This is my first attempt in learning how to train a simple Logistic Regression Model using Tensorflow. But first what are regression models? The simplest regression model is the linear regression model. It is defined as a linear approximation of a causal relationship between two or more variables. Suppose we want to predict the income for a random person given their years of education. Here years of education would be independent variable while income would be dependent variable. Given years of education, we can form a linear model which can predict income for any person. The model might look something like this:

**Income =B0+B1\*years of education**

Here B0 is the bias coefficient and B1 is the coefficient for the years of education. We use learning techniques to find good set of coefficients with the help of sample data sets. Once we find the best coefficients, we can use the model or equation to successfully predict income for any random person. However, we need to understand that linear regression assumes that the relationship between input (years of education) and output (income) is linear.

Now that we have basic idea of what linear regression is, it’s time to dive into logistic regression. Logistic regression is similar to linear regression, except logistic regression predicts whether something is True or False, instead of predicting something continuous like income. Given years of education, logistic regression model can predict whether somebody earns over 50,000$/year or not. Here the output is categorical. However, if we see logistic regression graph, we see a ‘S’ shaped curve which does not seem to be giving us a categorical result. This is because, it is giving us the probability. For different values of an independent variable (years of education), logistic regression model gives us corresponding probability of dependent variable (income > 50000) happening. We can introduce a threshold value (e.g 0.5) to make the classification easier. However, not every problem of classification will have just two classes. There is where softmax comes into existence. Softmax extends the idea of logistic regression into a multi-class world. Softmax assigns decimal probabilities to each class in a multi-class problem. Softmax is usually implemented through neural network layer just before the output layer.

Neural Network basically consists of: inputs, layer, and, neurons (output). In deep neural network there is at least 2 hidden layers. On basic level, all neuron does is sums up all the weighted inputs that it gets from the inputs or preceding layer. Then we use sigmoid activation function which will determine if the sum is enough to fire the neuron or not.

When it comes to training process, it can either be done in a single computer or using many computers in distributed manner. There are two major strategies. First one is model parallelism, in which the neural network itself gets split across multiple devices. This is used mainly in cases where the models consists of large number of parameters which would not fit on a single device. To solve this problem Tensorflow has a solution called Mesh. Second strategy, that we are about to dive deep into is data parallelism, in which models gets replicated in each worker node. Each replica needs to get updated about gradient calculated by the other workers. There are two ways the communication between different replicas takes place.However They are: synchronous training and asynchronous training. In sync training, all workers train over different slices of input data in sync, and aggregating gradients at each step. Until each and every worker node receives updated gradients from all the other worker node, each worker will wait. However, in async training all workers independently do training over the input data and update variables asynchronously. Typically, in Tensorflow, synchronous training is supported via all-reduce and asynchronous through parameter server architecture.

Asynchronous training typically achieves higher *throughput* (in terms of training data consumed per unit time) than synchronous training, because it never needs to block on another worker. Even if some worker node fails or are slow in their tasks, this model of training will work. However, whether this leads to a faster time to accuracy will depend on the model, and the performance of your network, because we will probably need to run a larger number of steps when doing asynchronous training to reach the same accuracy as a synchronous training process. However, many people have observed that, with modern GPU clusters and a fast interconnection, it is possible to make synchronous training run acceptably fast, and hence better to use that (for some problems) than asynchronous training.

We will use MultiWorkerMirroredStrategy to implement synchronous training of our model in multiple machines. After calculating gradients, each model’s parameters are aggregated and then synchronized to all devices using allReduce algorithm, hence the name ‘mirrored’.

Simplest way to configure Tensorflow is through TF\_CONFIG. It is a JSON string that contains two keys: cluster and task. The former defines the topology and also includes addresses of all worker nodes. It is identical among all the nodes. However, latter one is unique for each node and it describes their respective roles in the cluster. Once tf parses the JSON string, it starts GRPC servers based on the configuration string. Since we are using gcp, TF\_CONFIG is set automatically for us.

Steps involved in synchronous training are:

1. Packaging the python code:

First we need to get the .py file from the jupyter notebook. Then we need to put the file inside a new folder or python package named ‘trainer’. Aside from task.py, we also need to make an empty \_\_init\_\_.py file.

Graphical user interface, text, application, chat or text message

Description automatically generated

Here is the task.py that will train the model.

#!/usr/bin/env python

# coding: utf-8

import os

import json

import tensorflow as tf

from tensorflow.keras import layers

from matplotlib import pyplot as plt

per\_worker\_batch\_size = 64

tf\_config = json.loads(os.environ['TF\_CONFIG'])

num\_workers = len(tf\_config['cluster']['worker'])

(x\_train, y\_train),(x\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

x\_train=tf.keras.utils.normalize(x\_train, axis=1)

x\_test=tf.keras.utils.normalize(x\_test, axis=1)

strategy=tf.distribute.experimental.MultiWorkerMirroredStrategy()

with strategy.scope():

model=tf.keras.models.Sequential()

model.add(layers.Flatten())

model.add(layers.Dense(128, activation=tf.nn.relu))

model.add(layers.Dense(128, activation=tf.nn.relu))

model.add(layers.Dense(10, activation=tf.nn.softmax))

model.compile(optimizer='SGD',loss='sparse\_categorical\_crossentropy',metrics=['accuracy'])

model.fit(x\_train, y\_train, batch\_size=per\_worker\_batch\_size, epochs=3)

#Testing the model's accuracy

val\_loss, val\_acc=model.evaluate(x\_test, y\_test)

print(val\_loss, val\_acc)

1. Creating a configuration file: config.yaml. This will basically specify all machines configurations to train our model.

trainingInput:

runtimeVersion: "2.3"

pythonVersion: "3.7"

scaleTier: CUSTOM

masterType: n1-standard-4

workerType: n1-standard-4

workerCount: 2

scheduling:

maxWaitTime: 3600s

maxRunningTime: 7200s

1. Submitting training job to the cloud: Our original batch\_size was set to 64. Following is the result for it. It took about 10 minutes to train with accuracy reaching upto 88%.

##Submitting our training job to gcp

gcloud ml-engine jobs submit training mnist\_recognition2 --package-path=./trainer --module-name trainer.task --staging-bucket gs://pawan-bucket-9999 --config=config.yaml --region=us-east1

Table

Description automatically generated

Now, we decrease the batch\_size to 1 and see the change in accuracy and time taken to train of our model. It look much longer time (>35 minutes) but the accuracy improved.

Table

Description automatically generated with medium confidence

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Now let’s move on to Asynchronous training of our model. The key idea here is the necessity of a central server called Parameter Server. Each of the worker does not need to wait for one another. However, it will need to communicate with Parameter Server (PS) after it calculates the gradient. The central updated gradients in the PS will be used by each of the worker before beginning their new iteration. A detail description of how the PS works alongside each worker is given below:

1. The worker **reads** all of the shared model parameters in parallel from the PS task(s), and copies them to the worker task. These reads are uncoordinated with any concurrent writes, and no locks are acquired: in particular the worker may see partial updates from one or more other workers (e.g. a subset of the updates from another worker may have been applied, or a subset of the elements in a variable may have been updated).
2. The worker **computes** gradients locally, based on a batch of input data and the parameter values that it read in step 1.
3. The worker **sends** the gradients for each variable to the appropriate PS task, and **applies** the gradients to their respective variable, using an update rule that is determined by the optimization algorithm which is sgd in our case. The update rules typically use (approximately) commutative operations, so they may be applied independently on the updates from each worker, and the state of each variable will be a running aggregate of the sequence of updates received.

Asynchronous distributed training in TF involves a "cluster" with several "jobs", and each of the jobs may have one or more "tasks". When using parameter server training, it is recommended to have one coordinator job (which has the job name chief), multiple worker jobs (job name worker), and multiple parameter server jobs (job name ps). While the coordinator creates resources, dispatches training tasks, writes checkpoints, and deals with task failures, workers and parameter servers run [tf.distribute.Server](https://www.tensorflow.org/api_docs/python/tf/distribute/Server) that listen for requests from the coordinator.

The config file for the cluster we are about to create is provided below:

trainingInput:

runtimeVersion: "2.3"

pythonVersion: "3.7"

scaleTier: CUSTOM

masterType: n1-standard-4

workerType: n1-standard-4

parameterServerType: n1-standard-4

evaluatorType: n1-standard-4

workerCount: 9

parameterServerCount: 3

evaluatorCount: 1

##Submitting our asynchronous training job to gcp

gcloud ml-engine jobs submit training mnist\_recognition\_async8 --package-path=./trainerAsync --module-name trainerAsync.task --staging-bucket gs://pawan-bucket-9999 --config=configAsync.yaml --region=us-east1

After lots of tries, I was still unable to successfully train the model using this method. I think my codes for parameter server and coordinator didn’t work as I expected and therefore I could not finish the project.