AWS Machine Learning Blog

Text Classification with Gluon on Amazon SageMaker and AWS Batch

by Matt Krzus and Jason Berkowitz | on 30 MAR 2018 | in Apache MXNet On AWS, SageMaker | Permalink | Comments | Share

Our customer had a problem: The manual classification of warranty claims was causing a bottleneck. These claims were based on a text field that explained the event in short detail. An example of that text looked something like this: “The plutonium-fueled nuclear reactor overheated on a hot day in Arizona’s recent inclement weather. Burn damage to the flux capacitor. It will not time travel without replacing.” The classification of the claim might be something like “fire.”

A worker in the claims department had to read a batch of warranty claims, usually in the thousands, and manually classify each example with the proper warranty category to make sure that the submitter classified the claim correctly.

The company wanted to speed up the pipeline to minimize the time it took them to resolve the claims. This was central to keeping their customers satisfied. Because the company has such a large and extensive database of past warranty claims, turning toward a supervised learning solution was a logical next step.

This blog post outlines a solution for the classification of the sentiment of a given text that helped accelerate our customer’s workflow by bridging the gap between the non-technical claims team and the more solutions-focused data science team. For this example scenario, we will train our classification model using MXNet’s Gluon and Amazon SageMaker, then build an application based on AWS Lambda to process large batches of text using elastic resources on AWS Batch.

Before you begin: You’ll get the most out of this blog post if you have a familiarity with Python, the AWS Command Line Interface (CLI), and the AWS SDK for Python Boto3.

The Data

For the demo in this blog post, we use a subset of the IMDb Large Movie Review Dataset that has already been separated into a test and training set. The following is an example of the comma delimited text and the sentiment classification of that text:

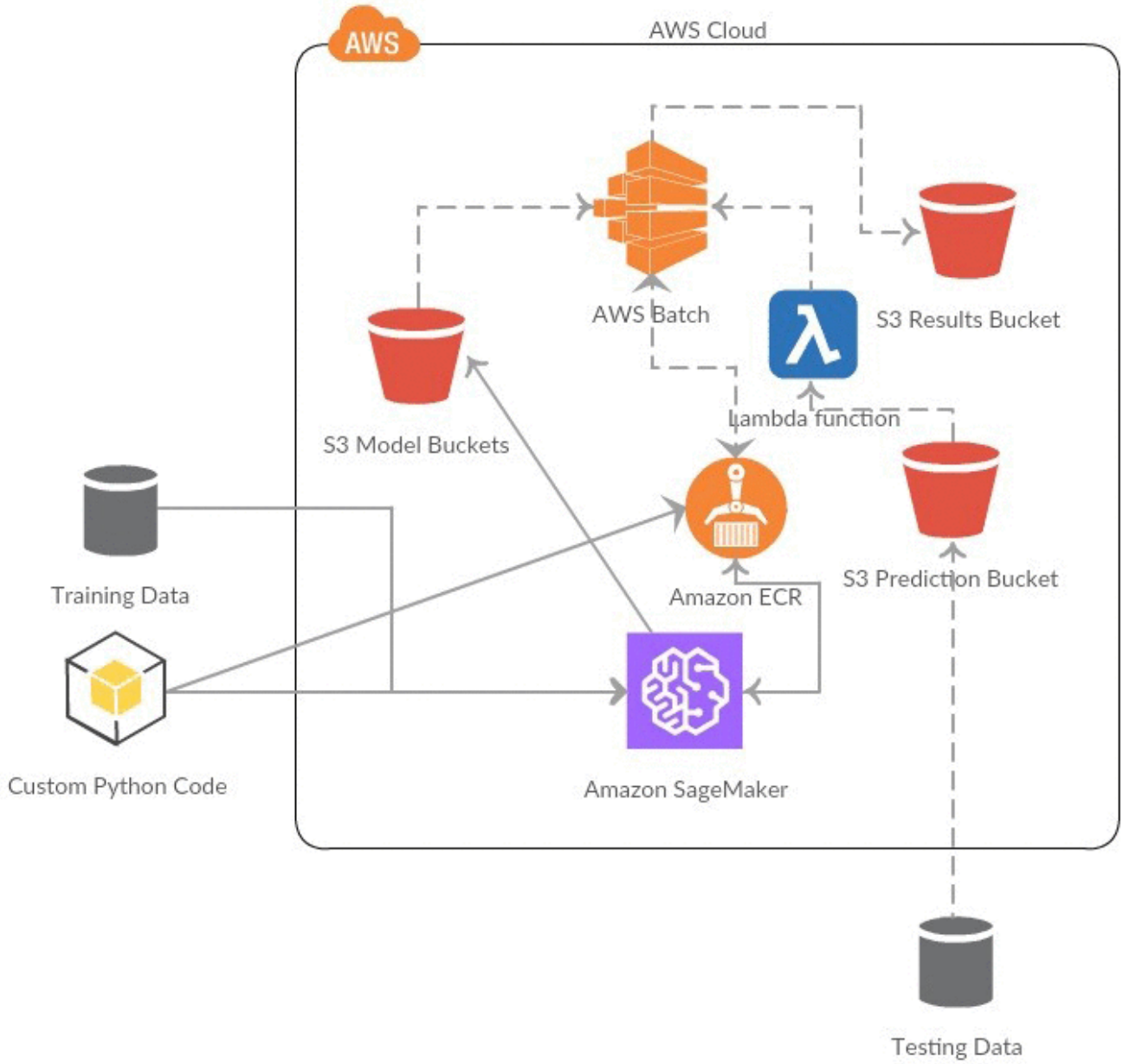
sentiment: 1, *text*: “Have you ever in your life, gone out for a sports activity, tried your best, and then found yourself in an important segment of it, where for a brief moment, you were given a chance to be a hero and a champion and . . . failed? I believe many of us have had that moment in our lives. This is the premise of the movie, \”The Best of Times.\” In this story a middle age banker, named Jack Dundee (Robin Williams) suffers from the deep melancholy of a football mistake, which happened years ago, is inspired to re-play the game . . again. In order to accomplish this he must convince the once great football quarterback, Reno Hightower (Kurt Russell) to make a comeback. For Reno, who is satisfied with his present lot in life, sees no need to change the past record, which gets better as he ages. Added to both their problems is the fact years have passed and in addition, both their marriages are floundering and in need of re-vamping. Not easy when his Father-in-law (Donald Moffat) habitually reminds him of the biggest drop. Nevertheless, Dundee is persistent and will do anything to try and correct the greatest blunder of his life. Great fun for anyone wishing to enjoy their youth again.”

The data files for this demo can be found here: test.csv and train.csv. [NOTE: You can either include the dataset within the GitHub repo or make a public Amazon S3 bucket. Whatever works]

System overview

We first create all the static pieces of our architecture: the security policies, the Amazon S3 buckets, the compute environment and job queue, the AWS Lambda functions, and the AWS Lambda triggers. We then place our training and prediction code into a Docker container that we’ll upload to Amazon Elastic Container Registry (Amazon ECR). After we have a working container, we can use Amazon SageMaker to quickly train our classification model.

When data is uploaded to the Predict Bucket, a trigger instructs the Predict Job Lambda function to use our container’s prediction script on the data we’ve just uploaded to Amazon S3. This job makes predictions on the event source data by using the trained parameters from the Amazon SageMaker task.



Building the Pipeline

Step 0: Prerequisites

Make sure you have a **bash**-enabled machine and install **Git**, **Docker**, and the **AWS CLI**.

If you haven't installed Sagemaker already, install it using the following command:

```
pip install sagemaker
```

Step 1: Download and unzip the repository from the Amazon S3 bucket

```
wget https://s3.amazonaws.com/aws-ml-blog/artifacts/text-classification-gluon/gluon
```

Step 2: Dockerize our code and upload the image to Amazon ECR

Because of the custom tie-in to AWS Batch, we chose to build a customer Amazon SageMaker image instead of using a pre-built tie in. As we did in the previous blog: Deep Learning on AWS Batch, we will be using Docker to create an image that Amazon SageMaker and our compute environment will utilize. Our Dockerfile will be built upon the mxnet/python:gpu Docker repository with a few added packages. The import line to note is the second RUN command, where we will use the spacy package to download pre-trained word embeddings to help speed up our training. We will add three scripts to help manage the training and prediction jobs: `train` (a python executable that SageMaker will utilize), `predict.py`, and `utils.py`. The `utils.py` manages the transfer of various files, defines the language model in Gluon, and creates the structure to encode incoming text. The `train` file will process our training data, train a language model on that data, and then upload the model's parameters and text encoder to Amazon S3. The `predict.py` script will download each respective parameter previously created, load the language model's weights, and then will make predictions on the file we want to analyze (the `test.csv` file).

```
Dockerfile


    FROM mxnet/python:gpu

    RUN pip2 install numpy scipy sklearn pandas awscli boto3 spacy -U
    RUN python2 -m spacy download en_core_web_md

    # add these files for batch prediction
    ADD mxnet_model/utils.py /usr/local/bin/utils.py
    ADD mxnet_model/predict.py /usr/local/bin/predict.py
    ADD mxnet_model/train /usr/local/bin/train.py

    ENV PYTHONUNBUFFERED=TRUE
    ENV PYTHONDONTWRITEBYTECODE=TRUE
    ENV PATH="/opt/program:${PATH}"

    COPY mxnet_model /opt/program
    WORKDIR /opt/program
```

In the event that our `train` file isn't an executable yet (it should be), we'll run this snippet just to be safe:

```
chmod +x train
```

To push our Docker image to Amazon ECR enter the following command.

```
cd container && build_and_push.sh languagemodel && cd ..
```

You will need the Amazon Repository URI of the image you have just created. Paste the URI into your favorite text editor so you don't forget it. (If do you forget the Repository URI, you can find it within the Amazon ECS console under "Repositories."). It will look something like this: *{account-id}.dkr.ecr.{region}.amazonaws.com/*languagemodel

Step 3*:* Create our AWS Lambda functions and push them to Amazon S3

Here we will assign an AWS Lambda trigger function to a specific Amazon S3 bucket. The act of placing a file within a given bucket will trigger the specific Lambda function. In our case, when data is placed in the prediction Amazon S3 bucket, the trigger will activate our Lambda function, which in turn will trigger the job that will make predictions on our data.

Before we can create an Amazon S3 bucket and assign triggers, we need to first upload our Lambda function's code to an Amazon S3 bucket of our choosing. We need to choose a name for our Lambda bucket. I used the name 'mxnet-lambda.' (You will need this name later, so copy and paste it into the text editor that has your URI information.) To create our Lambda function, it has to first have a .zip file to build itself upon (which in turn needs a place to be stored, for example, in an S3 bucket). By uploading the .zip file to Amazon S3 first, we can ensure no that dependency issues might arise within our AWS CloudFormation stack.

To do this, run the following command:

```
cd lambda-function && zip lambda_function.zip lambda_function.py

# mxnet-lambda is the name of the bucket we will create
    # mxnet_batch_lambda/lambda_function.zip is the name of the key
    # 1. create bucket with a unique name - I used mxnet-lambda

aws s3api create-bucket --bucket mxnet-lambda --region us-east-1

# 2. copy lambda_function.zip to our new bucket

aws s3 cp lambda_function.zip s3://mxnet-lambda/mxnet_batch_lambda/lambda_function.
    cd .. && cd ..
```

For similar dependency issues that might arise, we will not create the actual trigger until **Step 5**.

Step 4: Create our AWS architecture

This step will build an Amazon Virtual Private Cloud (Amazon VPC), create the necessary Amazon S3 buckets, assign the proper AWS Identity and Access Management (IAM) roles to our AWS Cloud resources, and ultimately build our AWS Batch environment. To start this step, we need to know the following:

- Remember the repository URI from **Step 2**; something like this:
    - *{account-id}.dkr.ecr.{region}.amazonaws.com/languagemodel*
- Remember the bucket we created in **Step 3** to store our Lambda function

    - *mxnet-lambda*
- Lastly, choose a unique prefix that will be appended to the front of every bucket created. I chose *my-mxnet-demo* so that upon creation, the various objects created look something like this:
    - *my-mxnet-demo-predict-bucket, my-mxnet-demo-transformer-bucket, etc.*

Using this information, we can now programmatically create our architecture using AWS CloudFormation and the AWS CLI:

```
cd cloudformation-templates &&
aws cloudformation create-stack
--stack-name mxnet-batch --template-body file://create.yaml --parameters ParameterK
```

In your AWS Management Console, navigate to the CloudFormation console and make sure the Stack Name *mxnet-batch* is running smoothly. Upon its completion (10-20 minutes), we need to create our job-queue. (NOTE: A job queue can only be created after a compute environment has been created and validated.) Make sure you're in the CloudFormation-templates folder and run the following command:

```
aws cloudformation create-stack --stack-name mxnet-batch-jobqueue --template-body f
```

Step 5: Creating AWS Lambda triggers

To create our AWS Lambda triggers all we need is our unique bucket prefix from **Step 4**:

```
cd lambda_triggers && bash launch_triggers.sh {MY_UNIQUE_PREFIX} && cd ..
```

This script ensures that our lambda function will launch when a new object is dropped into our prediction-bucket.

Step 6: Define our training configuration and push it to Amazon S3

To properly train the language model we defined in the training script, we need to set the training configurations we want for our model. This configuration (found in the *config* folder) will include things like the number of epochs to train for as well as architecture- and optimization-specific information. This file will be uploaded to the '*MY-UNIQUE-NAME-model-config'* bucket defined within our CloudFormation template.

The following is the training configuration I used. If you wish to experiment with different parameters please feel free to do so. The only necessary change you have to make is to ensure that the parameters *MODELPARAMSURL*, *TRANSFORMERURL*, and *MODELCONFIGURL* have been changed to fit your unique prefix from Step 4.

```
{    "n_classes" : 2,
```

```
    "num_epoch" : 5,
    "optimizer" : "adam",
    "seq_len" : 500,
    "vocab_size" : 5000,
    "batch_size" : 64,
    "hidden_size" : 500,
    "embed_size" : 300,
    "dropout" : 0.2,
    "n_layers" : 1
```

The following is the config description:

n_classes is the total number of class types that you want the language model to predict. The IMDb dataset is binary with only 2 classes—a positive review and a negative review.

num_epoch is the total number of training epochs the language model will train for, where one epoch is the number of times the model sees the entire training set.

optimizer is the learning algorithm you want to use, See http://mxnet.incubator.apache.org/api/python/optimization/optimization.html for available options.

seq_len stands for sequence length. This parameter is the maximum number of words in a given sentence. If a sentence does not have this length, it is padded with zeros. If the sentence is longer than this length, it is cut off. This is used to encode your training and test data. Note: This cannot be reset after training because it builds a model specific to this parameter.

vocab_size is the maximum number of most common words to use. If a word is less common, that is, if it appears fewer times than the most popular words, it will simply be ignored. The way to think about this is that we are constraining the vocabulary that exists to the *N* most popular words, everything else will be dismissed. Decreasing this parameter will make the model faster at the expense of accuracy. However, there is a point of diminishing returns.

batch_size is a parameter that controls how many sentences the model will train on at a given time. Increasing or decreasing this parameter is an optimization problem that is beyond the scope of this demo. However, if you want to get into the weeds, see this discussion: https://www.quora.com/Intuitively-how-does-mini-batch-size-affect-the-performance-of-stochastic-gradient-descent.

cnn parameter controls whether you want to use a convolutional network based language model or the LSTM based language model.

use_spacy_pretrained dictates whether or not to use spacy's pretrained word embeddings to initialize the network.

embed_size if you chose not to use spacy's pretrained word embeddings, you can set the size of the embedding you'd like to use. This is defaulted to 300 (spacy's size).

hidden_size controls how large a given layer in your LSTM will be (if you choose the LSTM option). While the size choice is specific to the problem at hand (sentiment analysis on the IMDb set), increasing or decreasing this number should be explored for your own dataset.

dropout is a regularization parameter that helps stop the model from overfitting. The recommended values are between .1 and .5 (note the value cannot exceed 1.0).

n_layers controls the number of recurrent layers within the LSTM (if you choose the LSTM option). Much like hidden size, this is dataset dependent.

bidirectional is a fun parameter that controls whether or not the recurrent function within the LSTM goes forward over the sequence as well as backward. It is currently set to True.

train_log_interval is the parameter that displays information about the loss and the perplexity of the given training batch. It will be displayed every *N* batches. This parameter is where you set that *N*.

val_log_interval is the parameter that displays information about the loss, perplexity, and accuracy of the validation set. It will be displayed every *N* batches. This parameter is where you set that *N*.

learning_rate is the decreasing function of time that controls the rate of learning of your optimization algorithm. The learning rate is also dataset dependent and should be played with for your own dataset.

save_name controls what the model's weights will be saved as with S3.

MODELPARAMSURL is the S3 bucket where the model's weights will be saved.

TRANSFORMERURL is the S3 bucket where the text transformer will be saved. The text transformer is important for ensuring that any incoming text will be constrained to the vocabulary of the training data set.

MODELCONFIGURL is the S3 bucket where the prediction task will pull the configuration file.

epoch_factor is starting point at which the learning rate will start to decay.

power_decay is the decision to simply decay the learning rate by the below decay function. by using this strategy it defaults to the current learning rate * 90.

```
decay = math.floor((epoch) / config['epoch_factor'])
    trainer.set_learning_rate(trainer.learning_rate * math.pow(0.1, decay))
```

wd stands for weight decay (L2 regularization coefficient) and is currently set to 0.0002. This modifies the objective function by adding a penalty for having large weights.

kernel_size controls the dimensions of the convolutional window if you choose the CNN option.

Now upload the config file to our Amazon S3 config-bucket.

```
aws s3 cp config/config.json s3://MY-UNIQUE-PREFIX-FROM-STEP-4-config-bucket/config
```

Step 7: Train our model

To begin training the model, we have provided two options: a simple script and a Jupyter Notebook. If you want to use the Python script, you can enter the following into the command line from the root directory of our code:

```
python train_sagemaker_model.py
```

You should see an output similar to the one that follows:

```
INFO:sagemaker:Creating training-job with name: lstm-2018-01-10-00-58-54-783
.....................................................................
[INFO] Starting the training process
[INFO] Loading data
[INFO] Initializing transformer
[INFO] Cleaning dataset
[INFO] Creating word-count index
[INFO] Creating training data
[INFO] Building CNN
[INFO] Loading pre-trained embedding matrix
[INFO] Uploading transformer to s3
[INFO] Beginning training!
[Epoch 1 Batch 5 Validation] time cost 24.45s, validation loss 0.73214,    validation perplexity 2.07953, accuracy 0.50160
...
...
[Epoch 5 Batch 590 Train] loss 0.04168, perplexity 1.04256
[Epoch 5 Batch 600 Train] loss 0.03622, perplexity 1.03688
[Epoch 5 Batch 610 Train] loss 0.04411, perplexity 1.04510
[Epoch 5 Batch 620 Train] loss 0.04113, perplexity 1.04199
[Epoch 5 Batch 624 Validation] time cost 101.81s, validation loss 0.23055,    validation perplexity 1.25929, accuracy 0.90976
[New Best Validation Accuracy] 0.90976 !!!!!!!
[Epoch 5 Validation] time cost 101.82s, validation loss 0.23055,    validation perplexity 1.25929, accuracy 0.90976
[Best Validation Accuracy] 0.90976
[INFO] Uploading best model parameters to s3
[INFO] Training complete.
===== Job Complete =====

In [7]:
```

Jupyter Notebook Option: First launch the Jupyter Notebook app using the command line from the root directory of our code:

```
jupyter notebook
```

In the Notebook Dashboard, navigate to the file named '*train_sagemaker_model.ipynb*' and click it. From here, you can either run the document cell-by-cell by pressing *Shift-Enter* or you can run the entire notebook by clicking *Cell* in the navigation pane and selecting *Run All.*

Upon completion of the training code, if the Best Validation Accuracy looks acceptable, then it's time to use our batch processer.

Step 8: Trigger our prediction Lambda function

We can now use the power of batch to process our data whenever we upload new content to our prediction bucket. To trigger our lambda function, upload test.csv into our prediction bucket using the following code:

```
aws s3 cp data/test.csv s3://MY-UNIQUE-PREFIX-FROM-STEP-4-predict-bucket/test.csv
```

We can log into our AWS Batch console to confirm RUNNING or SUCCEEDED and if you want, you can click the link from the Batch console to the job's STDERR/SDTOUT information in Amazon CloudWatch Logs and check on the progress.

Step 9: Download your predictions for Amazon S3

```
aws s3 cp s3:// MY-UNIQUE-PREFIX-FROM-STEP-4-batch-results/test_output.csv  test_ou
```

Conclusion

Using AWS-powered systems such as the one from this demo can give engineers the freedom to develop, test, and deploy code at scale while having the freedom of AWS managing their overhead. By using Amazon SageMaker to manage the training process on your behalf, a scientist can begin to build and orchestrate very complex systems for his or her company. Going further and combining Amazon SageMaker with large batch processes on AWS Batch, one can accelerate bottlenecks in any given workflow.

About the author

Matt Krzus is a Data Scientist with AWS Professional Services. He comes from a background of Machine Learning and Computer Vision on low power edge devices. He is currently working on the IoT and Analytics team within Professional Services.

Jason Berkowitz is a Big Data Practice Manager with a background in data science with AWS Professional Services. He comes from a background in Machine Learning, Data Lake Architectures and Enterprise Analytics Roadmaps. He is currently working helping customers shape their data lakes and analytic journeys on AWS within Professional Services.

Related Posts

- Model Server for Apache MXNet v1.0 released
- Using deep learning on AWS to lower property damage losses from natural disasters
- Pixm takes on phishing attacks with deep learning using Apache MXNet on AWS
- Curalate makes social sell with AI using Apache MXNet on AWS
- The importance of hyperparameter tuning for scaling deep learning training to multiple GPUs
- Model Server for Apache MXNet adds support for serving Gluon models
- Apache MXNet (incubating) adds support for Keras 2
- Use pre-trained models with Apache MXNet