Air Quality Monitor Data Prediction using Keras Multi GPU Model and Performance Comparison on Variable HyperParameters

1. Problem Description:

Predicting air quality at a specific site at a specific moment is definitely a demanding and challenging task. As, the training time heavily depends on the huge amount of monitored air pollution data, introduction of parallel processing is a must for this task. So, in this project, to improve training time of Air Quality System prediction, study of several parallel algorithm techniques have been done and Keras, a popular deep learning framework written on top of python, has been considered as the main experimental framework. With the recent commit and release of Keras 2.0.9, a new model named "*multi_gpu_model*" has been introduced, which has made the training of deep neural network really easy on multi gpu setup[5]. In this report, we have experimented the promise of multi_gpu_model from Keras for this dataset and some other different network configurations also. Result has distinctly shown how batch size affects the performance of multi_gpu_model training time.

2. Data Set Description:

To demonstrate the efficacy and efficiency of our new method, we explored daily PM2.5 data sets for comparison with the results from Tong et al.[6]. The data set was air pollution data from the EPA's AQS (Air Quality System) which provided 146,125 PM2.5 measurements collected at 955 monitoring sites on all 365 days of the year 2009. The dataset contains six columns namely site_id, year, month, day, longitude, latitude and the pm2.5 values.

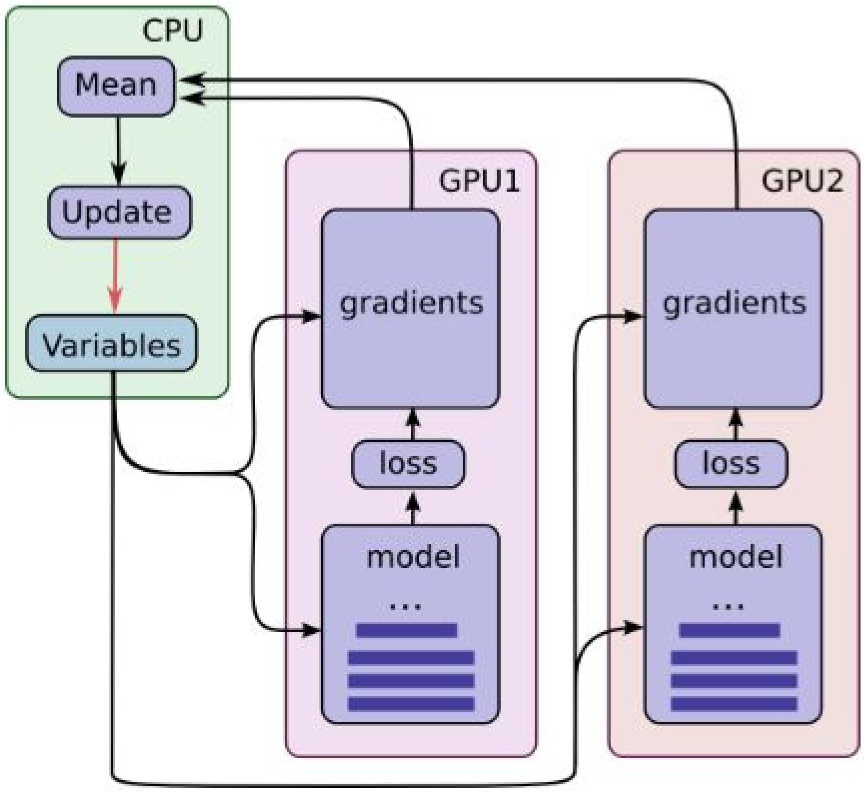
3.1. Keras multi_gpu_model:

As mentioned in the problem description, the library used for this experiment is Keras multi_gpu_model. The driving motive behind choosing this library is definitely the simplicity of model implementation. Keras multi_gpu_model replicates a model on different GPUs. Specifically, this function implements single-machine multi-GPU data parallelism. It works in the following way: Divide the model's input(s) into multiple sub-batches. Apply a model copy on each sub-batch. Every model copy is executed on a dedicated GPU. Concatenate the results (on CPU) into one big batch.

E.g. if your batch_size is 64 and you use gpus=2, then we will divide the input into 2 sub-batches of 32 samples, process each sub-batch on one GPU, then return the full batch of 64 processed samples. This induces quasi-linear speedup on up to 8 GPUs.[1][2]

3.2. Keras multi_gpu_model Architecture:

Here's the model architecture of Keras multi_gpu_model. The following figure clearly demonstrates how Keras multi_gpu_model handles gradient computation and model parameter update between cpu and gpus as explained in the previous section..



4. Methodology

4.1. Hardware:

Two NVIDIA Geforce GTX 1080 Ti gpus were used for the experiment.

4.2. Environment Setup:

For using Keras multi_gpu_model in this experiment we needed to setup the GPU version of Tensorflow. Anaconda was used as the package manager for this task.

4.3. Data Preprocessing:

The dataset contains six columns namely site_id, year, month, day, longitude, latitude and the pm2.5 values. $PM_{2.5}$ values were used as the label of the data and all the other columns except year were feeded as training feature. Year wasn't counted as it was providing a constant(2009) value. All the data were normalized feature wise. Then dataset was splitted on 8:2 ratio as training and testing data for avoiding overfitting in the model.

4.4. Experiments

Experiments been conducted in several stages. Different hyperparameters on the neural network was tuned for achieving the best accuracy and speedup on multi gpu. Even different depth and different types of Neural Networks were implemented to compare multi gpu performance on all these different configurations.

4.4.1. Experiment 1: Experiment with constant configuration model

For the first experiment the following configurations and Neural Network was used,

Model Hyperparameters

Batch Size = 512
Number of Epochs = 200
Dataset Size = 0.14 million(146126)
Neural Network Depth = 3

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 32)	192
dense_2 (Dense)	(None, 16)	528
dense_3 (Dense)	(None, 1)	17

This model was run several times and the mean of training time was plotted. The result was on the favour of the model used on cpu only. As the batch size was comparatively small. And, surprisingly in this experiment and on the other experiments also gpu1 took more time than gpu0. Here is the time taken by each configuration in the figure below.

Epochs	BatchSize	Time(sec)	Config.
200	512	94.74	gpu/0
200	512	128.97	gpu/1
200	512	96.6	gpu/0+gpu/1
200	***512***	***67***	***cpu0***

Figure: Training time taken for 200 epochs with 512 batch size in different configurations

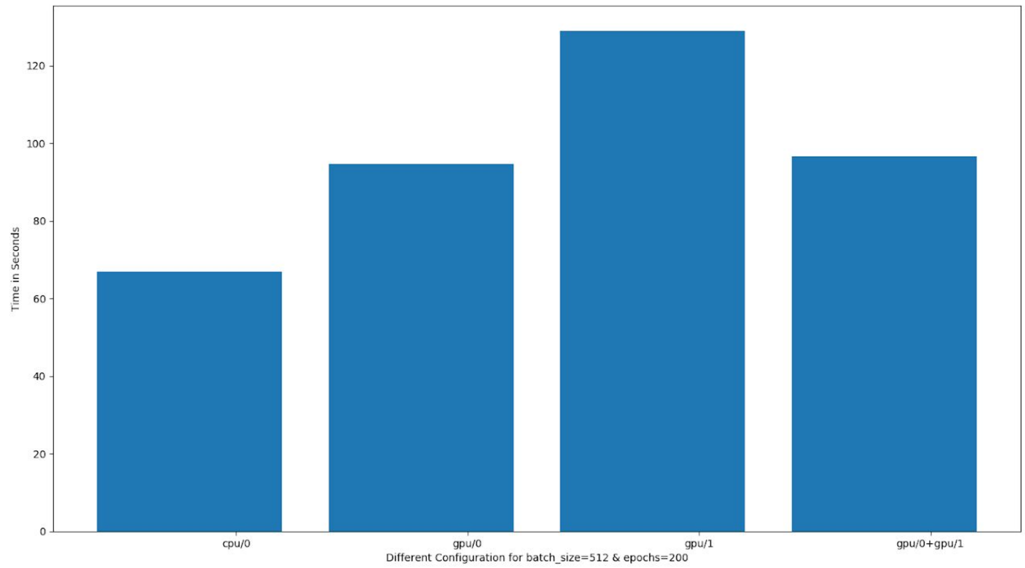


Figure: Training time taken for 200 epochs with 512 batch size and 3 depth Neural Network. In the x axis of the graph from left cpu0, gpu0, gpu1 and gpu0+gpu1

4.4.2. <u>Experiment 2: Experiment with variable depth of Neural Network</u>

For the second experiment below was the model configuration. The only variable thing was the depth of the neural network. We tried with 3, 6, 9 and 12 depth neural network with 64 nodes in each layer.

Model Hyperparameters
Batch Size = 512
Number of Epochs = 200
Dataset Size = 0.14 million(146126)
Neural Network Depth = 3, 6, 9, 12

The result still wasn't on favor of multi_gpu_model. Training time for cpu0 and gpu0 were still less than multi gpu(gpu0+gpu1), which definitely was not expected . Though the gpu usage increased a bit on higher depth configuration.

4.4.3. <u>Experiment 3: Training with variable number of epochs</u>

On the third experiment following was the model configuration. This experiment was done with different number of epochs with a bit larger batch size(2048 compared to 512) than the first experiment.

Model Hyperparameters
Batch Size = 2048
Number of Epochs = 500, 1000, 1500 and 2000
Dataset Size = 0.14 million(146126)
Neural Network Depth = 6

gpu0 and cpu0 were clearly again winner in terms of model training time. The only impressive outcome came out from this experiment was multi gpu started performing better than both of the previous experiments, though still the performance was way less than the expected performance. So, a clue was definite, "*multi gpu training time depends on the batch size of training data*". Which inspired us to conduct the fourth experiment on variable batch size of training data.

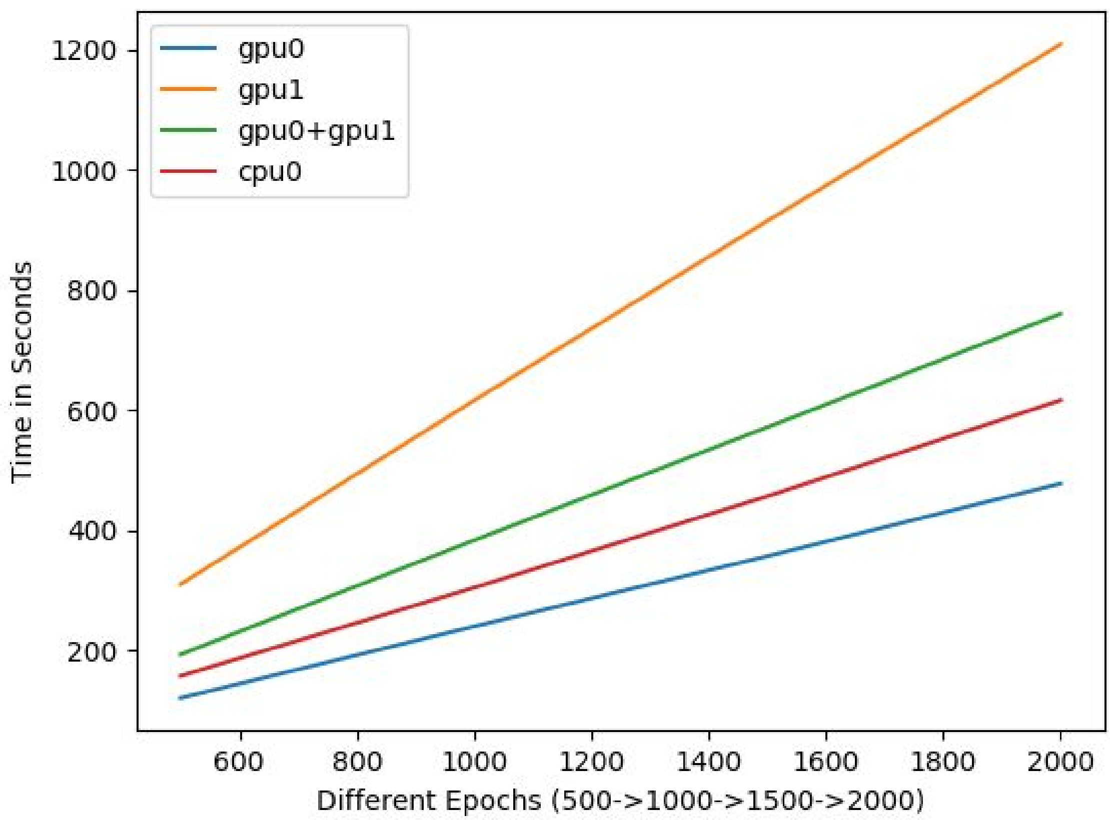


Figure: Training time plotted for different cpu, gpu combinations with different number of epochs

4.4.4. <u>Experiment 4: Training with variable batch size</u>

The third experiment brought up the possible configuration of getting better result in multi gpu model; variable length batch size. Different batch size of data was feed into the model starting from 512 upto 0.1 million. For that the data set size needed to have an increment also. The dataset size was 16 times larger than the original dataset for this experiment.

Model Hyperparameters

Batch Size = 512 to .1 million

Number of Epochs = 200

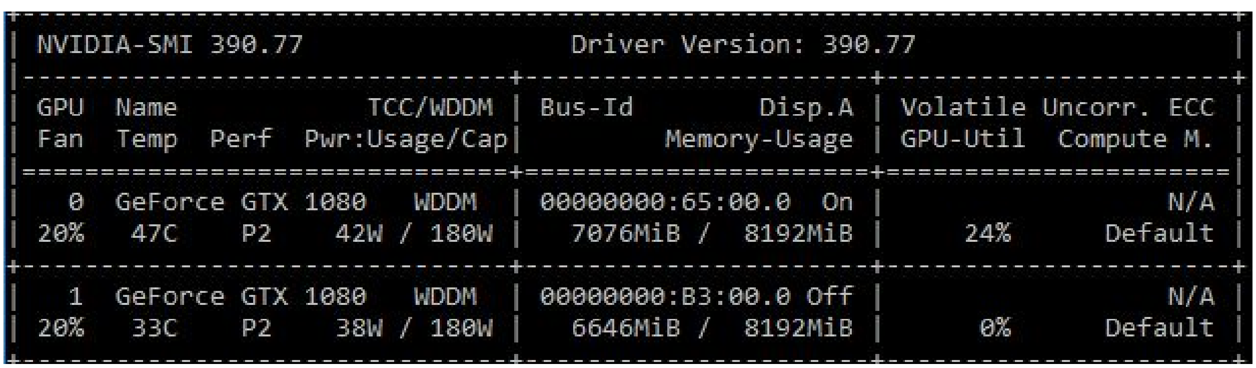
*Dataset Size = 0.14 million(146126) * 16*

Neural Network Depth = 6

The experiment took quite a handsome amount of time and the result came out with a good outcome. As predicted, for larger batch size the multi_gpu_model from Keras started performing better. After the batch_size became 10,000 the model started giving better result for multi gpu(gpu0 and gpu1 together). Though we couldn't reach the perfect speed up value of 2 for both gpu usage but still the result was convincing to say that, "For larger batch size gpu works better than cpu and multi gpu works better than single gpu. Because on larger batch size setup the time taken for gpus to do the computation on training data(*example: calculating gradient descent*) is much larger than time taken by cpu to do the communication tasks with gpus(*example: receiving updated gradient descents from gpus and resend the updated model to gpus*)".

5.1. <u>GPU Usage on different setup:</u>

For monitoring gpu performance on different configurations gpu stat was captured and the median value of data is shown below. For this purpose NVIDIA's built in driver toolkit nvidia-smi(System management interface) was used. It comes along with NVIDIA gpu library.



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.77                 Driver Version: 390.77                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080    WDDM | 00000000:65:00.0  On |                  N/A |
| 20%   47C    P2    42W / 180W |   7076MiB /  8192MiB |     24%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080    WDDM | 00000000:B3:00.0 Off |                  N/A |
| 20%   33C    P2    38W / 180W |   6646MiB /  8192MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

FIgure: GPU usage on both gpus while the model was run on gpu0 only. Obviously, gpu1 usage was 0% and gpu0 usage reached up to 24%-29%.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.77                 Driver Version: 390.77                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080   WDDM  | 00000000:65:00.0  On |                  N/A |
| 20%   50C    P2    42W / 180W |   7076MiB /  8192MiB |     16%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080   WDDM  | 00000000:B3:00.0 Off |                  N/A |
| 20%   35C    P2    39W / 180W |   6678MiB /  8192MiB |      6%      Default |
+-------------------------------+----------------------+----------------------+
```

Figure: Usage of multi gpu when model was run on both gpus. gpu1 usage is much less than gpu0 usage, because gpu0 gets priority while tasks are assigned. For less computational heavy tasks, gpu1 is used less than gpu0. And, gpu0 didn't reach more than 16-18%. The reason behind the low usage of gpus is the model complexity. Computationally heavy neural networks(example: Higher depth CNN, RNNs) utilized gpus better than this network model.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.77                 Driver Version: 390.77                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080   WDDM  | 00000000:65:00.0  On |                  N/A |
| 20%   51C    P2    42W / 180W |   6924MiB /  8192MiB |      9%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080   WDDM  | 00000000:B3:00.0 Off |                  N/A |
| 20%   39C    P2    33W / 180W |   6646MiB /  8192MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Figure: gpu usage when the model was run on cpu only. Surprisingly, cpu only model also used quite a decent amount of gpu0.

5.2. An interesting observation on GPU usage:

None of the above setup couldn't reach even 30% of gpu optimization. The reason came out after another rounds of experiment on different computational heavy neural networks. It came out neural networks those need more and more computation than basic feed forward neural networks require much more gpu usage. Below is the result of running Keras multi_gpu_model in a 12 depth of a CNN(Convolutional Neural Network) on MNIST data(Handwritten digit classification).

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.77                 Driver Version: 390.77                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080   WDDM  | 00000000:65:00.0  On |                  N/A |
| 20%   54C    P2    93W / 180W |   7196MiB /  8192MiB |     55%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080   WDDM  | 00000000:B3:00.0 Off |                  N/A |
| 20%   41C    P2    77W / 180W |   6798MiB /  8192MiB |     54%      Default |
+-------------------------------+----------------------+----------------------+
```

Figure: GPU utilization on higher depth CNN for MNIST digit classification task

6. <u>Accuracy:</u>

The accuracy was measured in terms of MAE(Mean Absolute Error). After hundreds of iteration, mean of the accuracy was taken. The mean value of MAE came out as 4.25.

7. <u>Future Works:</u>

- All the experiments were done on an environment having only 2 GTX-Geforce 1080 Ti gpu. Results may come out interesting if more gpus are feed for training the model. An interesting observation can be, from which batch size multi gpu model starts performing better can vary on higher gpu models.
- Besides Keras multi_gpu_model, the same task can be done on Tensorflow with manual assigning of models on gpu and cpu devices and manual computation of gradients on cpu and manual model update also. Another interesting observation maybe observing how Tensorflow handles this task and how is the gpu usage on that setup.
- Keras has a library dist keras and Tensorflow also has a popular library called Distributed Tensorflow for handling distributed training tasks. How the experimented models perform over there can be another crucial observation.

8. <u>Appendix:</u>

Here is a simple implementation code using Keras multi_gpu_model on EPA data to estimate air quality monitor.

```
*****************************************************************************************
from __future__ import absolute_import, division, print_function
import tensorflow as tf
import keras
from keras.utils import multi_gpu_model
import numpy as np
import pandas as pd
import numpy as np
import sklearn
import time
```

```
# specify the path
path =
"C:/Users/mr07520/PycharmProjects/HelloWorld/Data/pm25_2009_measured.csv"

# Load the data from local file into a dataframe
df = pd.read_csv(path)

# select input and output
Y = df['pm25'].values.reshape(df.shape[0], 1)  # select the label (correct output)
df = df.drop('pm25', 1)  # remove the label from input

dataset = df.values
X = dataset[:, 0:dataset.shape[1]]  # select features (input data)

# splitting the data into training and testing
from sklearn.model_selection import train_test_split

train_data, test_data, train_labels, test_labels = train_test_split(X, Y, test_size=0.2)  #
training to testing ratio is 0.8:0.2
print(train_data[0])

# Shuffle the training set
order = np.argsort(np.random.random(train_labels.shape[0]))
train_data = train_data[order]
train_labels = train_labels[order]

print("Training set: {}".format(train_data.shape))
print("Testing set:  {}".format(test_data.shape))

# Test data is *not* used when calculating the mean and std

print("before ", train_data[0])

from sklearn.preprocessing import StandardScaler

# Normalization
mean = train_data.mean(axis=0)
std = train_data.std(axis=0)
train_data = (train_data - mean) / std
test_data = (test_data - mean) / std
#
print("after ",train_data[0])  # First training sample, normalized
print("train_data.shape[0] ", train_data.shape[0])
print("train_data.shape[1] ", train_data.shape[1])
```

```
# Model
def build_model():
    model = keras.Sequential([
        keras.layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(32, activation='relu'),
        keras.layers.Dense(16, activation='relu'),
        keras.layers.Dense(1)
    ])

    return model


with tf.device('/cpu:0'):
    model = build_model()
# For running in multi gpus
parallel_model = multi_gpu_model(model, gpus=2)

optimizer = keras.optimizers.RMSprop(lr=0.001)
parallel_model.compile(loss='mse',
              optimizer=optimizer,
              metrics=['mae'])

model.summary()

# Display training progress by printing a single dot for each completed epoch
class PrintDot(keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs):
    if epoch % 100 == 0: print('')
    print('.', end='')

EPOCHS = 100

start_time = time.time()
history = parallel_model.fit(train_data, train_labels, epochs=EPOCHS,
                    validation_split=0.2,
                    batch_size=2048,
                    callbacks=[PrintDot()])
print("Training Time ")
print("--- %s seconds ---" % (time.time() - start_time))


import matplotlib.pyplot as plt
```

```
def plot_history(history):
 plt.figure()
 plt.xlabel('Epoch')
 plt.ylabel('Mean Abs Error')
 plt.plot(history.epoch, np.array(history.history['mean_absolute_error']),
         label='Train Loss')
 plt.plot(history.epoch, np.array(history.history['val_mean_absolute_error']),
         label = 'Val loss')
 plt.legend()
 plt.ylim([0, 10])
 plt.show()

plot_history(history)

early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=20)

predict = parallel_model.predict(test_data)
print("Predicted Values")
for i in range(0,10):
    print(predict[i])

[loss, mae] = parallel_model.evaluate(test_data, test_labels, verbose=0)
print("Real Value")
for i in range(0,10):
    print(test_labels[i])

print("MAE ", mae)
print("Testing set Mean Abs Error: ${:7.2f}".format(mae))
```

****

9. Reference:

1. https://keras.io/
2. https://keras.io/utils/#multi_gpu_model
3. https://developer.nvidia.com/nvidia-system-management-interface
4. https://www.tensorflow.org/tutorials/keras/basic_classification
5. https://www.pyimagesearch.com/2017/10/30/how-to-multi-gpu-training-with-keras-python-and-deep-learning/
6. https://edg.epa.gov/metadata/catalog/main/home.page
7. https://escholarship.org/uc/item/4dw721gn