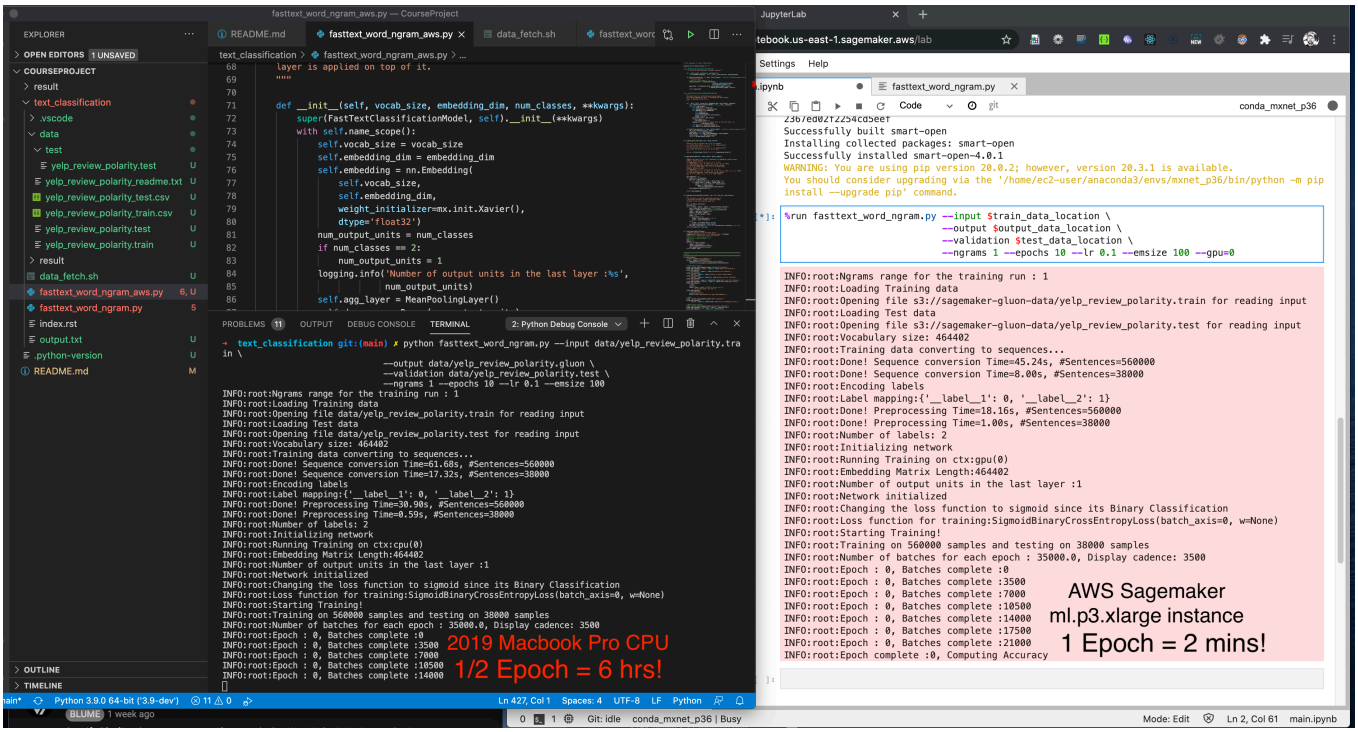
- Highest Test Accuracy: 0.9401578947368421 (Epoch 8)
- Final Test Accuracy: 0.939921052631579, Test Loss: 0.17803387705344548 (Epoch 10)

Jupyter notebook run 2 with `sgd` **as optimizer, 10 Epochs**

- Highest Test Accuracy: 0.9403157894736842 (Epoch 8)
- Final Test Accuracy: 0.9400526315789474, Test Loss: 0.17815197125596924 (Epoch 10)

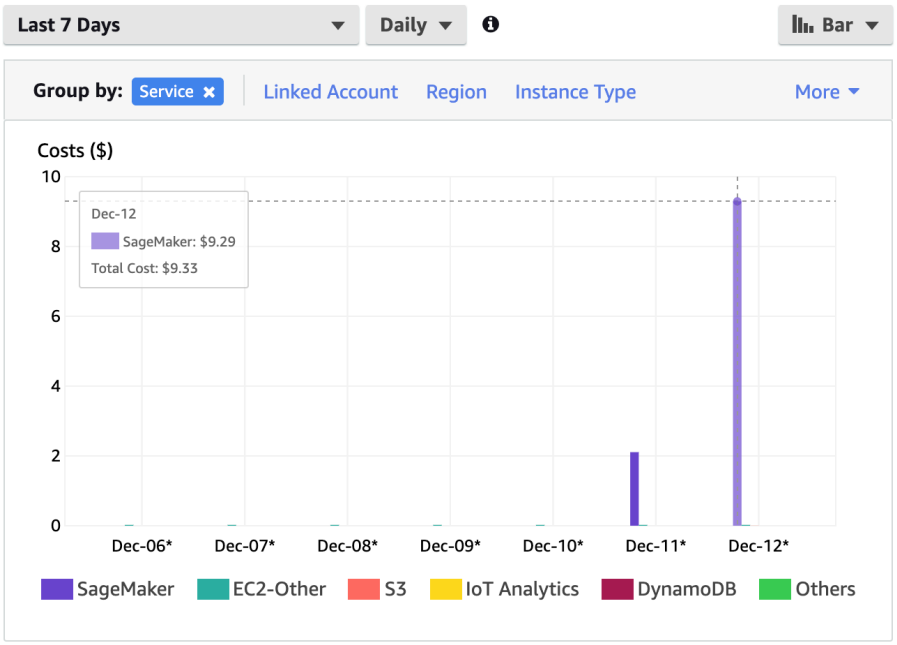
Jupyter notebook run 3 with `sgd` **as optimizer, 25 Epochs**

- Highest Test Accuracy: 0.9403157894736842 (Epoch 8)
- Final Test Accuracy: 0.9397105263157894, Test Loss: 0.17758843273002864 (Epoch 25)
- Note, the accuracy above was reached at Epoch 18 and remained stable.



To reiterate, my MacBook Pro CPU (`ctx=cpu(0)`) never completed a single epoch after 8 hours of run time, and the V100 GPU (`ctx=gpu(0)`) finished one epoch in less than 2 minutes! The entire job run of 10 epochs required only 17 minutes total, which is very satisfying.

The costs incurred for several hours of experimenting with the AWS SageMaker JupyterNotebook environment, plus the three training runs above (and a few failed ones when parameter output to file was not configured correctly) remained under $10. So we were able to capture reliable and conclusive training work at the expected cost of around $3/hr! The Cost Explorer returns this data analysis by service typically the next day.



Importantly, I shut off my P3 EC2 instance whenever I was not using it, and I deleted other experimental instances, such as P2, when I moved onto another solution. Not performing this cleanup task is a painful way to discover an exorbitant bill from AWS at the end of the month!

Further Work

1) The Gluon team notes that they simplified the model somewhat for training speed as follows:

> "We have added dropout to the final layer, and the optimizer is changed from 'sgd' to 'adam' These are made for demo purposes and we can get very good numbers with

> original settings too, but a complete async sgd with batch size = 1, might be very slow for training using a GPU."[10]

I was able to pass 'sgd' (or Stochastic Gradient Descent) as an optimizer parameter and get slightly better testing accuracy over 'adam' (0.94032 vs. .94016) with no noticeable decrease in training speed. I did not remove the dropout nor change the batch size, so these could be restored to the original fastText algorithm for more benchmarking on this particular GPU instance.

2) Rather than firing up JupyterLab in our P3 instance and copy/pasting in the custom algorithm for our training job, Amazon SageMaker can interact with a Docker container based on a pre-built image. This would be a better practice for continued use of this approach, since it can easily be deployed into different regions and scaled up or down on different EC2 instances for more or less GPU power.[11]

This workflow is covered in more detail here, which was also an early inspiration for this project (although they use GluonCV):

> **Deploying custom models built with Gluon and Apache MXNet on Amazon SageMaker**
> "When you build models with the Apache MXNet deep learning framework, you can take advantage of the expansive model zoo provided by GluonCV to quickly train state-of-the-art computer vision algorithms for image and video processing. A typical development environment for training consists of a Jupyter notebook hosted on a compute instance configured by the operating data scientist. To make sure this environment is replicated during use in production, the environment is wrapped inside a Docker container, which is launched and scaled according to the expected load."[12]

I felt great enjoyment in this project and learned a lot about NLP using AWS Sagemaker GPUs, and look forward to experimenting more with Gluon/MXNet.

---

[10] https://nlp.gluon.ai/model_zoo/text_classification/index.html

[11] https://docs.aws.amazon.com/sagemaker/latest/dg/your-algorithms-training-algo.html

[12] https://aws.amazon.com/blogs/machine-learning/deploying-custom-models-built-with-gluon-and-apache-mxnet-on-amazon-sagemaker/