Deploying a Sentiment Analysis Model on SageMaker

Author: Md Hafizur Rahman

Introduction:

The project Deploying a Sentiment Analysis by Udacity inc. deploys a recurrent neural network onto AWS for the purpose of determining the sentiment of a movie review using the IMDB data set. The notebook and Python files provided here create a simple web app which interacts with a recurrent neural network once deployed performing sentiment analysis on movie reviews. This project assumes some familiarity with SageMaker, the IMDB Sentiment Analysis using XGBoost and RNNs.

Instructions:

This step will involve setting up an AWS IAM role with the AmazonSageMakerFullAccess IAM policy attached. Once created navigate the notebook to the SageMaker Project.ipynb.

Once the notebook is open we can simply run the notebook. The notebook will do the following:

- Download or otherwise retrieve the data
- Process and Prepare the data
- Upload the processed data to S3
- Build and Train the PyTorch Model
- Test the trained model (typically using a batch transform job).
- Deploy the trained model.
- Use the deployed model.

- Deploying model on our web app.

Step 1. Download or otherwise retrieve the data:

We will be using the IMDb dataset. This dataset can be found and downloaded here Large Movie Review Dataset v1.0

Step 2. Process and Prepare the data:

We will be doing some initial data processing. To begin with, we will read in each of the reviews and combine them into a single input structure. Then, we will split the dataset into a training set and a testing set. Next, we make sure that any html tags that appear in the review are removed. In addition we tokenize our data input, that way words such as entertained and entertaining are considered the same with regard to sentiment analysis. The review_to_words method does 5 major things to prep for the vocabulary of a single review. The steps are shown below:

- `stemmer = PorterStemmer()`
- `text = BeautifulSoup(review, "html.parser").get_text() # Remove HTML tags`
- `text = re.sub(r"[^a-zA-Z0-9]", " ", text.lower()) # Convert to lower case`
- `words = text.split() # Split string into words`
- `words = [w for w in words if w not in stopwords.words("english")] # Remove stopwords`
- `words = [PorterStemmer().stem(w) for w in words] # stem`

Once the data is processed we pickle it into preprocessed_data.pkl.

For the model we are going to construct in this notebook we will construct a feature representation which is very similar to a bag of words feature representation. To start, we will represent each word as an integer. Of course, some of the words that appear in the reviews occur very infrequently and so likely don't contain much information for the purposes of sentiment analysis. The way we will deal with this problem is that we will fix the size of our working vocabulary and we will only include the words that appear most frequently. We will then combine all of the infrequent words into a single category and, in our case, we will label it as 1. Since we will be using a recurrent neural network, it will be convenient if the length of each review is the same. To do this, we will fix a size for our reviews and then pad short reviews with the category 'no word' (which we will label 0) and truncate long reviews.

To begin with, we need to construct a way to map words that appear in the reviews to integers. Here we fix the size of our vocabulary (including the 'no word' ('0') and 'infrequent' ('1') categories) to be 5000.

After creating the tokenized dictionary, word_dict, we notice the five most frequent tokenized words in the training set are movi, film, one, like, and time. It does make sense that these word would appear the most often because they are words that would be found in movie reviews but some really don't give great sentiment information. For this model we decided to just let it slide but if we wanted to improve the models success we might remove the top 2 from the dictionary because they could be over saturating the model.

Next, we pickle the word_dict so we can us it in our future AWS Lambda function when turning a review into a integer reprasentation. Then we create a function that will be used to truncate the training reviews to a set padding value. we started with 500 as my reviews fixed length.

The methods build_dict and preprocess_data are used to get the top 4998 words from the review dataset to reduce the processing speed but this also reduces accuracy of the model we're designing because of the loss of data when removing the other less frequent words.

The method convert_and_pad_data is used to convert the the review words representation into the bag of words representation. If a 0 no word found in that space. If a 1 is found, it is an infrequent words, so a word not in our top 4998. If

from 2-5000 if found, it is a frequent word. The lower the number, like 2, the more frequent the word is in the dataset. The higher the number, like 5000, the less frequent the word is int he dataset, but still frequent since its in the top 4998 of words in the dataset.

Step 3. Upload the processed data to S3:

We start this step by saving the traing dataset to train.csv.

```
pd.concat([pd.DataFrame(train_y), pd.DataFrame(train_X_len),
pd.DataFrame(train_X)], axis=1) \        .to_csv(os.path.join(data_dir,
'train.csv'), header=False, index=False)
```

Next, we need to upload the training data to the SageMaker default S3 bucket so that we can provide access to it while training our model.

```
import sagemakersagemaker_session = sagemaker.Session()bucket =
sagemaker_session.default_bucket()prefix = 'sagemaker/sentiment_rnn'role =
sagemaker.get_execution_role()

input_data = sagemaker_session.upload_data(path=data_dir, bucket=bucket,
key_prefix=prefix)
```

Step 4. Build and Train the PyTorch Model:

In the XGBoost notebook we discussed what a model is in the SageMaker framework. In particular, a model comprises three objects

- Model Artifacts,
- Training Code, and
- Inference Code,

each of which interact with one another. In the XGBoost example we used training and inference code that was provided by Amazon. Here we will still be using containers provided by Amazon with the added benefit of being able to include our own custom code.

We will start by implementing our own neural network in PyTorch along with a training script. For the purposes of this project we have provided the necessary

model object in the model.py file, inside of the train folder. A model comprises three objects

- Model Artifacts,
- Training Code, and
- Inference Code,

each of which interact with one another. We use training and inference code that is provided by Amazon. Here we will be using containers provided by Amazon with the added benefit of being able to include our own custom code. The train/model.py looks is defined as follows:

```
!pygmentize train/model.py
```

```
import torch.nn as nn

class LSTMClassifier(nn.Module):
    """
    This is the simple RNN model we will be using to perform Sentiment Analysis.
    """

    def __init__(self, embedding_dim, hidden_dim, vocab_size):
        """
        Initialize the model by settingg up the various layers.
        """
        super(LSTMClassifier, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)
        self.dense = nn.Linear(in_features=hidden_dim, out_features=1)
        self.sig = nn.Sigmoid()

        self.word_dict = None

    def forward(self, x):
        """
        Perform a forward pass of our model on some input.
        """
        x = x.t()
        lengths = x[0,:]
        reviews = x[1:,:]
        embeds = self.embedding(reviews)
        lstm_out, _ = self.lstm(embeds)
        out = self.dense(lstm_out)
        out = out[lengths - 1, range(len(lengths))]
        return self.sig(out.squeeze())
```

The important takeaway from the implementation provided is that there are three parameters that we may wish to tweak to improve the performance of our

model. These are the embedding dimension, the hidden dimension and the size of the vocabulary. We have made these parameters configurable in the training script so that if we wish to modify them we do not need to modify the script itself. We wrote some of the training code in the notebook so that we can more easily diagnose any issues that arise.

First we load a small portion of the training data set to use as a sample. It would be very time consuming to try and train the model completely in the notebook as we do not have access to a gpu and the compute instance that we are using is not particularly powerful. However, we were able to work on a small bit of the data to get a feel for how our training script is behaving.

After setting a training dataloader, we wrote the training method. We made this training method as simple as possible just to get a feel for the dataloader, so all loading and parameter loading will be implemented later in the notebook. We test the training method with a small training set over 5 epoches just to make sure everything is working and on the right track.

In order to construct a PyTorch model using SageMaker we must provide SageMaker with a training script. We may optionally include a directory which will be copied to the container and from which our training code will be run. When the training container is executed it will check the uploaded directory (if there is one) for a requirements.txt file and install any required Python libraries, after which the training script will be run.

When a PyTorch model is constructed in SageMaker, an entry point must be specified. This is the Python file which will be executed when the model is trained. Inside of the train directory is a file called train.py, which contains the necessary code to train our model. The way that SageMaker passes hyperparameters to the training script is by way of arguments. These arguments can then be parsed and used in the training script. To see how this is done take a look at the provided train/train.py file.

```
from sagemaker.pytorch import PyTorch

estimator = PyTorch(entry_point="train.py",
                    source_dir="train",
                    role=role,
                    framework_version='0.4.0',
                    train_instance_count=1,
                    train_instance_type='ml.p2.xlarge',
                    hyperparameters={
                        'epochs': 20,
                        'hidden_dim': 200,
                    })
estimator.fit({'training': input_data})
```

Results: