Neural Network Hyper Parameter Optimization (NHOP) Using Distributed TensorFlow

Srini Ananthakrishnan

Abstract— **Hyper parameter optimization of Neural Networks has one ultimate objective, that is to find a function that minimizes some expected loss over independent and identically distributed samples from a natural (grand truth) distribution. Learning algorithms produces this function through the optimization of a training criterion with respect to set of parameters. This paper describes a software framework in python that uses distributed TensorFlow [1] to optimize hyper parameters which are the bells and whistles of the learning algorithm. Random search [2] is used as default algorithm to search for these optimal parameters in hyperspace.**

***Keywords*— model selection, distributed tensorflow, hyperspace, random search, neural networks**

I. Introduction

The problem of identifying a good value for hyper-parameter λ usually through an iterative and approximate method is called the problem of *hyper-parameter optimization*. The computation performed by learning algorithm *A* involves an inner-loop which is iterative and an outer-loop algorithm that optimizes for hyper-parameters. Given in the form of equation:

$$\lambda^{(*)} = \operatorname{argmin}_{\lambda \in \Lambda} E_{x \sim Gx} [L(x; A_\lambda (X^{(\text{train})}))]$$

where,
E_x is generalization error
G_x is grand truth distribution
$X^{(\text{train})}$ is a finite set of samples x from G_x
L is expected loss $L(x; f)$ over finite samples
A_λ actual learning algorithm with λ

NHOP software framework helps achieve mentioned objective. The framework uses python as its frontend and distributed TensorFlow [1] as its backend for training neural network.

The inner-loop optimization is performed using TensorFlow [1] API which are implemented as dataflow-like models. The computations are expressed as stateful dataflow graphs. Outer-loop "Optimizer" is a python process that computes hyper-parameters from hyperspace using Random Search [2] algorithm and feeds them to inner-loop which trains the finite set of samples $X^{(\text{train})}$ iteratively that minimizes some of the expected loss. Given all that what we need in practice is a way to choose λ so as minimize generalization error. NHOP framework allows users to specify distribution bounds (or) search space for hyper-parameter.

II. Implementation

Note: Keywords representation in this paper. TensorFlow [1] specific acronyms are mentioned in *italics* starting from this section of paper. For example, *ps* represent parameter server task and *worker* is compute server task in TensorFlow [1] distributed layer. NHOP specific python process start with upper case letter, like Optimizer, PS and Worker. Keywords in quotes are user specified parameters or hyper-parameters, like "learning_rate".

A. *Framework:*

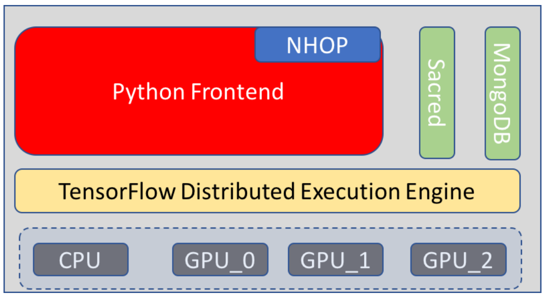


Fig. 1 NHOP software stack

NHOP framework works with python frontend and uses TensorFlow Distributed Execution Engine as backend. User parameters configurations are managed using Sacred interface either as dictionary or yaml format.

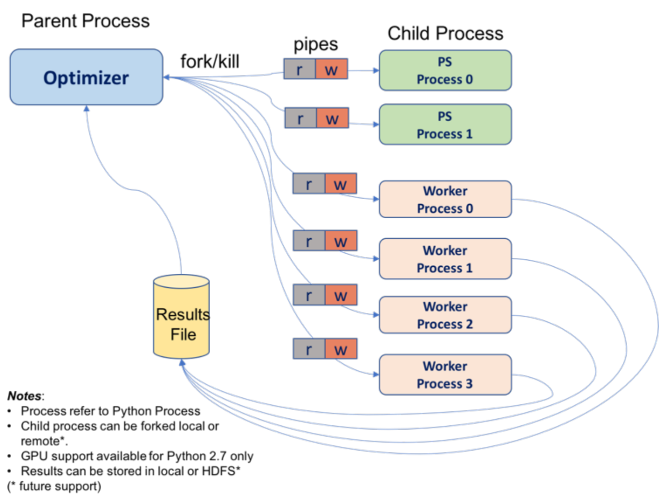


Fig. 2 Parent and child process

NHOP framework has three python process categories. Parent process called Optimizer, two child processes PS (also called Parameter Server) and Worker (also called Compute Server). Signalling between the Worker and Optimizer is accomplished through python sub-process pipes. Training results and best configurations are saved/read from file system. (Future work will support HDFS). Current NHOP architecture is capable of spawning multi-processes on single server machine, like AWS EC2. (Future work will support multi server processing, jobs spawned on multiple EC2)

The parent python process is designed to spawn multiple new child python process, connect to their output/error pipes and obtain their return codes. Once job done, these child process are terminated using simple inter-process communication signal SIGUSR1. Results of Worker processes and best optimized hyper-parameters are stored in file system (FS). Best hyper-parameters are the once with minimum loss during the optimizer training iteration or epoch.

B. *Optimizer Process:*

Optimizer process reads user specified configuration and determines the staged execution of hyper-parameter optimization. It builds the configuration for each stage, considering the search space or bounds from best hyper-parameters of previous stage. Optimizer determines smaller bounds around the best points from previous stage.

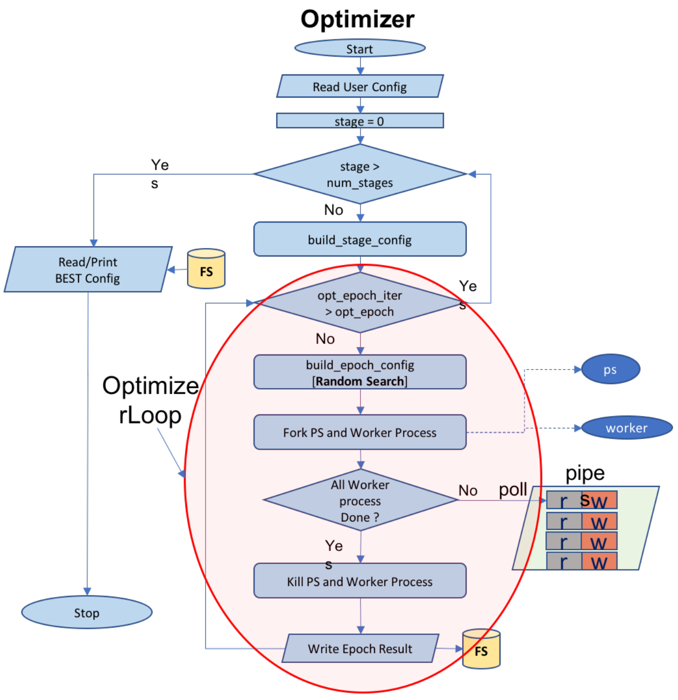


Fig. 3 Optimizer process flow diagram

Optimizer process runs stage loop and optimizer (outer) loop. The purpose of stage loop is to run user specified number of stages. For example, stage 1 could be the largest search with training (inner) loop of 1k and optimizer (outer) loop of 10k. Followed by stage 2 and 3 which are smaller volume around the best hyper-parameters points of previous stage. Intuition here is early stages would capture a larger hyperspace volume of hyper-parameters and later would narrow down to optimal hyper-parameters set which would minimize the expected loss.
For every optimizer (outer) loop iteration, hyper-parameters are determined using Random Search [2] (Future work will support other methods like Genetic

algorithm and Bayesian). Parameters like "nodes_per_layer" (number of hidden neural network nodes) are obtained from the 'discrete uniform' distribution specified within range of lower and upper bounds. Categorical parameters like activation and "train_optimizer" are obtained using numpy.random.randint. Continuous parameters like "learning_rate" is sampled from uniform distribution.

Fig.2 presents the execution flow of Optimizer process.

C. *TensorFlow Distributed model:*

Before exploring more about PS and Worker process it's important to understand TensorFlow distributed framework. Not to confuse with the terminology, *italics* representation are TensorFlow specific.

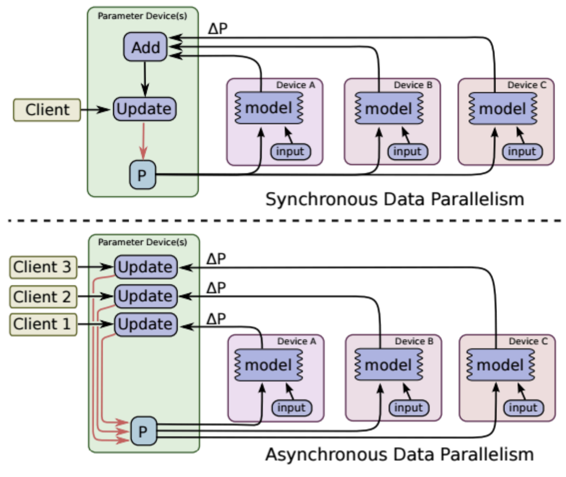


Fig. 4 TensorFlow Data Parallelism Methods

NHOP child processes (PS and Worker) further instantiate TensorFlow [1] server class which spawn's *tasks*. These *tasks* represent a TensorFlow [1] *cluster* which are set of *tasks* that participate in the distributed execution of a TensorFlow [1] graph. Each task is associated with a TensorFlow [1] *server*, which contains a master or chief that's used to create sessions, and a *worker* that executes operations in the graph. All training jobs have data parallelism meaning *worker* job training the same model on different mini-batches of data, updating shared parameters hosted in one or more *ps tasks*. Current version 1.0 of NHOP software allows multiple *worker*/*ps* processes on same machine with or without GPUs.

NHOP framework allows user to specify type of replicated training. When user parameter flag "sync_replicas" is set to true, synchronous training is performed. In this approach, all of the TensorFlow graph *replicas* read the same values of current parameters, compute the gradient in parallel, and then apply them together. It is compatible with both in-graph and between the graph replication. Fig. 4 shows both synchronous and asynchronous data parallelism between parameter *tasks or devices* (in case of multi-server). Fig.4 shows TensorFlow data parallelism methods.

D. *PS Process:*

PS process reads the epoch cluster configuration, instantiates a TensorFlow *ps server* and invokes sever.join(), which basically goes into listening mode.

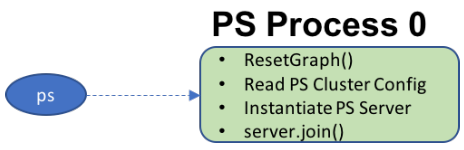


Fig. 5 Parameter Server process flow diagram

Parameter server *task* manages/updates the TensorFlow variables. A good example would be weights (W) and biases (b) used in simple linear regression (y_hat = Wx + b) as maintained by parameter server *ps*. PS process instantiate *ps server* that will listen to all *worker* and updates its variables.

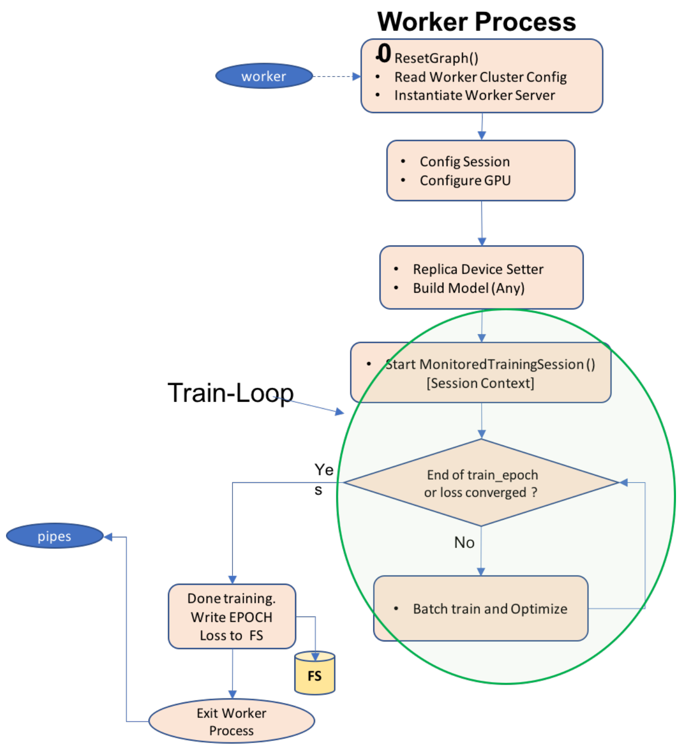
E. *Worker Process:*

Worker process reads the cluster configuration, instantiates a TensorFlow [1] *worker server* as `"/job:worker/task:%d/gpu:%d"`. Every GPU instance can run single or multiple *worker task*. When using single GPU for multiple *worker tasks*

`CUDA_VISIBLE_DEVICES=worker_task_id` must be specified while forking the Worker process.
Job of TensorFlow [1] *worker tasks* is to perform various compute operations like pre-processing, loss calculation and backpropagation.

Worker process starts with resetting default graph to avoid any stale entries from previous training epoch. Then, configure the GPU to not pre-allocate the entire specified GPU memory region, instead start small and grow as needed. With this, session is configured for use with devices found (like cpu or gpu).

Replica device setter TensorFlow [1] API will automatically place variables ops on separate parameter server *ps*. The non-Variable ops will be placed on the *workers*. The *ps* use CPU and workers use GPU. Now, build the neural network using NHOP **build_model** method of RegressorNN class. Generic build model can also be instantiated from CustomNN class.



Worker Process
worker
0 ResetGraph()
Read Worker Cluster Config
Instantiate Worker Server
Config Session
Configure GPU
Replica Device Setter
Build Model (Any)
Train-Loop
Start MonitoredTrainingSession() [Session Context]
End of train_epoch or loss converged ?
Yes
No
Batch train and Optimize
pipes
Done training. Write EPOCH Loss to FS
FS
Exit Worker Process

Fig. 6 Worker process flow diagram

Before we start the TensorFlow [1] *session*, create hooks to handle session initializations and assign the number of training epochs from configuration built by optimizer process. With all that done start Monitored Training Session *context* and start the training by invoking NHOP **train_model** method. Fig. 6 show detailed Worker process flow diagram.

A completed training will return the computed minimized loss and best hyper-parameter used back to Optimizer.

III. Configuration Space

Below listed are different hyper-parameters that user can configure.

```
nn_dimensions    = [4, 1]     # number of neural
network nodes for [input, output]
                              # input: should
match lenght of input matrix (X)
                              # output: should
be 1: for regressor (Y predict for a row in X)
batch_size       = [100, 2000]      # batch
size as [lower_bound, upper_bound]
learning_rate    = [0.001, 0.0001]  # learning
rate as [lower_bound, upper_bound]
hidden_layers    = [4, 10]    # hidden_layer
[min, max]
train_optimizer = ['Adam', 'sgd', 'Adagrad']
# optimizer to be used for train. Default is
'Adam'. Supports 'sgd', 'Adagrad'
activation       = ['relu','tanh']    #
activation for non-linearity. Default is
'relu'. Supports 'tanh'
opt_epoch        = 3          # optimizer epoch
is the outer loop for optimizing
hyperparameters
train_epoch      = 1000       # training epoch
is the inner loop for training input data
train_tolerance = 1e-8        # inner train loop
loss convergence threshold
opt_tolerance    = 1e-5       # outer optimizer
loop loss convergence threshold
rnn_max_seq_length   = 100    # int Maximum
length of the traning sequences to generate
rnn_state_dim    = [50, 50, 50]  # int or int[]
State dimension. Can use a list to stack RNNs.
```

IV. Stage Testing

This section describes the staged NHOP using synthetically generated complex cosine function. Fig.7 shows the X/Y grand truth relationship. Followed by outcomes of each stage. This test was run on AWS EC2 (p2.xlarge) instance with Tesla GPU, TensorFlow [1] version 1.0.1, CUDA 8.0 and python 2.7.

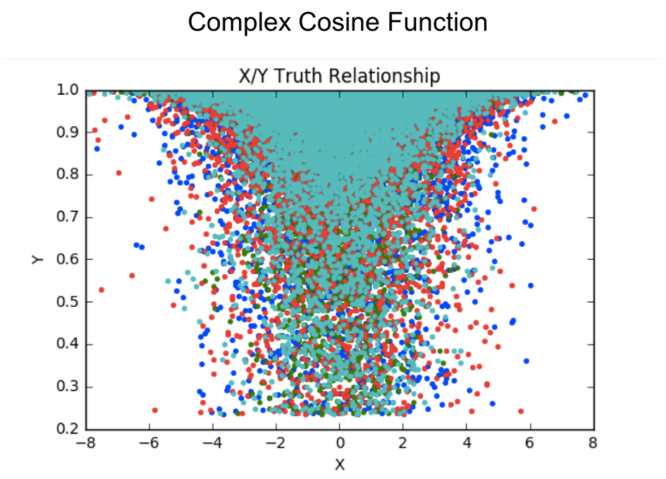


Fig. 7 Synthetic Data Truth Relationship

Stage 1:

Fig.8 show outcome of stage 1 which was run with train (inner) loop of 1000 and optimizer (outer) loop of 1000. Plot on the right bottom represent target output y_test and predicted y_hat_test. We could see the predictions has lots of error. Right upper plot of truth relationship predicts points (in cyan) outside the actual test data.

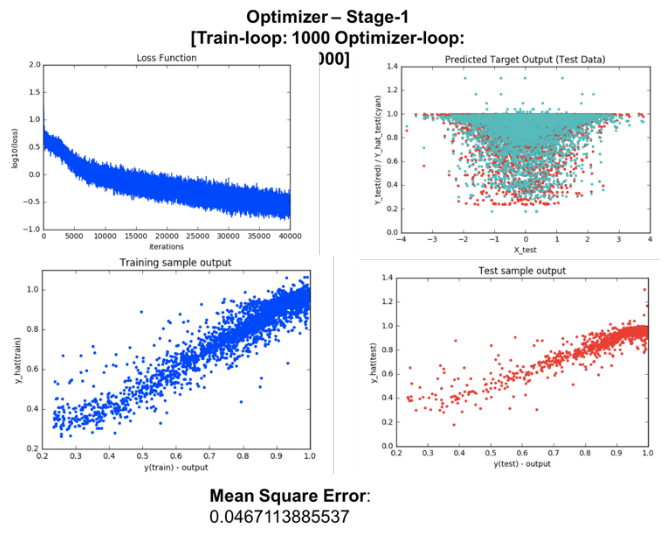


Fig. 8 Stage 1 Results with [Train-loop: 1000 Optimizer-loop: 1000]

Stage 2:

Fig.9 shows outcome of stag 2which was run with train (inner) loop of 5000 and optimizer (outer) loop of 750. In this case errors significantly reduced and target output y_test and predicted y_hat_test almost match.

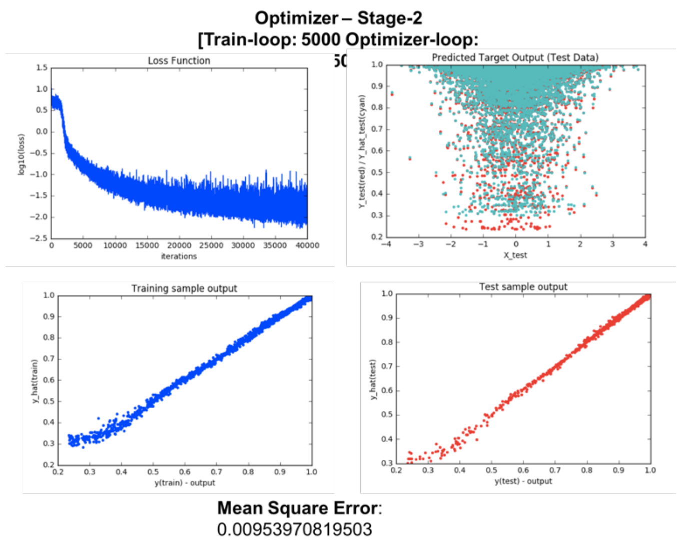


Fig. 9 Stage 2 Results with [Train-loop: 5000 Optimizer-loop: 750]

Stage 3:

Fig.10 show3 outcome of stage 3 which run with even smaller volume of hyper-parameter space around the best points from stage 2. We could see loss and MSE are lower than stag 2.

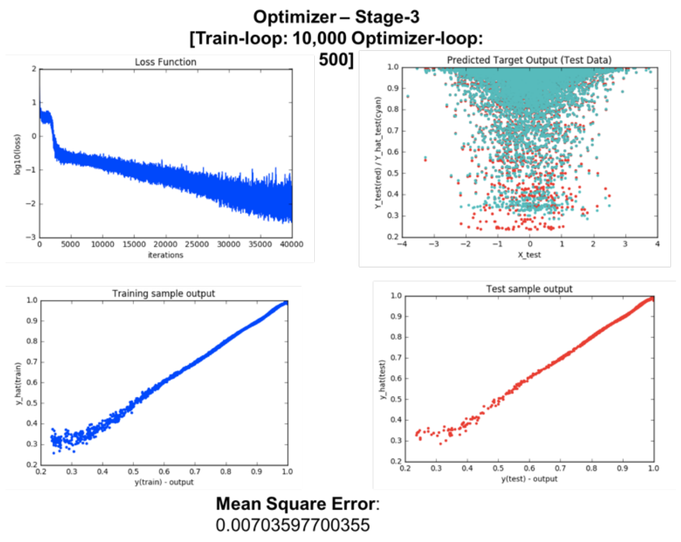


Fig. 10 Stage 3 Results with [Train-loop: 10,000 Optimizer-loop: 500]

V. Conclusion

It’s evident NHOP type of hyper-parameter optimization framework could find optimal hyper-parameters in complex high dimensional hyperspace. Random search is just a method and that works fairly well than manual or grid search. Methods like Genetic Algorithm and Bayesian methods could prove even more effective.

VI. Future work:

Goal is to open source NHOP to wider community usage and we believe following additions would bring more value to users:

- Optimization algorithm support:
    - Genetic
    - Bayesian methods
- Modify software API independent of sacred/mongoDB
- Add support for remote process (PS/Worker) forking
- Fix python 3.x related sync replica bugs

References

[1] Mart´ın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris ´ Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol ´ Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] James Bergstra, Yoshua Bengio. Random Search for Hyper-Parameter Optimization.