Suneeta Mall

Rambling of a curious engineer & data scientist

Posts · Projects · Talks · About

Realizing reproducible Machine Learning - with Tensorflow

Sunday, December 22, 2019, 12:00 AM Machine-learning, AI, Reproducible-ml

This is Part 2 - **Realizing reproducible Machine Learning - with Tensorflow** of technical blog series titled Reproducibility in Machine Learning. Part 1 & Part 3 can be found here & here respectively.

As discussed in Part 1, writing reproducible machine learning is not easy with challenges arising from every direction e.g. hardware, software, algorithms, process & practice, data. In this post, we will focus on what is needed to ensure ML code is reproducible.

First things first

There are a few very simple things that's needed to be done to before *thinking big* and focussing on writing reproducible ML code. These are:

- Code is version controlled

  Same input (data), same process (code) resulting into same output is essence of reproducibility. But code keeps evolving, since ML is so iterative. Hence, its important to version control code (including configuration). This allows obtaining same i.e. exact code (commit/version) from source repository.

- Reproducible runtime – pinned libraries

  So we version controlled code but what about environment/runtime? Sometimes, non-determinism is introduced not direct by user code but also dependencies. We talked about this at length in software section of `Challenges in realizing reproducible ML` in Part 1. Taking the example of Pyproj - a geospatial transform library, that I once used to compute geo-location based on some parameters. We changed nothing but just version of pyproj from V1.9.6 to V2.4.0 and suddenly all our calculation were giving different results. The difference was so much that location calculation for San Diego Convention Centre were coming out to be somewhere in Miramar off golf course (see figure 1) issue link. Now imagine ordering pizza delivery on the back of my computation snippet backed with unpinned version of pyproj?

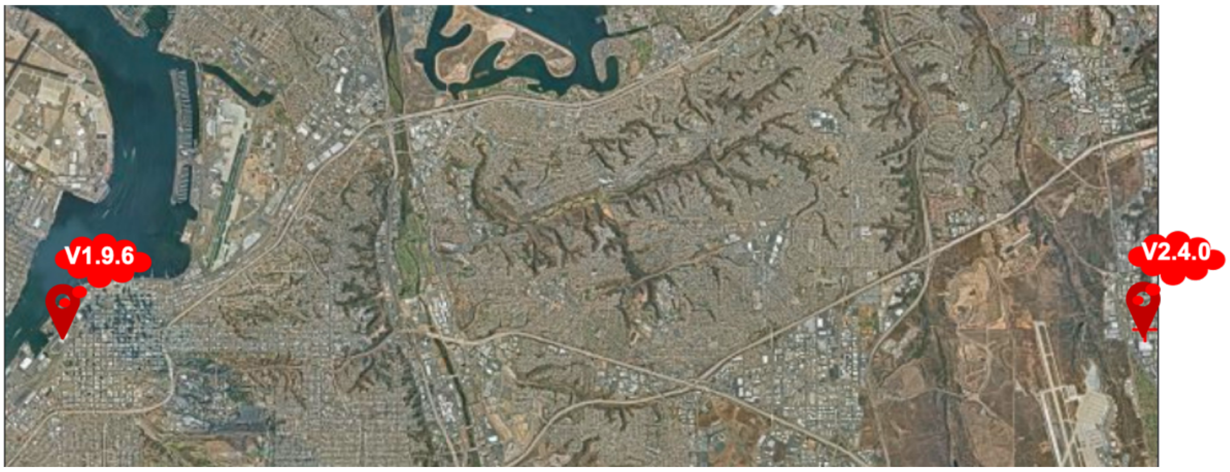


Figure 1: Example of why pinned libraries are important

Challenges like these occur quite often that we would like. That's why its important to pin/fix dependent runtime either by pinning version of libraries or using versioned containers (like docker).

- Smart randomness

  As discussed in Part 1, randomization plays a key role in most ML algorithms. Unseeded randomness is the simplest way to make code non-reproducible. It is also one of the easiest to manage amongst all things gotchas of non-reproducible ML. All we need to do is seed all the randomness and manage the seed via configuration (as code or external).

- Rounding precision, under-flows & overflows

  Floating point arithmetic is ubiquitous in ML. The complexity and intensity of floating point operations (FLOPS) are increasing everyday with current needs easily meeting Giga-Flops order of computations. To achieve the efficiency in terms of speed despite complexity, mixed precision floating point operations have also been proposed. As discussed Part 1, accelerated hardware such as (GPGPU), tensor processing unit (TPU) etc. due to their architecture and asynchronous computing do not guarantee reproducibility. In addition, when dealing with floating points, the issues related to overflow and underflow are expected. This just adds to the complexity.

- Dependent library's behavior aware

  As discussed in software section of `Challenges in realizing reproducible ML` in Part 1, some routines of ML libraries do not guarantee reproducibility. For instance, NVIDIA's cuda based deep learning library cudnn. Similarly, with

Tensorflow, using some methods may result into non-deterministic behaviour. One such example is backward pass of broadcasting on GPU[ref]. Awareness about behaviours of libraries being used and approaches to overcome the non-determinism should be explored.

Writing reproducible ML

To demonstrate reproducible ML, I will be using Oxford Pet dataset that has labels for pet images. I will be doing a semantic segmentation of pets images and will be using pixel level labelling. Each pixel of the pet image in Oxford Pet dataset is labelled as 1) Foreground (Pet area), 2) Background (not pet area) and 3) Unknown (edges). These labels by definition are mutually exclusive - i.e. a pixel can be only be one of the above 3.

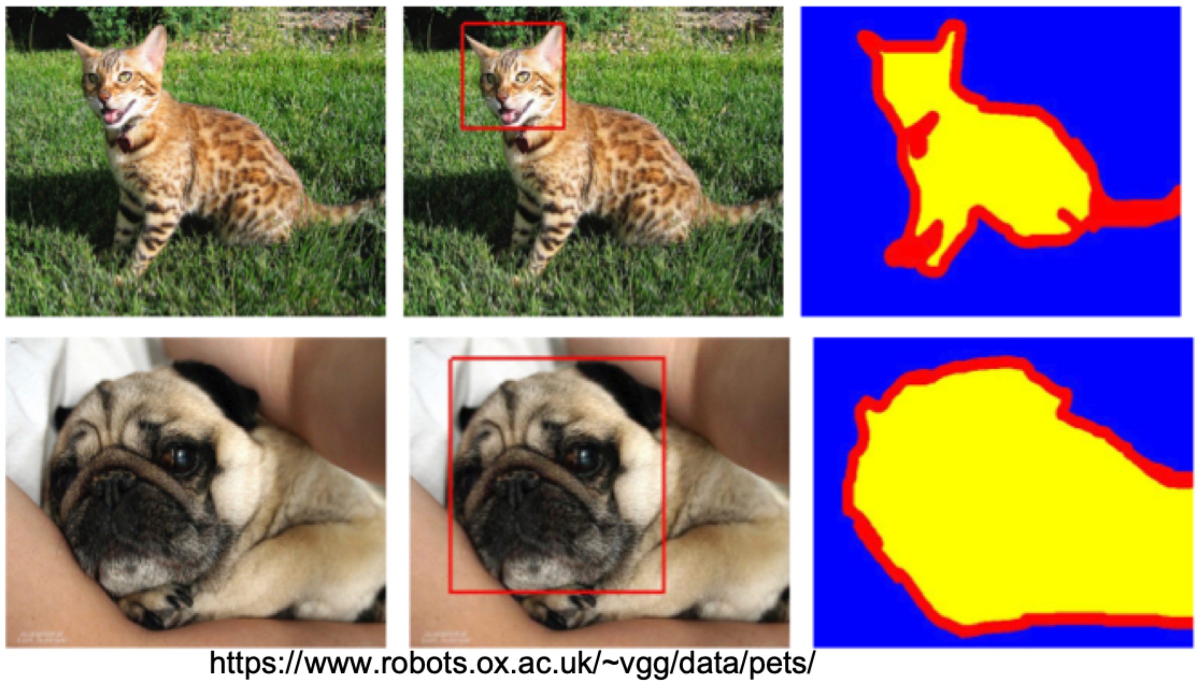
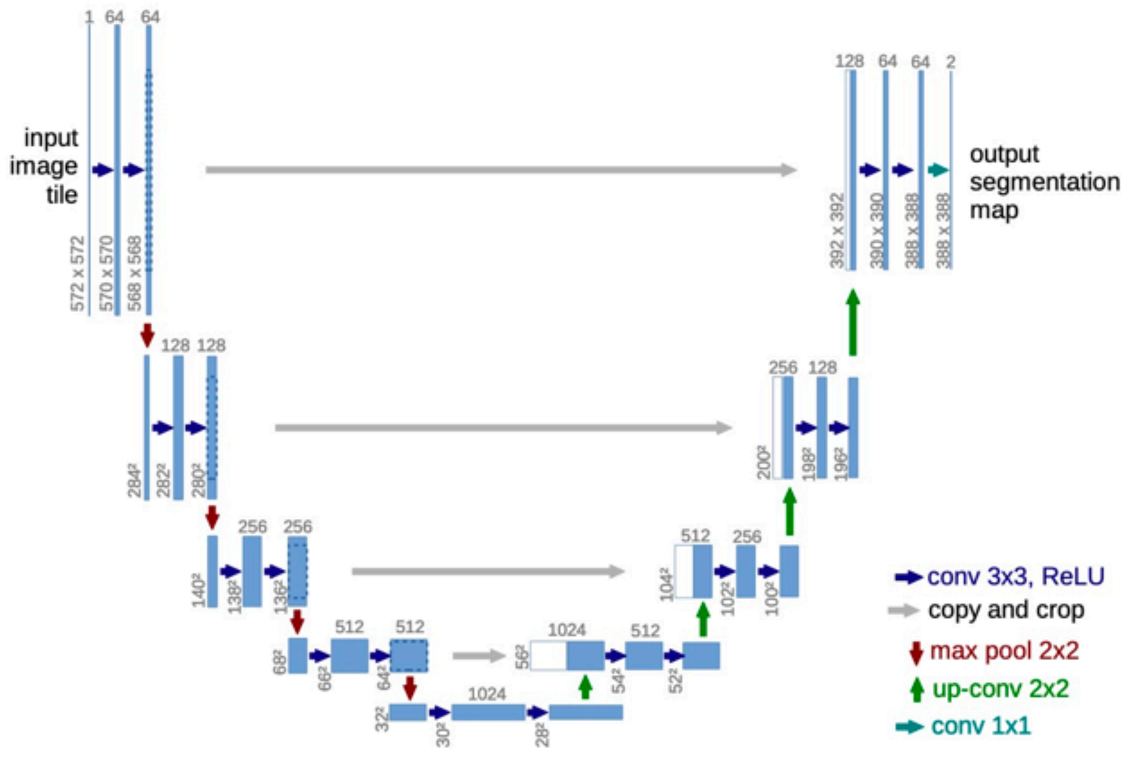


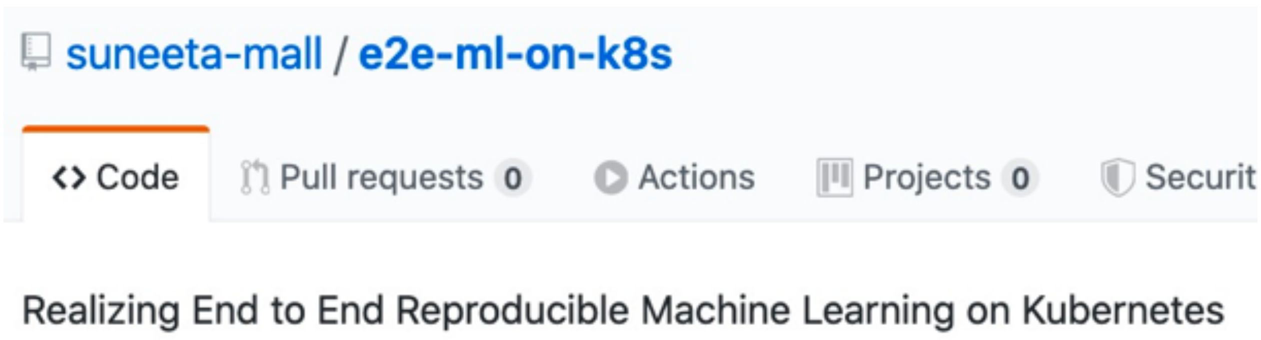
Figure 2: Oxford pet dataset

I will be using convolution neural network (ConvNet) for semantic segmentation. The network architecture is based on U-net. This is similar to standard semantic segmentation example by tensorflow.



Fihure 3: U-net architecture

The reproducible version of semantic segmentation is available in github repository. This example demonstrate reproducible ML and also performing end to end ML with provenance across process and data.


suneeta-mall / e2e-ml-on-k8s

<> Code   Pull requests 0   Actions   Projects 0   Securit

Realizing End to End Reproducible Machine Learning on Kubernetes

Figure 4: Reproducible ML sample - semantic segmentation of oxford pet

In this post, however, I will be discussing only reproducible ML aspect of it and will be referencing snippets of this example.

ML workflow

In reality a machine learning workflow is very complex and look somewhat similar to figure 5. In this post, however, we will only discuss data and model training part of it. The remaining workflow i.e. the end to end workflow will be discussed in next post.

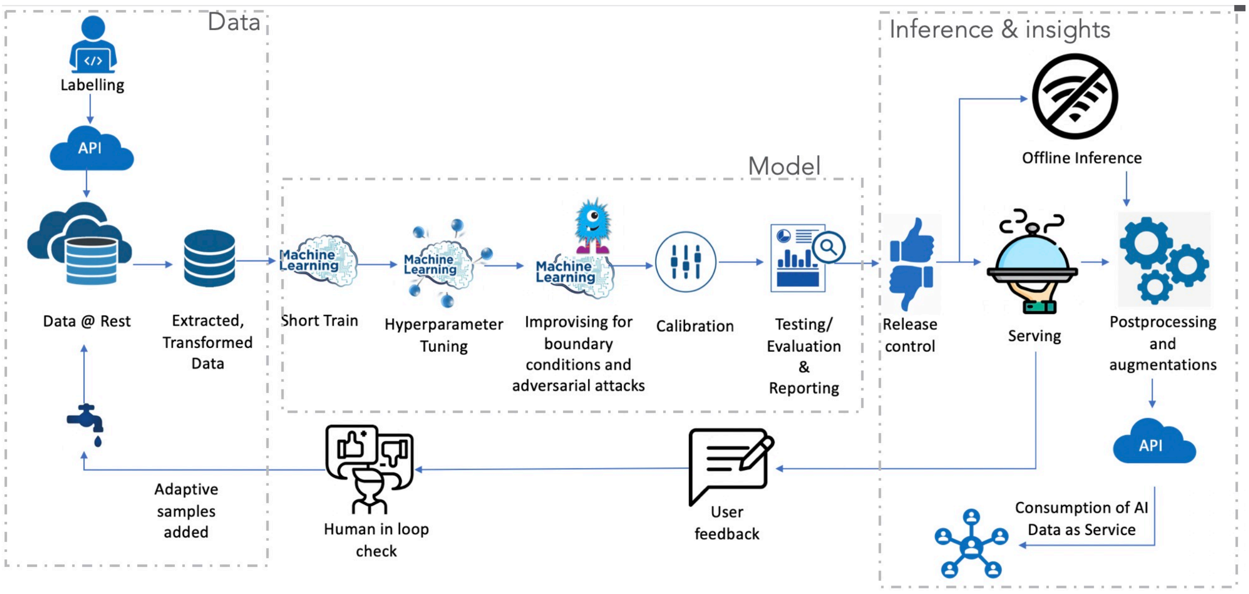


Figure 5: Machine learning workflow

Data

The source dataset is Oxford Pet dataset which contains multitude labels e.g. class outcome, pixelwise label, bounding boxes etc. First step is to process this data to generate the trainable dataset. In the example code, this is done by download_petset.py script.

```
python download_petset.py  --output /wks/petset
```

Result sample is show in figure 6.

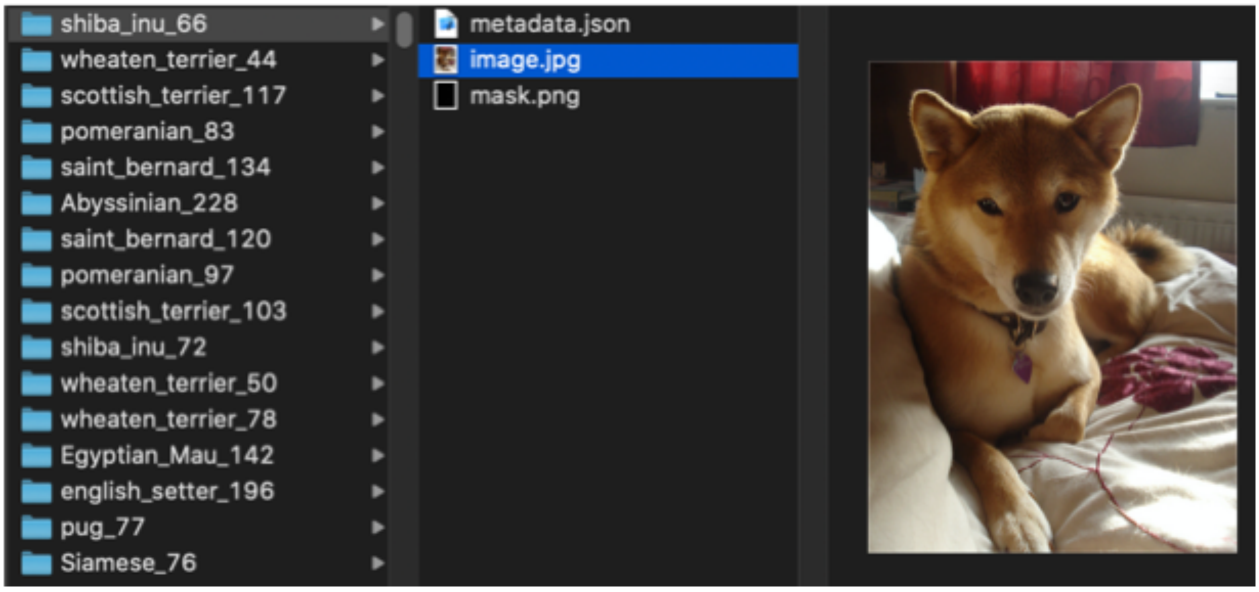


Figure 6: Pets data partitioned by Pets ID

Post data partition, the entire dataset is divided into 4 set: a) training, b) validation, c) calibration and d) test We would want this set partitioning strategy to be reproducible. By doing this, we ensure that if we have to blow away training dataset, or if accidental data loss occur then the *exact* dataset can be created.

In this sample, this is achieved by generating hash of petid and partitioning the hash into 10 folds (script below) to obtain partition index of pet id.

```
partition_idx = int(hashlib.md5((os.path.basename(petset_id)).encode()).hexdigest
```

With partition_idx 0-6 assigned for training, 7 for validation, 8 for calibration, and 9 for test, every generation will result in pets going into their respective partition.

In addition to set partitioning, any random data augmentation performed is seeded with seed controlled as configuration as code. See `tf.image.random_flip_left_right` used in this tensorflow data pipeline method.

Script for model dataset preparation is located in dataset_gen.py and used as following

```
python dataset_gen.py --input /wks/petset --output /wks/model_dataset
```

with results shown as below:

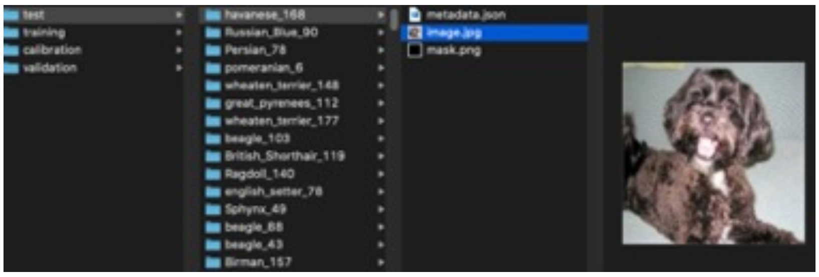


Figure 7: Pets data partitioned into training, validation, calibration and test set

Modelling semantic segmentation

The model for pet segmentation is based on U-net with backbone of either MobileNet-v2 or VGG-19 (defaults to VGG-19). As per this models network architecture, 5 activation layers of pre-trained backbone network are chosen. These layers are:

- MobileNet

```
'block_1_expand_relu'
'block_3_expand_relu'
'block_6_expand_relu'
'block_13_expand_relu'
'block_16_project'
```

- VGG

```
        'block1_pool'
        'block2_pool'
        'block3_pool'
        'block4_pool'
        'block5_pool'
```

Each of these layers are then concatenated with corresponding upsampling layer comprising of Conv2DTranspose layer forming whats known as skip connection. See model code for more info.

The training script train.py can be used as following:

```
python train.py --input /wks/model_dataset --output /wks/model --checkpoint_path ,
```

1. Seeding randomness

All methods exploiting randomness is used with appropriate seed.

- All random initialization is seeded e.g. tf.random_normal_initializer
- All dropout layers are seeded e.g. dropout

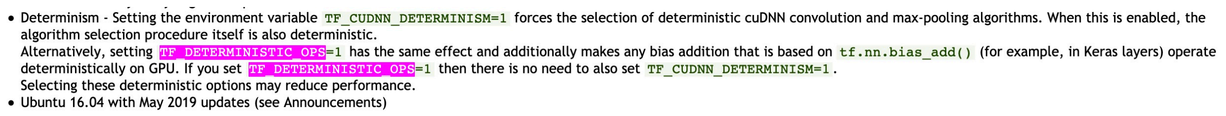
Global seed is set for any hidden methods that may be using randomness by calling in `set_seed(seed)` which sets seed for used libraries:

```
def set_seeds(seed=SEED):
    os.environ['PYTHONHASHSEED'] = str(seed)
    random.seed(seed)
    tf.random.set_seed(seed)
    np.random.seed(seed)
```

2. Handing library behaviours

2.1 CuDNN

CuDNN do not guarantee reproducibility in some of its routine. Environment variable `TF_DETERMINISTIC_OPS` & `TF_CUDNN_DETERMINISTIC` can be used to control this behavior as per this snippet (figure 8) from cudnn release page.



- Determinism - Setting the environment variable `TF_CUDNN_DETERMINISM=1` forces the selection of deterministic cuDNN convolution and max-pooling algorithms. When this is enabled, the algorithm selection procedure itself is also deterministic.
  Alternatively, setting `TF_DETERMINISTIC_OPS=1` has the same effect and additionally makes any bias addition that is based on `tf.nn.bias_add()` (for example, in Keras layers) operate deterministically on GPU. If you set `TF_DETERMINISTIC_OPS=1` then there is no need to also set `TF_CUDNN_DETERMINISM=1`.
  Selecting these deterministic options may reduce performance.
- Ubuntu 16.04 with May 2019 updates (see Announcements)

Figure 8: NVIDIA release page snippet reproducibility

2.2 CPU thread parallelism

As discussed in Part 1, using high parallelism with compute intensive workflow may not be reproducible. Configurations for inter[ref] and intra[ref] operation parallelism should be set if 100% parallelism is desired.

In this example, I have chosen *1* to avoid any non-determinism arising from inter operation parallelism. *Warning* setting this will considerably slow down training.

```
tf.config.threading.set_inter_op_parallelism_threads(1)
tf.config.threading.set_intra_op_parallelism_threads(1)
```

2.3 Tensorflow determinism

Following are some of the application of Tensorflow that are not reproducible:

- Backward pass of broadcasting on GPU is non-deterministic[link]
- Mention that GPU reductions are nondeterministic in docs[link]
- Problems Getting TensorFlow to behave Deterministically[link]

Duncan Riach, along with several other contributors have created tensorflow_determinism package that can be used to overcome non-reproducibility related challenges from tensorflow. It should be used in addition to above measures we have discussed so far.

If we combine all the approaches discussed above (aside from using seeded randomness), they can be wrapped into a light weight method like one below:

```
def set_global_determinism(seed=SEED, fast_n_close=False):
    """
        Enable 100% reproducibility on operations related to tensor and randomnes
        Parameters:
        seed (int): seed value for global randomness
        fast_n_close (bool): whether to achieve efficient at the cost of determin
    """
    set_seeds(seed=seed)
    if fast_n_close:
        return

    logging.warning("*************************************************************
    logging.warning("*** set_global_determinism is called,setting full determinis
    logging.warning("*************************************************************
    os.environ['TF_DETERMINISTIC_OPS'] = '1'
    os.environ['TF_CUDNN_DETERMINISTIC'] = '1'
    # https://www.tensorflow.org/api_docs/python/tf/config/threading/set_inter_op
    tf.config.threading.set_inter_op_parallelism_threads(1)
    tf.config.threading.set_intra_op_parallelism_threads(1)
    from tfdeterminism import patch
    patch()
```

which can then be used on top of ML algorithm/process code to generate 100% reproducible code.

Result

What happens if we dont write reproducible ML? What kind of difference we are really talking about? Last two column of figure 9 shows results obtained by model trained on exact same dataset, exactly same code with EXACTLY one exception. The dropout layer used in the network were unseeded. Every other measures discussed above were taken into account.

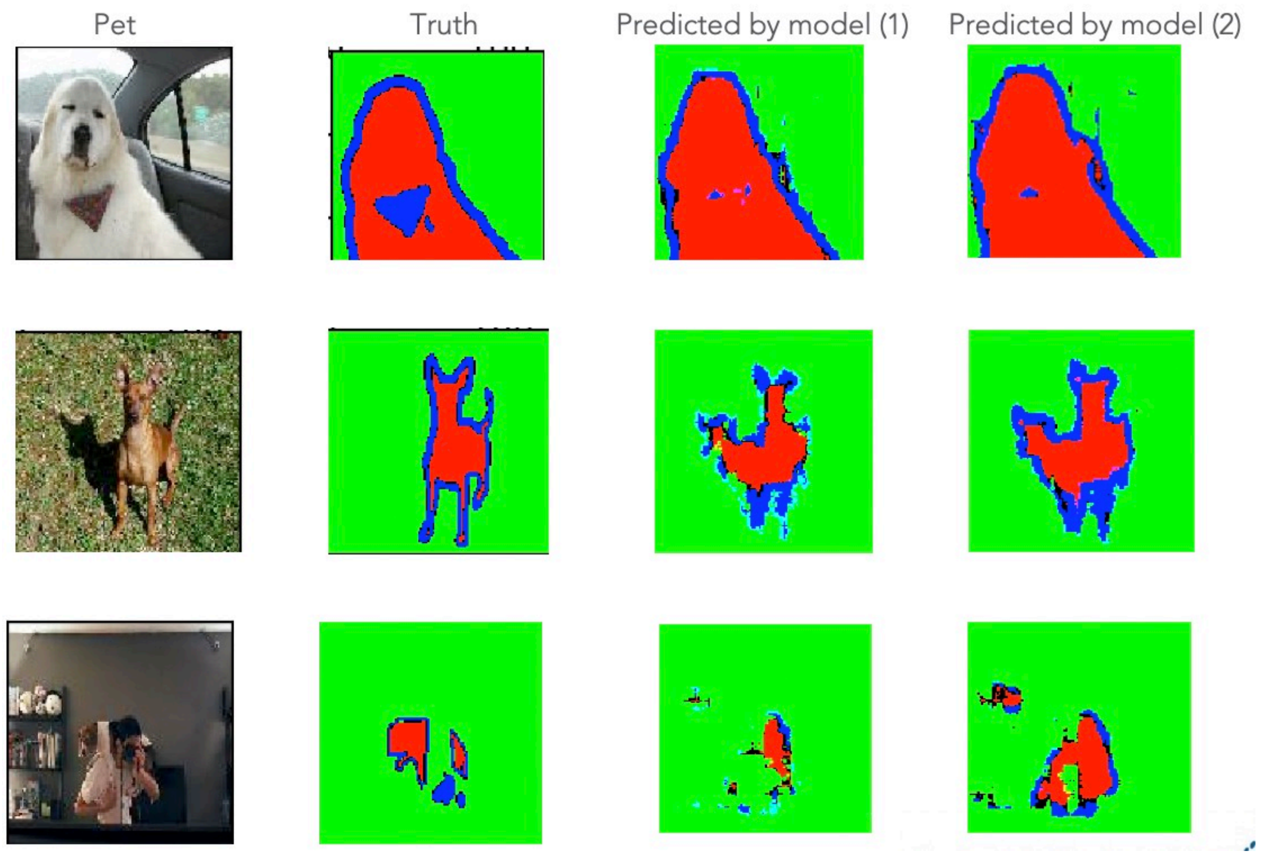


Figure 9: Effect of just forgetting to set one seed amidst many

Looking at the result of first pet which is very simple case, we can see subtle difference in outcome of these two models. Second pet case is slightly complicated due to shadow and we can see obvious differences in the outcome. But what about the third case, this is very hard case for pre-trained frozen backbone model we are using but we can see major differences in result from the two models.

If we were to use all the measures discussed above then 100% reproducible ML can be obatined. This is shown in the following 2 logs obtained by running the following:

```
python train.py \
  --input /wks/model_dataset \
  --hyperparam_fn_path best_hyper_parameters.json \
  --output logs \
  --checkpoint_path "logs/ckpts" \
  --tensorboard_path logs
```

wherein content of `best_hyper_parameters.json` are:

```
{
   "batch_size":60,
   "epochs":12,
   "iterations":100,
   "learning_rate":0.0018464290366223407,
   "model_arch":"MobileNetV2",
   "steps_per_epoch":84
}
```

Run attempt 1:

```
WARNING:root:******* set_global_determinism is called, setting seeds and determin
TensorFlow version 2.0.0 has been patched using tfdeterminism version 0.3.0
Input: tf-data, Model: MobileNetV2, Batch Size: 60, Epochs: 12, Learning Rate: 0.
Train for 84 steps, validate for 14 steps
Epoch 1/12
2019-11-07 12:48:27.576286: I tensorflow/core/profiler/lib/profiler_session.cc:18
84/84 [==============================] - 505s 6s/step - loss: 0.8187 - iou_score:
Epoch 2/12
84/84 [==============================] - 533s 6s/step - loss: 0.6116 - iou_score:
Epoch 3/12
84/84 [==============================] - 527s 6s/step - loss: 0.5829 - iou_score:
Epoch 4/12
84/84 [==============================] - 503s 6s/step - loss: 0.5733 - iou_score:
Epoch 5/12
84/84 [==============================] - 484s 6s/step - loss: 0.5566 - iou_score:
Epoch 6/12
84/84 [==============================] - 509s 6s/step - loss: 0.5524 - iou_score:
Epoch 7/12
```

```
84/84 [==============================] - 526s 6s/step - loss: 0.5439 - iou_score:
Epoch 8/12
84/84 [==============================] - 523s 6s/step - loss: 0.5339 - iou_score:
Epoch 9/12
84/84 [==============================] - 518s 6s/step - loss: 0.5287 - iou_score:
Epoch 10/12
84/84 [==============================] - 506s 6s/step - loss: 0.5259 - iou_score:
Epoch 11/12
84/84 [==============================] - 521s 6s/step - loss: 0.5146 - iou_score:
Epoch 12/12
84/84 [==============================] - 507s 6s/step - loss: 0.5114 - iou_score:
```

Run attempt 2:

```
WARNING:root:******* set_global_determinism is called, setting seeds and determin
TensorFlow version 2.0.0 has been patched using tfdeterminism version 0.3.0
Input: tf-data, Model: MobileNetV2, Batch Size: 60, Epochs: 12, Learning Rate: 0.
Train for 84 steps, validate for 14 steps

Epoch 1/12
2019-11-07 10:45:51.549715: I tensorflow/core/profiler/lib/profiler_session.cc:18
84/84 [==============================] - 549s 7s/step - loss: 0.8187 - iou_score:
Epoch 2/12
84/84 [==============================] - 515s 6s/step - loss: 0.6116 - iou_score:
Epoch 3/12
84/84 [==============================] - 492s 6s/step - loss: 0.5829 - iou_score:
Epoch 4/12
84/84 [==============================] - 515s 6s/step - loss: 0.5733 - iou_score:
Epoch 5/12
84/84 [==============================] - 534s 6s/step - loss: 0.5566 - iou_score:
Epoch 6/12
84/84 [==============================] - 494s 6s/step - loss: 0.5524 - iou_score:
Epoch 7/12
84/84 [==============================] - 506s 6s/step - loss: 0.5439 - iou_score:
Epoch 8/12
84/84 [==============================] - 514s 6s/step - loss: 0.5339 - iou_score:
Epoch 9/12
84/84 [==============================] - 518s 6s/step - loss: 0.5287 - iou_score:
Epoch 10/12
84/84 [==============================] - 531s 6s/step - loss: 0.5259 - iou_score:
Epoch 11/12
84/84 [==============================] - 495s 6s/step - loss: 0.5146 - iou_score:
Epoch 12/12
84/84 [==============================] - 483s 6s/step - loss: 0.5114 - iou_score:
```

So we have 100% reproducible ML code now but saying **training is snail-ish is an understatement**. Training time has increased (CPU based measures) from 28 minutes vs 1 hr 45 minutes as we give away with inter thread parallelism and also asynchronous computation optimization. This is not practical in reality. This is also why reproducibility in ML is more focussed around *having a road map to reach the same conclusions*- *Dodge*. This is realized by maintaining a system capable of capturing full provenance over everything involved in ML process including data, code, processes and infrastructure/environment. This will be the focus of part 3 of this blog series.

Next part of technical blog series, Reproducibility in Machine Learning, is End-to-end reproducible Machine Learning pipelines on Kubernetes.

« End-to-end reproducible Machine Learning pipelines on Kubernetes

Reproducibility in Machine Learning - Research and Industry »