

PORTING APPLICATIONS

June 11, 2024

Alexander Tsyplikhin

GRAPHCORE

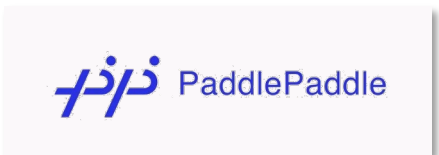
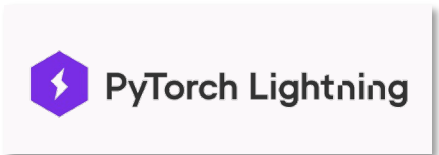
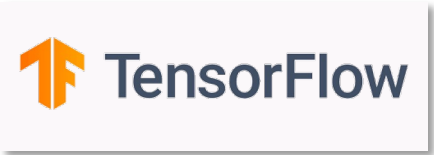


AGENDA

- Porting TensorFlow2/Keras
 - Porting a Keras script, leverage loop on device, replicate and run data-parallel, pipeline
- Porting PyTorch
 - PopTorch example, DataLoader, options to optimize performance

STANDARD ML FRAMEWORK SUPPORT

Develop models using standard high-level frameworks or port existing models



Existing models on alternative platforms

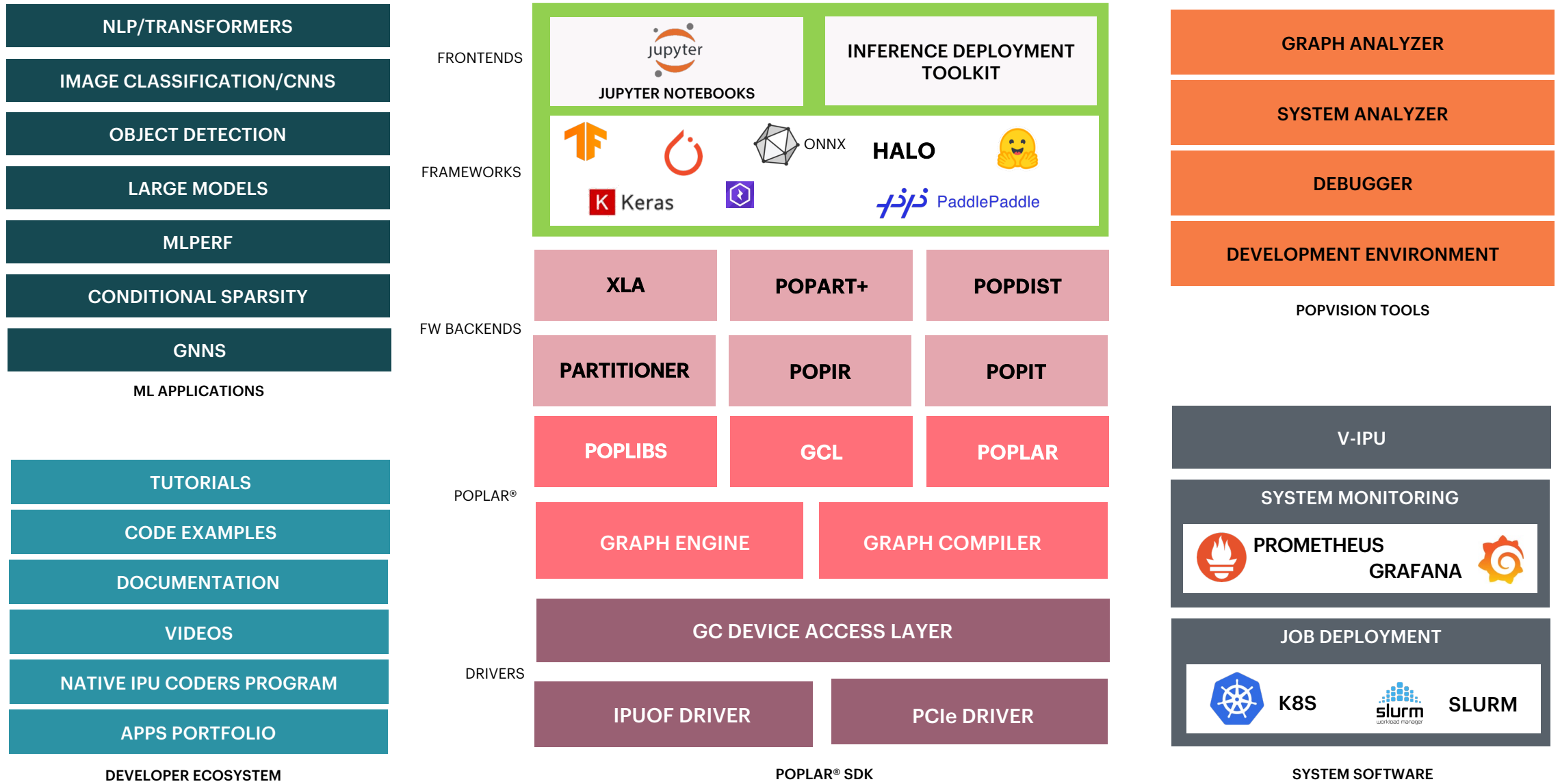


Easy port of high-level framework models



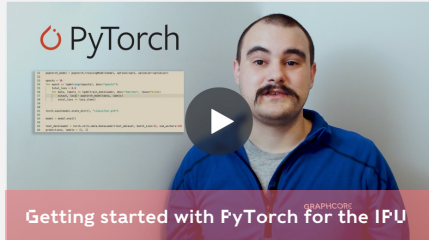
IPU-Processor Platforms

GRAPHCORE SOFTWARE MATURITY



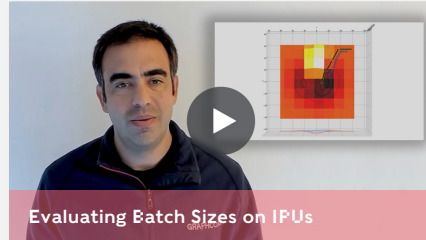
VIDEO + GITHUB TUTORIALS

A comprehensive set of online developer training materials and educational content

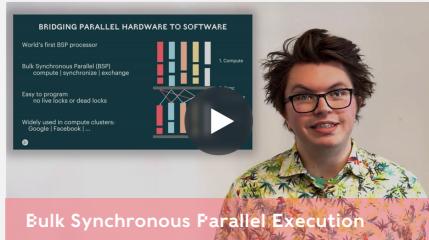


PyTorch

Getting started with PyTorch for the IPU



Evaluating Batch Sizes on IPU



BRIDGING PARALLEL HARDWARE TO SOFTWARE

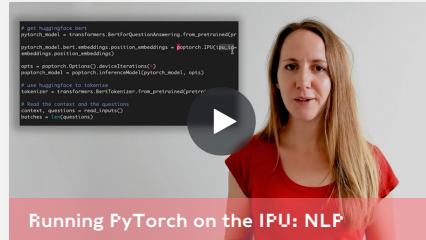
World's first BSP processor

Bulk Synchronous Parallel (BSP) compute | synchronize | exchange

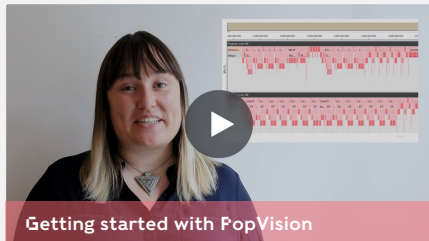
Easy to program, no low-level details

Widely used in compute clusters (Google | Facebook)

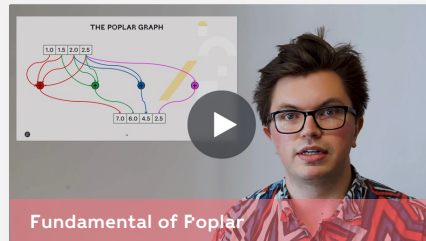
Bulk Synchronous Parallel Execution



Running PyTorch on the IPU: NLP

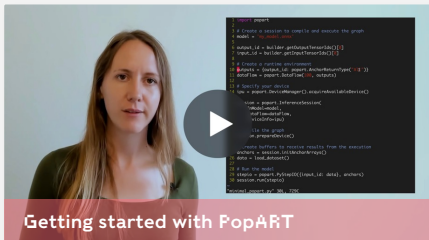


Getting started with PopVision

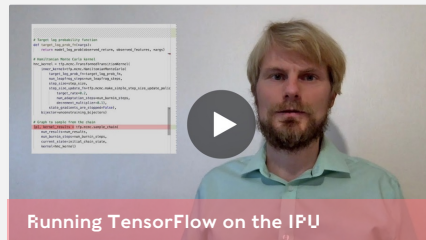


THE POPLAR GRAPH

Fundamental of Poplar



Getting started with PopART



Running TensorFlow on the IPU

TUTORIALS

Learn how to create and run programs using Poplar and PopLibs with our hands-on programming tutorials.

Programs and Variables

Using PopLibs

Writing Vertex Code

Profiling Output

Basic Machine Learning Example

Matrix-Vector Multiplication

Matrix-Vector Multiplication Optimisation

Simple PyTorch for the IPU

NEW

Tutorial 1: programs and variables

Copy the file `tut1_variables/start_here/tut1.cpp` to your working directory and open it in an editor. The file contains the outline of a C++ program including some Poplar library headers and a namespace.

Graphs, variables and programs

All Poplar programs require a `Graph` object to construct the computation graph. Graphs are always created for a specific target (where the target is a description of the hardware being targeted, such as an IPU). To obtain the target we need to choose a device.

The tutorials use a simulated target by default, so will run on any machine even if it has no Graphcore hardware attached. On systems with accelerator hardware, the header file `poplar/DeviceManager.hpp` contains API calls to enumerate and return `Device` objects for the attached hardware.

Simulated devices are created with the `IPUModel` class, which models the functionality of an IPU on the host. The `createDevice` function creates a new virtual device to work with. Once we have this device we can create a `Graph` object to target it.

- Add the following code to the body of `main`:

```
// Create the IPU Model device
IPUModel ipuModel;
Device device = ipuModel.createDevice();
Target target = device.getTarget();

// Create the Graph object
Graph graph(target);
```

Any program running on an IPU needs data to work on. These are defined as variables in the graph.

- Add the following code to create the first variable in the program:

Tutorial 5: a basic machine learning example

This tutorial contains a complete training program that performs a logistic regression on the MNIST data set, using gradient descent. The files for the demo are in `tut5_m1`. There are no coding steps in the tutorial. The task is to understand the code, build it and run it. You can build the code using the supplied makefile.

Before you can run the code you will need to run the `get_mnist.sh` script to download the MNIST data.

The program accepts an optional command line argument to make it use the IPU hardware instead of a simulated IPU.

As you would expect, training is significantly faster on the IPU hardware.

Copyright (c) 2018 Graphcore Ltd. All rights reserved.

TF2/KERAS ON IPU



KERAS ON IPU

IPU optimized Keras `Model` and `Sequential` with the following features:

- On-device training loop for reduction of communication overhead.
- Gradient accumulation for simulating larger batch sizes.
- Automatic data-parallelisation of the model when placed on a multi-IPU device.



LSTM Encoder Decoder

Keras

```
import tensorflow as tf
from tensorflow.keras.layers import *
```

GPU

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train = x_train.astype('float32') / 255.0
y_train = tf.keras.utils.to_categorical(y_train, 10)
ds_train = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(64, drop_remainder=True)
```

```
model = tf.keras.Sequential([
    Conv2D(32, (3, 3), padding='same', input_shape=x_train.shape[1:]),
    Activation('relu'),
    Conv2D(32, (3, 3)),
    Activation('relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Conv2D(64, (3, 3), padding='same'),
    Activation('relu'),
    Conv2D(32, (3, 3)),
    Activation('relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(512),
    Activation('relu'),
    Dropout(0.5),
    Dense(10),
    Activation('softmax')
])
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=tf.optimizers.SGD(learning_rate=0.016),
              metrics=['accuracy'])
```

```
model.fit(ds_train, epochs=40)
```

```
import tensorflow as tf
from tensorflow.keras.layers import *
+ from tensorflow.python import ipu
```

IPU

```
+ cfg = ipu.config.IPUConfig()
+ cfg.auto_select_ipus = 1
+ cfg.configure_ipu_system()
+ with ipu.ipu_strategy.IPUStrategy().scope():
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
    x_train = x_train.astype('float32') / 255.0
    y_train = tf.keras.utils.to_categorical(y_train, 10)
    ds_train = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(64, drop_remainder=True)
```

```
model = tf.keras.Sequential([
    Conv2D(32, (3, 3), padding='same', input_shape=x_train.shape[1:]),
    Activation('relu'),
    Conv2D(32, (3, 3)),
    Activation('relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Conv2D(64, (3, 3), padding='same'),
    Activation('relu'),
    Conv2D(32, (3, 3)),
    Activation('relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(512),
    Activation('relu'),
    Dropout(0.5),
    Dense(10),
    Activation('softmax')
])
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=tf.optimizers.SGD(learning_rate=0.016),
              metrics=['accuracy'])
```

```
model.fit(ds_train, epochs=40)
```


TF2/KERAS TUTORIALS

Sample commands: <https://bit.ly/ALCF2406>

Continued in the repositories below (follow the READMEs)

github.com/graphcore/examples/tree/master/tutorials/tutorials/tensorflow2/keras

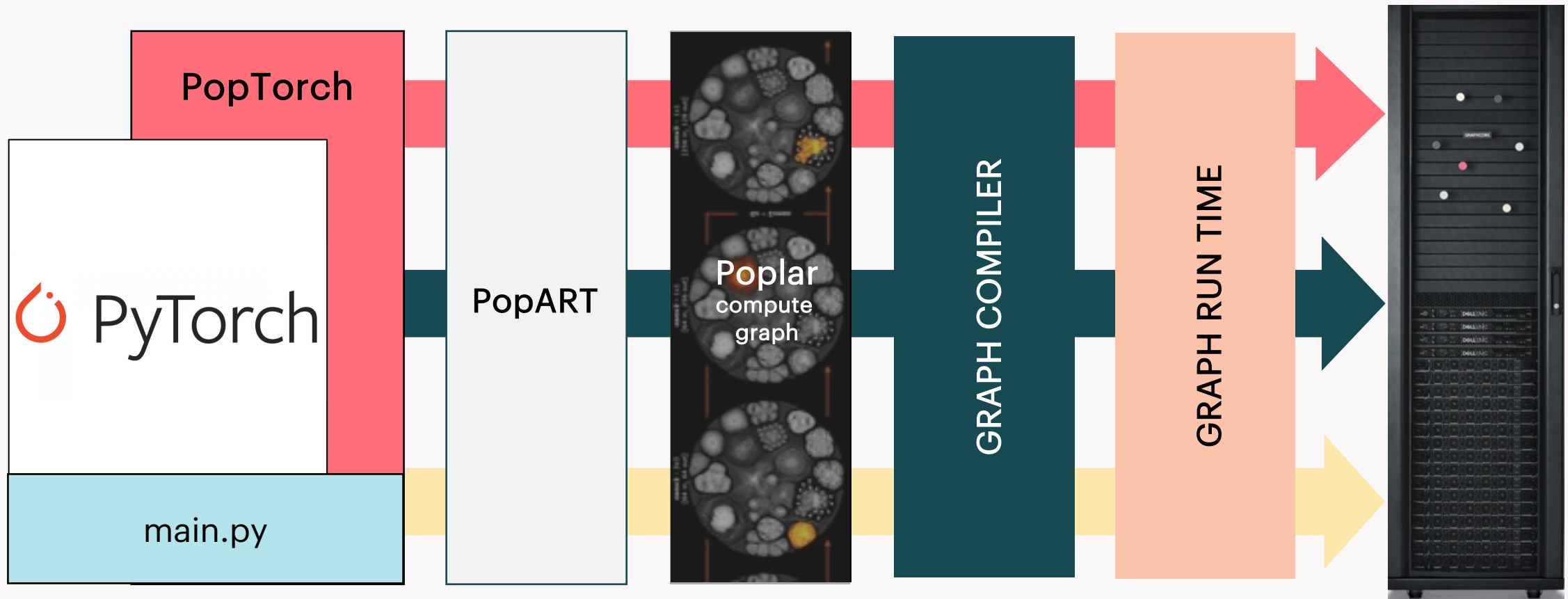


INTRO TO POPTORCH

GRAPHCORE



WHAT IS POPTORCH?



WHAT IS POPTORCH?

- PopTorch is a set of extensions for PyTorch to enable PyTorch models to run on Graphcore's IPU hardware.
- PopTorch supports both inference and training. To run a model on the IPU you wrap your existing PyTorch model in either a PopTorch inference wrapper or a PopTorch training wrapper.
- You can provide further annotations to partition the model across multiple IPUs. Using the user-provided annotations, PopTorch will use PopART to parallelise the model over the given number of IPUs.
- Additional parallelism can be expressed via a replication factor which enables you to data-parallelise the model over more IPUs.

GETTING STARTED: TRAINING A MODEL



TRAINING A MODEL

1. Import packages

PopTorch is a separate package from PyTorch, and must be imported.

2. Load dataset using torchvision.datasets and poptorch.DataLoader

In order to make data loading easier and more efficient, PopTorch offers an extension of `torch.utils.data.DataLoader` class:

`poptorch.DataLoader` class is specialised for the way the underlying PopART framework handles batching of data.

3. Define model and loss function using torch API

The only difference here from pure PyTorch is the loss computation, which has to be part of the forward function. This is to ensure the loss is computed on the IPU and not on the CPU, and to give us as much flexibility as possible when designing more complex loss functions.



TRAINING A MODEL

4. Prepare training

Instantiate compilation and execution options, these are used by PopTorch's wrappers such as `poptorch.DataLoader` and `poptorch.trainingModel`.

5. Train the model

Define the optimizer using PyTorch's API.

Use `poptorch.trainingModel` wrapper, to wrap your PyTorch model. This wrapper will trigger the compilation of our model, using TorchScript, and manage its translation to a program the IPU can run. Then run your training loop.



PyTorch

GPU

```
_, ind = torch.max(predictions, 1)
# provide labels only for samples, where prediction is available (during the training, not
predictions.size()[0]:]
torch.eq(ind, labels)).item() / labels.size(0)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='MNIST training in PopTorch')
    parser.add_argument('--batch-size', type=int, default=8, help='batch size for training (default: 8)')
    parser.add_argument('--test-batch-size', type=int, default=8, help='batch size for testing (default: 8)')
    parser.add_argument('--epochs', type=int, default=10, help='number of epochs to train (default: 10)')
    parser.add_argument('--lr', type=float, default=0.05, help='learning rate (default: 0.05)')

    args = parser.parse_args()

    training_data = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST('mnist_data/', train=True, download=True, transform=None,
        batch_size=args.batch_size, shuffle=True, drop_last=True)

    test_data = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST('mnist_data/', train=False, download=True, transform=None,
        batch_size=args.batch_size, shuffle=True, drop_last=True)

    model = Network()
    training_model = TrainingModelWithLoss(model)
    optimizer=optim.SGD(model.parameters(), lr=args.lr)

    # Run training
    for _ in range(args.epochs):
        for data, labels in training_data:
            preds, losses = training_model(data, labels)
            optimizer.zero_grad()
            losses.backward()
            optimizer.step()

    # Run validation
    sum_acc = 0.0
    with torch.no_grad():
        for data, labels in test_data:
            output = model(data)
            sum_acc += accuracy(output, labels)
    print("Accuracy on test set: {:.2f}%".format(sum_acc / len(test_data)))
```

```
_, ind = torch.max(predictions, 1)
# provide labels only for samples, where prediction is available (during the training, not
predictions.size()[0]:]
accuracy = torch.sum(torch.eq(ind, labels)).item() / labels.size(0)
return accuracy
```

IPU

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='MNIST training in PopTorch')
    parser.add_argument('--batch-size', type=int, default=8, help='batch size for training (default: 8)')
    parser.add_argument('--test-batch-size', type=int, default=8, help='batch size for testing (default: 8)')
    parser.add_argument('--epochs', type=int, default=10, help='number of epochs to train (default: 10)')
    parser.add_argument('--lr', type=float, default=0.05, help='learning rate (default: 0.05)')
    parser.add_argument('--device-iterations', type=int, default=50, help='device iterations (default: 50)')

    args = parser.parse_args()

    + opts = poptorch.Options().deviceIterations(args.device_iterations)
    + training_data = poptorch.DataLoader(opts,
        torchvision.datasets.MNIST('mnist_data/', train=True, download=True, transform=None,
        batch_size=args.batch_size, shuffle=True, drop_last=True)

    + test_data = poptorch.DataLoader(opts,
        torchvision.datasets.MNIST('mnist_data/', train=False, download=True, transform=None,
        batch_size=args.batch_size, shuffle=True, drop_last=True)

    model = Network()
    training_model = TrainingModelWithLoss(model)
    optimizer=optim.SGD(model.parameters(), lr=args.lr)
    + training_model = poptorch.trainingModel(training_model, opts, optimizer=optimizer)
    + inference_model = poptorch.inferenceModel(model)

    # Run training
    for _ in range(args.epochs):
        for data, labels in training_data:
            preds, losses = training_model(data, labels)

    + # Detach the training model so that the same IPU could be used for validation
    + training_model.detachFromDevice()

    # Run validation
    sum_acc = 0.0
    with torch.no_grad():
        for data, labels in test_data:
            output = inference_model(data)
            sum_acc += accuracy(output, labels)
    print("Accuracy on test set: {:.2f}%".format(sum_acc / len(test_data)))
```


POPTORCH TUTORIALS

Continued in the repositories below (follow the READMEs)

github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/basics

github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/mixed_precision

github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/efficient_data_loading

github.com/graphcore/examples/tree/master/tutorials/tutorials/pytorch/pipelining



POPTORCH.OPTIONS

- The compilation and execution on the IPU can be controlled using `poptorch.Options`
- Full list of options available here: <https://docs.graphcore.ai/projects/poptorch-user-guide/en/latest/overview.html#options>

- Some examples:

(i) **`deviceIterations`**

This option specifies the number of batches that is prepared by the host (CPU) for the IPU. The higher this number, the less the IPU has to interact with the CPU, for example to request and wait for data, so that the IPU can loop faster. However, the user will have to wait for the IPU to go over all the iterations before getting the results back. The maximum is the total number of batches in your dataset, and the default value is 1.

(ii) **`replicationFactor`**

This is the number of replicas of a model. We use replicas as an implementation of data parallelism. To achieve the same behavior in pure PyTorch, you'd wrap your model with `torch.nn.DataParallel`, but with PopTorch, this is an option.



USEFUL ENV VARIABLES



LOGGING

Logging messages can be generated when your program runs. This is controlled by the environment variables described below. For more detailed information see the docs:

<https://docs.graphcore.ai/projects/poplar-user-guide/en/latest/env-vars.html>

POPLAR_LOG_LEVEL: Enable logging for Poplar

POPLAR_LOG_DEST: Specify the destination for Poplar logging (“stdout”, “stderr” or a file name)

“OFF”	No logging information. The default.
“ERR”	Only error conditions will be reported.
“WARN”	Warnings when, for example, the software cannot achieve what was requested (for example, if the convolution planner can’t keep to the memory budget, or Poplar has determined that the model won’t fit in memory but the debug.allowOutOfMemory option is enabled).
“INFO”	Very high level information, such as PopLibs function calls.
“DEBUG”	Useful per-graph information.
“TRACE”	The most verbose level. All useful per-tile information.

CREATE EXECUTION PROFILE

```
POPLAR_ENGINE_OPTIONS='{ "autoReport.all": "true", "autoReport.directory": "./report" }'
```

- The PopVision Graph Analyser uses report files generated during compilation and execution by the Poplar SDK.
- These files can be created using POPLAR_ENGINE_OPTIONS.
- In order to capture the reports needed for the PopVision Graph Analyser you only need to set POPLAR_ENGINE_OPTIONS='{ "autoReport.all": "true" }' before you run a program. By default this will enable instrumentation and capture all the required reports to the current working directory.

EXECUTABLE CACHE

If you often run the same models you might want to enable executable caching to save time:

POPTORCH:

- You can do this by either setting the POPTORCH_CACHE_DIR environment variable or by calling `poptorch.Options.enableExecutableCaching`.

TENSORFLOW:

- You can use the flag `--executable_cache_path` to specify a directory where compiled files will be placed. Fused XLA/HLO graphs are hashed with a 64-bit hash and stored in this directory.

Warning

The cache directory might grow large quickly. Poplar doesn't evict old models from the cache and, depending on the number and size of your models and the number of IPUs used, the executables might be quite large.

It is your responsibility to delete the unwanted cache files.

SYNTHETIC-DATA

```
TF_POPLAR_FLAGS= "--use_synthetic_data --synthetic_data_initializer=random"
```

Used for measuring the IPU-only throughput and disregards any host/CPU activity.

GRAPHCORE COMMAND LINE TOOLS

- gc-docker*** Allows you to use IPU devices in Docker containers using the Docker container engine.
- gc-flops*** Allows you to benchmark the number of floating point operations per second on one or more IPU processors.
- gc-info*** Determines what IPU cards are present in the system.
- gc-inventory*** Lists device IDs, physical parameters and firmware version numbers.
- gc-links*** Displays the status and connectivity of each of the IPU-Links that connect IPU's. See also *IPU-Link channel mapping* for connectivity in an IPU Server containing C2 cards.
- gc-monitor*** Monitors IPU activity on shared systems.
- gc-reset*** Resets IPU devices.
- gc-exchangetest*** Allows you to test the internal exchange fabric in an IPU.
- gc-exchangewritetest*** Tests direct writes to the IPU's tile memory via the host.
- gc-gwlinkstraffictest*** Tests GW-Links on multi-rack IPU-POD systems.
- gc-hostsynclatencytest*** Reports the latency of transfers between the host machine and the IPU's (in both directions).
- gc-hosttraffictest*** Allows you to test the data transfer between the host machine and the IPU's (in both directions).
- gc-iputraffictest*** Allows you to test the data transfer between IPU's.
- gc-memorytest*** Tests all the memory in an IPU, reporting any tiles that fail.
- gc-podman*** Allows you to use IPU devices in Docker containers using the Podman container engine.
- gc-powertest*** Tests power consumption and temperature of the IPU processors.

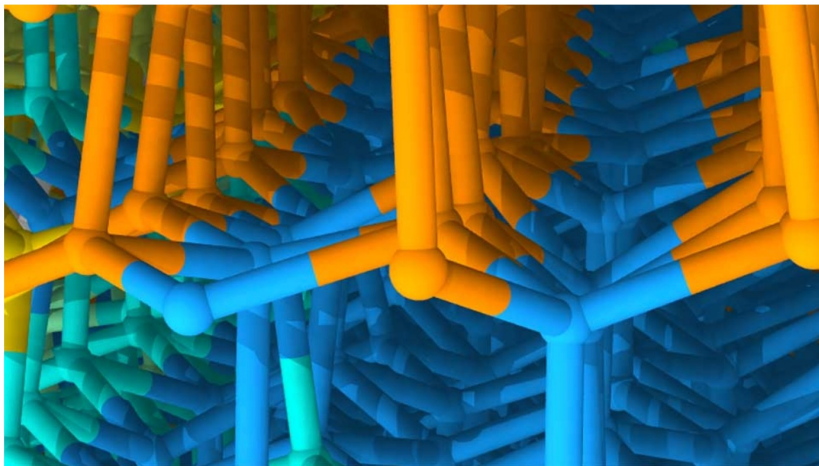


APPLY AND JOIN TODAY



Director's Discretionary Allocation Program

The ALCF Director's Discretionary program provides “start up” awards to researchers working to achieve computational readiness for for a major allocation award.



Molecular dynamics simulations based on machine learning help scientists learn about the movement of the boundary between ice grains (yellow/green/cyan) and the stacking disorder that occurs when hexagonal (orange) and cubic (blue) pieces of ice freeze together. Image: Henry Chan and Subramanian Sankaranarayanan, Argonne National Laboratory

Apply at alcf.anl.gov/science/directors-discretionary-allocation-program

general

charlieb 6:05 AM

Pleased to share with you all some new work from the Graphcore research team! 🎉

Our paper *Unit Scaling* introduces a new method for low-precision number formats, making FP16. We've managed to train BERT in these formats for the first time without loss scaling.

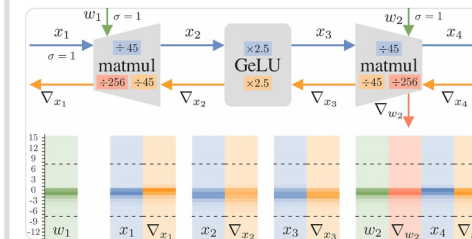
- You can find our blog post here: <https://www.graphcore.ai/posts/simple-fp16-and-fp8-training>
- Paperspace notebook (try it yourself!): <https://ipu.dev/qXfm2a>
- Arxiv paper: <https://arxiv.org/abs/2303.11257>

(& we were also featured on Davis Blalock's popular [ML newsletter](#) this week) (edited)

graphcore.ai

Simple FP16 and FP8 training with unit scaling

Unit Scaling is a new low-precision machine learning method able to train language models in FP16 and FP8 without loss scaling. (69 kB)



arXiv.org

Unit Scaling: Out-of-the-Box Low-Precision Training

We present unit scaling, a paradigm for designing deep learning models that simplifies the use of low-precision number formats. Training in FP16 or the recently proposed FP8 formats offers substantial efficiency gains, but can lack sufficient range for out-of-the-box training. Unit scaling addresses this by introducing a principled approach to model numerics: seeking unit variance of

[Show more](#)



Join at graphcore.ai/join-community

TUESDAY, 11 JUNE



- 1:00 PM** → 1:15 PM **Introduction** ⌚ 15m
- 1:15 PM** → 1:45 PM **Graphcore BowPod64 Hardware** ⌚ 30m
- 1:45 PM** → 2:30 PM **Software Stack: TensorFlow, PyTorch, and Poplar** ⌚ 45m
- 2:30 PM** → 2:45 PM **Break** ⌚ 15m
- 2:45 PM** → 3:15 PM **Porting applications with Poplar** ⌚ 30m
- 3:15 PM** → 4:00 PM **How to use Bow Pod64@ ALCF** ⌚ 45m

WEDNESDAY, 12 JUNE



- 1:00 PM** → 1:45 PM **Deep Dive on Graph neural networks and Large Language Models** ⌚ 45m
- 1:45 PM** → 2:15 PM **Profiling with PopVision** ⌚ 30m
- 2:15 PM** → 2:30 PM **Break** ⌚ 15m
- 2:30 PM** → 3:15 PM **Hands-on session** ⌚ 45m
- 3:15 PM** → 4:00 PM **Best Practices, Q&A** ⌚ 45m



THANK YOU

Alexander Tsyplikhin
alex@graphcore.ai