Deep learning at scale
Tools and techniques
Part I: How DL training scales
1:30-2:20
   Introduction
   Challenges when scaling
   Components to a codebase
   CLI install

Part II: Topic deep dives
2:20-3:00 Reproducibility
3:00-3:30 Conference Tea Break (30 mins)
3:30-4:10 Hyperparameter Tuning
4:10-4:20 Break (10 mins)
4:20-5:00 Model Serving
Team of one

- Relatively small experiments on single machine
- No need to share resources or productionize results
- Relatively simple infrastructure to manage
- Simple experiment tracking is sufficient:

  results/lstm.dataset.batchsize-16.epochs-500.opt-adam.log
As scalability kicks in

Challenges surface:

- Resource/GPU sharing and infrastructure management
- Repetitive and time-consuming work in training, e.g. hyperparameter search
- Experiment tracking for reproducibility and collaboration
- Model deployment in production
Deep Learning Today (For Everyone Else)

Existing Tools (e.g., TensorFlow):
- Mostly focused on 1 researcher training 1 model on 1 GPU

Minimal Support For:
- Teams of researchers, clusters of GPUs, many models
- Deployment, ops, and collaboration

Focus of existing open-source tools
- Teams of researchers, clusters of GPUs, many models
- Deployment, ops, and collaboration
- Model training
- H.P. Tuning
- Data management
- Cluster management
What kind of AI infrastructure is needed to address scalability?

- TensorFlow
- PyTorch
- Keras

- GPU
- TPU

- AWS
- Google Cloud
We need **holistic** and **specialized** AI-native software infrastructure.
Determined AI offers holistic and specialized AI-native infrastructure
Cluster sharing and resource management

- Automated hyperparameter search
- Model deployment optimization
- Distributed Training
- Experiment tracking

**Cluster sharing and resource management**

**DIY**
- Static allocation
- Calendar-based system
- Queuing with jobs starving
- GPUs under-utilized

**Point Solutions**
- Kubernetes
- Mesos

**State of the Art**
- Dynamic flexible sharing
- Fault tolerance
- Simultaneous progress
- Priority and quota
Automated hyperparameter search

- Automated hyperparameter search
- Model deployment optimization
- Distributed Training
- Experiment tracking
- Cluster sharing and resource management

DIY

- Grid search
- Long delay to market

Point Solution

- Bayesian optimization
- Marginally better model performance

State of the Art

- Resource aware hyperparameter optimization
- 5x-50x faster

$\$$

$\$$

$\$$
Parallel and distributed training

**DIY**
- Single node training
- tf.distribute
- torch.distributed
- Framework dependency
- Manual configuration
- No fault tolerance

**Point Solution**
- Horovod
- Configure MPI/Gloo
- Change Model Code

**State of the Art**
- Simple distributed toggle (model code unchanged)
- Efficient Scaling
Model deployment optimization

DIY
-手动实施的剪枝策略
-手工优化
-实验技术从研究

Point Solution
-手实施的剪枝策略
-使用实验技术从研究

State of the Art
-全自动
-部署约束设计时
-系统辅助导航
-准确性与性能权衡空间
Experiment tracking

- Automated hyperparameter search
- Model deployment optimization
- Distributed training
- Experiment tracking
- Cluster sharing and resource management

DIY

- Disorganized metadata
- lstm.dataset.try3.log

Point Solution

- Experiment run logging

State of the Art

- Containerized execution
- Complete tracking of training environment
- Automatic metrics capture
- DL-specific reproducibility
- Query-able checkpoint database
- Fostering collaboration

State of the Art
What does holistic AI-native software infrastructure look like?
Pre-existing Keras implementation

- Code
- Hyperparameters
- Logs
- Checkpoints
- Visualizations
- Datasets
- Training Loss Curves
- Horovod
- Validation Metrics
- Library Version(s)
Components to a Keras codebase

- Data Loading
- Model Architecture
- Experiment Config

- Preprocessing
- keras.layer
- Specifies crucial metadata
Experiment Configuration

- data setup
- package versioning
- train time resources (distributed vs parallel)
- other plumbing
class CIFARTrial(TFKerasTrial):
    def __init__(self, *args: Any, **kwargs: Any):
        super().__init__(*args, **kwargs)

        self._base_learning_rate = self._hparams['learning_rate']  # type: float
        self._learning_rate_decay = self._hparams['learning_rate_decay']  # type: float
        self._layer1_dropout = self._hparams['layer1_dropout']  # type: float
        self._layer2_dropout = self._hparams['layer2_dropout']  # type: float
        self._layer3_dropout = self._hparams['layer3_dropout']  # type: float
        self._batch_size = self._hparams['batch_size']  # type: int

    def session_config(self, hparams: Dict[str, Any]) -> tf.ConfigProto:
        if hparams.get('disable_CPU_parallelism', False):
            return tf.ConfigProto(intra_op_parallelism_threads=1, inter_op_parallelism_threads=1)
        else:
            return tf.ConfigProto()

    def build_model(self, hparams: Dict[str, Any]) -> Sequential:
        model = Sequential()
        model.add(
            Conv2D(32, (3, 3), padding='same', input_shape=[IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS])
        )
        model.add(Activation('relu'))
        model.add(Conv2D(32, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(self._layer1_dropout))

        model.add(Conv2D(64, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(Conv2D(64, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(self._layer2_dropout))

        model.add(Flatten())
        model.add(Dense(512))
        model.add(Activation('relu'))
        model.add(Dropout(self._layer3_dropout))
        model.add(Dense(NUM_CLASSES))
        model.add(Activation('softmax'))

        return model
def make_data_loaders(
    experiment_config: Dict[str, Any], hparams: Dict[str, Any]
) -> Tuple[Sequence, Sequence]:
    
    """Provides training and validation data for model training.

    This example demonstrates how you could configure PEDL to help you optimize your data loading.

    In this example we added some fields of note under the `data` field in the YAML experiment configuration: the `acceleration` field. Under this field, you can configure multithreading by setting `use_multiprocessing` to `False`, or set it to `True` for multiprocessing. You can also configure the number of workers (processes or threads depending on `use_multiprocessing`).

    Another thing of note are the data augmentation fields in hyperparameters. The fields here get passed through to Keras' `ImageDataGenerator` for real-time data augmentation.
    """

    acceleration = experiment_config["data"]["acceleration"]
    url = experiment_config["data"]["url"]
    width_shift_range = hparams.get("width_shift_range", 0.0)
    height_shift_range = hparams.get("height_shift_range", 0.0)
    horizontal_flip = hparams.get("horizontal_flip", False)
    batch_size = hparams["batch_size"]

    (train_data, train_labels), (test_data, test_labels) = get_data(url)

    # Setup training data loader.
    data_augmentation = {
        "width_shift_range": width_shift_range,
        "height_shift_range": height_shift_range,
        "horizontal_flip": horizontal_flip,
    }
    train = augment_data(train_data, train_labels, batch_size, data_augmentation)

    if acceleration:
        workers = acceleration.get("Workers", 1)
        use_multiprocessing = acceleration.get("use_multiprocessing", False)
        train = KerasDataAdapter(train, workers=workers, use_multiprocessing=use_multiprocessing)

    # Setup validation data loader.
    test = CIFARSequence(test_data, test_labels, batch_size)

    return train, test
Example: RCNN with Inception V2 on COCO
http://52.38.76.131:8080/
Part I: how DL training scales
1:30-2:20 Introduction
   Challenges when scaling
   Components to a codebase
   CLI install

Part II: topic deep dives
2:20-3:00 Reproducibility
   Conference Tea Break (30 min)
3:30-4:10 Hyperparameter Tuning
   Break (10 min)
4:20-5:00 Model Serving
Reproducibility
Death by a thousand cuts: data ordering, software versions, hyperparameters, random seeds, model weights
Why should we care?

Scientific Progress

Reproducibility is a fundamental tenet of scientific progress

Hidden sources of randomness can lead to erroneous conclusions
Why should we care?

- **Scientific Progress**
  - Enable sharing & encourages experimentation
  - Easily ramp-up new hires
  - Reduce dependency on individual team member

- **Collaboration**

- **Accountability**
Why should we care?

- Scientific Progress
- Collaboration
- Accountability

Avoid lossy translation between training and deployment

Easily roll back in case of system crash or poor performance
Wait...isn’t this a solved problem?

**Traditional Software Engineering**

```
compile(code, deps) → binary
```

**Deep Learning Engineering**

```
optimize(
    architecture, deps, data, init state
) → ML model
```

Additional inputs + noisy optimizer = **ML reproducibility is hard!**
We’re taking over for Leslie
to improve on the model she trained then deployed
Artifacts from Leslie

```python
class MultiTaskMLLTrial(KerasTrial):
    def __init__(self, hparams: Dict[str, Any]):
        super().__init__(hparams)
        self._dropout = hparams.get("dropout", 0.5)

    def build_model(self, hparams: Dict[str, Any]):
        model = Sequential()
        model.add(Conv2D(32, kernel_size=(3, 3), activation="relu")
        model.add(Conv2D(64, (3, 3), activation="relu")
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(self._dropout))
        model.add(Flatten())
        model.add(Dense(128, activation="relu")
        model.add(Dropout(0.5))
        model.add(Dense(NUM_CLASSES, activation="softmax"))
        return model

    def batch_size(self) -> int:
        return BATCH_SIZE

    def optimizer(self) -> keras.optimizers.Optimizer:
        return SGD(lr=0.001)

    def loss(self) -> dict:
        return categorical_crossentropy

    def training_metrics(self) -> dict:
        return {"accuracy": categorical_accuracy}

    def validation_metrics(self) -> dict:
        return {"accuracy": categorical_accuracy}
```

files @ OReilly/reproducibility

Exp 30 @ `<cluster_ip>:8080`

Let’s retrain to get a baseline!
In PEDL, for simplicity:

Leslie’s artifacts: Experiment 30 at <cluster_ip>:8080

Let’s rerun her script:

```
pedl e create leslie.yaml
```

from OReilly/reproducibility/model_code

Reminder, to set up you need to:
1) Clone https://github.com/determined-ai/OReilly
2) Install PEDL CLI
3) Point PEDL CLI to cluster address: export PEDL_MASTER=<cluster_ip>
Results...

What could’ve gone wrong?
After digging, we find out...

**Training data**: 30K data points were added to her data directory recently, so she had actually trained on a subset of the data.

**Hyperparameters**: Leslie didn’t use default values, but specified hyperparameters:
- batch size = 64
- dropout = 0.6684
- learning rate = 0.00362
- momentum = 0.953

Can we fix this?

`model_leslie.py`
in OReilly/reproducibility/model_code
Fixes so far

**Training data**: Trained on correct subset of data

**Hyperparameters**: Specified the same hyperparameters at runtime as Leslie

<table>
<thead>
<tr>
<th></th>
<th>Validation Accuracy</th>
<th>Difference from Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>98.75%</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>Test1: code as is</strong></td>
<td>93.50%</td>
<td>5.25%</td>
</tr>
<tr>
<td><strong>Test2: data, hp fixes</strong></td>
<td>98.90%</td>
<td><strong>-0.15%</strong></td>
</tr>
</tbody>
</table>

**Why is this happening?**
Randomness is an intrinsic part of training

- e.g., weight initialization, shuffling and augmentation of datasets, noisy hidden layers (e.g. dropout)

Fix random seeds!

- There are lots of them!
- ML framework dependent
- Must be recorded for reuse

Can we fix this?
Fix random seeds!

- There are lots of them!
- ML framework dependent
- Must be recorded for reuse

Can we fix this?

Setting random seed handled by PEDL

docs.determined.ai -> reference -> experiment configuration

reproducibility:
experiment_seed: 999

in leslie.yaml
Variation across specialized software

• Within versions and across ML frameworks (TF, Keras, PyTorch)
• Underlying libraries (NumPy, cuDNN, CUDA, MKL)

Leverage the power of containerization!

Requires non-trivial engineering infrastructure

Already fixed!
Ugh…Debug…Hardware

Inherent System/Hardware Level Randomness
• non-deterministic GPU operations
• CPU multi-threading

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>Validation Error</th>
<th>Difference from Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td></td>
<td>98.75%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Test1: code as is</td>
<td></td>
<td>93.50%</td>
<td>5.25%</td>
</tr>
<tr>
<td>Test2: data, hp fixes</td>
<td></td>
<td>98.90%</td>
<td>-0.15%</td>
</tr>
<tr>
<td>Test3: +library, random seeds fixes</td>
<td></td>
<td>98.80%</td>
<td>-0.05%</td>
</tr>
</tbody>
</table>

UGH!!!
## Taming hardware non-determinism

**Fundamental issue:** floating point computations in underlying hardware operations
gathering partial computations from separate threads

<table>
<thead>
<tr>
<th>PyTorch</th>
<th>Tensorflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>● torch manual seed</td>
<td>● NGC Tensorflow containers 19.06, 19.07</td>
</tr>
<tr>
<td>● CuDNN backend flags</td>
<td>● Or, with TF 1.14</td>
</tr>
<tr>
<td>cudnn.deterministic</td>
<td>TF_DETERMINISTIC_OPS=1</td>
</tr>
<tr>
<td>cudnn.benchmark</td>
<td>TF_CUDNN_DETERMINISM=1</td>
</tr>
<tr>
<td>Reproducibility not guaranteed if functions</td>
<td>and tfdeterminism patch</td>
</tr>
<tr>
<td>directly call CUDA ops</td>
<td>Imperfect determinism</td>
</tr>
</tbody>
</table>
Feature Request: Support for configuring deterministic options of cudNN Conv routines #18096

Open  yoavz opened this issue on Mar 29, 2018 · 13 comments
At last...we *could* achieve perfect reproducibility!

Requires **CPU-only** training with multi-threading disabled... **SLOW**
What is needed for perfect reproducibility?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version control for model definitions</td>
<td>Track changes in model architecture, optimization algorithm, data preprocessing pipeline</td>
</tr>
<tr>
<td>Metadata capture and storage</td>
<td>Record training + validation metrics, training logs, model hyperparameters</td>
</tr>
<tr>
<td>Dependency management</td>
<td>Ensure ML framework and all dependencies are consistent between runs</td>
</tr>
<tr>
<td>Experiment seed management</td>
<td>Generate the same pseudo-random values every run</td>
</tr>
<tr>
<td>Hardware resource flexibility</td>
<td>Allow users to disable multi-threading and GPU usage, if desired</td>
</tr>
</tbody>
</table>
Schedule

Part I: how DL training scales
1:30-2:20 Introduction
   Challenges when scaling
   Components to a codebase
   CLI install

Part II: topic deep dives
2:20-3:00 Reproducibility
   Conference Tea Break (30 min)
3:30-4:10 Hyperparameter Tuning
   Break (10 min)
4:20-5:00 Model Serving
Hyperparameter Tuning
• What is hyperparameter search?

• Hyperparameter search algorithms
  • Random Search
  • Bayesian Methods
  • Successive Halving Algorithm and Hyperband
What are hyperparameters?
Model space defined by ‘hyperparameters’

- activation function \((a)\)
- # units per layer \((u)\)
- # hidden layers \((h)\)
- regularization \((r)\)
Training a Black-box Solver
Training

Predictive Error
0.058
Everything is a hyperparameter!

- activation function \( a \)
- # units per layer \( u \)
- # hidden layers \( h \)
- regularization \( r \)
- The subset of input features you decide to include in the model.
- How you decide to preprocess your data (e.g. adding random augmentation)
- The seed you are using to shuffle input data.
- The optimization procedure you decide to use.
- The total size of your training data.
- What type of early stopping procedure you decide to use, if any.
Exercise 1: Train a model with fixed hyperparameters

- Navigate to the hyperparameter_tuning folder:
  ```
  $ cd hyperparameter_tuning
  ```

- Edit single.yaml to change the fixed hyperparameters:
  ```
  $ open single.yaml
  ```

- Kick off a single model training job:
  ```
  $ pedl experiment create single.yaml
  ```

Repeat!
Iterative Model Development

1. Train model
2. Analyze Results
3. Modify Hyperparameters

The cycle repeats until the model is optimized.
What is hyperparameter search?
How can we efficiently identify high-quality hyperparameters?

Efficiency $\iff$ resources consumed

Quality $\iff$ predictive error
“One major drawback of current architectures is that they are expensive to train, typically requiring \textit{days to weeks of GPU time}.”

“We report results … corresponding to over \textit{250,000 GPU hours} on the standard WMT English to German translation task”

“Massive Exploration of Neural Machine Translation Architectures” (Britz et. al., 2017)

\textbf{10K Train Models, 29 Compute Years, $200K}
Search Methods: Random / Grid
Classic Methods

**Random**

**Pros:** Simple

**Cons:** Curse of dimensionality for large search spaces

**Grid**
Exercise 2: Random hyperparameter search

$ open random.yaml

rameter_tuning folder:

$ cd hyperparameter_tuning

Edit random.yaml to your desired search space:

Kick off a random hyperparameter search job:

$ pedl experiment create random.yaml

Repeat!
Search Driven Model Development
Configuration Selection Methods

**e.g., Bayesian Optimization**

**Pros:** Better than classical methods in low-dimensional spaces

**Cons:** Curse of dimensionality for large search spaces

Difficult to parallelize
Search Methods: Successive Halving and Hyperband
Downsampling Iterations

Wasted iterations

predictive error

# iterations
Downsampling Iterations

Focus efforts here!
Successive Halving Algorithm
What could go wrong?

Sequences can be **non-monotonic, non-smooth**

How can we “safely” discard a configuration?
What could go wrong?

**Hyperband**: Novel downsampling approach

[AISTATS16, ICLR17, JMLR18]

- ✔ State-of-the-art empirical performance
- ✔ Provably correct
Massively Parallel Hyperparameter Optimization

**Speedups**
- >50x over Random
- 10x over PBT

- ✔ Lower final error
- ✔ Lower variance

![CIFAR10 Using Small Cuda Convnet Model](image)
Exercise 3: Hyperband Search

Navigate to the hyperparameter_tuning folder:

```
$ open adaptive.yaml
```

```
$ cd hyperparameter_tuning
```

Edit adaptive.yaml to your desired search space:

```
$ pedl experiment create adaptive.yaml
```

Kick off an adaptive hyperparameter search job:

Repeat!
Part I: How DL training scales
1:30-2:20
   Introduction
   Challenges when scaling
   Components to a codebase
   CLI install

Part II: Topic deep dives
2:20-3:00 Reproducibility
3:00-3:30 Conference Tea Break (30 mins)
3:30-4:10 Hyperparameter Tuning
4:10-4:20 Break (10 mins)
4:20-5:00 Model Serving
Model Serving and Optimization
What is model serving?
Model Serving Architecture

- model files
- model server
- client
- image
- prediction
Goals In Production Serving

- Low Latency Online Inference
- Experimentation
- Rollout and Rollback
- Dynamic Scaling
- Monitoring
Goals In Production Serving

- Low Latency Online Inference
- Experimentation
- Rollout and Rollback
- Dynamic Scaling
- Monitoring
Goals In Production Serving

- Low Latency Online Inference
- **Experimentation**
- Rollout and Rollback
- Dynamic Scaling
- Monitoring
Goals In Production Serving

- **Low Latency Online Inference**
- **Experimentation**

**Rollout and Rollback**

- **Dynamic Scaling**
- **Monitoring**
Goals In Production Serving

- Low Latency Online Inference
- Experimentation
- Rollout and Rollback
- Dynamic Scaling
- Monitoring
Goals In Production Serving

- Low Latency Online Inference
- Experimentation
- Rollout and Rollback
- Dynamic Scaling
- Monitoring
How can we accomplish these goals?
Model Serving Tools

- NVIDIA
- TensorFlow
- docker
Some Model Deployment Challenges

- Model Size
- Deployment Constraints
- Adaptive Batching
How can we optimize our models for production?
The model you want to train

is not

the model you want to use for inference
Optimization Methods

General Methods
- Distillation
- Pruning
- Folding Batch Normalization
- Quantization

For TensorFlow Models Specifically
- Freezing
- Constant Folding
Optimization Methods

- Distillation
- Pruning
- Folding Batch Normalization
- Quantization

For TensorFlow Models Specifically
- Freezing
- Constant Folding
Model Distillation
Model Pruning

before pruning

pruning synapses

pruning neurons

after pruning
Quantization
Optimization Methods

General Methods
- Distillation
- Pruning
- Folding Batch Normalization
- Quantization

For TensorFlow Models Specifically
- Freezing
- Constant Folding
Optimization Methods

- General Methods
  - Distillation
  - Pruning
  - Folding Batch Normalization
  - Quantization

For TensorFlow Models Specifically

- Freezing
- Constant Folding
In this exercise we will:

1. Train a model on MNIST
2. Convert our model into a servable artifact
3. Serve the model artifact
4. Optimize our artifacts to reduce size and latency
5. Redeploy our newly optimized models
Feedback

How did we do? Link at the bottom of github page https://forms.gle/PBxr9YmtXbDCmt1s9
Thank you.
Bonus: Object Detection
pedl notebook start --config-file model_serving/config.yaml

Navigate to object-detection.ipynb