GRAPHCORE

PopART User Guide

Version latest

Graphcore Ltd

Aug 25, 2021

CONTENTS

1 Introduction 1

2 Importing graphs 2
- 2.1 Creating a session 2
- 2.2 Session control options 3

3 Building graphs in PopART 4
- 3.1 Adding operations to the graph 4
- 3.2 Adding parameters to the graph 5
- 3.3 Setting outputs 5
- 3.4 Setting the IPU number for operations 5

4 Executing graphs 6
- 4.1 Setting input/output data buffers for an execution 6
  - 4.1.1 Retrieving results 7
- 4.2 Selecting a device for execution 7
- 4.3 Executing a session 8
- 4.4 Saving and loading a model 8
- 4.5 Retrieving profiling reports 8
- 4.6 Turning on execution tracing 9
  - 4.6.1 Programming interface 10
  - 4.6.2 Output format 10
- 4.7 Errors 10
  - 4.7.1 Application errors 10
  - 4.7.2 System errors 11

5 Distributed training with Horovod 12
- 5.1 How to modify a PopART program for distributed training 12
- 5.2 Install 13
- 5.3 Configuring and running distributed training 13
- 5.4 Full distributed training example 13

6 Performance optimisation 16
- 6.1 Pipelined execution 16
- 6.2 Graph replication 16
  - 6.2.1 Local replication 17
  - 6.2.2 Global replication 17
- 6.3 Sync configuration 18
  - 6.3.1 Sync patterns 18

7 Supported operators 20
- 7.1 Domain: ai.onnx 20
- 7.2 Domain: ai.graphcore 25

8 Custom operators 27

8.1 Overview . . . 27
8.1.1 Custom op classes . . . 27
8.2 Implementing a custom op . . . 29
8.2.1 The op class . . . 30
8.2.2 The grad op class . . . 31
8.2.3 The opx class . . . 32
8.2.4 The grad opx class . . . 32
8.3 Making the op available to PopART . . . 33
8.3.1 Define the op identifier . . . 33
8.3.2 Define the op creator . . . 33
8.3.3 Define the opx creator . . . 34
8.4 ONNX schema and shape inference . . . 34
8.5 Using the op in a program . . . 35

9 Environment variables **36**
9.1 Logging . . . 36
9.1.1 POPART_LOG_LEVEL . . . 36
9.1.2 POPART_LOG_DEST . . . 36
9.1.3 POPART_LOG_CONFIG . . . 36
9.2 Generating DOT files . . . 37
9.2.1 POPART_DOT_CHECKS . . . 37
9.3 Inspecting the Ir . . . 37
9.3.1 POPART_IR_DUMP . . . 37

10 References **38**

11 Glossary **39**
11.1 Sample . . . 39
11.2 Micro-batch size . . . 39
11.3 Replication factor . . . 39
11.4 Accumulation factor . . . 39
11.5 Batch size . . . 39
11.6 Batches per step . . . 40
11.7 Step size . . . 40
11.8 Input data shape . . . 40

12 Trademarks & copyright **41**

CHAPTER

ONE

INTRODUCTION

The Poplar Advanced Run Time (PopART) is part of the Poplar SDK for implementing and running algorithms on networks of Graphcore IPU processors. It enables you to import models using the Open Neural Network Exchange (ONNX) and run them using the Poplar tools. ONNX is a serialisation format for neural network systems that can be created and read by several frameworks including Caffe2, PyTorch and MXNet.

This document describes the features of PopART. It assumes that you are familiar with machine learning and the ONNX framework.

An overview of the IPU architecture and programming model can be found in the IPU Programmer's Guide. For more information on the Poplar framework refer to the Poplar and PopLibs User Guide.

PopART has three main features:

1) It can import ONNX graphs into a runtime environment (see *Importing graphs*).
2) It provides a simple interface for constructing ONNX graphs without needing a third party framework (described in *Building graphs in PopART*).
3) It runs imported graphs in inference, evaluation or training modes, by building a Poplar engine, connecting data feeds and scheduling the execution of the Engine (see *Executing graphs*).

IPU-specific annotations on ONNX operations allow the provider of the graph to control IPU-specific features, such as mapping an algorithm across multiple IPUs.

PopART has both a C++ API and a Python API. Most of the examples in this document use the Python API.

CHAPTER

TWO

IMPORTING GRAPHS

The PopART `Session` class creates the runtime environment for executing graphs on IPU hardware. It can read an ONNX graph from a serialised ONNX model protobuf (ModelProto), either directly from a file or from memory. A session object can be constructed either as an `InferenceSession` or a `TrainingSession`

Some metadata must be supplied to augment the data present in the ONNX graph in order to run it, as described below.

In the following example of importing a graph for inference, TorchVision is used to create a pre-trained AlexNet graph, with a 4 x 3 x 244 x 244 input. The graph has an ONNX output called `out`, and the `DataFlow` object contains an entry to fetch that anchor.

```
# Copyright (c) 2020 Graphcore Ltd. All rights reserved.
import popart

import torch.onnx
import torchvision

input_ = torch.FloatTensor(torch.randn(4, 3, 224, 224))
model = torchvision.models.alexnet(pretrained=True)

output_name = "output"

torch.onnx.export(model, input_, "alexnet.onnx", output_names=[output_name])

# Create a runtime environment
anchors = {output_name: popart.AnchorReturnType("All")}
dataFlow = popart.DataFlow(100, anchors)
device = popart.DeviceManager().createCpuDevice()

session = popart.InferenceSession("alexnet.onnx", dataFlow, device)
```

The DataFlow object is described in more detail in *Executing graphs*.

2.1 Creating a session

The `Session` class takes the name of a protobuf file, or the protobuf itself. It also takes a `DataFlow` object which has information about how to execute the graph:

- The number of times to conduct a forward pass (and a backward pass, if training) of the graph on the IPU before returning to the host for more data.
- The names of the tensors in the graph used to return the results to the host.

In some ONNX graphs, the sizes of input tensors might not be specified. In this case, the `inputShapeInfo` parameter can be used to specify the input shapes. The Poplar framework uses statically allocated memory buffers and so it needs to know the size of tensors before the compilation.

The `patterns` parameter allows the user to select a set of graph transformation patterns which will be applied to the graph. Without this parameter, a default set of optimisation transformations will be applied.

Other parameters to the `Session` object are used when you are training the network instead of performing inference. They describe the types of loss to apply to the network and the optimiser to use.

An example of creating a session object from an ONNX model is shown below.

```
# Copyright (c) 2020 Graphcore Ltd. All rights reserved.
import popart

import torch.onnx
import torchvision

input_ = torch.FloatTensor(torch.randn(4, 3, 224, 224))
model = torchvision.models.alexnet(pretrained=False)

output_name = "output"

torch.onnx.export(model, input_, "alexnet.onnx", output_names=[output_name])

# Create a runtime environment
anchors = {output_name: popart.AnchorReturnType("All")}
dataFlow = popart.DataFlow(100, anchors)

# Append an Nll loss operation to the model
builder = popart.Builder("alexnet.onnx")
labels = builder.addInputTensor("INT32", [4])
nlll = builder.aiGraphcore.nllloss([output_name, labels])

optimizer = popart.ConstSGD(0.001)

# Run session on CPU
device = popart.DeviceManager().createCpuDevice()
session = popart.TrainingSession(builder.getModelProto(),
                                 deviceInfo=device,
                                 dataFlow=dataFlow,
                                 loss=nlll,
                                 optimizer=optimizer)
```

In this example, when the `Session` object is asked to train the graph, an Nll loss node will be added to the end of the graph, and a `ConstSGD` optimiser will be used to optimise the parameters in the network.

2.2 Session control options

The `userOptions` parameter passes options to the session. The available options are listed in the PopART C++ API Reference. As well as options to control specific features of the PopART session, there are also some that allow you to pass options to the underlying Poplar functions:

- `engineOptions` passes options to the Poplar `Engine` object created to run the graph.
- `convolutionOptions` passes options to the PopLibs convolution functions.
- `reportOptions` Controls the instrumentation and generation of profiling information.

See *Retrieving profiling reports* for examples of using some of these options.

Full details of the Poplar options can be found in the Poplar and PopLibs API Reference.

CHAPTER

THREE

BUILDING GRAPHS IN POPART

PopART has a `Builder` class for constructing ONNX graphs without needing a third party framework.

In the example below, a simple addition is prepared for execution. The steps involved are described in the following sections and in *Executing graphs*.

```
import popart

builder = popart.Builder()

# Build a simple graph
i1 = builder.addInputTensor(popart.TensorInfo("FLOAT", [1, 2, 32, 32]))
i2 = builder.addInputTensor(popart.TensorInfo("FLOAT", [1, 2, 32, 32]))

o = builder.aiOnnx.add([i1, i2])

builder.addOutputTensor(o)

# Get the ONNX protobuf from the builder to pass to the Session
proto = builder.getModelProto()

# Create a runtime environment
anchors = {o : popart.AnchorReturnType("ALL")}
dataFlow = popart.DataFlow(1, anchors)
device = popart.DeviceManager().createCpuDevice()

# Create the session from the graph, data feed and device information
session = popart.InferenceSession(proto, dataFlow, device)
```

The DataFlow object is described in more detail in *Executing graphs*.

3.1 Adding operations to the graph

The builder adds operations to the graph by calling one of the many operation methods. Each of these methods has a common signature. For example, `relu` will add an ONNX Relu operation to the graph:

```
output = builder.aiOnnx.relu([input], "debug-name")
```

They take a list of arguments which are the input tensor names, and an optional string to assign to the node. This name is passed to the Poplar nodes and used in debugging and profiling reports.

The operation method returns the name of the tensor that is an output of the newly added node.

In some cases other arguments are required, for instance:

```
output = builder.aiOnnx.gather(['input', 'indices'], axis=1, debugContext="My-Gather")
```

3.2 Adding parameters to the graph

Parameters, for instance the weights of a convolution, are represented as initialised inputs to the graph. They can be added with the `addInitializedInputTensor` method:

```
w_data = np.random.rand(64, 4, 3, 3).astype(np.float16)
w1 = builder.addInitializedInputTensor(w_data)
```

3.3 Setting outputs

The outputs of the graph should be marked appropriately, using the `addOutputTensor` method:

```
builder.addOutputTensor(output)
```

3.4 Setting the IPU number for operations

When creating a graph which will run on a multiple IPU system, nodes need to be marked with an annotation to describe which IPU they will run upon.

For instance, to place a specific convolution onto IPU 1:

```
we = builder.addInitializedInputTensor(np.zeros([32, 4, 3, 3], np.float16))
bi = builder.addInitializedInputTensor(np.zeros([32], np.float16))
o = builder.aiOnnx.conv([x, we, bi],
                        dilations=[1, 1],
                        pads=[1, 1, 1, 1],
                        strides=[1, 1])
# place operation on IPU 1
builder.virtualGraph(o, 1)
```

A context manager is available for placing multiple operations together onto a specific IPU:

```
builder = popart.Builder()

i1 = builder.addInputTensor(popart.TensorInfo("FLOAT", [1]))
i2 = builder.addInputTensor(popart.TensorInfo("FLOAT", [1]))
i3 = builder.addInputTensor(popart.TensorInfo("FLOAT", [1]))
i4 = builder.addInputTensor(popart.TensorInfo("FLOAT", [1]))

# place two add operations on IPU 0
with builder.virtualGraph(0):
    o1 = builder.aiOnnx.add([i1, i2])
    o2 = builder.aiOnnx.add([i3, i4])

# place one add operation on IPU 1
with builder.virtualGraph(1):
    o = builder.aiOnnx.add([o1, o2])
```

Alternatively, for automatic placement of nodes on available IPUs, set the session option `virtualGraphMode` to `popart.VirtualGraphMode.Auto`. See `SessionOptions` in the PopART C++ API Reference.

CHAPTER

FOUR

EXECUTING GRAPHS

The Session class is used to run graphs on an IPU device. Before the graph can be run, the way in which data will be transferred to and from the IPU must be specified. Then an IPU device can be selected to execute the graph.

4.1 Setting input/output data buffers for an execution

Input and output data is passed to and from a Session object via IStepIO objects. Each call to session.run(...) takes such a IStepIO object. For every input tensor, this object contains a number of buffers that the session can read input data from. And for every anchored tensor, it contains a number of buffers to write output data to. There is more information about anchors in Section 4.1.1, Retrieving results.

The number and shape of these buffers depend on a variety of factors including

1) the shape of associated tensor in the ONNX model
2) the DataFlow configuration (see next section) as passed to the Session object's constructor
3) the number of local replicas, and
4) the accumulation factor.

This is explained in more detail in the C++ API documentation under the IStepIO class (for inputs) and under the DataFlow class (for outputs).

When using Python, the PyStepIO class is a convenient way of providing a session with input and output buffers. For both input and output, this class takes a dictionary with tensor names as keys and Python (or Numpy) arrays as values. PopART splits up these arrays internally to provide the Session object with the buffers that it needs.

Note that Session has a convenience method, initAnchorArrays, that can create the output dictionary that PyStepIO needs automatically.

An alternative to PyStepIO is the PyStepIOCallback class, which you can use to implement IStepIO by means of a callback mechanism.

The C++ equivalents of PyStepIO and PyStepIOCallback are StepIO and StepIOCallback, respectively.

Below is an example of how to use PyStepIO:

```
# Create buffers to receive results from the execution
anchors = session.initAnchorArrays()

# Generate some random input data
data_a = np.random.rand(1).astype(np.float32)
data_b = np.random.rand(1).astype(np.float32)

stepio = popart.PyStepIO({'a': data_a, 'b': data_b}, anchors)

session.run(stepio)
```

If there are any pre-defined inputs (such as weights or biases) in the graph then they will not be specified in the IStepIO object. However, before executing the graph, they will need to the copied to the hardware. If there are any optimiser-specific parameters which can be modified, then these must be written to the device. For example:

```
session.weightsFromHost()
```

These can also be updated between executions.

```
# Update learning rate parameter between training steps
stepLr = learningRate[step]
session.updateOptimizerFromHost(popart.SGD(stepLr))
```

4.1.1 Retrieving results

The `DataFlow` class describes how to execute the graph. When you construct a `DataFlow` class it expects two parameters:

```
df = popart.DataFlow(1, {o: popart.AnchorReturnType("ALL")})
```

The first argument is `batchesPerStep`. This is the the number of batches a call to `session.run(...)` executes for before returning control to the caller.

The second argument is a Python dictionary with keys that are the names of the tensors to retrieve from the model via the `IStepIO` object. We call such tensors *anchors*. The associated values are `AnchorReturnType` values, which are one of:

- `popart.AnchorReturnType("ALL")`: a vector of results is returned, one for each iteration of the graph.
- `popart.AnchorReturnType("EVERYN", N)`: a vector containing the tensor, but only for iterations which are divisible by `N`.
- `popart.AnchorReturnType("FINAL")`: the value of the tensor on the final iteration through the graph.
- `popart.AnchorReturnType("SUM")`: the sum of the values of the tensor from each iteration through the graph.

The effect of this setting on the number of output buffers is explained in more detail in our C++ API documentation documentation (see documentation for the `DataFlow` class).

Note that the set of tensors that are *anchored* may differ from those tensors marked as ONNX model *outputs* (via `builder.addOutputTensor(...)`). That is, a model's output tensor need not be anchored and an anchored tensor need not be a model output – any tensor can be anchored. It is the anchored tensors that are considered 'output' in the context of a `IStepIO` object.

4.2 Selecting a device for execution

The device manager allows the selection of an IPU configuration for executing the session. The device must be passed into the session constructor.

```
df = popart.DataFlow(1, {o: popart.AnchorReturnType("ALL")})
device = popart.DeviceManager().createCpuDevice()
s = popart.InferenceSession("onnx.pb", deviceInfo=device, dataFlow=df)
```

The device manager can enumerate the available devices with the `enumerateDevices` method. The `acquireAvailableDevice` method will acquire the next available device. The first parameter specifies how many IPUs to acquire.

```
# Acquire a two-IPU pair
dev = popart.DeviceManager().acquireAvailableDevice(2)
```

Using `acquireDeviceById` will select a device from the list of IPU configurations, as given by the `enumerateDevices` method, or by the `gc-info` command-line tool. This may be a single IPU or a group of IPUs.

```
# Acquire IPU configuration 5
dev = popart.DeviceManager().acquireDeviceById(5)
```

The method createIpuModelDevice is used to create a Poplar software emulation of an IPU device. Similarly, the method createCpuDevice creates a simple Poplar CPU backend. See the PopART C++ API Reference for details.

By default the functions acquireAvailableDevice and acquireDeviceById will attach the device immediately to the running process. You can pass the DeviceConnectionType.OnDemand option to the DeviceManager to defer the device attachment until it is required by PopART.

```
# Acquire four IPUs on demand
connectionType=popart.DeviceConnectionType.OnDemand
dev = popart.DeviceManager().acquireAvailableDevice(4, connectionType=connectionType)
```

4.3 Executing a session

Once the device has been selected, the graph can be compiled for it, and loaded into the hardware. The prepareDevice method is used for this:

```
session.prepareDevice()
```

To execute the session you need to call the session's run method.

```
session.run(stepio)
```

If the session is created for inference, the user is responsible for ensuring that the forward graph finishes with the appropriate operation for an inference. If losses are provided to the inference session the forward pass and the losses will be executed, and the final loss value will be returned.

If the session was created for training, any pre-initialised parameters will be updated to reflect the changes made to them by the optimiser.

4.4 Saving and loading a model

The method modelToHost writes a model with updated weights to the specified file.

```
session.modelToHost("trained_model.onnx")
```

Note that if you plan to run your program in multiple processes simultaneously, you should avoid possible race conditions by writing to different files, for example by using temporary files.

A file of saved parameters, for example from an earlier execution session, can be loaded into the current session.

```
session.resetHostWeights("test.onnx")
session.weightsFromHost()
```

4.5 Retrieving profiling reports

Poplar can provide profiling information on the compilation and execution of the graph. Profiling is not enabled by default.

To get profiling reports in PopART, you will need to enable profiling in the Poplar engine. For example:

```
opts = popart.SessionOptions()
opts.engineOptions = {"autoReport.all": "true"}
```

You can also control what information is included in the profiling report:

```
opts.reportOptions = {"showExecutionSteps": "true"}
```

There are two method functions of the session object to access the profiling information:

- `getSummaryReport` retrieves a text summary of the compilation and execution of the graph.
- `getReport` returns a libpva `Report` object containing details of the compilation and execution of the graph. the graph

If profiling is not enabled, then the summary report will say 'Execution profiling not enabled' and the report will contain no information in the execution.

For more information on the libpva Report, see the pva user guide and api document: * Libpva User Guide * Libpva C++ API Reference * Libpva Python API Reference.

For more information on profiling control and the information returned by these functions, see the Profiling chapter of the Poplar and PopLibs User Guide.

4.6 Turning on execution tracing

PopART contains an internal logging system that can show the progress of graph compilation and execution.

Logging information is generated from the following modules:

popart	Generic PopART module, if no module specified
session	The ONNX session (the PopART API)
ir	The intermediate representation
devicex	The Poplar backend
transform	The transform module
pattern	The pattern module
builder	The builder module
op	The op module
opx	The opx module
ces	The constant expression module
python	The Python module
none	An unidentified module

The logging levels, in decreasing verbosity, are shown below.

TRACE	The highest level, shows the order of method calls
DEBUG	
INFO	
WARN	Warnings
ERR	Errors
CRITICAL	Only critical errors
OFF	No logging

The default is "OFF". You can change this, and where the logging information is written to, by setting environment variables, see *Environment variables*.

4.6.1 Programming interface

You can also control the logging level for each module in your program.

For example, in Python:

```
# Set all modules to DEBUG level
popart.getLogger().setLevel("DEBUG")
# Turn off logging for the session module
popart.getLogger("session").setLevel("OFF")
```

And in C++:

```
// Set all modules to DEBUG level
popart::logger::setLevel("popart", "DEBUG")
// Turn off logging for the session module
popart::logger::setLevel("session", "OFF")
```

4.6.2 Output format

The information is output in the following format:

```
[<timestamp>] [<module>] [<level>] <logging string>
```

For example:

```
[2019-10-16 13:55:05.359] [popart:devicex] [debug] Creating poplar::Tensor 1
[2019-10-16 13:55:05.359] [popart:devicex] [debug] Creating host-to-device FIFO 1
[2019-10-16 13:55:05.359] [popart:devicex] [debug] Creating device-to-host FIFO 1
```

4.7 Errors

The full hierarchy of error that can be thrown from a popart program is:

```
popart_exception
  popart_internal_exception
  popart_runtime_error
poplibs_exception
poplar_exception
  poplar_runtime_error
    poplar_application_runtime_error
    poplar_system_runtime_error
      poplar_recoverable_runtime_error
      poplar_unrecoverable_runtime_error
      poplar_unknown_runtime_error
```

4.7.1 Application errors

Application errors will be due to a bug in either the users code or in the framework. These are:

```
popart.popart_exception
popart.popart_internal_exception
popart.popart_runtime_error
popart.poplibs_exception
popart.poplar_application_runtime_error
```

4.7.2 System errors

These are:

```
popart.poplar_recoverable_runtime_error
popart.poplar_unrecoverable_runtime_error
popart.poplar_unknown_runtime_error
```

An instance of a `poplar_recoverable_error` has an attribute `recoveryAction` which contains the action required to recover from this error. This will be one of the values:

```
.popart.RecoveryAction.IPU_RESET
.popart.RecoveryAction.PARTITION_RESET
.popart.RecoveryAction.POWER_CYCLE
```

A `poplar_unrecoverable_error` suggests that the user needs to contact Graphcore support and that this issue could either be an SDK bug or a machine issue.

An `unknown_runtime_error` could be either recoverable or unrecoverable, but there is not enough information to know for sure. In this instance, the 3 recovery options (IPU_RESET, PARTITION_RESET, and POWER_CYCLE) should be tried and if none resolve the issue, then Graphcore support should be contacted.

CHAPTER

FIVE

DISTRIBUTED TRAINING WITH HOROVOD

In order to scale out training with PopART across multiple machines we use Horovod to setup and run collective operations. There is support for the `Broadcast` and `AllReduce` collective operations. The `Broadcast` operation is typically run at the start of a training to initialise the weights to have the same values across the instances. Gradients produced during the backwards pass will be aggregated and averaged across the instances by running the `AllReduce` operation. This ensures that each rank applies the same gradients to its weights during the weight update step.

5.1 How to modify a PopART program for distributed training

Import the Horovod PopART extension:

```
import horovod.popart as hvd
```

Enable the `hostAllReduce` PopART session option:

```
userOpts = popart.SessionOptions()

# Enable host side AllReduce operations in the graph
userOpts.hostAllReduce = True
```

Initialise the Horovod runtime:

```
hvd.init()
```

Initialise the Horovod `DistributedOptimizer` object. The constructor takes the PopART optimiser, training session and session options objects as arguments. The `DistributedOptimizer` object will add operations to copy gradients into and out of the IPU and run the Horovod `AllReduce` operation:

```
distributed_optimizer = hvd.DistributedOptimizer(optimizer, training.session, userOpts)
```

Insert the all reduce operation:

```
distributed_optimizer.insert_host_allreduce()
```

Broadcast the initial weights from the rank zero process to the other PopART instances:

```
hvd.broadcast_weights(training.session, root_rank=0)
```

5.2 Install

The Horovod PopART extension Python wheel can be found in the Poplar SDK downloaded from https://downloads.graphcore.ai/. System prerequisites for installing the Horovod PopART extension can be found here: Horovod install.

5.3 Configuring and running distributed training

Running distributed training with the Horovod PopART extension can be done in the same way as with other frameworks. For instance, running distributed training across two processes on the same machine can be done with the following command:

```
$ horovodrun -np 2 -H localhost:2 python train.py
```

Alternatively we can use the Gloo backend for the collective operations as shown below:

```
$ horovodrun --gloo -np 2 -H localhost:2 python train.py
```

Additional documentation on flags that can be passed to `horovodrun` can be found here: Horovod documentation.

5.4 Full distributed training example

A small example illustrating how to use Horovod with PopART:

```
import numpy as np
import os
from collections import namedtuple

# import the PopART Horovod extension
import horovod.popart as hvd
import popart

Session = namedtuple('Session', ['session', 'anchors'])
batch_size = 1
IN_SHAPE = 784
OUT_SHAPE = 10


def create_model():
    builder = popart.Builder()
    dtype = np.float32

    np.random.seed(42)
    input_shape = popart.TensorInfo(dtype, [batch_size, IN_SHAPE])
    x = builder.addInputTensor(input_shape)
    init_weights = np.random.normal(0, 1, [IN_SHAPE, OUT_SHAPE]).astype(dtype)
    w = builder.addInitializedInputTensor(init_weights)
    init_biases = np.random.normal(0, 1, [OUT_SHAPE]).astype(dtype)
    b = builder.addInitializedInputTensor(init_biases)
    h = builder.aiOnnx.matmul([x, w])
    a = builder.aiOnnx.add([h, b])

    output = a
    probs = builder.aiOnnx.softmax([output])
    label_shape = popart.TensorInfo("INT32", [batch_size])
    label = builder.addInputTensor(label_shape)
    nll = builder.aiGraphcore.nllloss([output, label])

    proto = builder.getModelProto()

    return builder, proto, x, label, output, nll
```

(continues on next page)

(continued from previous page)

```
def get_device(simulation=True):
    num_ipus = 1
    deviceManager = popart.DeviceManager()
    if simulation:
        print("Creating ipu sim")
        ipu_options = {
            "compileIPUCode": True,
            'numIPUs': num_ipus,
            "tilesPerIPU": 1216
        }
        device = deviceManager.createIpuModelDevice(ipu_options)
        if device is None:
            raise OSError("Failed to acquire IPU.")
    else:
        print("Aquiring IPU")
        device = deviceManager.acquireAvailableDevice(num_ipus)
        if device is None:
            raise OSError("Failed to acquire IPU.")
        else:
            print("Acquired IPU: {}".format(device))

    return device


def init_session(proto, loss, dataFlow, userOpts, device):
    # Create a session to compile and execute the graph
    optimizer = popart.SGD({"defaultLearningRate": (0.1, False)})
    session = popart.TrainingSession(fnModel=proto,
                                     loss=loss,
                                     deviceInfo=device,
                                     optimizer=optimizer,
                                     dataFlow=dataFlow,
                                     userOptions=userOpts)

    session.prepareDevice()
    session.setRandomSeed(42)

    # Create buffers to receive results from the execution
    anchors = session.initAnchorArrays()

    return Session(session, anchors), optimizer


def train():
    builder, proto, data_in, labels_in, output, loss = create_model()

    batches_per_step = 32
    anchor_desc = {
        output: popart.AnchorReturnType("All"),
        loss: popart.AnchorReturnType("All")
    }
    dataFlow = popart.DataFlow(batches_per_step, anchor_desc)

    userOpts = popart.SessionOptions()
    device = get_device()

    # Enable host side AllReduce operations in the graph
    userOpts.hostAllReduce = True
    training, optimizer = init_session(proto, loss, dataFlow, userOpts, device)
    if userOpts.hostAllReduce:
        hvd.init()

        distributed_optimizer = hvd.DistributedOptimizer(
            optimizer, training.session, userOpts)
        distributed_optimizer.insert_host_allreduce()

        # Broadcast weights to all the other processes
        hvd.broadcast_weights(training.session, root_rank=0)
```

(continues on next page)

(continued from previous page)

```
    training.session.weightsFromHost()

    # Synthetic data
    data = np.random.normal(size=(batches_per_step, batch_size, 784)).astype(
        np.float32)
    labels = np.zeros((batches_per_step, batch_size, 1)).astype(np.int32)

    num_training_steps = 10

    for _ in range(num_training_steps):
        stepio = popart.PyStepIO({
            data_in: data,
            labels_in: labels
        }, training.anchors)
        training.session.run(stepio)


train()
```

A more comprehensive example on a real dataset can be found in https://github.com/graphcore/tutorials/tree/sdk-release-2.2/feature_examples/popart/distributed_training/horovod.

CHAPTER

SIX

PERFORMANCE OPTIMISATION

6.1 Pipelined execution

Pipelining is a feature for optimising utilization of a multi-IPU system by parallelizing the execution of model partitions, with each partition operating on a separate mini-batch of data. We refer to these partitions here as pipeline stages.

You can split a model into pipeline stages by annotating operations in the ONNX model using the `Builder` class. There are two APIs. For instance, to place a specific convolution onto pipeline stage 1:

```
o = builder.aiOnnx.conv([x, w])
builder.pipelineStage(o, 1)
```

Or using the context manager:

```
with builder.pipelineStage(1):
    o = builder.aiOnnx.conv([x, w])
```

Alternatively, if you have annotated the operations with `VirtualGraph` attributes, then you can defer annotating pipeline stage attributes to the `Session` constructor. However, it is recommended that you profile the model to choose a partitioning with the optimal utilization.

You can enable pipelined execution by setting the session option `enablePipelining` to `True`. See `SessionOptions` in the PopART C++ API Reference.

Note that, by default, pipelining a training model with variable tensors stored over different pipeline stages results in 'stale weights' (see Zhang et al., arXiv:1912.12675). This can be avoided by enabling gradient accumulation. In this case, the pipeline is flushed before the weight update applies the accumulated gradients.

6.2 Graph replication

PopART has the ability to run multiple copies of your model in parallel on distinct sets of IPUs. This is called *graph replication*. Informally, replication is a means of parallelising your inference or training workloads.

When training, weight updates are coordinated between replicas to ensure replicas benefit from each other's weight updates. A reduction is applied on the weight updates across replicas according to the `ReductionType` specified by the `accumulationAndReplicationReductionType` session option. The reductions involve some communication between replicas. This communication is managed by PopART.

When you use replication, PopART also manages the splitting and distribution of input data, making sure the data specified in the `StepIO` is split evenly between replicas. This does mean you need to provide enough input data to satisfy all (local) replicas.

There are two tiers of replication available in PopART, *local* and *global*, each of which we will describe below.

Note that replication is not supported on IPU model targets.

6.2.1 Local replication

Local replications are replications managed by a single PopART process. This means local replication is limited to those IPUs that are accessible to the host machine that PopART is running on. To enable local replication, set session option `enableReplicatedGraphs` to `True` and set `replicatedGraphCount` to the number of times you want to replicate your model. For example, to replicate a model twice, pass the following session options to your session:

```
opts = popart.SessionOptions()
opts.enableReplicatedGraphs = True
opts.replicatedGraphCount = 2
```

Note that if one replica of your model uses, say, 3 IPUs then with a `replicatedGraphCount` of 2 you will need 6 IPUs to run both replicas. Also, you will need to provide twice the volume of input data. The data returned for each anchor will include a local replication dimension for all values of `AnchorReturnType`.

More details on the expected shapes of input and output data (for a given set of session options) can be found in the API documentation of the `IStepIO` and `DataFlow` classes, respectively.

6.2.2 Global replication

In addition to local replication, it is possible for multiple PopART processes to work together using *global replication*. With this option, as the PopART processes may run on separate hosts, you are not limited to using only the IPUs that are available to a single host. It is also possible to combine local and global replication.

To enable global replication, set `enableDistributedReplicatedGraphs` to `True` and set `globalReplicationFactor` to the desired total number of replications (*including* local replication). Finally, `globalReplicaOffset` must be set to a different offset for each PopART processes involved, using offsets starting from 0 and incrementing by the local replication factor for each process.

For example, if the local replication factor is 2 and we want to replicate this over four PopART processes then we need to configure a global replication factor of 8. We then expect the `globalReplicaOffset` in the respective PopART processes to be set to 0, 2, 4 and 6, respectively, going up in increments equivalent to the local replication factor. As an example, the configuration of the PopART session on the second host is shown below:

```
opts = popart.SessionOptions()
# Local replication settings.
opts.enableReplicatedGraphs = True
opts.replicatedGraphCount = 2
# Global replication settings.
opts.enableDistributedReplicatedGraphs = True
opts.globalReplicationFactor = 8
opts.globalReplicaOffset = 2 # <-- Different offset for each PopART instance
```

Note that when local and global replication are used in tandem the data provided to each PopART instance (in the `IStepIO` instance passed to `Session::run`) should contain only the data required for the local replicas. Moreover, the output anchors will also only contain the output data for the local replicas. Essentially, input and output data shapes are unaffected by global replication settings.

More details on the input and output shapes can be found in the API documentation of the `IStepIO` and `DataFlow` classes, respectively.

6.3 Sync configuration

In a multi-IPU system, synchronisation (sync) signals are used to ensure that IPUs are ready to exchange data and that data exchange is complete. These sync signals are also used to synchronise host transfers and access to remote buffers.

Each IPU can be allocated to one or more "sync groups". At a synchronization point, all the IPUs in a sync group will wait until all the other IPUs in the group are ready.

Sync groups can be used to to allow subsets of IPUs to overlap their operations. For example, one sync group can be performing data transfers to or from the host, while another group is processing a previous batch of data.

You can configure the sync groups using the PopART `syncPatterns` option when creating a device.

For example, the following code shows how to set the sync configuration to "ping-pong" mode.

```
sync_pattern = popart.SyncPattern.Full
if args.execution_mode == "PHASED":
    sync_pattern = popart.SyncPattern.ReplicaAndLadder
device = popart.DeviceManager().acquireAvailableDevice(
    request_ipus,
    pattern=sync_pattern)
```

6.3.1 Sync patterns

There are three sync patterns available. These control how the IPUs are allocated to the two sync groups, GS1 and GS2.

The sync patterns are described with reference to the diagram below, which shows four IPUs: A, B, C and D.

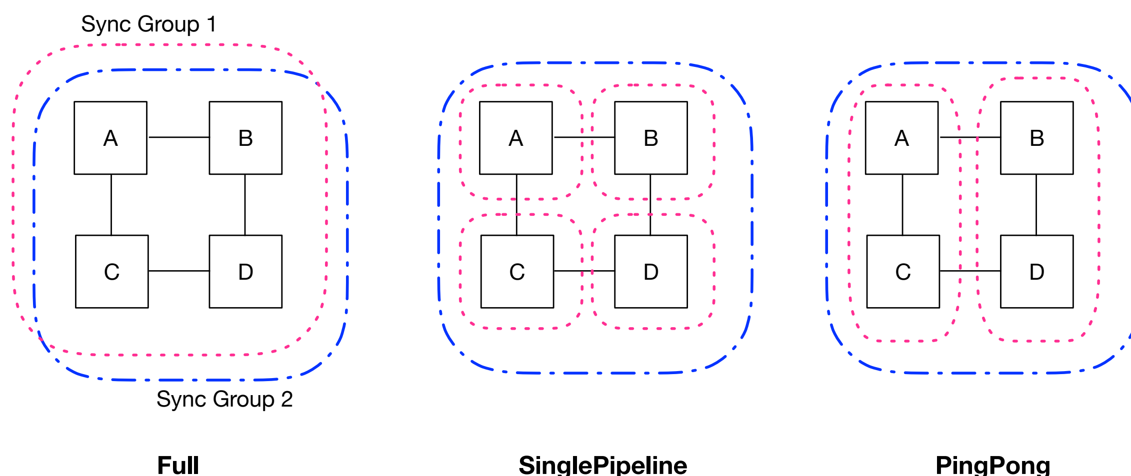

Fig. 6.1: Sync patterns

- **Full:** All four IPUs are in both sync groups. Any communication between the IPUs or with the host, will require all IPUs to synchronise.
- **SinglePipeline:** One sync group contains all four of the IPUs. So any communication using that sync group will synchronise all the IPUs.

  The other sync group is used separately by each IPU. This means that they can each sync with the host independently, without syncing with each other. This allows any IPU to be doing host IO, for example, while others are processing data.
- **ReplicaAndLadder:** One sync group contains all the IPUs. The other sync group is used independently by sets of IPUs, for example A+C and B+D. This means that each subset can communicate independently of each other. The two groups of IPUs can then alternate between host I/O and processing.

For more information on how the sync groups are used by the Poplar framework, please refer to the Poplar and PopLibs User Guide.

CHAPTER

SEVEN

SUPPORTED OPERATORS

PopART is compatible with ONNX versions up to and including 1.6. (see ONNX Versioning). This section lists the supported operators.

The Graphcore (ai.graphcore) and ONNX (ai.onnx) operators, and versions supported, are listed below. See ONNX Operators for more information.

7.1 Domain: ai.onnx

- Abs-6
- Add-6
- Add-7
- And-1
- And-7
- ArgMax-1
- ArgMax-11
- ArgMin-1
- ArgMin-11
- Asin-7
- Atan-7
- AveragePool-1
- AveragePool-7
- AveragePool-10
- AveragePool-11
- BatchNormalization-6
- BatchNormalization-7
- BatchNormalization-9
- Cast-6
- Cast-9
- Ceil-1
- Ceil-6
- Clip-6
- Clip-11

- Concat-1
- Concat-4
- Concat-11
- Constant-9
- Constant-11
- ConstantOfShape-9
- Conv-1
- Conv-11
- ConvTranspose-1
- ConvTranspose-11
- Cos-7
- Cosh-9
- CumSum-11
- Div-6
- Div-7
- Dropout-6
- Dropout-7
- Dropout-10
- Elu-1
- Elu-6
- Equal-1
- Equal-7
- Equal-11
- Erf-9
- Exp-6
- Expand-8
- Flatten-1
- Flatten-9
- Flatten-11
- Floor-1
- Floor-6
- GRU-3
- GRU-7
- Gather-1
- Gather-11
- GlobalAveragePool-1
- GlobalMaxPool-1
- Greater-1
- Greater-7

- Greater-9
- HardSigmoid-1
- HardSigmoid-6
- Identity-1
- If-1
- If-11
- InstanceNormalization-6
- IsInf-10
- IsNaN-9
- LRN-1
- LSTM-1
- LSTM-7
- LeakyRelu-1
- LeakyRelu-6
- Less-7
- Less-9
- Log-6
- LogSoftmax-1
- LogSoftmax-11
- Loop-1
- Loop-11
- MatMul-1
- MatMul-9
- Max-6
- Max-8
- MaxPool-1
- MaxPool-8
- MaxPool-10
- MaxPool-11
- Mean-6
- Mean-8
- Min-6
- Min-8
- Mul-6
- Mul-7
- Neg-6
- Not-1
- OneHot-9
- OneHot-11

- Or-1
- Or-7
- PRelu-9
- Pad-2
- Pad-11
- Pow-1
- Pow-7
- RandomNormal-1
- RandomNormalLike-1
- RandomUniform-1
- RandomUniformLike-1
- Reciprocal-6
- ReduceL1-1
- ReduceL1-11
- ReduceL2-1
- ReduceL2-11
- ReduceLogSum-1
- ReduceLogSum-11
- ReduceLogSumExp-1
- ReduceLogSumExp-11
- ReduceMax-1
- ReduceMax-11
- ReduceMean-1
- ReduceMean-11
- ReduceMin-1
- ReduceMin-11
- ReduceProd-1
- ReduceProd-11
- ReduceSum-1
- ReduceSum-11
- ReduceSumSquare-1
- ReduceSumSquare-11
- Relu-6
- Reshape-5
- Resize-10
- Resize-11
- Round-11
- Scan-9
- Scan-11

- Scatter-9
- Selu-1
- Selu-6
- Shape-1
- Shrink-9
- Sigmoid-6
- Sign-9
- Sin-7
- Sinh-9
- Slice-1
- Slice-10
- Slice-11
- Softmax-1
- Softmax-11
- Softplus-1
- Softsign-1
- Split-2
- Split-11
- Sqrt-6
- Squeeze-1
- Squeeze-11
- Sub-6
- Sub-7
- Sum-6
- Sum-8
- Tanh-6
- ThresholdedRelu-10
- Tile-1
- Tile-6
- TopK-1
- TopK-10
- TopK-11
- Transpose-1
- Unsqueeze-1
- Unsqueeze-11
- Upsample-9
- Where-9

7.2 Domain: ai.graphcore

- Abort-1
- AddLhsInplace-1
- Atan2-1
- AutoLossScaleProxy-1
- BinaryConstScalar-1
- BitwiseAnd-1
- BitwiseNot-1
- BitwiseOr-1
- BitwiseXnor-1
- BitwiseXor-1
- Call-1
- ConvFlipWeights-1
- Ctc-1
- CtcBeamSearchDecoder-1
- Detach-1
- DynamicAdd-1
- DynamicSlice-1
- DynamicUpdate-1
- DynamicZero-1
- Expm1-1
- Fmod-1
- Gelu-1
- GroupNormalization-1
- Histogram-1
- HostLoad-1
- HostStore-1
- IdentityLoss-1
- Init-1
- L1-1
- LSTM-1
- Log1p-1
- LossScaleUpdate-1
- MultiConv-1
- Nll-1
- Nop-1
- PackedDataBlock-1
- PrintTensor-1

- ReduceMedian-1
- RemoteLoad-1
- RemoteStore-1
- ReplicatedAllGather-1
- ReplicatedAllReduce-1
- ReplicatedAllReduceInplace-1
- ReplicatedReduceScatter-1
- Reshape-1
- Reverse-1
- Round-1
- Scale-1
- ScaledAdd-1
- ScatterReduce-1
- SequenceSlice-1
- ShapedDropout-1
- Square-1
- Subsample-1
- UnaryZeroGrad-1
- Zeros-1
- ZerosLike-1

CHAPTER

EIGHT

CUSTOM OPERATORS

This section explains how to implement a custom operator (op) in PopART. Code from the Leaky ReLU custom op example in the Graphcore GitHub repository will be used to illustrate the concepts.

8.1 Overview

You need to write some C++ classes to implement the op.

One is an implementations of the op as PopART's intermediate representation (IR). This is used during PopART's compilation process to transform and optimise the graph. There is also a Poplar implementation of the op, which provides the code that is run when the graph is executed. If the op will be used for training, then you also need gradient versions of these.

These classes are compiled to create a shared object library that can be linked with a Python program when it is run.

You also need to define an "operator identifier". This consists of a unique combination of domain, operator name and operator version strings. This is used to register the custom op classes with PopART so that it can be used.

There are two ways of using the new custom op: from the builder API or from an ONNX file.

- **Builder API**: You can include the new op with the builder API using the domain, op name and op version that match the custom op definition.
- **ONNX file:** You can reference the op from an ONNX file using a `NodeProto` definition that matches the custom op definition.

The custom op will then be instantiated from the shared object library and treated like any other op in PopART.

You can also provide an "op definition" when you register the custom op. PopART will use that to check that the correct inputs, outputs and attributes are provided, and are of the expected types.

8.1.1 Custom op classes

The two key base classes in PopART that define an op are:

- `Op`: the intermediate representation (IR) of an op in PopART. This provides methods that are called during PopART's optimisation passes and transformations of the compute graph. This representation of the op is decoupled from the Poplar implementation.
- `Opx`: a Poplar implementation of the op. This is the code that will actually be run on the IPU.

If the op is required for training, then a `GradOp` and `GradOpx` must also be defined for the gradient operation (see Fig. 8.1 and Fig. 8.2).

To make these classes visible to PopART, you must instantiate `OpCreator` and `OpxCreator` objects. These map from the string identifier of the new op (for example, "LeakyRelu"; see Section 8.3.1, Define the op identifier) to constructors for your newly-defined `Op` and `Opx` C++ classes.

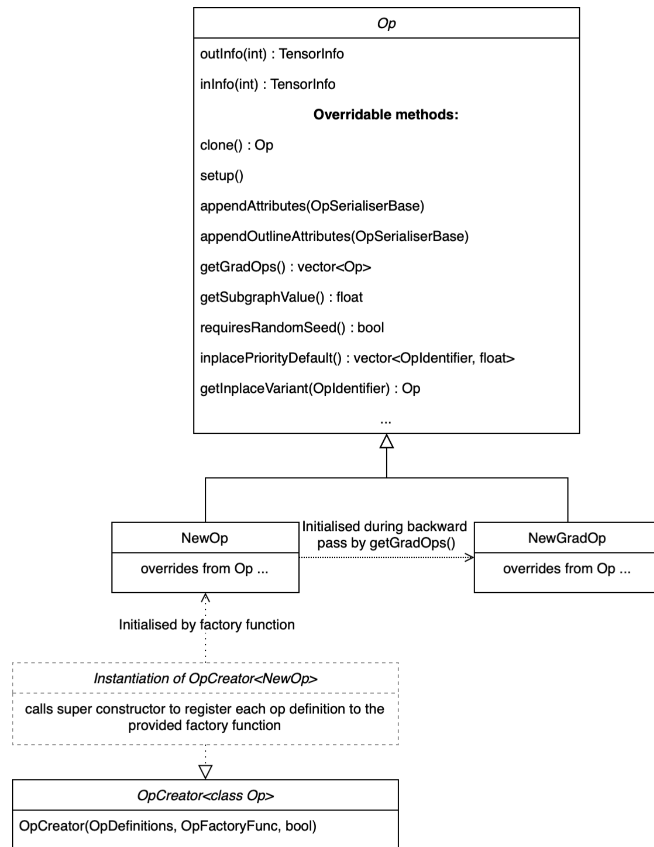

Fig. 8.1: Op class diagram

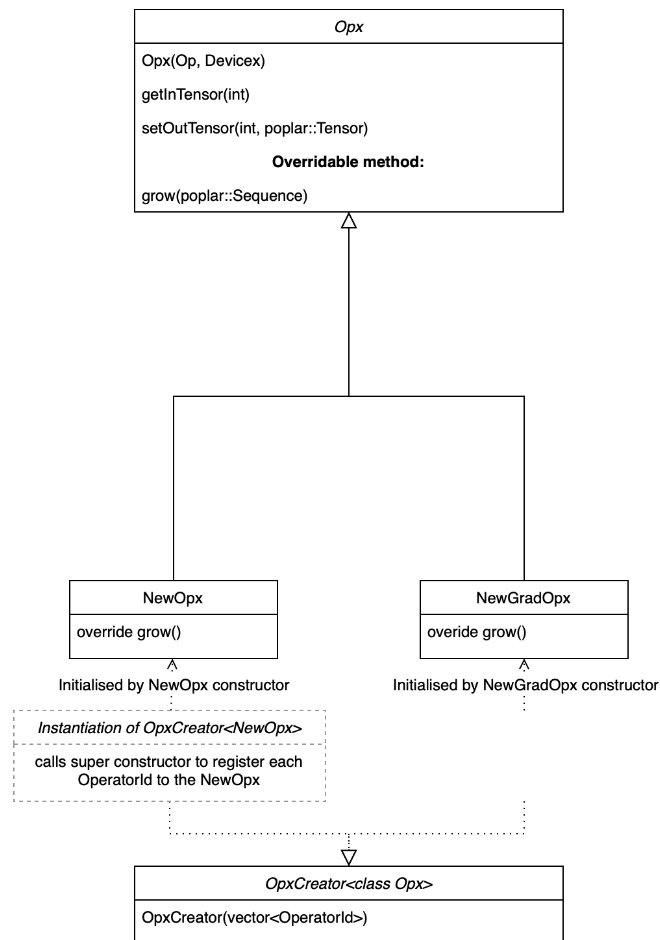

Fig. 8.2: Opx class diagram

These classes are compiled to create a shared object library that can be dynamically linked into the Python program at runtime, as shown below:

```
import ctypes

ctypes.cdll.LoadLibrary(so_path)
```

You can see how this is done in the LeakyReLU example.

8.2 Implementing a custom op

Some of the examples in the GitHub repository have a single C++ file that defines all of the classes for a custom op. Although this can make it easier to see everything in one place, it can be more difficult to follow. So, in this section the main elements of the `LeakyRelu` example are extracted with some more detailed descriptions of each method.

8.2.1 The op class

The Op base class provides the methods necessary for the PopART IR passes and transformations.

The main methods that you need to override or implement are:

- Attributes should be passed into the constructor and corresponding accessors defined.
- `clone()`: returns a copy of the op. Usually, this means returning a `std::make_unique` copy of the op. This must be implemented.
- `setup()`: sets the shape and type of the arguments to the op. This must set the type and shape information for all the output TensorInfo objects.
- `appendAttributes()`: appends attributes when serialising the op to a stream. This is used for some debugging purposes but also for generating the PopART IR hash. This hash is used to determine whether a Poplar cache can be reused so it is important that op attributes which may alter the Poplar compilation are appended to this stream. If this method is overridden, then it must also call the base class method.
- `appendOutlineAttributes()`: determines which ops are functionally equivalent during outlining.
- `getGradOps()`: returns a vector of `GradOp` object for each `Op` in the forward graph to automatically generate the backward pass. There can be a separate grad op for each input (this is usually cleaner to implement) or a single grad op that generates gradients for all inputs.

  The mapping from the index of each output tensor of the grad op to the index of each input tensor of the non-grad op is configured using the `gradOutToNonGradIn()` method that should be overridden in the `GradOp` definitions (see below).
- `getSubgraphValue()`: this is used by outlining algorithm to determine whether or not to outline ops. There are high and low bounding values retrieved by `getHighSubgraphValue()` (for expensive ops such as Conv) or `getLowSubgraphValue()` (for inexpensive ops such as Relu).
- `requiresRandomSeed()`: this is set to false by default. This should be overridden and set to true if an IPU random seed tensor is required by the op. If so it will be connected to `inTensor(getSeedInIndex())` by the IR process.
- `inplacePriorityDefault()`: if the op can be replaced by an in-place variant of itself, this method should be overridden to return a vector of `<OperatorIdentifier, float>` tuples in descending order of preference. For example, the LeakyReLU implementation for this is:

  ```
  return {{Onnx::CustomOperators::LeakyReluInplace, 10}};
  ```
- `getInplaceVariant()`: this is called to instantiate a particular in-place variant of the Op with a specified `OperatorIdentifier` from the vector returned by `inplacePriorityDefault()`.

LeakyReluOp example

```
class LeakyReluOp : public popart::Op {
public:
  LeakyReluOp(const popart::OperatorIdentifier &_opid, float _alpha,
              const popart::Op::Settings &settings_)
      : popart::Op(_opid, settings_), alpha(_alpha) {}

  std::unique_ptr<Op> clone() const final {
    return std::make_unique<LeakyReluOp>(*this);
  }

  void setup() final { outInfo(0) = inInfo(0); }

  void appendAttributes(popart::OpSerialiserBase &os) const override {
    Op::appendAttributes(os);
    os.appendAttribute("alpha", getAlpha());
  }
```

(continues on next page)

(continued from previous page)

```
  void appendOutlineAttributes(popart::OpSerialiserBase &os) const override {
    Op::appendOutlineAttributes(os);
    os.appendAttribute("alpha", getAlpha());
  }

  std::vector<std::unique_ptr<popart::Op>> getGradOps() {
    std::vector<std::unique_ptr<Op>> upops;
    upops.emplace_back(new LeakyReluGradOp(*this));
    return upops;
  }

  float getSubgraphValue() const final { return getHighSubgraphValue(); }

  bool requiresRandomSeed() const override { return false; }

  // Attributes
  float getAlpha() const { return alpha; }

private:
  float alpha;
};
```

8.2.2 The grad op class

```
class LeakyReluGradOp : public popart::Op {
public:
  LeakyReluGradOp::LeakyReluGradOp(const LeakyReluOp &fwdOp)
      : popart::Op(CustomGradOperators::LeakyReluGrad_6, fwdOp.settings),
        alpha(fwdOp.getAlpha()) {}

  std::unique_ptr<popart::Op> clone() const final {
    return std::make_unique<LeakyReluGradOp>(*this);
  }
  void setup() final { outInfo(0) = inInfo(0); };

  const std::vector<popart::GradInOutMapper> &gradInputInfo() const {
    static const std::vector<popart::GradInOutMapper> inInfo = {
        {0, 0, popart::GradOpInType::GradOut},
        {1, 0, popart::GradOpInType::In}};
    return inInfo;
  }

  // The Grad Op has 1 output, which is the gradient of the only input
  const std::map<int, int> &gradOutToNonGradIn() const {
    static const std::map<int, int> outInfo = {{0, 0}};
    return outInfo;
  }

  bool requiresRandomSeed() const override { return false; }

  // an estimate of how valuable sub-graph matching will be
  float getSubgraphValue() const final { return getHighSubgraphValue(); }

  float getAlpha() const { return alpha; }

  // Implementation defined below
  void appendAttributes(popart::OpSerialiserBase &os) const override {
    Op::appendAttributes(os);
    os.appendAttribute("alpha", getAlpha());
  }

  // Implementation defined below
  void appendOutlineAttributes(popart::OpSerialiserBase &os) const override {
    Op::appendOutlineAttributes(os);
    os.appendAttribute("alpha", getAlpha());
  }

private:
```

(continues on next page)

(continued from previous page)

```
  float alpha;
};
```

8.2.3 The opx class

The Opx class provides a `grow()` function that implements the corresponding `Op` definition as Poplar or PopLibs calls using the provided `program::Sequence`. Since `OpxCreator` uses a generic constructor, you should also check that the `Op` passed in is of the expected type and matches the `OperatorIdentifier`.

```
class LeakyReluOpx : public popart::popx::Opx {
public:
  LeakyReluOpx(popart::Op *op, popart::popx::Devicex *devicex)
      : popart::popx::Opx(op, devicex) {
    verifyOp<LeakyReluOp>(
        op, {CustomOperators::LeakyRelu_1, CustomOperators::LeakyRelu_6});
  }

  void grow(poplar::program::Sequence &prog) const final {

    auto op = getOp<LeakyReluOp>();

    poplar::Tensor input = getInTensor(0);

    float alpha = op.getAlpha();

    // x < 0.0f ? alpha * x : x
    auto expression = pe::Select(pe::Mul(pe::Const(alpha), pe::_1), pe::_1,
                                 pe::Lt(pe::_1, pe::Const(0.0f)));

    popops::mapInPlace(graph(), expression, {input}, prog,
                       debugContext("LeakyRelu"), poplar::OptionFlags());

    setOutTensor(0, input);
  }
};
```

8.2.4 The grad opx class

```
class LeakyReluGradOpx : public popart::popx::Opx {
public:
  LeakyReluGradOpx(popart::Op *op, popart::popx::Devicex *devicex)
      : popart::popx::Opx(op, devicex) {
    verifyOp<LeakyReluGradOp>(op, {CustomGradOperators::LeakyReluGrad_1,
                                   CustomGradOperators::LeakyReluGrad_6});
  }

  void grow(poplar::program::Sequence &prog) const final {

    auto op = getOp<LeakyReluGradOp>();

    poplar::Tensor grad = getInTensor(0);
    poplar::Tensor input = getInTensor(1);

    float alpha = op.getAlpha();

    // (grad * (x < 0.0f ? alpha : 1))
    pe::Mul expression = pe::Mul(pe::Select(pe::Const(alpha), pe::Const(1.0f),
                                            pe::Lt(pe::_2, pe::Const(0.0f))),
                                 pe::_1);

    auto output =
        popops::map(graph(), expression, {grad, input}, prog,
                    debugContext("LeakyReluGrad"), poplar::OptionFlags());
```

(continues on next page)

(continued from previous page)

```
    setOutTensor(0, output);
  }
};
```

8.3 Making the op available to PopART

After you have written the classes that implement the op, you will need to make the op available to PopART. This means defining an op identifier and using the op creator class to register the op with PopART.

8.3.1 Define the op identifier

The first step is to define an `OperatorIdentifier` with the domain, op name and op version so that the op can be found by the `builder.customOp()` call in PopART or by a reference to the op in an ONNX file.

The `OperatorIdentifier` is a structure with the components `domain`, `opName` and `opVersion`.

For example, from leaky_relu_custom_op.cpp:

```
namespace CustomOperators {
  const popart::OperatorIdentifier LeakyRelu_1 = {"ai.onnx", "LeakyRelu", 1};
  const popart::OperatorIdentifier LeakyRelu_6 = {"ai.onnx", "LeakyRelu", 6};
} // namespace CustomOperators

namespace CustomGradOperators {
  const popart::OperatorIdentifier LeakyReluGrad_1 = {"ai.onnx", "LeakyReluGrad", 1};
  const popart::OperatorIdentifier LeakyReluGrad_6 = {"ai.onnx", "LeakyReluGrad", 6};
} // namespace CustomGradOperators
```

8.3.2 Define the op creator

The op creator registers the the new Op with PopART.

The `OperatorIdentifier` and a factory function that generates the new Op class are passed to the constructor of `OpCreator` to create a mapping. When your program loads the shared object library, this `OpCreator` is instantiated and registers the new Op.

You can also pass in an `OpDefinition` that allows the `inputs`, `outputs` and `attributes` to be checked against those provided in the model implementation.

The `GradOp` class will be implicitly created when the overridden method `getGradOps()` is called during the backwards pass.

```
namespace {
static OpDefinition::DataTypes T = {DataType::FLOAT16, DataType::FLOAT};

static OpDefinition
    leakyReluOpDef({OpDefinition::Inputs({{"input", T}}),
                    OpDefinition::Outputs({{"output", T}}),
                    OpDefinition::Attributes({{"alpha", {"*"}}})});

static OpCreator<LeakyReluOp> leakyReluOpCreator(
    popart::OpDefinitions({{Onnx::Operators::LeakyRelu_1, leakyReluOpDef},
                           {Onnx::Operators::LeakyRelu_6, leakyReluOpDef}}),
    [](const OpCreatorInfo &info) {
      float alpha = info.attributes.getAttribute<popart::Attributes::Float>(
          "alpha", 1e-2f);
      // default epsilon is 10**(-2)
      return std::make_unique<LeakyReluOp>(info.opid, alpha, info.settings);
    },
    true);
} // namespace
```

8.3.3 Define the opx creator

You add the `Opx` definitions in a similar to the `Op`. In this case, a generic constructor of the Opx is always used of the form `Opx(Op *op, Devicex *devicex)`. For example:

```
static popart::popx::OpxCreator<LeakyReluOpx> LeakyReluOpxCreator(
    {CustomOperators::LeakyRelu_1, CustomOperators::LeakyRelu_6});
static popart::popx::OpxCreator<LeakyReluGradOpx>
    LeakyReluGradOpxCreator({CustomGradOperators::LeakyReluGrad_1,
                             CustomGradOperators::LeakyReluGrad_6});
```

8.4 ONNX schema and shape inference

To enable ONNX to use the op as part of an ONNX model, you must define a schema for it. This includes inputs, outputs, domain, and versions.

To register an `OpSchema`, you can use the macro `ONNX_OPERATOR_SCHEMA(name)` and then append the various functions in the class. See schema.h for more examples.

```
namespace ONNX {

void LeakyReluShapeInference(InferenceContext &ctx) {
  propagateShapeAndTypeFromFirstInput(ctx);
}

static const char LeakyReluDoc[] = "Performs a leaky ReLU operation on the input.";

ONNX_OPERATOR_SET_SCHEMA_EX(
    LeakyRelu,
    comAcme,
    "com.acme",
    1,
    false,
    OpSchema()
        .SetDoc(LeakyReluDoc)
        .Input(0, "X", "Input tensor", "T")
        .Output(0, "Y", "Output tensor", "T")
        .TypeConstraint(
            "T",
            {"tensor(float)", "tensor(int32)", "tensor(float16)"},
            "Constrain input and output types to signed numeric tensors.")
        .TypeAndShapeInferenceFunction(LeakyReluShapeInference));

static bool registerOps() {
  auto &d = ONNX_NAMESPACE::OpSchemaRegistry::DomainToVersionRange::Instance();
  d.AddDomainToVersion("com.acme", 1, 1);

  ONNX_NAMESPACE::RegisterSchema(
      GetOpSchema<ONNX_OPERATOR_SET_SCHEMA_CLASS_NAME(comAcme, 1, LeakyRelu)>());

  return true;
}

} // namespace ONNX
```

In the same namespace you can define the shape inference for the op. This allows ONNX to infer from the shape of the inputs the shape of the outputs. With simple operations, such as this example, the output shape is the same as the first input, so you can use the ONNX function `propagateShapeAndTypeFromFirstInput` from shape_inference.h.

There are other methods to use for shape inference in ONNX contained in that header. For example, numpy-style broadcasting, shape from attributes, and so on. Defining shape inference is optional, however you may encounter issues with operations later in your model if ONNX is not able to infer the input shape of an operation from earlier inputs.

8.5 Using the op in a program

The op can be referenced, using the values in the op identifer, in a Python program using the `builder`. For example, from run_leaky_relu.py:

```
output_tensor = builder.customOp(opName="LeakyRelu",
                                 opVersion=6,
                                 domain="ai.onnx",
                                 inputs=[input_tensor],
                                 attributes={"alpha": alpha})[0]
```

Or the op can be referenced from an ONNX file using a NodeProto definition that matches the domain, name and version of the op.

CHAPTER

NINE

ENVIRONMENT VARIABLES

There are several environment variables which you can use to control the behaviour of PopART.

9.1 Logging

PopART can output information about its activity as described in *Turning on execution tracing*. You can control the default level of logging information using environment variables.

9.1.1 POPART_LOG_LEVEL

This controls the amount of information written to the log output for all modules. Finer control can be achieved using *POPART_LOG_CONFIG*.

9.1.2 POPART_LOG_DEST

This variable defines the output for the logging information. The value can be "stdout", "stderr" or a file name.

The default, if not defined, is "stderr".

9.1.3 POPART_LOG_CONFIG

If set, this variable defines the name of a configuration file which specifies the logging level for each module. This is a JSON format file with pairs of module:level strings. For example, a file called `conf.py` can be specified by setting the environment variable:

```
export POPART_LOG_CONFIG=conf.py
```

To set the logging level of the devicex and session modules, `conf.py` would contain:

```
{
  "devicex":"INFO",
  "session":"WARN"
}
```

These values override the value specified in POPART_LOG_LEVEL.

9.2 Generating DOT files

9.2.1 POPART_DOT_CHECKS

PopART can output a graphical representation of the graph, in DOT format, when it constructs the intermediate representation (IR). The stages of IR construction where the DOT files is generated is controlled by this variable.

Supported values:

- FWD0
- FWD1
- BWD0
- PREALIAS
- FINAL
- ALL

These values may be combined using ":" as a separator. The example below shows how to set `POPART_DOT_CHECKS` to export DOT graphs for the FWD0 and FINAL stages.

```
export POPART_DOT_CHECKS=FWD0:FINAL
```

The values in `POPART_DOT_CHECKS` will be combined with any values that are defined in the session options.

9.3 Inspecting the Ir

9.3.1 POPART_IR_DUMP

If set, this variable defines the name of a file where the serialised ir will be written. The ir will be written either at the end of the ir preparation phase, or when an exception is thrown during the ir preparation phase.

CHAPTER

TEN

REFERENCES

- https://onnx.ai/
- https://pytorch.org/docs/stable/index.html
- Poplar and PopLibs User Guide

CHAPTER

ELEVEN

GLOSSARY

11.1 Sample

The smallest division of a data set.

11.2 Micro-batch size

The number of samples processed in a single execution of a graph on a single device. Also referred to as the machine batch size. The micro-batch shape, or the shape of input data as defined in the ONNX model, is therefore `[micro_batch_size, *sample_shape]`.

11.3 Replication factor

The number of graphs to be run in parallel over multiple devices. The weight gradients from each device will be accumulated before a weight update. Also referred to as "device replication factor" or "spatial replication factor". This is sometimes called data-parallel execution.

11.4 Accumulation factor

The weight gradients will be accumulated over this number of micro-batches in series before a weight update. Also referred to as "temporal replication factor".

Accumulation can be thought of as doing replication on a single device.

11.5 Batch size

This is defined as `micro-batch size * replication factor * accumulation factor`. This is the number of samples per weight update.

11.6 Batches per step

The number of batches to run in a single call to `Session::run`.

11.7 Step size

This is defined as `batch size * batches per step`. This is the number of samples per step.

11.8 Input data shape

Inputs to a `session.run()` call are read in with the assumption that data is arranged in the shape:

`[batches_per_step, accl_factor, repl_factor, micro_batch_size, *sample_shape]`

However, there is no constraint of the shape of the input array, except that it has the correct number of elements.

CHAPTER

TWELVE

TRADEMARKS & COPYRIGHT

Graphcore® and Poplar® are registered trademarks of Graphcore Ltd.

AI-Float™, Colossus™, Exchange Memory™, Graphcloud™, In-Processor-Memory™, IPU-Core™, IPU-Exchange™, IPU-Fabric™, IPU-Link™, IPU-M2000™, IPU-Machine™, IPU-POD™, IPU-Tile™, PopART™, PopLibs™, PopVision™, PopTorch™, Streaming Memory™ and Virtual-IPU™ are trademarks of Graphcore Ltd.

All other trademarks are the property of their respective owners.

© Copyright 2016-2020, Graphcore Ltd.