2. Parallax

김정훈,한정민,모유찬

1. 구현해야 할 것 (5점): parallax의 shard API를 이용하여 각 worker가 서로 다른 data files을 처리하도록 구현한다.

- cifar_input.py
- parallax 의 shard 추가

```
"""CIFAR dataset input module.
"""

import tensorflow as tf
import parallax
from parallax import shard
```

```
  ##############################################################################
  #### FIXME: Partition dataset so that each worker can read disjoint data files
  ####
  ##############################################################################
  data_files = tf.gfile.Glob(data_path)
  data_files.sort()
  ds = tf.data.Dataset.from_tensor_slices(data_files).repeat()
  ds = shard.shard(ds)
  ds = ds.apply(
      # Read and preprocess cycle_length files concurrently
      tf.contrib.data.parallel_interleave(
          lambda filename: tf.data.FixedLengthRecordDataset(filename, record_b
ytes),
          cycle_length=len(data_files)))

  # Prefetch BATCH_SIZE items
  ds = ds.prefetch(batch_size)
```

2. 구현해야 할 것 (5점): Parallax API를 이용하여 주어진 단일 머신 모델 구조 함수를 사용하여 분산학습을 진행한다. 100 step에 한번씩 학습된 파라미터의 checkpoint 파일을 생성한다.

- parallax_config.py 생성(practice 폴더로 부터 파일 복사, 과제 기준에 맞춰 수정)
- save_ckpt_steps : 100

```
import tensorflow as tf
import parallax

flags = tf.app.flags
flags.DEFINE_boolean('replicate_variables', True, """replicate_variables""")
flags.DEFINE_string('protocol', 'grpc', """The method for managing variables""")
flags.DEFINE_boolean('use_allgatherv', False, """use allgatherv instead of allga
ther""")
flags.DEFINE_string('mpirun_options', '', 'The option for mpirun')
flags.DEFINE_string('redirect_path', None, """redirect path to keep the log of d
istributed workers""")
flags.DEFINE_integer('save_ckpt_steps', 100,
                     """Number of steps between two consecutive checkpoints""")
flags.DEFINE_string('profile_dir', None, """Directory to save RunMetadata""")
flags.DEFINE_string('profile_steps', None, """Comma separated porfile steps""")
flags.DEFINE_string('profile_range', None, """profile start step,profile end ste
```

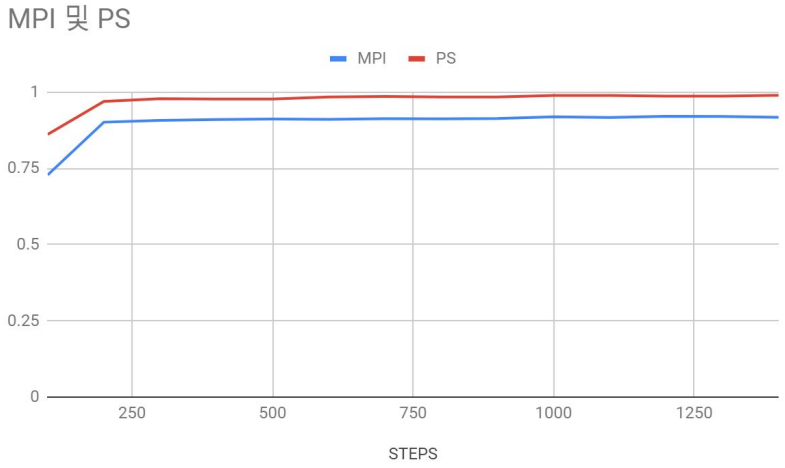
- 100 번 수행시 마다 아래와 같이 model.ckpt 내 save

```
step: 691, loss: 1.241, precision: 0.641
INFO:tensorflow:Saving checkpoints for 700 into ckpt/model.ckpt.
INFO:tensorflow:global_step/sec: 0.912595
step: 701, loss: 1.384, precision: 0.594
step: 701, loss: 1.312, precision: 0.609
step: 701, loss: 1.262, precision: 0.641
step: 711, loss: 1.371, precision: 0.586
step: 711, loss: 1.087, precision: 0.719
```

3. PS와 MPI(default)의 성능비교 (5점): Parallax의 run_option을 달리하여 PS와 MPI 방식의 성능(global_step/sec)을 비교하고 그 결과를 보고서에 첨부한다. (node 한개에 하나의 worker/server를 띄운 것을 기준으로 한다.)

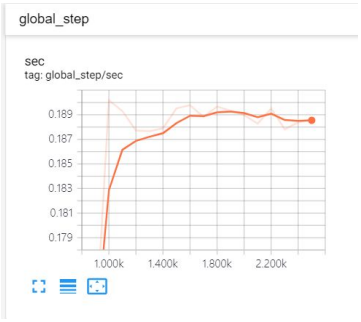
1) **PS - MPI global_step/sec(~1400steps**

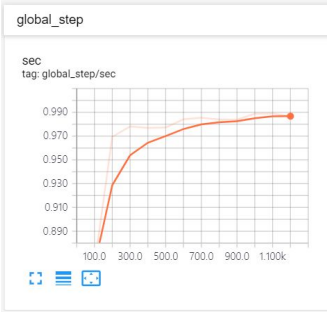
- **결과) MPI : 0.9 / PS : 0.97**
- **사유로는 AWS가 PS 수행시 불안정하였었고, 추가적으로 NNI task를 동시에 작업하고 있었음 => 그에 따라서 병렬 처리에 성능이 100% 나오지 않아 PS를 재실행 1400 steps 까지 재진행 후 비교**



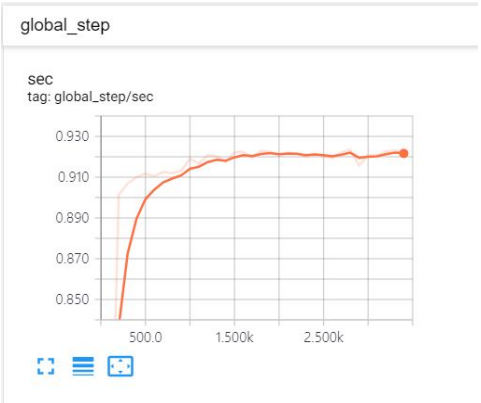
2) **PS : 수행 당시 NNI를 동시 진행하여서 초당 진행 global_step이 낮게 나옴=>재수행하여 0.189 => 0.9 성능 확인**

(with NNI Task) **(without NNI Task)**





3) **MPI : Parallax만 단독 수행하여 빠른 속도로 수행 확인**

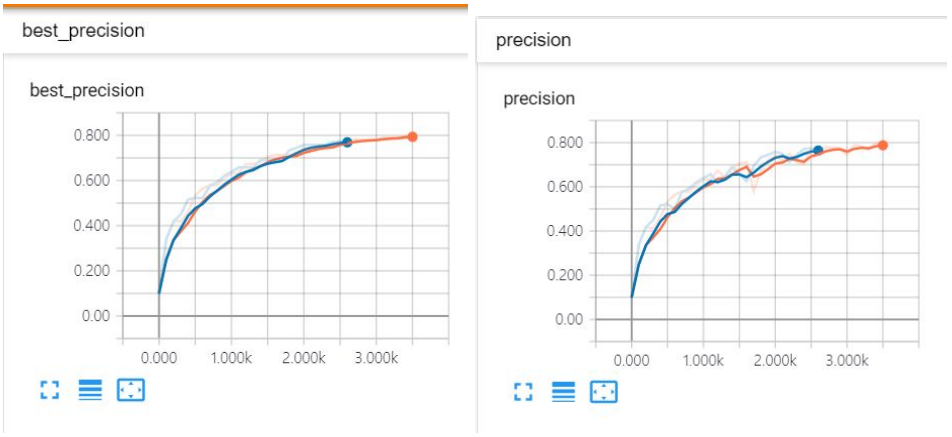


4. 구현해야 할 것 (5점): run_eval.sh을 수행하여 정확도를 측정한다. 또한 각 체크포인트 파일에 대하여 계산된 precision과 best_precision값을 summary로 FLAGS.eval_dir에 남겨, 모든 체크포인트에 대하여 evaluation이 완료된 후 TensorBoard를 이용하여 확인한다.

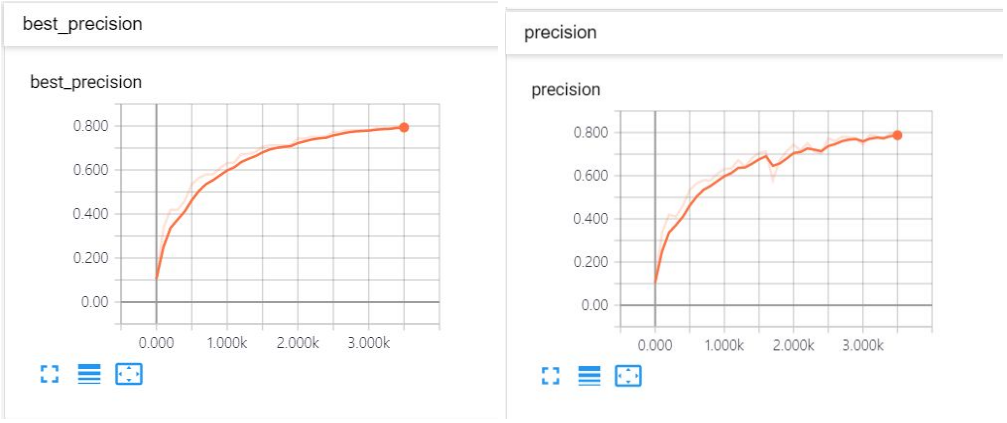
TENSOR BOARD

- MPI vs PS : (주황색 : MPI , 파란색 : PS) ⇒ 두개간의 성능 차이는 없어보임

-서버 상태상 PS는 2400steps 정도까지만 진행, MPI는 3000steps 이상 진행



- MPI : precision(0.796), best precision(0.796) ⇒ steps 3500



- PS : precision(0.774), best precision(0.774) ⇒ steps 2600